Program Analysis and Verification - Homework 3 and Final Project

Shaked Rafaeli

January 17, 2019

Contents

1 Numerical Analysis

1.1 Overview

The first type of analysis has been implemented in C++. The abstract semantics used are the Cartesian Product of Constant Propagation and Variable Equality. This type of analysis allows both the *Parity Analysis* and the *Sum Analysis* required, and uses *Reduce* operations to derive conclusions between the two different lattices.

The following table describes the classes used and their use. Then, a short description will be given of the analysis algorithm. Finally, I will conclude with advantages and disadvantages of this analysis with code examples. To run the code, Change the path to the input .txt file in *main.cpp*. Notice that correct brackets are critical for programs to be parsed correctly.

1.2 Instructions

To run the code, change the path to the input .txt file in *main.cpp*. Notice that correct brackets are critical for programs to be parsed correctly. Then, compile and run using your preferred C++ IDE (tested with Visual Studio 2017 Community).

1.3 Objects

The following table describes the objects used. Although the code is thoroughly documented, this overview should simplify it.

Class	**Description**
CFG	Control Flow Graph. contains the states and edges
StateNode(abstract)	A node describing a state in the CFG
NA-StateNode	Derived StateNode for a state in a Numerical Analysis CFG Contains a list $(var1, var2) \in E$ and assignments $(var, constant) \in A$
Edge	CFG edge Points to its source, destination Contains the command performed when edge is traversed
Command (abstract)	Code command on an edge
NA-Command	Derived Command for Numerical Analysis Contains variables or constants depending on command type Contains an expression if its an *Assert* or *assume*
Expression(abstract)	Logical expression, can be conjunction or disjunction
NA-Expression	Logical expression for Numerical Analysis A disjunction of expressions or a conjunction of predicates
AtomicExpression (abstract)	Basic predicates for any analysis
NA-AtomicExpression	Basic predicates for Numerical Analysis Types are:(even i) , (odd i) (i == j) , (i != j) (i == K) , (i != K) (K_1 == K_2) , (K_1 != K_2)
Variable	Named variable for any analysis
Constant(abstract)	Abstract constant for analyses
NA-Constant	Constant for Numerical Analysis. Contains a value, its parity and a $\top/\bot$ indicator

1.4 Analysis Algorithm

The analysis follows Algorithm 1:

Algorithm 1 Analysis

1: Parse the given code into a control-flow graph , and add a "fail" node to which any *assert* command is also connected. Denote the resulting graph $G = (V, E)$.
2: Let S be a stack of states (our worklist)
3: Let $u =$ first node added *Push* u
4: Let $v = S.pop()$. If $v = fail$, throw exception
5: For every edge $e = (v, w) \in E$:

1. Perform command in e on v's current state, i.e. $t = e(v)$
2. **if** Command is "assume" **then** use $join$ and push if condition $\neq$ FALSE
3. **if** Command is "assert" **then**
   - If assertion $\neq$ TRUE, $S.push(fail)$, else update
4. **else**
   - $w = join(t, w)$
   - If w was changed, $S.push(w)$

6: **if** S not empty **then** go to step 4
7: **else** Fixed point

In the numerical analysis two lattices are used: Constant Propagation and Variable Equality. The proof can be seen in the Lesson 6 slides 23-34. The state of every NA-StateNode is built out of a value out of the Variable Equality lattice and a value out of the Constant Propagation lattice (a NA-Constant). A Constant Propagation NA-constant contains an integer $i \in \mathbb{N} \cup \{\bot, \top\}$ and a parity bit.

Expressions in the analysis use 3-value logic, and so the following rules have been set regarding 3-value logic:

- Assume edges are handled conservatively: The destination is updated and set to the analysis if the condition is either TRUE or UNKNOWN.
- Assert edges are also conservative: They *fail* if the condition is anything other than TRUE

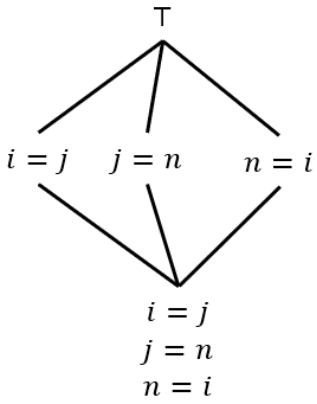

Figure 1: Variable Equality lattice of 3 variables (n,j,i)

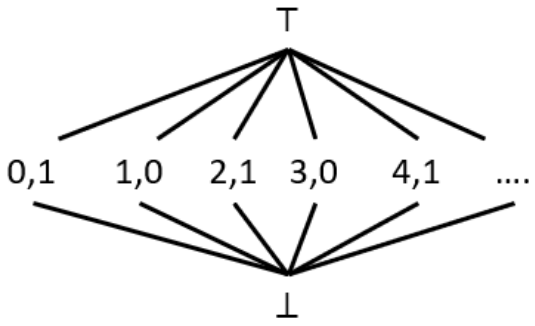

Figure 2: Constant Propagation lattice

Notice that the Numerical Analysis provided uses *Reduce right* and *Reduce left* to derive information between the two lattices. This process allows for successful assertions in some cases, but information loss still occurs. Included are 5 code examples that use both `even/odd` and `sum` operations. The ability to prove an assertion is heavily dependent on the conditions of the loop, and whether we can draw information from the `assume` command that leads from the loop.

