Term Project Report

Termination Analysis and Ranking Function

Freya Mehta

4th May, 2020

INDEX

Abstract

In this report, we present a tool which classifies whether a program is terminating or non-terminating. Additionally, we generate a ranking function and set of linear inductive supporting invariants as termination proofs for boogie and C programs following the method presented in [6] for proving termination of C loops. The scope of this tool is narrowed to certain type of programs with constraints on kinds of its loops. The tool accepts C and boogie programs and produces results for them.

What is Boogie?

Boogie is an intermediate verification language (IVL), intended as a layer on which to build program verifiers for other languages. Several program verifiers have been built in this way, including the VCC and HAVOC verifiers for C.

Boogie is also the name of a tool. The tool accepts the Boogie language as input, optionally infers some invariants in the given Boogie program, and then generates verification conditions that are passed to an SMT solver. The default SMT solver is Z3. A front-end translates a program written in some language to the Boogie IVL and a back-end tries to verify the translated program. This provides “separation of concerns” as
• The front-end does need to care about what method is used to verify the program.
• The back-end does not need to care about the semantics of several different programming languages. It only needs to care about the Boogie IVL.
This idea is analogous to the “intermediate representation” (IR) used in modern compilers.

Boogie Tools: Boogaloo

Boogaloo is an interpreter for Boogie available via web interface3.

- Displays possible executions of a Boogie program
- Use option -o to control number of executions, e.g. -o 5 for 5 executions.
- To get more diverse executions, use -n, e.g. -n 3 for at most 3 executions with the same sequence of statements.
- Other interesting options: -c=0 turns off "concrete mode", -p specifies entry procedure.
- Output with assume f: print "text"g true
- User Manual available

Boogaloo example:

Running the following program through Boogaloo with option -o 3 produces the output below, listing arguments, output, and return value.

```
procedure Square(a : int) returns (square: int) {
    square := a * a;
    if (square == 0) {
        assume {: print "a is zero" } true;
    } else {
        assume {: print "a = ", a } true;
    }
}
```

```
Execution 0: Square(0) passed
a is zero
Outs: square -> 0

Execution 1: Square(-1) passed
a = -1
Outs: square -> 1

Execution 2: Square(1) passed
a = 1
Outs: square -> 1
```

Running the following program through Boogaloo with options -o 4 -n 1 -c=0 produces the output below.

```
procedure ZeroInit(a : [int]int, lo : int, hi : int) returns (b :
      [int]int)
{
    var i : int;
    b := a;
    i := lo;
    while (i <= hi) {
        b[i] := 0;
        i := i+1;
    }
}
```

```
Execution 0: ZeroInit([], 0, -1) passed
Outs: b -> []

Execution 1: ZeroInit([0 -> 0], 0, 0) passed
Outs: b -> [0 -> 0]

Execution 2: ZeroInit([0 -> 0, 1 -> 0], 0, 1) passed
Outs: b -> [0 -> 0, 1 -> 0]

Execution 3: ZeroInit([0 -> 0, 1 -> 0, 2 -> 0], 0, 2) passed
Outs: b -> [0 -> 0, 1 -> 0, 2 -> 0]
```

- If you only pass -o, Boogaloo will only produce executions with lo > hi. This is because it first chooses a sequence of statements (go through the loop once), and then searches variable values to fit that sequence. Because there are infinitely many (unlike in the first example), it will never consider another sequence.
- Additionally passing -n fixes this problem: It allows only the given number of executions per sequence of statements. However, only 2 instead of 4 executions will be found. This is because the number of possible values for the input parameters is restricted (Boogaloo calls this the concrete mode).
- Additionally passing -c=0 turns off this concrete mode, finally showing the diverse executions above.

Different combinations of these options can often help get the desired test cases for a program. However, always using all of them is not necessarily the solution in every case.

In this project we focus on the Boogie Language. The language is called the “intermediate language” because it is designed to be the middle part of a program verifier. Boogie is a language that is like programming languages suitable for writing down algorithms in a machine readable from. However, Boogie is typically not used by programmers to write code. It is typically used to model computer programs written in other programming languages.

