Long Title

ABSTRACT

ACM Reference Format:
. 2020. Long Title. In *Proceedings of The 9th Computer Science Education Research Conference (CSERC'20)*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

2 EXAMPLES

2.1 Example: Propositional & Predicate Logic

2.1.1 De Morgan's Laws. **Input File:**

```
p: BOOLEAN
q: BOOLEAN
-- formula is tautology
verify not (p and q) <=> not p or not q
verify not (p or q) <=> not p and not q
-- formula is not tautology
verify not (p and q) <=> not p and not q
```

Output Result-Part 1:

```
((not (p and q)) = ((not p) or (not q)))
Is a tautology.

((not (p or q)) = ((not p) and (not q)))
Is a tautology.
```

Part 1: This part indicates that the original De Morgan's Laws are tautologies.

Output Result-Part 2:

```
((not (p and q)) = ((not p) and (not q)))
Where:
    p : BOOLEAN
    q : BOOLEAN
Is not a tautology. Here is a counter example:
    p : false
    q : true
```

Part 2: This part indicates that the revised formula is not tautology (i.e., one which holds under all circumstances). The output also shows a counterexample containing the variable type and their values. Here it means when boolean variable p is *false* and boolean variable q is *true*, this formula will be evaluated to *false*. Therefore it is not a tautology.

2.1.2 Quantification: Single forall (∀).

For quantification verification, there is no need to declare variables separately. **Input File:**

```
-- formula is tautology
verify forall i:INTEGER | i <= i * i
-- formula is not tautology
verify forall j:INTEGER | j < j * j
```

Output Result:

```
forall i | (i <= (i * i))
Is a tautology.

forall j | (j < (j * j))
Where:
    j : INTEGER
Is not a tautology. Here is a counter example:
    j : 1
```

Similarly, this output indicates that the first formula is a tautology, where the second formula is not a tautology with an counterexample that when the value of integer variable j is 1, this formula will be evaluated to *false*.

2.1.3 Quantification: Single exists (∃).

Input File:

```
-- formula is tautology
verify exists p,q,r : BOOLEAN | (p or not q) and (q or
     not r) and (r or not p)
-- formula is not tautology
verify exists s,t,v : BOOLEAN | (s and not t) and (t and
     not v) and (v and not s)
```

Output Result:

```
exists p,q,r | (((p or (not q)) and (q or (not r))) and (
     r or (not p)))
Is a tautology.

exists s,t,v | (((s and (not t)) and (t and (not v))) and
      (v and (not s)))
Where:
    s : BOOLEAN
    t : BOOLEAN
    v : BOOLEAN
Is not a tautology.
Counterexample is not available.
```

In this example, counterexample is not available because z3 SMT Solver is used as the backend tool for *Verifier*, and when the formula contains quantifications, z3 may not provide accurate counterexamples.

Previous example that only contains single forall (∀) could be easily transformed into normal proposition (e.g., $\forall i | i \leq i * i$ is logically equivalent to $i \leq i * i$), therefore *Verifier* could correctly provide the counterexample.

2.1.4 Quantification: Nested quantification (∀ / ∃).

Input File:

```
-- formula is tautology
verify forall i:INTEGER | exists j: INTEGER | i <= j and
     j <= i => i = j
-- formula is not tautology
verify exists k:INTEGER | forall n: INTEGER | k <= n and
     n <= k => k = n + 1
```

Output Result:

```
forall i | exists j | (((i <= j) and (j <= i)) => (i = j)
     )
Is a tautology.
```


CSERC'20, October 2019, Campus The Hague, Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

```
exists k | forall n | (((k <= n) and (n <= k)) => (k = (n
     + 1)))
Where:
    k : INTEGER
    n : INTEGER
Is not a tautology.
Counterexample is not available.
```

Similarly, for the formula that contains nested quantification, counterexample may not be available as well.

2.2 Example: Program Verification

For verifying programs, *Verifier* supports the input of assignments, alternations, sequential compositions, and loops.

2.2.1 Compute Tax.

Input File:

```
compute_tax(status: INTEGER ; income: INTEGER) : REAL
require
  positive_income: income >= 0
local
  part1: REAL
  part2: REAL
  part3: REAL
do
  if status = 1 or status = 2 then
    if status = 1 then
      if income <= 8350 then
        part1 := income * 0.1;
        Result := part1;
      elseif income <= 33950 then
        part1 := 8350 * 0.1;
        part2 := (income - 8350) * 0.15;
        Result := part1 + part2;
      else
        part1 := 8350 * 0.1;
        part2 := (33950 - 8350) * 0.15;
        part3 := (income - 33950) * 0.25;
        Result := part1 + part2 + part3;
      end
    else
      if income <= 16700 then
        part1 := income * 0.1;
        Result := part1;
      elseif income <= 67900 then
        part1 := 16700 * 0.1;
        part2 := (income - 16700) * 0.15;
        Result := part1 + part2;
      else
        part1 := 16700 * 0.1;
        part2 := (67900 - 16700) * 0.15;
        part3 := (income - 67900) * 0.25;
        Result := part1 + part2 + part3;
      end
    end
  else
    Result := -1;
  end
ensure
  -- Discharged postcondition example
  discharged: (status = 1 and income = 34870) => (Result
       = part1 + part2 + part3)
  -- Not discharged postcondition example
  not_discharged: (status = 2 and income > 67900) => (
       Result = 16700 * 0.1 + (67900 - 16700) * 0.15)
end

verify compute_tax
```