The strength of this analysis is that it terminates in very few iterations, while not losing all its information. The loops in the example programs are analyzed only several times, even in programs such as *ParityTest2.txt* which in reality have a large number of iterations. If, for example, we were to use disjunctive completion, the overhead in memory and analysis time for such a program would greatly increase.

1.5 Code Examples

In the following code examples, notice that the assertions are proven in all but *ParityText4.txt*, which shows a weakness of this analysis: The loop causes information loss, since the invariant contains no equality and is too conservative over the value of m. Since the condition that ends the loop also contains no information over m (it depends on n only), no assertion can be proven about m, it is simply $\top$.

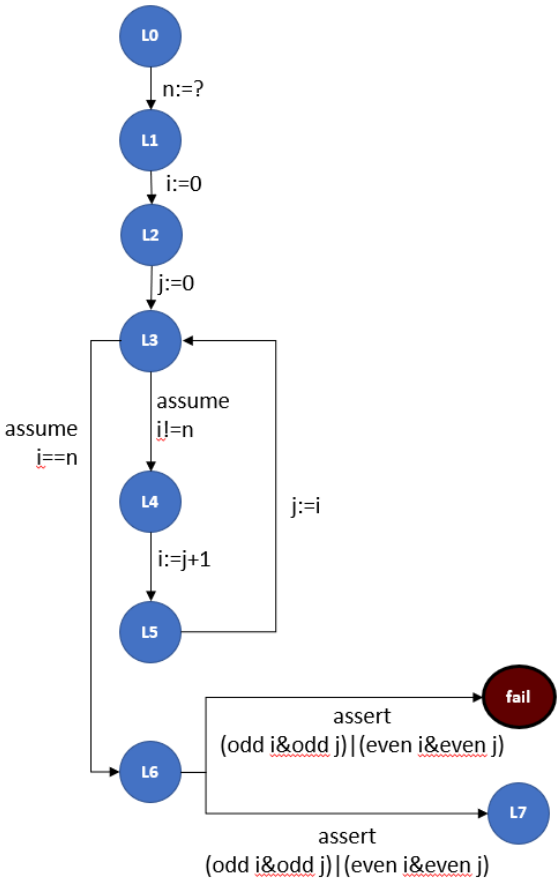

Figure 3: ParityTest1

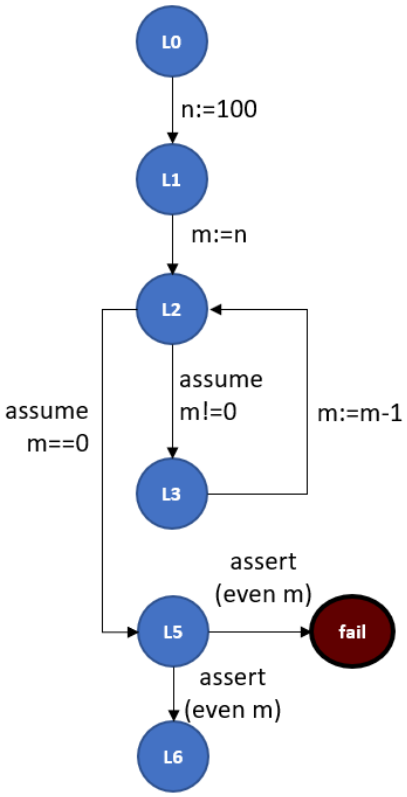

Figure 4: ParityTest2

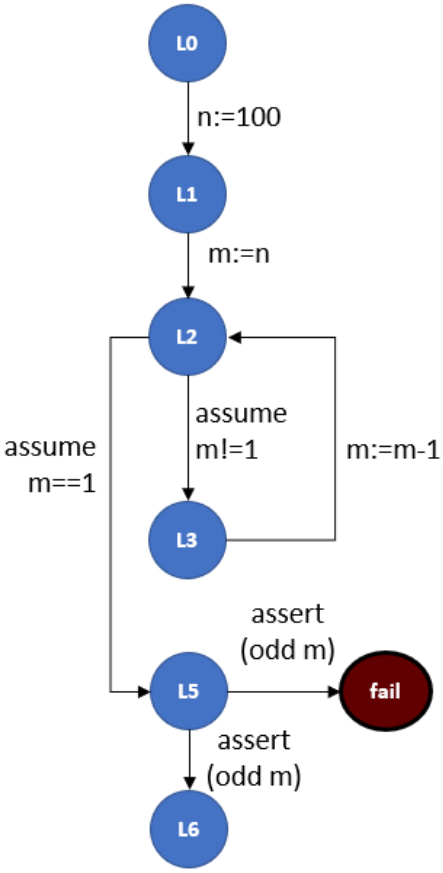

Figure 5: ParityTest3

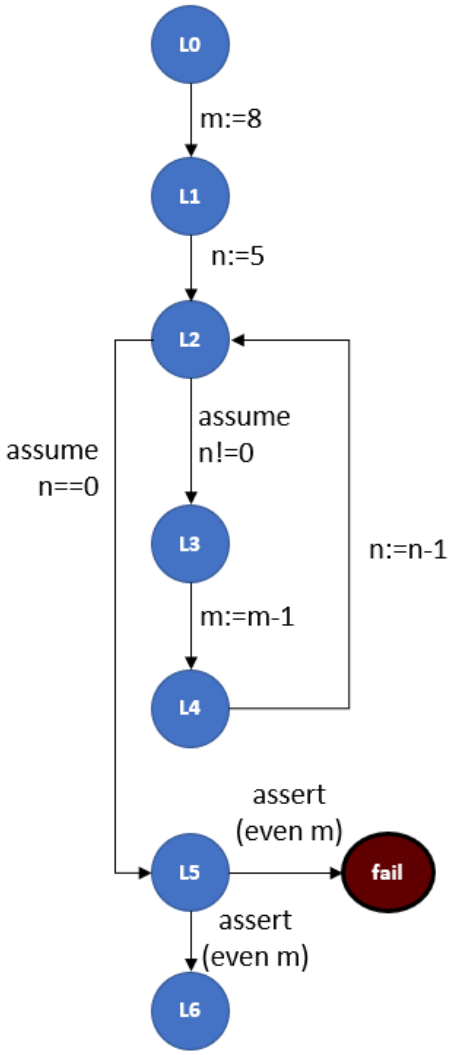

Figure 6: ParityTest4

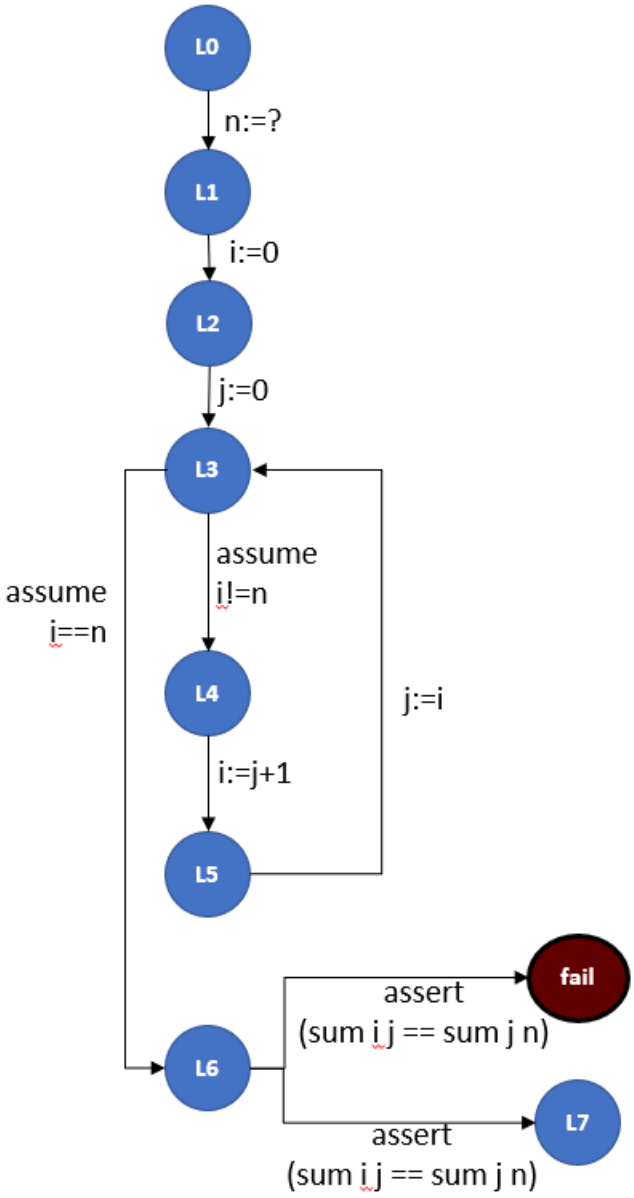

Figure 7: SumTest1

Claim. *The analysis is terminating.*

Proof. An analysis terminates when the work-list stack is empty, or a "fail" state had been drawn from the stack. A state holds an item from the VE lattice and one item from the CP lattice per variable. Since a state changes only using the *join* operation, by the fixed-point theorem applied to each lattice item, the transformers on this state all fulfill the fixed-point theorem as well. If every state reaches a fixed point using the given transformers then using chaotic iteration all states reach a fixed point, resulting in no more states added to the work-list. □