Boogie doesn’t directly support termination measures hence proving termination of any kind of boogie programs isn’t explored yet.

Important Definitions

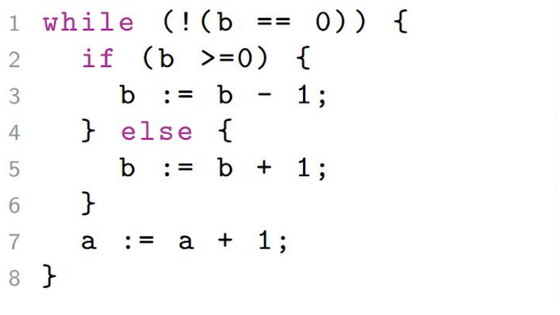
A **Boogie program** is a triple P = (V, μ, T) where,

- V is a set of (program) variables,
- μ is a map that assigns each variable either Z or {true, false}
- T is a derivation tree for the start symbol S_{Boo} in the Boogie grammar

such that the translation of each expression/type to an SMT term/sort is well-sorted wrt. the map μ. Given a variable v ∈ V we call μ(v) the domain of v.

Example:

$P_{ab} = (V_{ab}, μ_{ab}, T_{ab})$ where

- V_{ab} = {a, b},
- μ(a) = Z, μ(b) = Z, and
- T_{ab} is the derivation tree for the text (right) 

```
while (!(b == 0)) {
  if (b >=0) {
    b := b - 1;
  } else {
    b := b + 1;
  }
  a := a + 1;
}
```

.

Program State

Given a program P = (V, μ, T), a program state is a map that assigns each variable v ∈ V a value of the variable’s domain. We use $S_{V,μ}$ to denote the set of all program states.

Example:

The map that assigns the variable a to 23 and the variable a to 42 is an element of $S_{Vab,μab.}$

Notation:
There are several notations for maps. We can e.g. write the state above

- as a set of pairs {(a, 23), (b, 42)}.
- Alternatively, we can write the pairs using an arrow symbol: {a --> 23, b --> 42}.
- Furthermore, we can give that state a name, e.g., s and define the state via the equalities s(a) = 23 and s(b) = 42.

Convention:
We will use FOL formulas to denote sets of program states.

- The set of variables in the formulas will be the program variables.
- The constant symbols, function symbols, and predicate symbols are given by the SMT theories.
- The model M is defined by the SMT theories.
- A formula φ denotes that the set of all program states s such that for s = ρ the evaluation $[[\varphi]]_{M,\rho}$ is true.
- Example:
    - The formula a = 23 ^ b = 42 denotes the singleton set {{a --> 23,b --> 42}} ⊆ $S_{Vab,\mu ab}$
- We will define a program semantics such that the set of states in which P_{ab} can be after executing the while loop "is" b = 0.

Termination

There are four different properties of programs. One property was terminated and the other properties were related to termination. We provide formal definitions here. In each case, we consider a program P with a CFG (Loc, ∆, l_{init}, l_{ex}).

1. We say that P can reach the exit location if there exists a finite execution, such that the first configuration (l, s) is initial, and the last configuration is (l_{ex} ,s') for some states s'.
2. We say that P can stop if there exists a reachable configuration (l, s) such that there exists no configuration (l', s') and statement st with (l, st, l') ∈ ∆ and (s, s') ∈ [[st]].
3. We say that P always reaches the exit location if there exist no infinite executions, and all finite executions end in a configuration (l', s') where we either have a successor (i.e., there exists a configuration (l'', s'') and statement st with (l', st, l'') ∈ ∆ and (s', s'') ∈ [[st]]) or we have that l' is l_{ex}.
4. We say that P always stops resp. P terminates if there exist no infinite executions.

Let P = (V, µ, st) be a program and G = (Loc, ∆, l_{init}, l_{ex}) be a control-flow graph for P.

Infinite Execution:
We call a sequence of program configurations (l_0, s_0), an infinite execution of P if there exists an infinite sequence of statements st_1 . . . such that for each i ∈ N

- (l_i, st_{i+1}, l_{i+1}) ∈ Δ and
- (s_i, s_{i+1}) ∈ [[st_{i+1}]]

We call P terminating if P does not have an infinite execution that starts in an initial configuration.

