Lazy Heap Analysis with Symbolic Memory Graphs

Alexander Driemeyer

Outline

1. Motivation

2. CPAchecker and Symbolic Memory Graphs

3. Abstractions of Symbolic Memory Graphs

4. Using counterexample guided abstraction refinement with Symbolic Memory Graphs

5. Challenges and conclusion

Motivation

- Use symbolic memory graphs to verify programs with complex heap structures
- Use abstraction to be able to check all possible states of a program for the specified safety property
- Use abstraction refinement to find a level of abstraction that is as coarse as possible while still fine enough to eliminate all spurious safety property violation

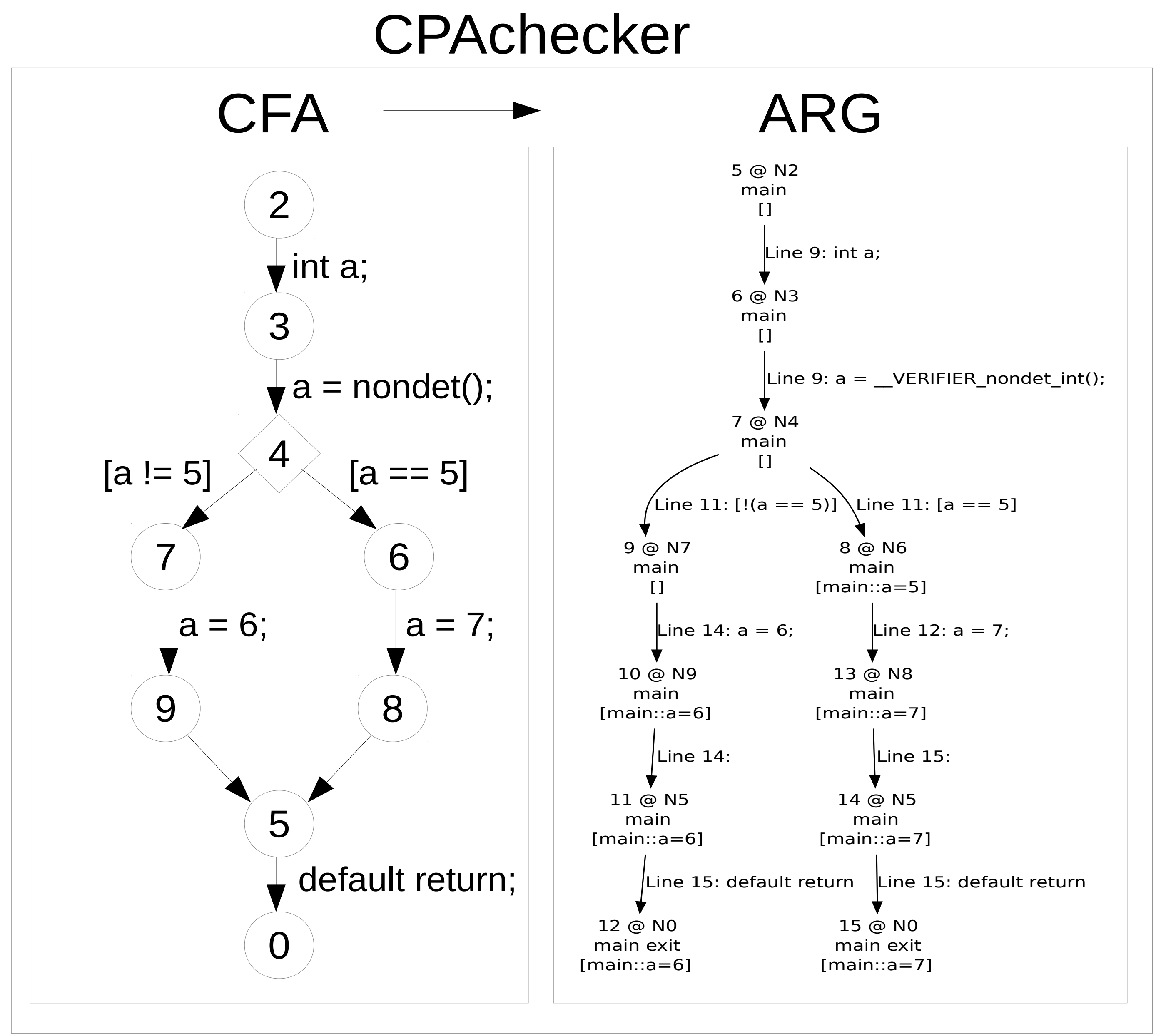
Outline

1. Motivation
2. **CPAchecker and Symbolic Memory Graphs**
3. Abstractions of Symbolic Memory Graphs
4. Using counterexample guided abstraction refinement with Symbolic Memory Graphs
5. Challenges and conclusion