The last program I added is a version of factorial calculation using the project's given language. This analysis does not successfully assert that 5! was calculated successfully. The reason is once again loss of information inside loops. The variable named *collected* is where the value is continuously incremented until it reaches the factorial, but the loop-head states lose its value as it becomes $\top$.

Algorithm 2 Factorial 5 - PseudoCode

```
1: fact = 5
2: res = 120
3: collected = fact
4: for current-mul in range(fact-1, 1, -1):
    1. current-collected = collected
    2. for times-to-add in range(current-mul, 1, -1):
        (a) for inc-counter in range(current-collected, 0, -1):
                i. collected += 1
5: assert collected == res
```

Algorithm 3 Factorial 5 - CFG syntax

```
fact res collected current-mul current-collected times-to-add counter
L0 fact:=5 L1
L1 res:=120 L2
L2 collected:=fact L12
L12 current-mul:=fact-1 L3
L3 assume(current-mul!=1) L4
L3 assume(current-mul==1) L10
L4 current-collected:=collected L5
L5 times-to-add:=current-mul L6
L6 assume(times-to-add!=1) L7
L6 assume(times-to-add==1) L15
L15 current-mul:=current-mul-1 L3
L7 counter:=current-collected L8
L8 assume(counter!=0) L9
L8 assume(counter==0) L14
L14 times-to-add:=times-to-add-1 L6
L9 collected:=collected+1 L13
L13 counter:=counter-1 L8
L10 assert((collected==res)) L11
```