Our main means for proving termination will be ranking functions. Informally, a ranking function for a loop is a function whose value is bounded from below but decreasing in every iteration. (Hence, we can conclude by reductio ad absurdum that only a finite number of loop iterations is possible). On Wikipedia ranking functions are called Loop variants. In the research community on termination analysis, the term ranking function is however used more often.

Ranking Function

Ranking Function Definition:
Given a program P = (V, μ, st), a while loop while(expr){st} and a set W together with a well-founded relation R ⊆ W × W , we call a function f : $S_{V,\mu}$ --> W a ranking function if for each pair of states (s, s') ∈ [[assume expr; st]] the relation (f (s), f (s')) ∈ R holds.
(Well-founded - Let X be a set. We call a binary relation R ⊆ X × X well-founded if there is no infinite sequence x1, x2,. . . such that (x_i, x_{i+1}) ∈ R for all i ∈ N).
Example-

If we choose (W, R) as (N, >) then f(s) = 100 − s(x) − s(y) is a ranking function for this program.
Notation-
In order to improve legibility, people usually write f(x, y) = 100 − x − y instead of f(s) = 100 − s(x) − s(y).

Is every loop that has a ranking function terminating?
No. There might be a non terminating loop inside the loop that has a ranking function. For eg.

```
while (x < 100) {
  x := x + 1;
  while (y < 100) {
    y := y -1;
  }
}
```

Theorem: Let P be a program. If every while loop of P has a ranking function then P is terminating.

Proof: Assume there is an infinite execution (l_0,s_0), (l_1,s_1), . . . that starts in an initial configuration. Let l'_0, l'_1, . . . be the subsequence of all locations that are loop heads. Because of the structure of control flow graphs the sequence l'_0, l'_1, . . . is an infinite subsequence. Because there are only finitely many different locations in a control-flow graph, at least one loop head occurs infinitely often. Let l^ be a loop head that occurs infinitely often in the sequence. Between each two visits of l^ the ranking function of the corresponding loop is decreasing which is a contradiction to well-foundedness.

Are ranking functions into (N; >) always convenient?

Problem: Negative return value of function after the last loop iteration.

```
while (x>=0) {
  x := x - 1;
}
```

For (N, >) the function f(x) = x is not a ranking function because after the last loop iteration the function returns −1. f(x) = x + 1 is a ranking function.

```
while (x>=0) {
  assume y >= 1;
  x := x - y;
}
```

For (N, >) the function the function f(x, y) = x is not a ranking function. f (x, y) = x + y is a ranking function.

```
while (x>=0) {
  havoc y;
  assume y >= 1;
  x := x - y;
}
```

For (N, >) there is no ranking function.

Solution: Do not use (N, >) but (Z, >N) whose relation we define as follows.

x >N y iff x > y and x ∈ N

This relation also solves another problem: If we require that the function f is defined for all states $s \in S_{V;\mu}$ (i.e., f is a total function) then the functions that we saw so far were in fact not well-defined.

How can we check if a function f is a ranking function?

We present a solution for the schematic example below where we assume

1. we have one loop in the loop body,
2. the program's variables are x1, . . . ,xn,
3. that we can express the function as an expression f_{expr}(x1,...,xn) over the program's variables, and
4. the range of f is Z and we consider the well-founded ordering >N.

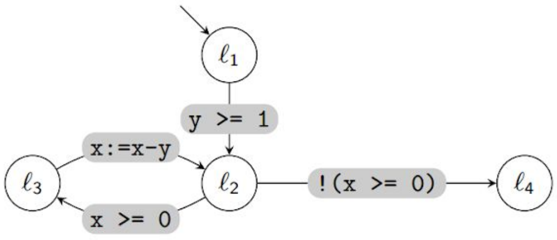
```
while (expr1) {
  // outer loop body part 1
  while (expr2) {
    // inner loop body
  }
  //outer loop body part 2
}
```

