Program Analysis and Verification
Course 0368-4479
Final Project

Noam Rinetzky

Due 1/September/2022

Design, prove correct, implement, and document the following static analyses. The analyses should be conservative and terminating. You should also write five interesting test programs for each analysis. The test programs should demonstrate the strengths and weaknesses of each analysis. After you submit the project, I will schedule a meeting with each group in which you should present the different analyses you implemented, intuitively argue about their correctness, and show how they run on your test programs.

1 Analyzing Integer Programs

1.1 Programming language

Consider a language of programs which manipulate natural numbers (including zero) with the following primitive commands:

C	::=	`skip` | i := j | i := K | i := ? | i := j + 1 | i := j -1
	|	`assume` E | `assert` ORC
E	::=	i = j | i != j | i = K | i != K | `TRUE` | `FALSE`
ORC	::=	(ANDC) | (ANDC) ORC
ANDC	::=	b | b ANDC

In the above syntax, K is s constant and ? is an unknown arbitrary value. An `assert` command aborts the program if the assertion does not hold. It takes as an argument a sequence of parenthesized expressions and should be interpreted as a disjunction: The assertion holds if at least one of the parenthesized expressions hold. A parenthesized expression should be interpreted as a conjunction of atomic predicates b whose syntax is defined soon: For the parenthesized expression to hold all of its atomic predicates should be true.

Assume that the program is written in a CFG syntax,

$$P = \bar{i}\ \overline{\ell\ C\ \ell},$$

where $\bar{i}$ is the sequence of variable names (strings comprised of lowercase letters) and $\ell\ C\ \ell$ is an edge in the control flow graph. For example, the following program increments i and j until their value is equal to that of n:

```
n i j

L0  n := ?          L1
L1  i := 0          L2
L2  j := 0          L3
L3  assume(i = n)   L6
L3  assume(i != n)  L4
L4  i := j + 1      L5
L5  j:= i           L3
L6  skip            L7
```

Simplifying assumptions You may assume the following:

1. The only one node with no incoming edges is the one from which the execution starts.
2. Every node in the control flow graph has at most two out-going edges.
3. If a node has more than one outgoing edges than these edges are annotated with `assume` commands.
4. Programs are syntactically legal, i.e., there is no need to handle syntax errors.

1.2 Parity Analysis

Design and implement an abstract domain and transformers which can prove that a program does not violate assertions pertaining to the parity of each variables. Specifically, consider atomic predicates of the following form:

$$\text{b} \quad ::= \quad \texttt{EVEN}\ \text{i} \mid \texttt{ODD}\ \text{i}$$

For example, assume we replace the `skip` command in the example program with the following command:

$$\texttt{assert}\ (\texttt{ODD}\ i\ \ \texttt{ODD}\ j)\ (\texttt{EVEN}\ i\ \ \texttt{EVEN}\ j)$$

The analysis should be able to prove that the program does not violate the assertion, i.e., that i has the same parity as j when the program reaches L6.

1.3 Summation Analysis

Design and implement an abstract domain and transformers which can prove that a program does not violate assertions pertaining to the equality of the summation of two sets of variables. Specifically, consider atomic predicates of the following form:

$$\mathrm{b} \quad ::= \quad \texttt{SUM}\ \bar{i} = \texttt{SUM}\ \bar{j}$$

For example, if we replace the `skip` command in the example program with the following command:

$$\texttt{assert}\ (\texttt{SUM}\ i\ j\ \ =\ \ \texttt{SUM}\ j\ n)$$

then the analysis should be able to prove that the program does not violate this assertion.

1.4 Combined Analysis

Combine the Parity and Summation analyses in two different ways and find examples that show the cost/precision differences between the analyses.

2 Shape Analysis of Acyclic Linked Lists

Consider programs comprised of the following primitive commands (the rest of the programming language is as described in Section 1.1, except that the only constant K is NULL and that the primitive predicates b are as described below.)

$$\begin{array}{lll} \mathrm{C} & ::= & \texttt{skip} \mid \mathrm{x := y} \mid \mathrm{x := NULL} \mid \mathrm{x := y.n} \\ & \mid & \mathrm{x.n := y} \mid \mathrm{x := new} \mid \texttt{assume}(\mathrm{E}) \mid \texttt{assert}(\mathrm{ORC}) \end{array}$$

The program manipulates acyclic (possibly shared) singly-linked lists. Design an abstract domain and transformers which can prove that the program does not violate assertions comprised of the following atomic predicates:

b ::= x = NULL | x != NULL | x = y | x != y | x = y.n | x != y.n
| `LS` x y | `NOLS` x y | `ODD` x y | `EVEN` x y

`LS` x y means that there is a list going out from the node pointed to by x which reaches the node pointed to by y (specifically, if `LS` x y holds then x and y must have a non-NULL value). `NOLS` x y means that either x or y has a NULL value or that there is not a list going out from the node pointed to by x which reaches the node pointed to by y . `ODD` x y resp. `EVEN` x y means that `LS` x y holds and that the number of nodes in the list segment between x and y, including the nodes pointed to by x and y, is odd resp. even.

For example, the following program creates three lists segments of the same length and concatenate one to the tail of the other two. Your analysis should be able to prove the assertions in labels 44, 45, and 46.

```
x y z xx yy zz t p

L8 t := new L9
L9 t.n := NULL L10
L10 t.n := x L11
L11 x := t L12

L12 t := new  L13
L13 t.n := NULL L14
L14 t.n := y L15
L15 y := t L6

L6 assume(TRUE) L8
L6 assume(TRUE) L18

L18 z := new L20

L20 xx := x            L21
L21 t := xx.n          L22
L22 assume(t = NULL)   L30
```

```
L22 assume(t != NULL)  L23
L23 xx := t            L21

L30 yy := y            L31
L31 t := yy.n          L32
L34 assume(t = NULL)   L40
L34 assume(t != NULL)  L35
L35 yy := t            L31


L40 xx.n := NULL       L41
L41 xx.n := z          L42

L42 yy.n := NULL       L43
L43 yy.n := z          L50

L50 assert (t = NULL)  L51
L51 assert (z != NULL) L52
L52 assert (LS x xx)   L53
L53 assert (LS y z)    L54
L54 assert (NOLS x y)  L55
L55 assert (t = z.n)   L56
L56 assert (ODD x xx EVEN x z) (EVEN x xx ODD x z) L57
```

Simplifying assumptions You may assume the following:

1. Every node contains a single field `n`.
2. Pointer variables are initialized to NULL. Similarly, the `n`-field of newly allocated nodes is initialized to NULL.
3. Every command which sets the `n`-field of a node to the value of a variable is preceded by a command which sets it to NULL.
4. At most one node in the heap can be heap-shared, i.e., may be pointed to by more than one `n`-field.

Your analysis should detect errors that stems from dereferencing NULL-valued pointers, executing a command which creates a cycle in the heap or which makes more than one node heap-shared. Bonus will be given if your analysis can handle an unbounded number of shared nodes (i.e., you do not use simplifying assumption number 4).