CPAchecker

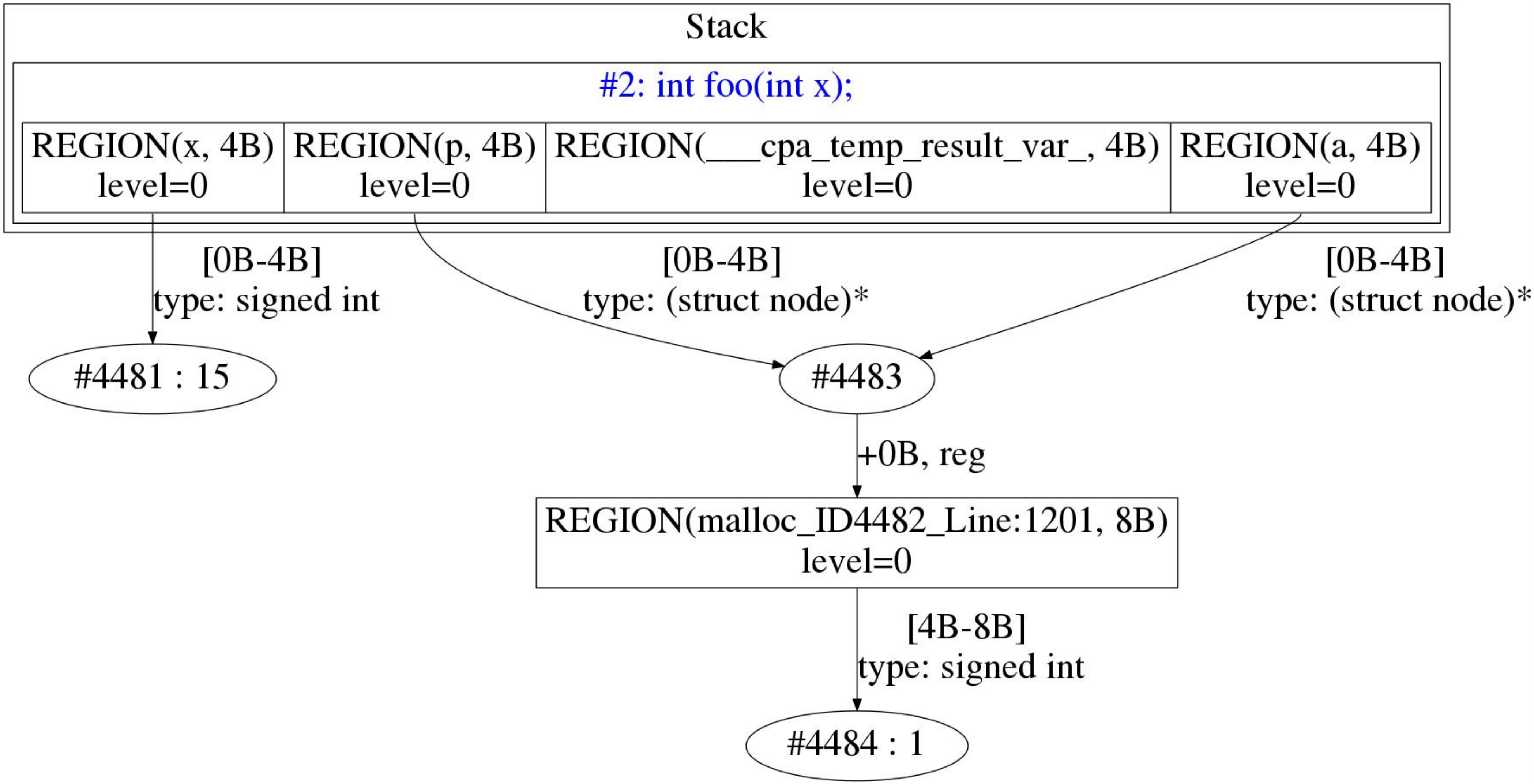
program +
specification

```
1  int main() {
2
3    int a = nondet_int();
5
6    if(a == 5) {
7      a = 7;
8    } else {
9      a = 6;
10   }
11 }
```



Symbolic Memory Graph (SMG)

- Represents sets of heap graphs of a program at a program location
- Supports read and write operations, join of smgs, checking values for equality and inequality, and list abstraction
- Detects memory leaks and invalid read, write or free operations



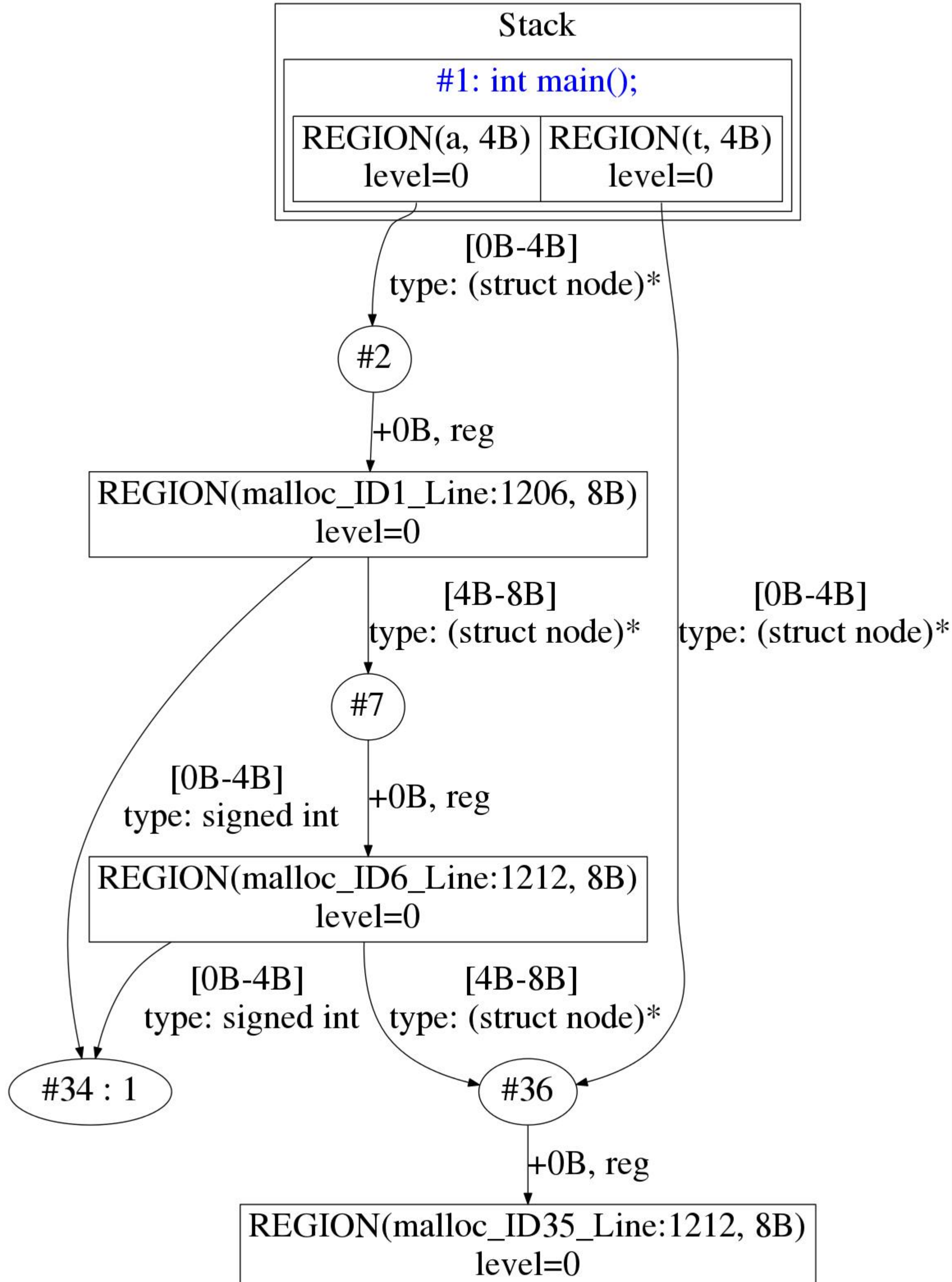
Location: line 22: N24 -{p->i = 1;}-> N25 on N25