We introduce a new variable oldf whose values are integers and transform the program above to the program below. The function f is a ranking function for the outer while loop iff the program below is safe.

```
oldf := fexpr(x1,..,xn);
while (expr1) {
  // outer loop body part 1
  while (expr2) {
    // inner loop body
  }
  // outer loop body part 2
  assert fexpr(x1,...,xn)<oldf && oldf>=0;
  oldf := fexpr(x1,...,xn);
}
```

What if a ranking function is only decreasing for reachable states?
Consider the following program which is obviously terminating.

```
assume(y >= 1);
while (x >= 0) {
  x := x - y;
}
```


$$\begin{tikzcd} & \ell_1 \arrow[d, "y >= 1"] & \\ \ell_3 \arrow[r, bend left, "x:=x-y"] & \ell_2 \arrow[l, bend left, "x >= 0"] \arrow[r, "!(x >= 0)"] & \ell_4 \end{tikzcd}$$

- ❏ Is f (x, y) = x a ranking function for this loop?
  No. The value of x is increasing if y is negative.
- ❏ Is there a ranking function that allows us to prove termination of this program?
  Our first definition of a ranking function refers only to a loop. In order to prove termination of the program above, we also have to take the reachable states into account.

How can we compute ranking functions?

- For general programs: very difficult. Although there is some research that follows this direction.
- For infinite traces: doable. For several termination analyses, it is sufficient to compute ranking functions for ultimately periodic traces. An ultimately periodic trace is an infinite trace in which some (finite) sequence of statements is repeated infinitely often. This ultimately periodic trace is then considered as a program that consists of a single while loop. For programs of this form several approaches are available. Example: the trace where x>0, y>0, y:=y-1 is repeated infinitely often is an ultimately periodic trace of the program below. For my project, I have limited the scope to such programs only.

```
while (x > 0) {
  if (y > 0) {
    y := y-1;

  } else {
    x := x-1;
    havoc y;
  }
}
```

Basic Idea of the approach:

- Compute one transition formula τ_{loop} for all statements on the path from the loop entry to the loop entry.
- Do not set up constraints that encode the existence of a general ranking function. Set up constraints that encode the existence of a ranking function that has a certain form.
- If the variables of the program are x and y and r_x, r_y, r_0, Z then we call $r_x \cdot x + r_y \cdot y + r_0$ a linear function. Constraints for the existence of a linear ranking function. There exist r_x, r_y, r_0 such that
  - $\forall$ x, y, x', y', $\tau_{loop}(x, x')$ --> $(r_x \cdot x + r_y \cdot y + r_0) - (r_x \cdot x' + r_y \cdot y' + r_0) \geq 1$
  - $\forall$ x, y, x', y': $\tau_{loop}(x, x')$ --> $r_x \cdot x + r_y \cdot y + r_0 \geq 0$
  - The first line states that the function $f(x, y) = r_x \cdot x + r_y \cdot y + r_0$ is decreasing in every iteration.
  - The second line states that the function $f(x, y) = r_x \cdot x + r_y \cdot y + r_0$ is bounded from below.
- Use Farkas' Lemma to simplify the constraints.
- Apply an SMT solver to the resulting constraints. E.g., in the schematic example above we use the satisfying assignments to r_x, r_y, r_0 to build our linear ranking function.

How can we build an algorithm for checking termination?

Basic idea of the approach of the Terminator tool.

Iteratively collect ranking functions until termination of all loops is shown.

1. Start with the empty set of ranking functions.

2. Pick an ultimately periodic trace for which termination is not yet shown (if termination is not yet proven).
3. Compute a ranking function for this trace and add it to our collection (if the trace does not have in infinite execution).
4. Check if the collection of ranking functions is sufficient to prove termination and continue with the second step.

A strength of Terminator's approach is that it does not need one (possibly complicated) ranking function for each loop but that it can use a combination of several ranking functions to prove termination of a single loop. The theoretical basis for this are disjunctively well-founded transition invariants. The basic idea demonstration using an example:

```
while (x>0 && y>0) {
  if (*) {
    x := x-1;
    havoc y;
  } else {
    y := y-1;
  }
}
```

Let us prove that the program whose code is depicted above is terminating.
The if (*) means that the computer which runs the program can nondeterministically pick one of the two branches. This is the syntax of Boogie.
We will need three iterations and two ranking functions.