2 Shape Analysis

2.1 Overview

The Shape analysis has been implemented in Python, with a graphic output and debugging interface using the *networkx* library. It follows the analysis logic described in *"The Compiler Design Handbook: Optimizations and Machine Code Generation", Chapter 12, "Shape Analysis and Applications" - Reps, Sagiv, Wilhelm.* The analysis algorithm constructs a CFG and analyzes it by maintaining a worklist queue. The queue holds the CFG's next states to be analyzed. Every state holds a set of abstract "heaps", which are implemented using collections of Kleene-logic predicates. These heaps are the heaps possible in that specific state, i.e., the lattice used for every state is the disjunctive completion of predicate-represented heaps. Every iteration, the next state is drawn from the queue and each out-going edge from that state is analyzed, with the result of the transformer of that edge updating that edge's destination using *join*.

In order to avoid information loss, I implemented multi-stage abstract semantics. Every time an edge is analyzed, every heap in its source-state undergoes a *focus* (partial concretization) operation,

which separates the abstract heap, in which all predicates may evaluate to $\frac{1}{2}$, to partially concrete heaps in which a specific area of interest evaluates to 0 or 1 (and by that brings that area "into focus"). After partial concretization, the transformer on the edge is applied to each of the heaps, and then each undergoes a *coerce* operation, designed to discard contradictory heaps and to deduct wherever possible whether a predicate evaluating to $\frac{1}{2}$ can be in fact evaluated to 0 or 1 using the instrumentation predicates.

The analysis requires only two of the simplifying assumptions mentioned in the project guidelines:

1. Every node contains a single field $.n$. The code does check for this property to hold in many cases, and the effort required to relaxing this assumption as well is not substantial, e.g., the *focus* operation can also focus multiple outgoing $.n$ fields.

2. Pointer variables are initialized to NULL and when a node is instantiated on the heap its $.n$ field is initialized to NULL.

The other two assumptions are not necessary for this analysis, namely, $.n$ fields can be set without a preceding `x.n:=NULL` command and more importantly, the analysis *works* even when an arbitrary number of nodes share the same node as their $.n$. Furthermore, the analysis does not halt when cycles are introduced by the program, and instead maintains an instrumentation predicate $\{c(v)\}$.

2.2 Instructions

The Shape Analysis is most comfortably accessed using the command line, unless intermediary cfgs and heaps are of interest and then use any IDE to access the debug options. The script uses the first argument passed to it as the file name that should sit in the same directory as "SA.py". For example, to analyze the code in 'ShapeAnalysisBasicLoop' the following command should be used:
`python SA.py ShapeAnalysisBasicLoop.txt`

Notice that when running SA.py from the command line, Python won't run it well if there's Hebrew in the path.

The code then runs and prints the states it analyzes. Whenever it reaches an 'assert' command, it prints out that the assertion was either a success or a failure, and outputs '.png' files that give the graphic view of the heaps in the state on which the assertion was made. These '.png' files are time-stamped and saved in the same directory as `SA.py`. When the analysis is complete, the last state analyzed is also output in the same way. Since in large code files that result in a large number of heaps outputting the asserted state every time is cumbersome, this output begins disabled. It can be enabled/disabled by removing/adding the comment from the command `draw_state_to_png(current_src_state)` in `Cfg.analyze()` (line 150).

Additional functionality can be accessed when debugging the code. At any moment the code can be paused and graphic views of both the CFG and any heap are possible using the following Python commands:

- `draw(Heap name)` - Shows a graphic view of the variable in Heap name, e.g. `draw(self)` if we're in a Heap method or `draw(self.states['L10'].heaps[0])` in a Cfg method.
- `draw_state(State name)` - Opens a graphic window for every heap in the input State.
- `draw_state_to_png(State name)` - Saves all the heaps of that state to '.png' files.
- `draw_cfg(Cfg name)` - Opens a window with a graphic view of the given Cfg.
- `cd()` - Clears all windows to avoid drawing over previous graphic figures.

In the graphic view of a heap, red nodes are regular nodes, grey nodes are summary nodes and the 'NULL' node is purple. Links are portrayed by the arrows when a black arrow is a 1 in the $.n$ field and a faded arrow is a $.n$ field evaluated to $\frac{1}{2}$.

2.3 Objects

The following table describes the classes in the code:

Class	Members
Cfg	* dictionary of (state name, State) * list of edges * State - fail state * State - start state * list of variables - variables
State	* list of Heap - heaps * list of Edge - out_edges * list of Edge - in_edges * Cfg - the containing graph (pointer)
Edge	* State - src * State - dst * Transformer - op
Heap	* binary predicate - var_pts_to * unary predicate - is_shared * binary predicate (predecessor, next) - next * unary predicate - is_summary * binary predicate - reachable * unary predicate - in_cycle * int - max heap index, "next new" node * list of int - heap items (nodes)
Command (abstract)	Code command on an edge
Transformer	* string - operation * Expression - expression * string - arg1 * string - arg2 * string - entire command Contains an expression if its an *Assert* or *assume*
Atomic Expression	int - type string - arg1 string - arg2 string - arg3 string - arg4
Expression	* list of atomic expressions - atomics * bool - disjunction

2.4 Algorithm

The main algorithm that analyzes the CFG is described in Algorithm 4. The worklist is a data structure that does not hold duplicates, so if an edge to a state L_x is analyzed, L_x is not enqueued more than once. The algorithm starts at the start node and iterates until reaching a fixed point. Whenever an edge is analyzed, the algorithm checks whether the analysis caused any change in the destination state. If no change is detected, the destination is *not* enqueued to the worklist.