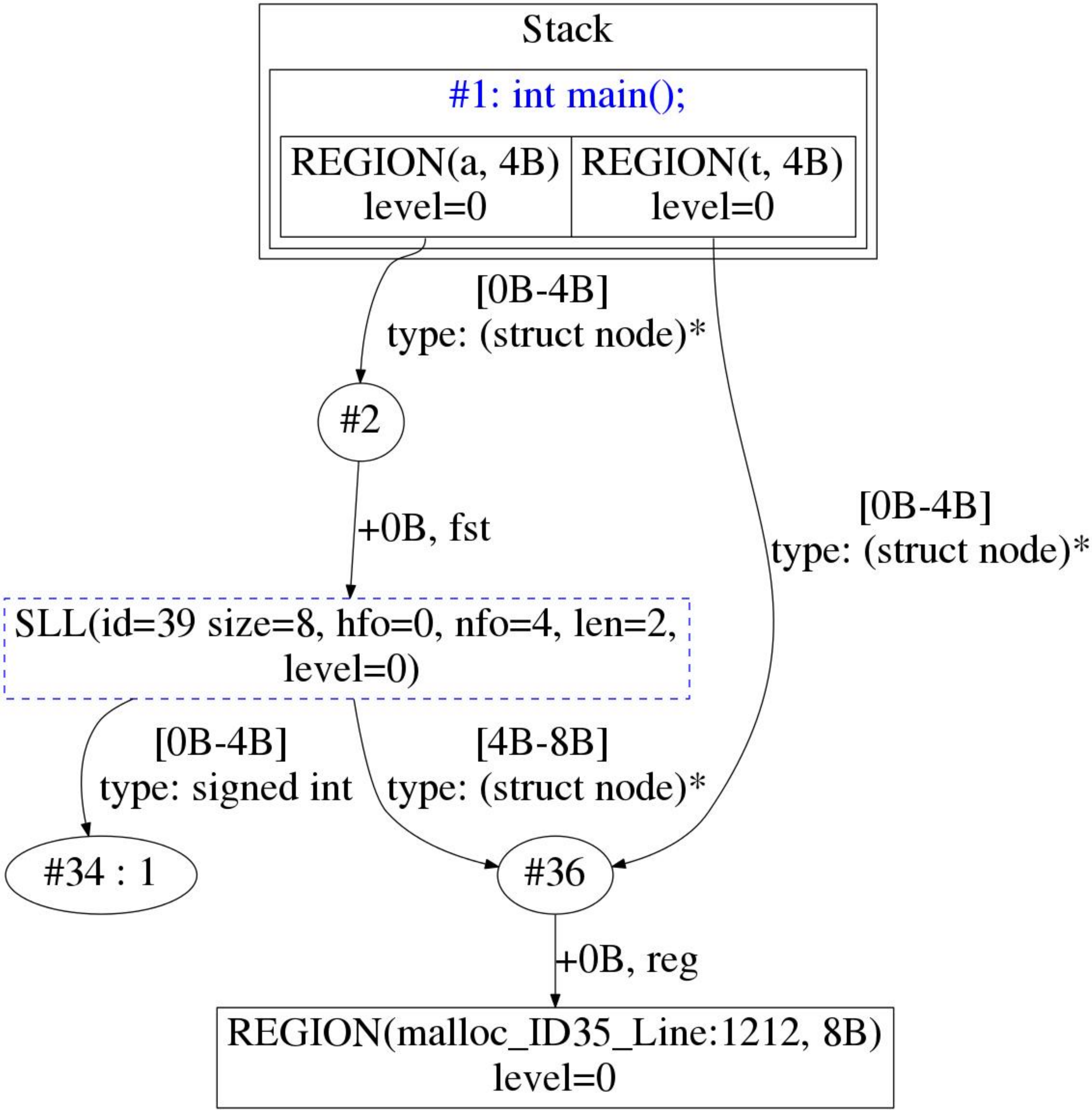
Outline

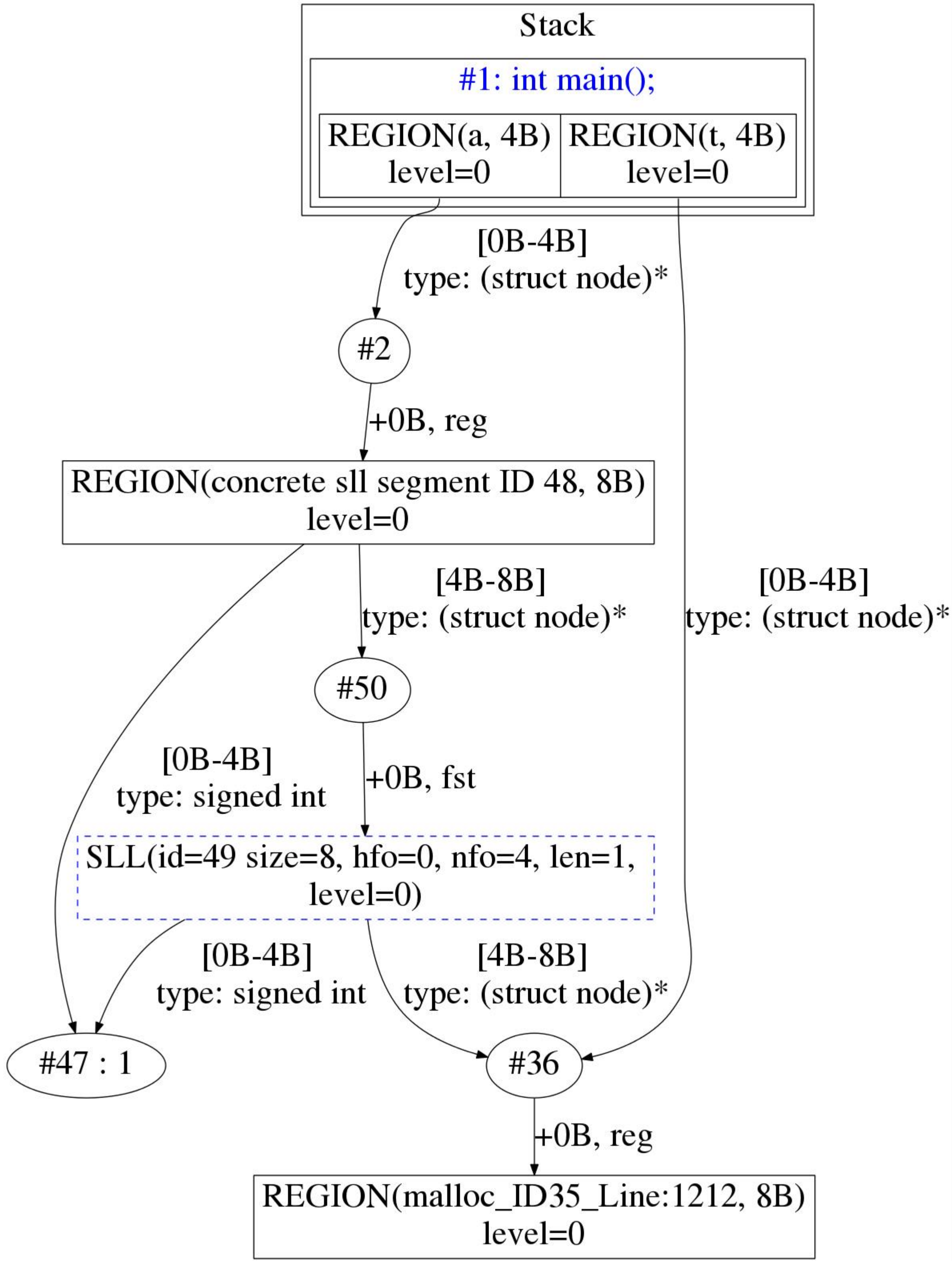
1. Motivation
2. CPAchecker and Symbolic Memory Graphs
3. **Abstractions of Symbolic Memory Graphs**
4. Using counterexample guided abstraction refinement with Symbolic Memory Graphs
5. Challenges and conclusion

List Abstraction

- Used to handle infinitely recursive list segments
- Heap objects are abstracted to list segments and the sub-graphs of the heap objects are joined together
- Whether to execute a possible list abstraction depends on the number of heap objects that can be abstracted into a list, and the loss of information when joining their sub graphs