- Initially, our set of ranking functions is empty. We construct the following program and pass it to a tool that checks safety (resp. that every assert statement is valid). The safety checker tells us that the assert is reachable via the if-branch and we conclude that there is an ultimately periodic infinite trace that repeats the if-branch. We pass this trace to a tool that infers ranking functions and obtain f1(x, y) = x.

```
while (x>0 && y>0) {
  if (*) {
    x := x-1;
    havoc y;
  } else {
    y := y-1;
  }
  assert false;
}
```

- In the second iteration we construct the following program in order to check whether f1 is sufficient to prove termination. The (re-)initialization of the oldf1 variable is done nondeterministically. The safety checker tells us that the assert can be violated by an execution that takes the else branch. We conclude that there is an ultimately periodic trace that repeats the else-branch whose termination cannot be shown by the ranking function f1. We pass this trace to a tool that infers ranking functions and obtain f2(x, y) = y.

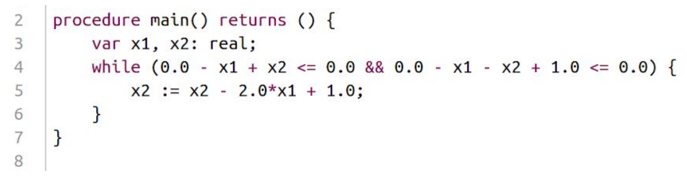
```
if (*) {
  oldf1 := f1(x,y);
}
while (x>0 && y>0) {
  if (*) {
    x := x-1;
    havoc y;
  } else {
    y := y-1;
  }
  assert oldf1 > f1(x,y) &&
    oldf1 >= 0;
  if (*) {
    oldf1 := f1(x,y);
  }
}
```

- In the third iteration we construct the following program in order to check whether the combination of f1 and f2 is sufficient to prove termination. The safety checker tells us that the assert statement is valid and we conclude termination of the original program.

```
if (*) {
  oldf1 := f1(x,y);
  oldf2 := f2(x,y);
}
while (x>0 && y>0) {
  if (*) {
    x := x-1;
    havoc y;
  } else {
    y := y-1;
  }
  assert (oldf1 > f1(x,y) &&
     oldf1 >= 0)
      || (oldf2 > f2(x,y) &&
     oldf2 >= 0);
  if (*) {
    oldf1 := f1(x,y);
    oldf2 := f2(x,y);
  }
}
```

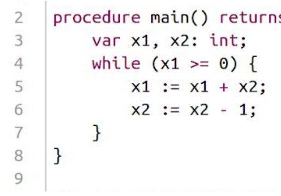
- We note that the expression of the assert statement is a disjunction; we do not require that both ranking functions are decreasing, we only require that at least one of ranking functions is decreasing. For concluding termination the nondeterministic assignments to oldf1 and oldf1 are vital. The safety proof does not only show that in each iteration the function f1 or the function f2 is decreasing, the safety proof shows that between every two (not necessarily consecutive) visits of the loop head the function f1 or the function f2 is decreasing.

Results

1. 

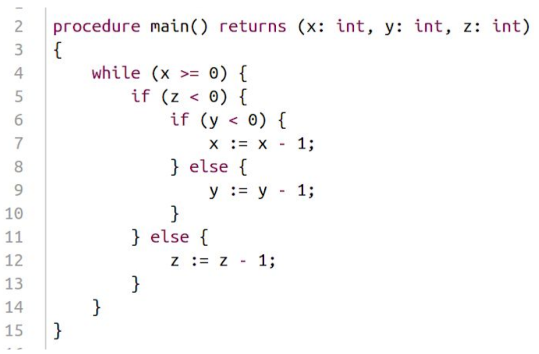
```
procedure main() returns () {
    var x1, x2: real;
    while (0.0 - x1 + x2 <= 0.0 && 0.0 - x1 - x2 + 1.0 <= 0.0) {
        x2 := x2 - 2.0*x1 + 1.0;
    }
}
```

Does not terminate over the reals. Has linear ranking function f(x1,x2)=x1+x2-1 over the integers. Loop is not integral.