Additional notes regarding the algorithm:

- The analysis accepts and analyzes cycles successfully, while keeping track of the *cycle* instrumentation predicate.

- The analysis supports more than just singly linked, acyclic lists. As such, more than one path may exist between nodes on the heap.
- For evaluation of `EVEN`, `ODD` and `LEN` the shortest path is taken into account, e.g., if there are two paths between v and u, the expression is evaluated with regard to the shortest non-recurring path.
- Heaps may be duplicated but with slightly different node *id*s. That is a result of using disjunctive completion and a relatively simple method of evaluating whether two heaps are equivalent, instead of trying to check graph isomorphism between heaps.

Algorithm 4 Shape Analysis

1: Parse the given code into a control-flow graph , and add a "fail" node to which any *assert* command is also connected. Denote the resulting graph $G = (V, E)$.
2: Let S be a stack of states (our worklist)
3: Let $u =$ first node added *Push* u
4: $v = S.pop()$
5: $\forall e = (src, dst) \in E$:

1. $H = \{Focus(h) | h \in src.heaps\}$
2. **if** Command is "assert" **then**
3. **if** $\exists h \in H$ s.t. condition doesn't hold **then** failed assertion, abort
4. Apply transformer f to H to get $f(H)$
5. **if** `NULL` dereference detected **then** Throw exception, abort
6. $W = coerce(f(H))$
7. **if** $join(canonical\_abstraction(W), dst.heaps) \neq dst.heaps$ **then**
   - $dst.heaps = join(canonical\_abstraction(W), dst.heaps)$
   - $\forall h \in dst.heaps : h.rename()$
   - $S.push(dst)$

6: **if** S not empty **then** go to step 4
7: **else** Fixed point

The analysis separates the predicates in the following way:

- Abstraction predicates: $is\_shared(v), x(v), reachable(x, v)$
- Instrumentation predicates: $next(v1, v2), cycle(v)$

Claim. *The analysis in Algorithm 4 is terminating*

Proof. The proof is similar to the proof of Algorithm 1 , only now the use of disjunctive completion of all possible heaps may produce a lattice of infinite height. For this purpose, a *Renaming* algorithm has been implemented in the code.

Definition 2.1 (Renaming)**.** Given a graph $G = (V, E)$ in which every vertex has an arbitrary id, a *Renaming* protocol outputs the same graph G in which $\forall v \in V, id_x \leq |V|$.

The key to the proof is that unlike the entire disjunctive completion lattice, there is a finite upper bound to the lattice of every State in the Cfg, since there is a finite maximum number of

nodes possible in heaps of a specific state. This can be proved using Structural Induction on all the transformers in the analysis. Informally, since the number of variables in the Cfg is constant, and *is_shared* nodes are limited by the structure of the graph, the canonical abstraction merges nodes to summary nodes resulting in a local maximum number of nodes possible in every state.

Since the *Renaming* algorithm is run over all heaps using the `Heap.rename()` method, even though two states with the same structure may be included in the same State, there is only a finite number of possible permutations of $|V|$ names.

So, the number of possibilities for the abstraction predicates is finite, afterwards any additional nodes must, somewhere in the graph, be merged by canonical abstraction in the graph. So there is a finite number of nodes per heap. The number of node-*id* permutations is finite. Thus, there is a finite number of possible heaps per state. So for every state, the analysis reaches a point it no longer pushes that state to the worklist. When that happened for every state, the analysis terminates.

□

In reality, the way the analysis runs scarcely creates such duplicates. The possibility of two heaps in the same state looking the same with different node *id*s is hard to avoid - as that would require solving a variant of Graph Isomorphism.

2.5 Code Examples

The following code examples show some of the advantages and disadvantages of the analysis. For each program, output examples are provided as well as a graphic of the resulting Cfg.

2.5.1 Basic Loop

Algorithm 5 ShapeAnalysisBasicLoop.txt

```
x y z w
L0 x:=new L1
L1 y:=new L2
L2 x.n:=y L3
L3 z:=new L4
L4 y.n:=z L5
L5 y:=z L6
L6 z:=new L7
L7 y.n:=z L12
L12 y:=y.n L13
L13 z:=new L14
L14 y.n:=z L15
L15 y:=y.n L8
L8 w:=z L9
L9 x:=x.n L10
L10 assume(x==w) L11
L10 assume(x!=w) L9
L11 assert(x==w) L20
```