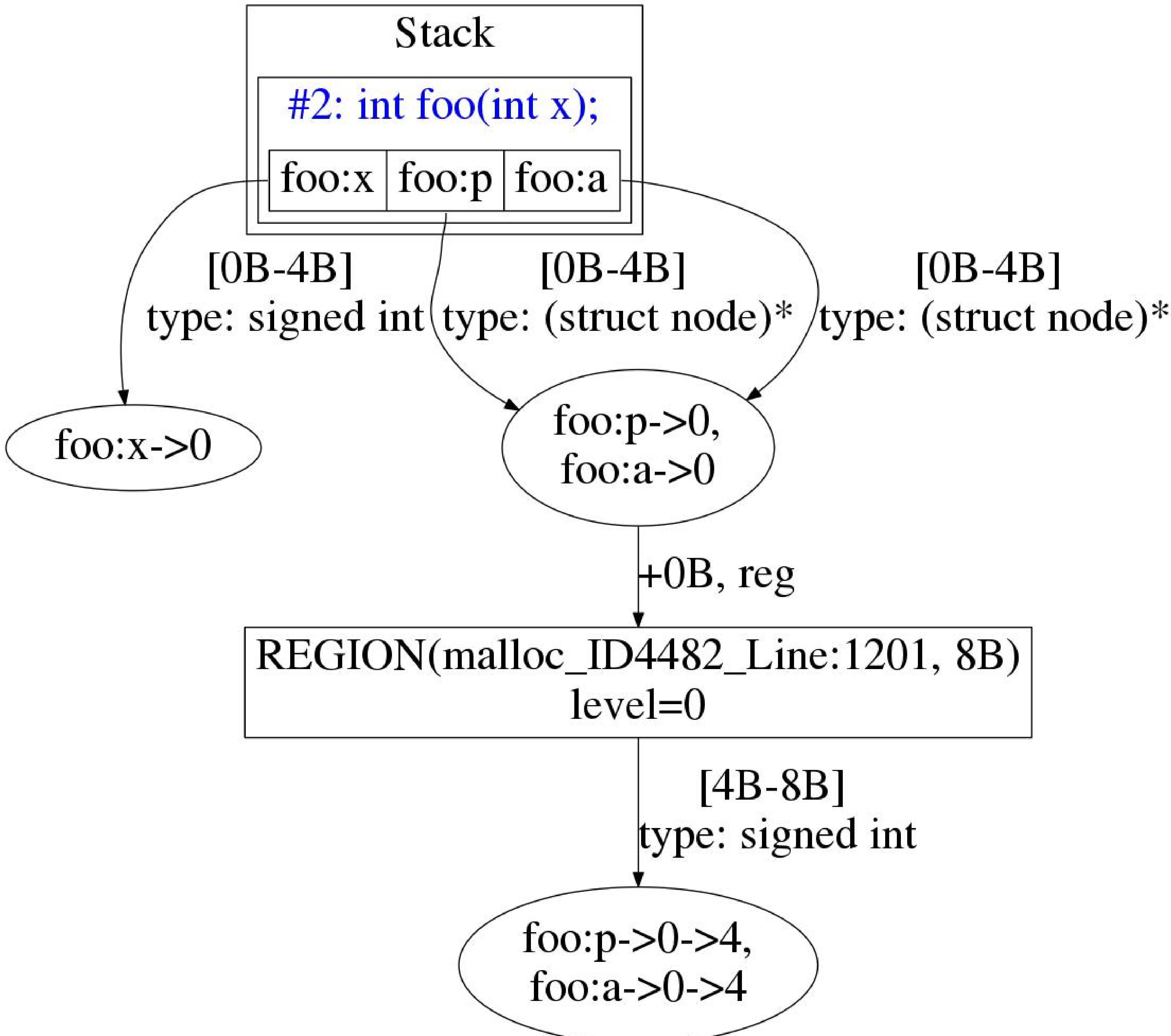

Stack
#1: int main();
REGION(a, 4B)
level=0
REGION(t, 4B)
level=0
[0B-4B]
type: (struct node)*
#2
+0B, reg
REGION(malloc_ID1_Line:1206, 8B)
level=0
[4B-8B]
type: (struct node)*
[0B-4B]
type: (struct node)*
#7
[0B-4B]
type: signed int
+0B, reg
REGION(malloc_ID6_Line:1212, 8B)
level=0
[0B-4B]
type: signed int
[4B-8B]
type: (struct node)*
#34 : 1
#36
+0B, reg
REGION(malloc_ID35_Line:1212, 8B)
level=0


Stack
#1: int main();
REGION(a, 4B)
level=0
REGION(t, 4B)
level=0
[0B-4B]
type: (struct node)*
#2
+0B, fst
[0B-4B]
type: (struct node)*
SLL(id=39 size=8, hfo=0, nfo=4, len=2,
level=0)
[0B-4B]
type: signed int
[4B-8B]
type: (struct node)*
#34 : 1
#36
+0B, reg
REGION(malloc_ID35_Line:1212, 8B)
level=0


Stack
#1: int main();
REGION(a, 4B)
level=0
REGION(t, 4B)
level=0
[0B-4B]
type: (struct node)*
#2
+0B, reg
REGION(concrete sll segment ID 48, 8B)
level=0
[4B-8B]
type: (struct node)*
[0B-4B]
type: (struct node)*
#50
[0B-4B]
type: signed int
+0B, fst
SLL(id=49 size=8, hfo=0, nfo=4, len=1,
level=0)
[0B-4B]
type: signed int
[4B-8B]
type: (struct node)*
#47 : 1
#36
+0B, reg
REGION(malloc_ID35_Line:1212, 8B)
level=0

SMG Precision

- Determines the level of abstraction of a program verification with symbolic memory graphs
- Consist of sets of memory locations, memory paths and locks for list abstractions for every program location
- Adjusts symbolic memory graphs of abstract states in the ARG after each calculation of new abstract states for the ARG


Stack
#2: int foo(int x);
foo:x
foo:p
foo:a
[0B-4B]
type: signed int
[0B-4B]
type: (struct node)*
[0B-4B]
type: (struct node)*
foo:x->0
foo:p->0,
foo:a->0
+0B, reg
REGION(malloc_ID4482_Line:1201, 8B)
level=0
[4B-8B]
type: signed int
foo:p->0->4,
foo:a->0->4

Outline

1. Motivation
2. CPAchecker and Symbolic Memory Graphs
3. Abstractions of Symbolic Memory Graphs
4. **Using counterexample guided abstraction refinement with Symbolic Memory Graphs**
5. Challenges and conclusion

Counterexample guided abstraction refinement

- Method to obtain a good level of abstraction for an analysis for a program
- 1 Step Abstraction : Construct an abstract model of the program
- 2 Step Verification: Check if the model violates a chosen safety property
- 3 Step Refinement: Refine the level of abstraction based on a found spurious counterexample