2. 

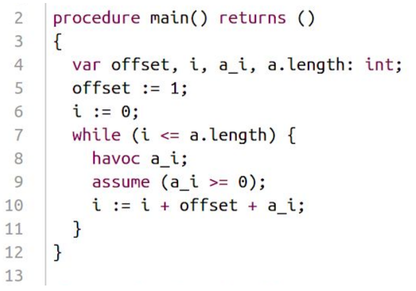
```
procedure main() returns () {
    var x1, x2: int;
    while (x1 >= 0) {
        x1 := x1 + x2;
        x2 := x2 - 1;
    }
}
```

Does not have a linear ranking function.

3. 

```
procedure main() returns (x: int, y: int, z: int)
{
    while (x >= 0) {
        if (z < 0) {
            if (y < 0) {
                x := x - 1;
            } else {
                y := y - 1;
            }
        } else {
            z := z - 1;
        }
    }
}
```

Has the 3-parallel ranking function, f = max{0, x + 1} + max{0, y + 1} + max{0, z + 1}. Doesn't cover in scope of project hence output will be could not prove termination or non-termination of given program.

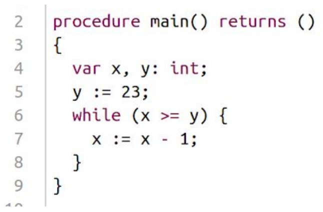
4. 

```
procedure main() returns ()
{
  var offset, i, a_i, a.length: int;
  offset := 1;
  i := 0;
  while (i <= a.length) {
    havoc a_i;
    assume (a_i >= 0);
    i := i + offset + a_i;
  }
}
```

Has linear Ranking function f(offset, i, a_i, a.length)=a.length-i with linear supporting invariant offset>=1. Occurred as abstraction of the following program with an array that contains arbitrary positive values.

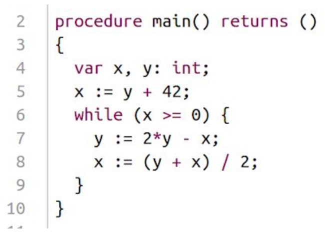
offset := 1; i := 0;

```
while (i <= a.length) {
        i := i + offset + a[i];
        }
```

5. 

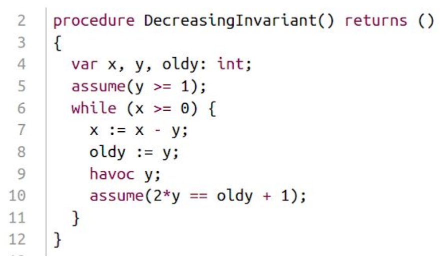
```
procedure main() returns ()
{
  var x, y: int;
  y := 23;
  while (x >= y) {
    x := x - 1;
  }
}
```

Has linear Ranking function f(x,y)=x-y with non-decreasing linear supporting invariant 0>=0

6. 

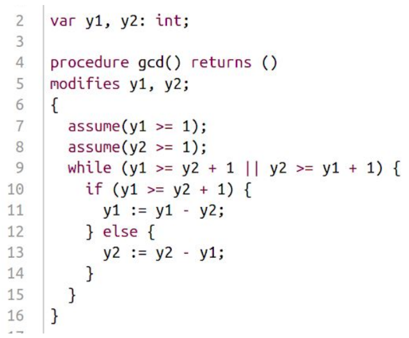
```
procedure main() returns ()
{
  var x, y: int;
  x := y + 42;
  while (x >= 0) {
    y := 2*y - x;
    x := (y + x) / 2;
  }
}
```

Has linear Ranking function f(x,y)=x with non-decreasing linear supporting invariant x-y>=42

7. 

```
procedure DecreasingInvariant() returns ()
{
  var x, y, oldy: int;
  assume(y >= 1);
  while (x >= 0) {
    x := x - y;
    oldy := y;
    havoc y;
    assume(2*y == oldy + 1);
  }
}
```