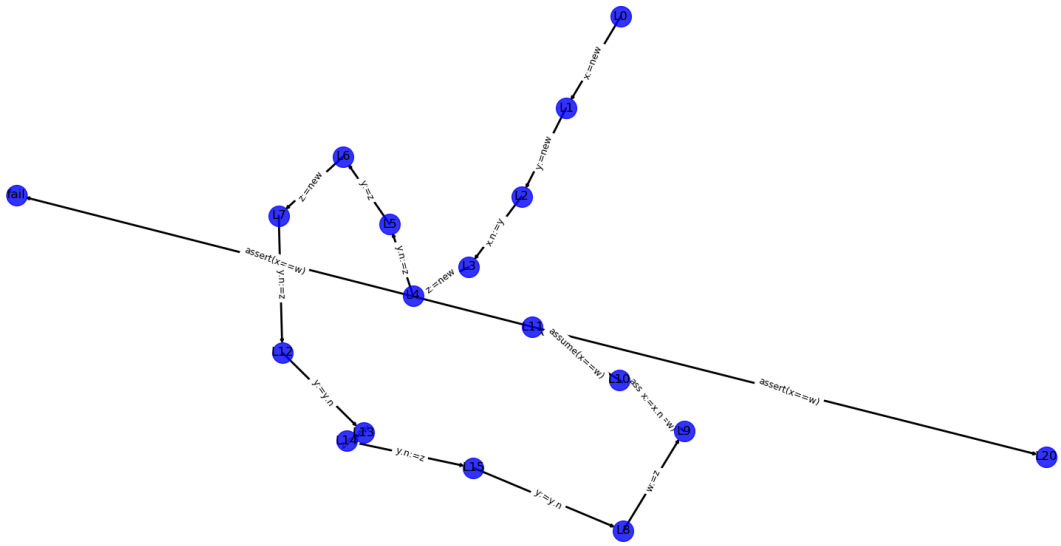

Figure 8: CFG created for *"ShapeAnalysisBasicLoop.txt"*

The code in *"ShapeAnalysisBasicLoop.txt"* shows how the analysis handles loops, and that assertions can be reached even if a large part of the heap has been summed in summary nodes. In the code, a singly linked list is created with several nodes, and then x is advanced until reaching the end of the list. The resulting Cfg from this code can be seen in the following figure. When running the analysis, it completes and successfully asserts that x had reached w at the end of the list. The analysis finds 2 possible heaps for state L20, representing different possible lengths of the list.

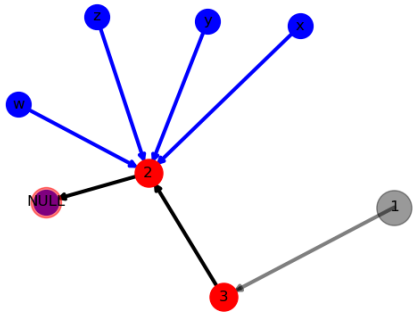

Figure 9: L20 heap 1

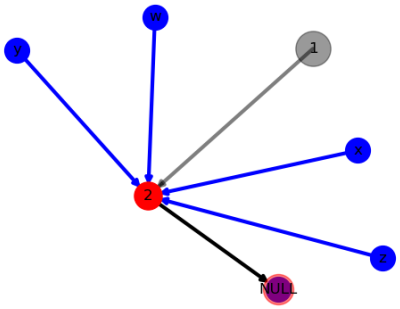

Figure 10: L20 heap 2

2.5.2 Null Dereference

Algorithm 6 ShapeAnalysisNullDereference.txt

```
x y z
L0 x:=new L1
L1 y:=new L2
L2 z:=new L3
L3 x.n:=y L4
L4 y.n:=z L5
L5 y:=new L6
L6 z.n:=y L7
L7 z:=new L8
L8 y.n:=z L9
L9 y:=new L11
L11 z.n:=y L12
L12 y:=NULL L13
L13 z:=NULL L14
L14 x:=x.n L15
L15 assume(TRUE) L14
```

"ShapeAnalysisNullDereference.txt" shows an example of a non-trivial NULL reference detection by the analysis. A list is created with several nodes, so the analysis would sum them up. The program then advances `x` indefinitely. Obviously, `x` would have eventually reached null. The program quickly analyses the possible null dereference and outputs the state it evaluated. Added are two of the possible heaps at L14 - one is legit but the other is a null reference.

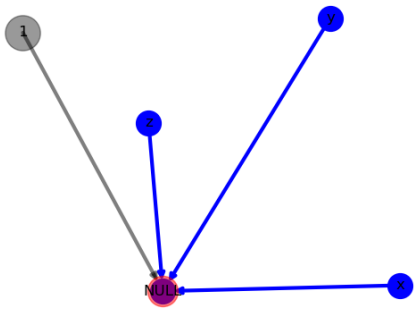

Figure 11: `x` gives a null dereference

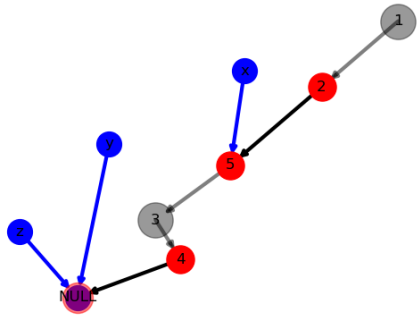

Figure 12: `x` advances through the list

2.5.3 Multiple Shared

Algorithm 7 ShapeAnalysisMultipleShared.txt

```
x y z w xx yy end
L0 x:=new L1
L1 y:=new L2
L2 z:=new L3
L3 x.n:=z L4
L4 y.n:=z L5
L5 w:=new L6
L6 z.n:=w L7
L7 xx:=new L8
L8 yy:=new L9
L9 z:=new L10
L10 xx.n:=z L11
L11 yy.n:=z L12
L12 z.n:=w L13
L13 z:=new L14
L14 w.n:=z L15
L15 end:=new L16
L16 z.n:=end L17
L17 assert(ODD x end) L18
L18 assert(ODD yy end) L19
L19 w:=end L20
L20 z:=end L21
L21 assert(LEN xx z == LEN y w) L22
L22 assert(ODD yy end) L22
```