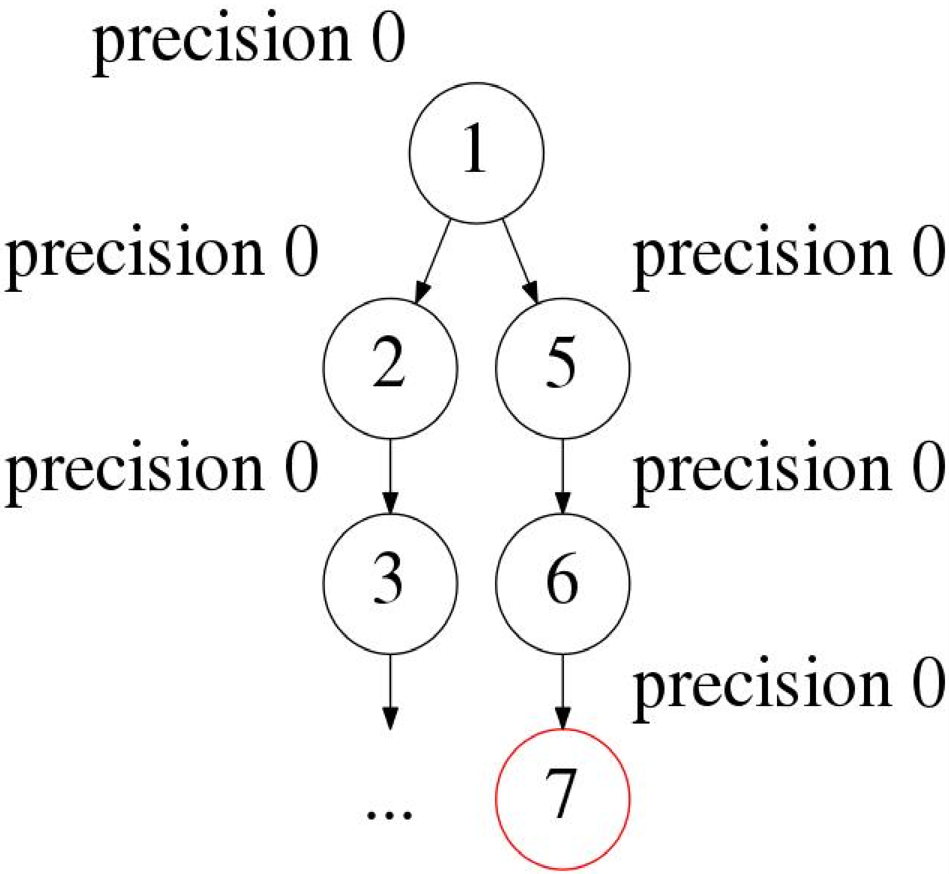
CEGAR with Symbolic Memory Graphs

- Use SMG precision to determine the level of abstraction for Step 1
- Use the full SMG precision on a path to check if a found counterexample is feasible for Step 2
- Use the flow dependence of the SMGs of the spurious counterexample to calculate the new SMG precision for step 3

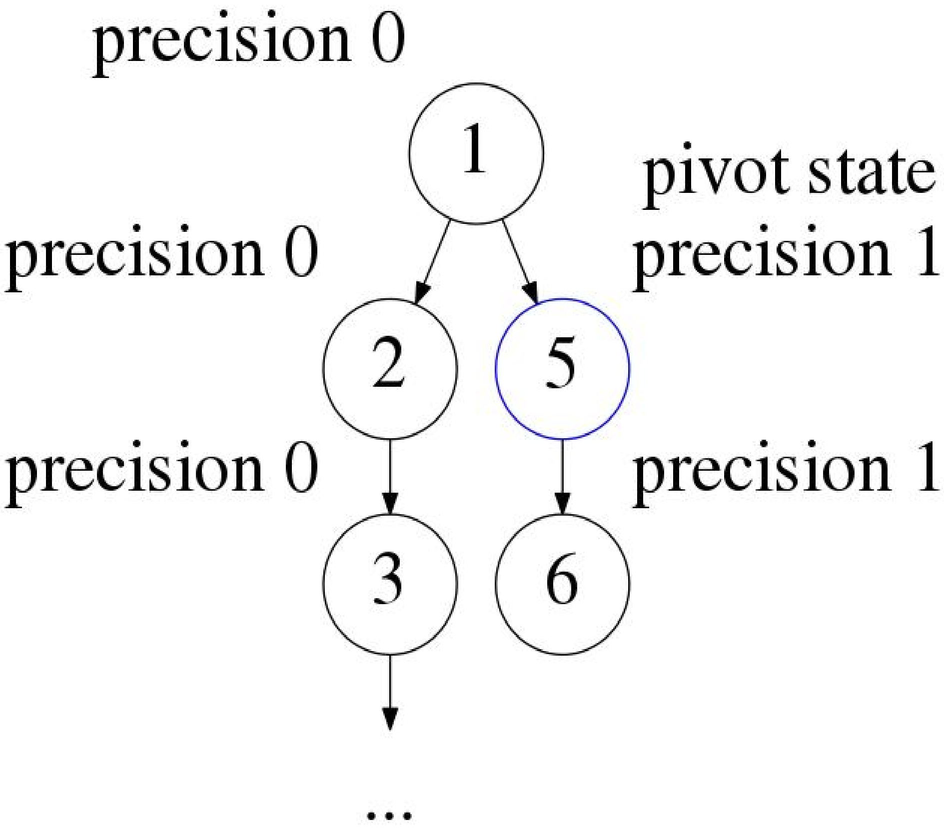
Lazy Abstraction

- Used to improve performance of Counterexample guided abstraction refinement
- Instead of continuously recalculating the abstract model after each refinement step, calculate the model and the refinement of the model on the fly