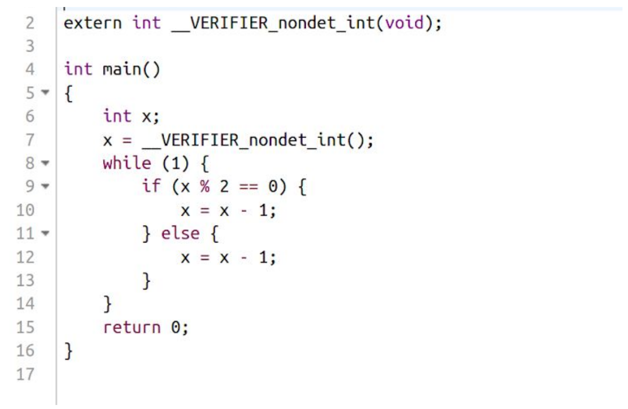
Ranking function: f(x, y, oldy) = x; needs the loop invariant y >= 1, which decreases.

8. 

```
var y1, y2: int;

procedure gcd() returns ()
modifies y1, y2;
{
  assume(y1 >= 1);
  assume(y2 >= 1);
  while (y1 >= y2 + 1 || y2 >= y1 + 1) {
    if (y1 >= y2 + 1) {
      y1 := y1 - y2;
    } else {
      y2 := y2 - y1;
    }
  }
}
```

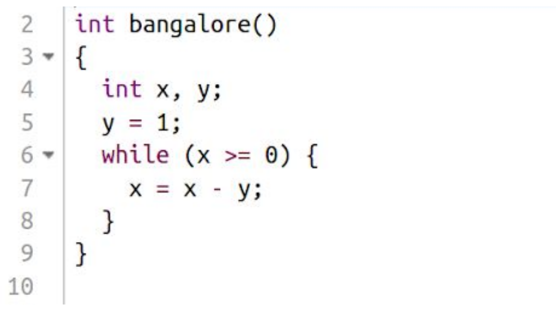
Has the ranking function f(y1, y2) = y1 + y2 provided the supporting invariants y1 >= 0, y2 >= 0.



checkc1.c :

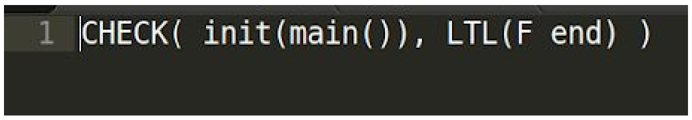
```c
extern int __VERIFIER_nondet_int(void);

int main()
{
    int x;
    x = __VERIFIER_nondet_int();
    while (1) {
        if (x % 2 == 0) {
            x = x - 1;
        } else {
            x = x - 1;
        }
    }
    return 0;
}
```



checkc2.c :

```c
int bangalore()
{
  int x, y;
  y = 1;
  while (x >= 0) {
    x = x - y;
  }
}
```