ShapeAnalysisMultipleShared.txt demonstrates how the analysis faces multiple shared nodes on the heap. It creates several shared nodes and asserts both on equality of the length of paths and whether a different path is of odd length. The analysis successfully proves the assertions and outputs this heap at state L22:

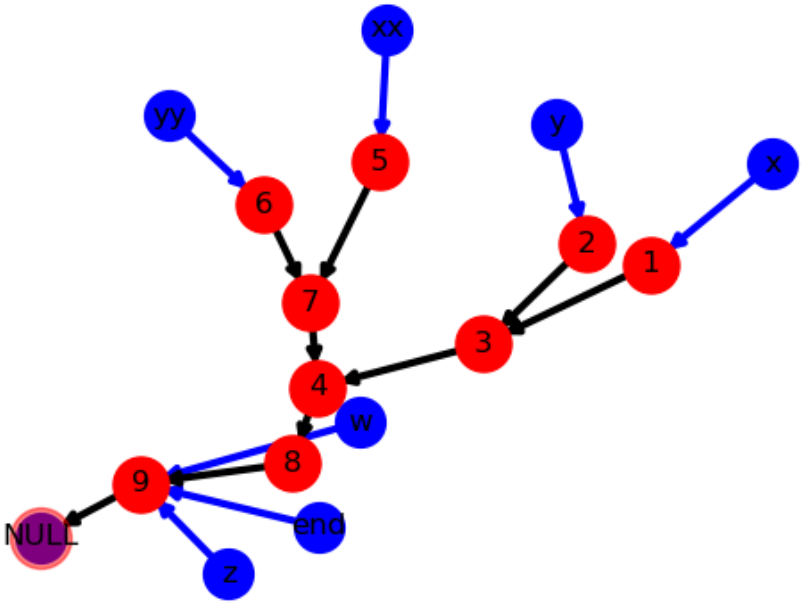

Figure 13: Output heap at L22

2.5.4 Infinite Shared

Algorithm 8 ShapeAnalysisInfiniteShared.txt

```
x y z w
L0 x:=new L1
L1 y:=new L2
L2 z:=new L3
L3 w:=new L4
L4 x.n:=z L5
L5 y.n:=z L6
L6 z.n:=w L7
L7 z:=new L8
L8 x:=new L9
L9 w.n:=z L10
L10 w:=new L11
L11 x.n:=w L12
L12 z.n:=w L13
L13 assume(TRUE) L7
L13 assume(TRUE) L20
L20 assert (LS x w) L21
```

ShapeAnalysisInfiniteShared.txt is a more extreme version of *ShapeAnalysisMultipleShared.txt*. It involves an infinte loop that keeps creating more reverse forks, and thus more and more shared

nodes. The analysis still manages to maintain its grip and make assertions, while protecting against null dereferences. Attached are two examples of heaps at L20 when the assertion is made, out of a total of 5. These heaps show that even though a many shared nodes are involved, the analysis successfully sums them according to the abstraction predicates and maintains relevant heaps, enough to make assertions.

Notice that had we changed `L20 assert (LS x w) L21` to `L20 assert (LS y w) L21` , the assertion would fail. this is because the two heaps in the example are summed by the analysis - and y is connected to the area "beyond" the summary nodes only using $\frac{1}{2}$ edges. If $\frac{1}{2}$ edges stand for "unknown", we cannot prove any assertion about the reachability.

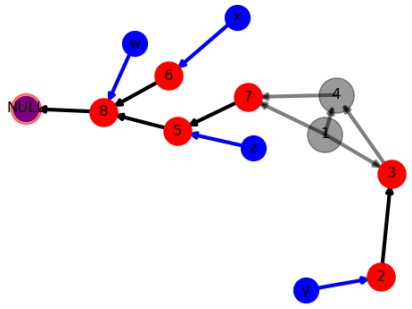

Figure 14: Heap example from L20

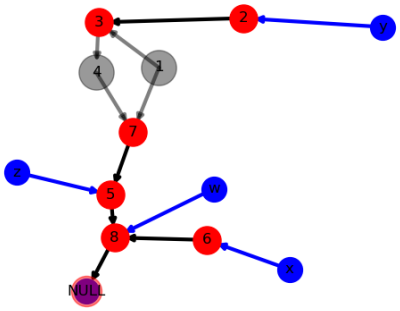

Figure 15: Heap example from L20

2.5.5 Cycle

Algorithm 9 ShapeAnalysisCycle.txt

x y z holder runner
L0 x:=new L1
L1 y:=new L2
L2 x.n:=y L3
L3 z:=new L30
L30 holder:=z L4
L4 y.n:=z L5
L5 z.n:=x L6
L6 x:=new L7
L7 z:=new L8
L8 x.n:=y L9
L9 z.n:=x L10
L10 x:=new L11
L11 x.n:=z L12
L12 runner:=x L13
L13 runner:=runner.n L14
L14 assume(runner==y) L15
L14 assume(runner!=y) L13
L15 runner:=runner.n L16
L16 assume(runner!=y) L15
L16 assume(runner==y) L17
L17 assert(LS x y) L18

ShapeAnalysisCycle.txt is an example of how SA.py handles cycles in the heap and still manages to make assertions. The code creates a small cycle and then a tail, and sends the *runner* variable to traverse the tail, into the cycle and then through the cycle again. Notice that the *holder* variable was added to prevent the analysis from summing up the cycle, making it easier to see in the figure.