Lazy Abstraction

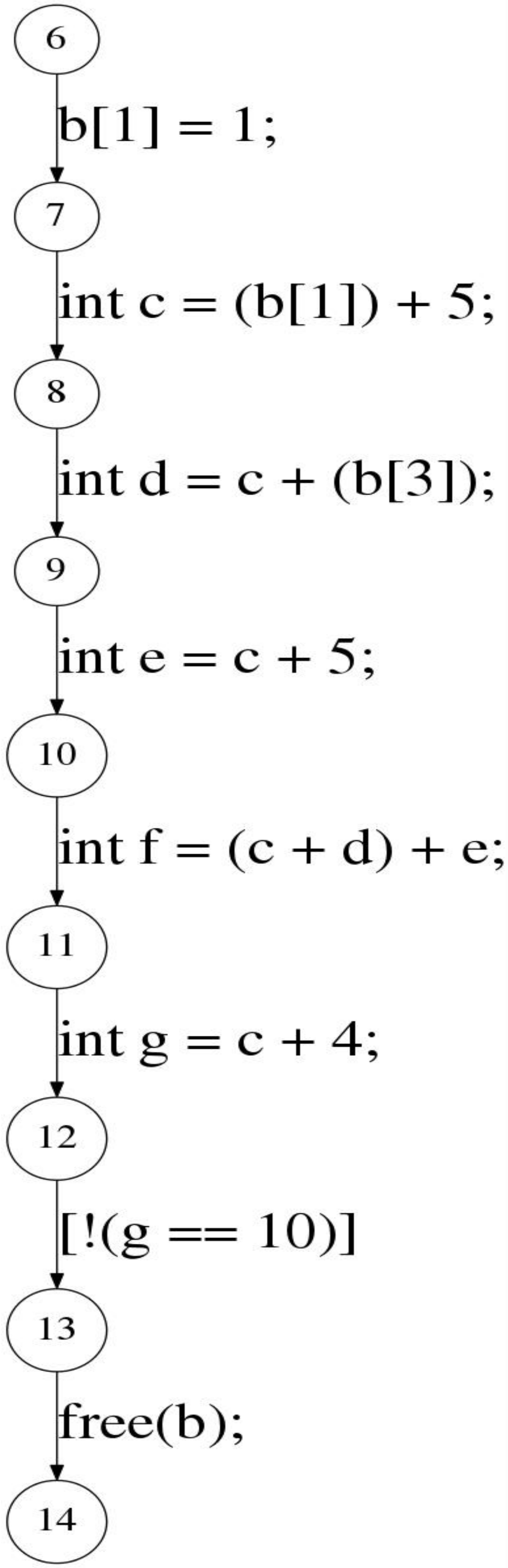


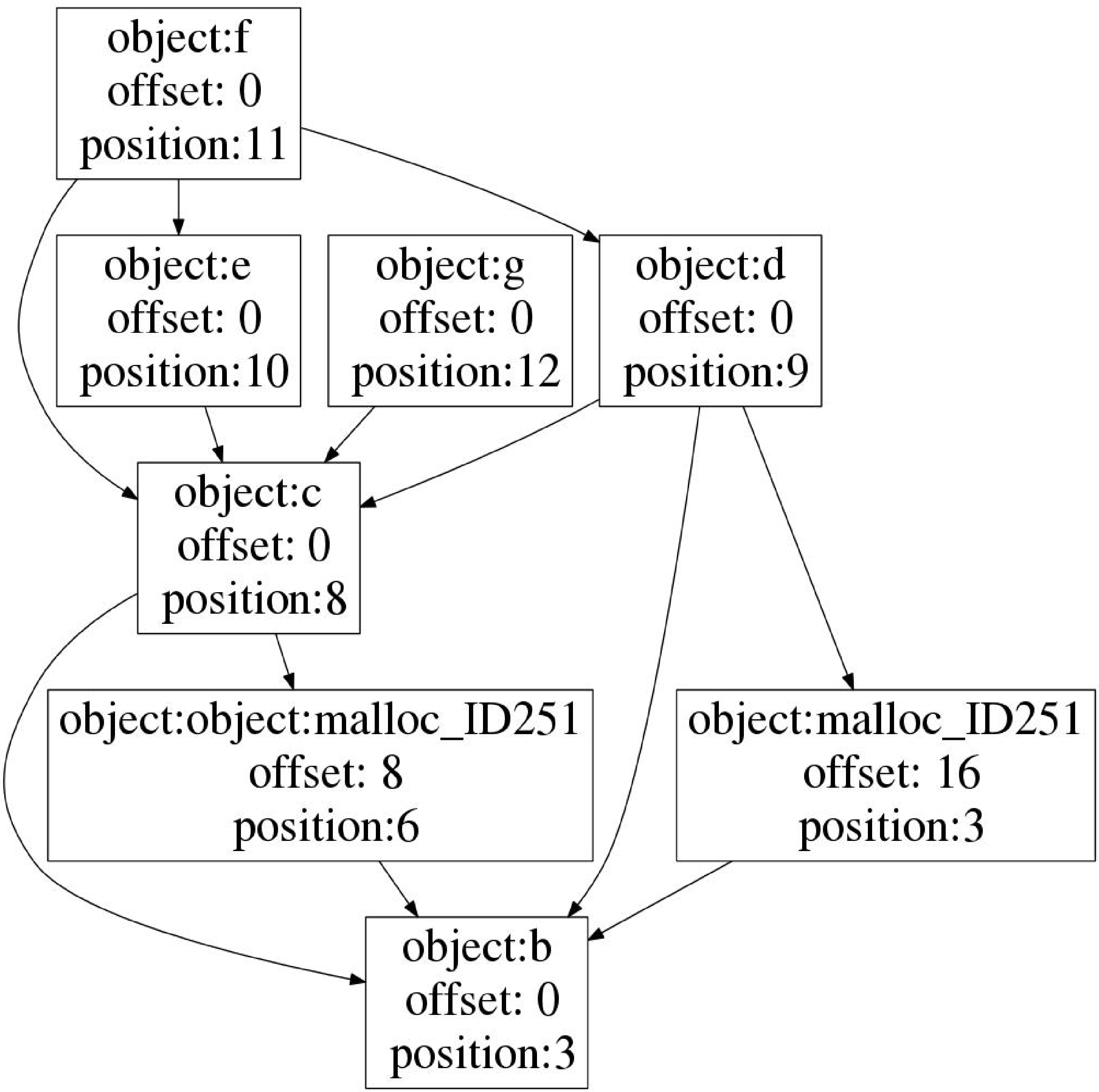
ARG

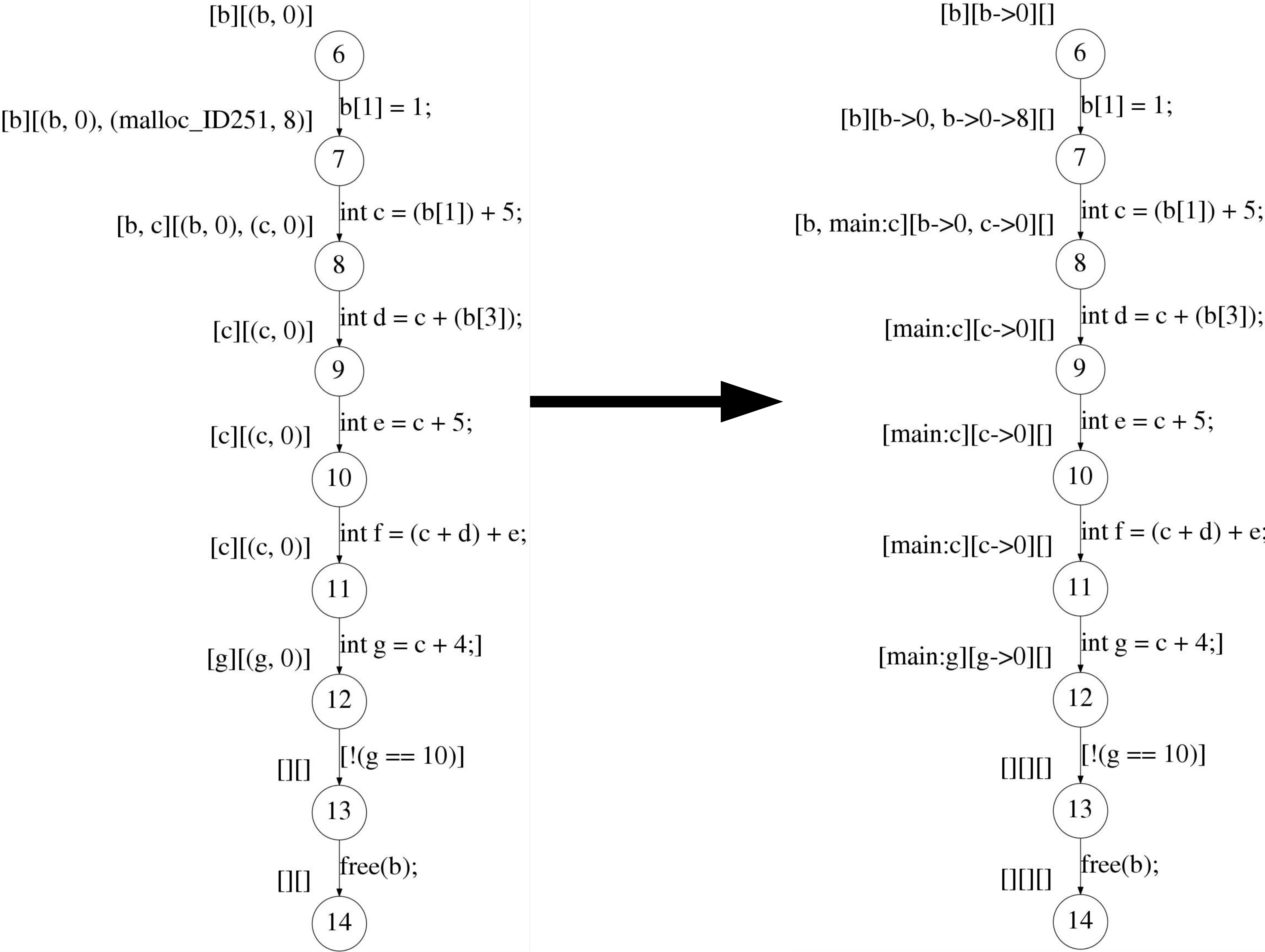


ARG

Example


6
b[1] = 1;
7
int c = (b[1]) + 5;
8
int d = c + (b[3]);
9
int e = c + 5;
10
int f = (c + d) + e;
11
int g = c + 4;
12
[!(g == 10)]
13
free(b);
14


object:f
offset: 0
position:11
object:e
offset: 0
position:10
object:g
offset: 0
position:12
object:d
offset: 0
position:9
object:c
offset: 0
position:8
object:object:malloc_ID251
offset: 8
position:6
object:malloc_ID251
offset: 16
position:3
object:b
offset: 0
position:3



[b][(b, 0)]
6
b[1] = 1;
[b][(b, 0), (malloc_ID251, 8)]
7
int c = (b[1]) + 5;
[b, c][(b, 0), (c, 0)]
8
int d = c + (b[3]);
[c][(c, 0)]
9
int e = c + 5;
[c][(c, 0)]
10
int f = (c + d) + e;
[c][(c, 0)]
11
int g = c + 4;]
[g][(g, 0)]
12
[!(g == 10)]
[][]
13
free(b);
[][]
14
[b][b->0][]
6
b[1] = 1;
[b][b->0, b->0->8][]
7
int c = (b[1]) + 5;
[b, main:c][b->0, c->0][]
8
int d = c + (b[3]);
[main:c][c->0][]
9
int e = c + 5;
[main:c][c->0][]
10
int f = (c + d) + e;
[main:c][c->0][]
11
int g = c + 4;]
[main:g][g->0][]
12
[!(g == 10)]
[][][]
13
free(b);
[][][]
14

Outline

1. Motivation
2. CPAchecker and Symbolic Memory Graphs
3. Abstractions of Symbolic Memory Graphs
4. Using counterexample guided abstraction refinement with Symbolic Memory Graphs
5. **Challenges and conclusion**

Challenges And Conclusion

- Finding a better refinement method for list abstractions
- A method to reduce the loss of information when writing to program location that is not known at the current level of abstraction
- Heap abstraction for trees and other data structures