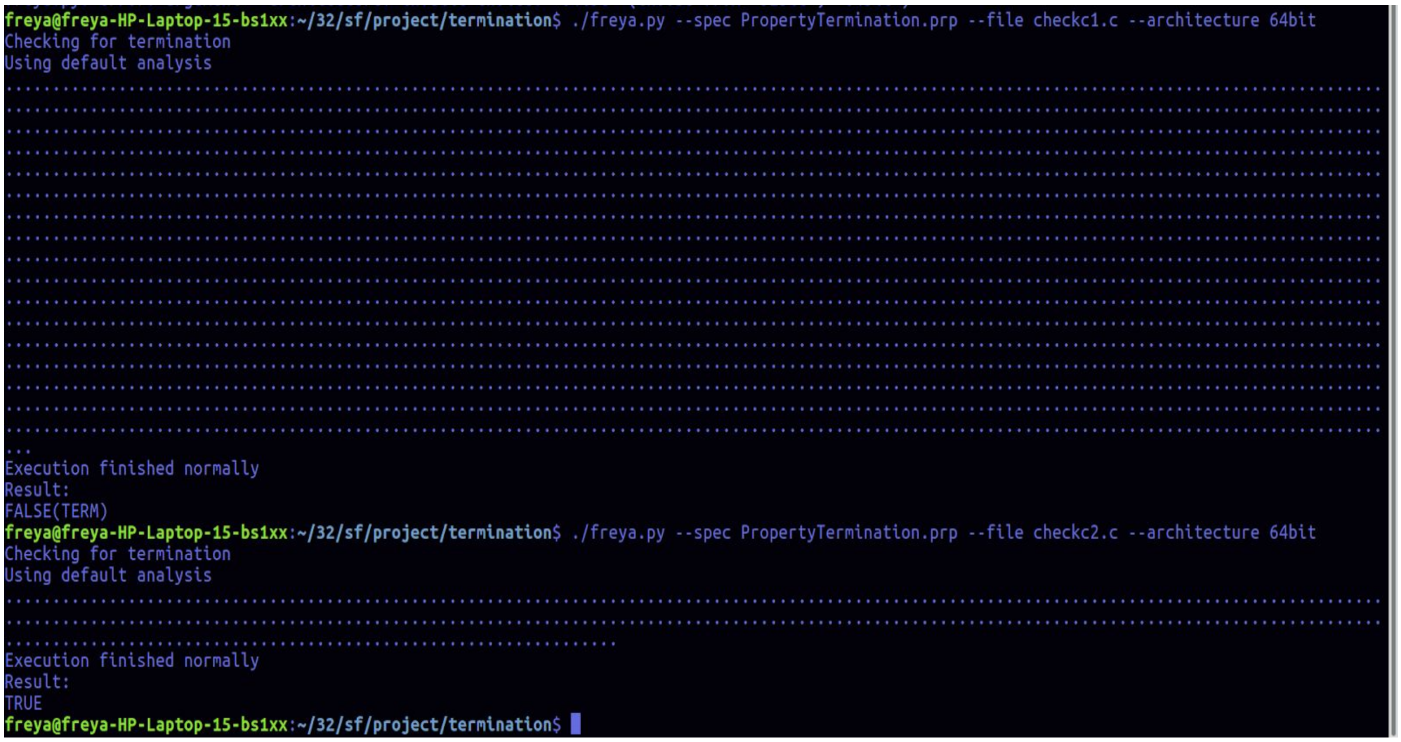


PropertyTermination.prp (property file)

```
CHECK( init(main()), LTL(F end) )
```

The result is 'TRUE' iff every program execution reaches the end of the program, i.e., iff all program executions are finite.

Result:



```
freya@freya-HP-Laptop-15-bs1xx:~/32/sf/project/termination$ ./freya.py --spec PropertyTermination.prp --file checkc1.c --architecture 64bit
Checking for termination
Using default analysis
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
....................................................................................................................................................
...
Execution finished normally
Result:
FALSE(TERM)
freya@freya-HP-Laptop-15-bs1xx:~/32/sf/project/termination$ ./freya.py --spec PropertyTermination.prp --file checkc2.c --architecture 64bit
Checking for termination
Using default analysis
....................................................................................................................................................
....................................................................................................................................................
..................................................................
Execution finished normally
Result:
TRUE
freya@freya-HP-Laptop-15-bs1xx:~/32/sf/project/termination$
```

References

1) Amir M. Ben-Amram and Samir Genaim. “On Multiphase-Linear Ranking Functions". In: CAV (2). Vol. 10427. Lecture Notes in Computer Science. Springer, 2017, pp. 601-620
2) Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. “Linear Ranking with Reachability". In: CAV. Vol. 3576. Lecture Notes in Computer Science. Springer, 2005, pp. 491-504
3) Michael Colon and Henny Sipma. “Synthesis of Linear Ranking Functions". In: TACAS. Vol. 2031. Lecture Notes in Computer Science. Springer, 2001, pp. 67-81
4) Byron Cook, Andreas Podelski, and Andrey Rybalchenko. “Termination proofs for systems code". In: PLDI. ACM, 2006, pp. 415-426
5) Bradley, Aaron Manna, Zohar Sipma, Henny. (2005). “Linear Ranking with Reachability”. Lecture Notes in Computer Science. 3576. 491-504.
6) Leino, K. Rustan M. “This is Boogie 2”. Microsoft Research. June 2008.