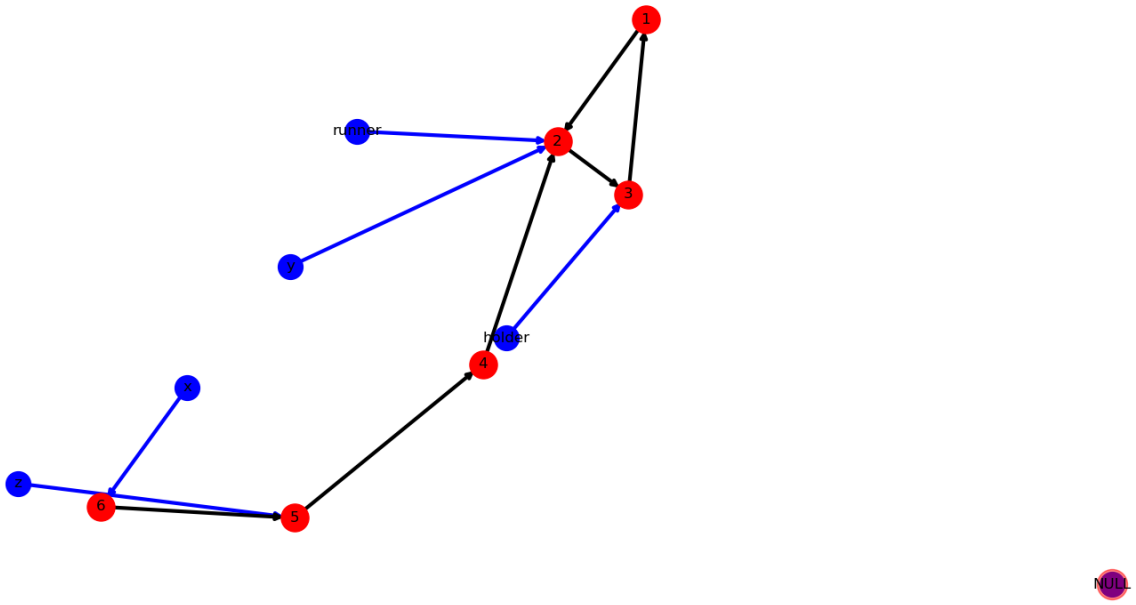

Figure 16: Output heap at L18

In addition, a version without the *holder* variable is also included, i.e. , the code is the same except that the line `L30 holder:=z L4` is skipped. The same assertions can be made and the analysis successfully follows with possible positions of *runner*. Also, we see a good example of how the focus operation changes summary nodes in several ways, since when *runner* traverses the summarized cycle we consider several partially concrete heaps. The following heaps are output at L18:

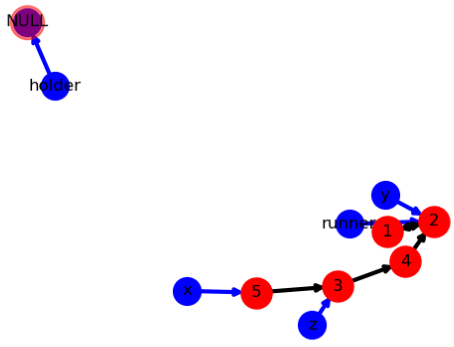

Figure 17: Heap output from `ShapeAnalysisCycle2.txt` at L18

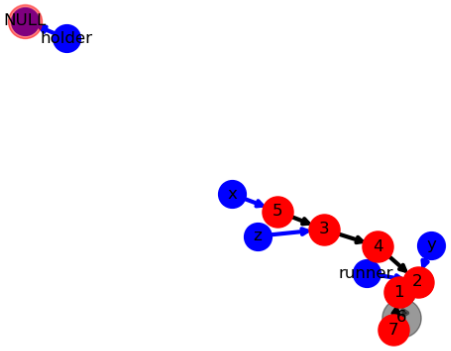

Figure 18: Heap output from `ShapeAnalysisCycle2.txt` at L18

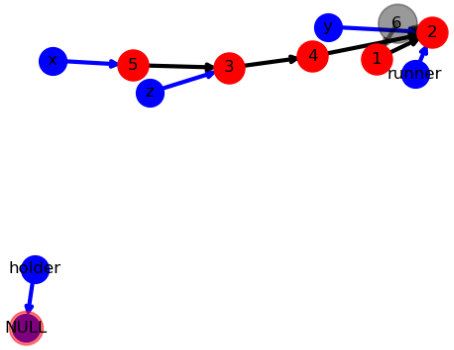

Figure 19: Heap output from `ShapeAnalysisCycle2.txt` at L18