Output Result-Part 1:

```
compute_tax(status : INTEGER ; income : INTEGER) : REAL
    ... (Omitted)
Where:
Precondition(Q) :
    positive_income : (income >= 0)
Postcondition(R) :
    case1_3_specific : (((status = 1) and (income =
         34870)) => (Result = ((part1 + part2) + part3)))
    not_discharged : (((status = 2) and (income > 67900))
          => (Result = ((16700 * 0.1) + ((67900 - 16700)
         * 0.15))))
Implementation(S) :
    ... (Omitted)
```

Part 1: In the beginning of the output is the original program details, and the program specifications such as precondition(Q), postcondition(R), as well as the implementations(S) will be displayed separately (the original program details and implementations are omitted because they are the same as input file).

Output Result-Part 2:

```
wp(S, discharged)
  ((((status = 1) or (status = 2)) => (((status = 1) =>
       ((((income <= 8350) => (((status = 1) and (income
       = 34870)) => ((income * 0.1) = (((income * 0.1) +
       part2) + part3)))) and (((not (income <= 8350))
       and (income <= 33950)) => (((status = 1) and (
       income = 34870)) => (((8350 * 0.1) + ((income -
       8350) * 0.15)) = (((8350 * 0.1) + ((income - 8350)
        * 0.15)) + part3))))) and (((not (income <= 8350)
       ) and (not (income <= 33950))) => (((status = 1)
       and (income = 34870)) => ((((8350 * 0.1) + ((33950
        - 8350) * 0.15)) + ((income - 33950) * 0.25)) =
       (((8350 * 0.1) + ((33950 - 8350) * 0.15)) + ((
       income - 33950) * 0.25))))))) and ((not (status =
       1)) => ((((income <= 16700) => (((status = 1) and
       (income = 34870)) => ((income * 0.1) = (((income *
        0.1) + part2) + part3)))) and (((not (income <=
       16700)) and (income <= 67900)) => (((status = 1)
       and (income = 34870)) => (((16700 * 0.1) + ((
       income - 16700) * 0.15)) = (((16700 * 0.1) + ((
       income - 16700) * 0.15)) + part3))))) and (((not (
       income <= 16700)) and (not (income <= 67900))) =>
       (((status = 1) and (income = 34870)) => ((((16700
       * 0.1) + ((67900 - 16700) * 0.15)) + ((income -
       67900) * 0.25)) = (((16700 * 0.1) + ((67900 -
       16700) * 0.15)) + ((income - 67900) * 0.25))))))))
       ) and ((not ((status = 1) or (status = 2))) => (((
       status = 1) and (income = 34870)) => (-1 = ((part1
        + part2) + part3)))))
wp(S, not_discharged)
  ... (Omitted)
```

Part 2: Following part 1, this part will output the *weakest precondition (wp)*. Note that *Verifier* will require user to provide tags for each contract in precondition and postcondition, and calculate the *wp* separately for each postcondition.

wp(S, discharged) is the *weakest precondition* for implementations(S) to establish the postcondition with tag *discharged*, and same as *wp(S, not_discharged)*.

Output Result-Part 3:

```
Proof Obligation:
(positive_income) => wp(S, discharged)
Discharged.

(positive_income) => wp(S, not_discharged)
Not discharged.
Counterexample:
    income : 67901
    status : 2
```

Part 3: This part will show the proof obligation for program correctness using Hoare Logic by generating the boolean predicate *precondition(Q)* ⇒ *wp* for each postcondition, and indicate if the result is *Discharged* (i.e., the boolean predicate could be proved true), or *Not discharged* with an counterexample.

The separation of proof obligation could help to differentiate the *wp* if there exists multiple postconditions.

2.2.2 *Loop: indices_of.*

For programs that contain loop, there are five conditions that needs to be proved, which will be illustrated in the following example.

Input File:

```
indices_of(a: ARRAY[INTEGER]; value: INTEGER): ARRAY[INTEGER]
require
  not_empty: a.count > 0
local
  i: INTEGER
  j: INTEGER
do
  from
    i := 1;
    j := 1;
  invariant
    j <= i
  until
    i > a.upper
  loop
    if a[i] = value then
      Result[j] := i;
      j := j + 1;
    end
    i := i + 1;
  variant
    loop_variant: a.upper - i + 1
  end
ensure
  -- discharged postcondition
  case1: exists k1: INTEGER | a[k1] = value => exists s1: INTEGER | Result[s1] = k1
end

verify indices_of
```

Output Result-Part 1:

```
indices_of(a : ARRAY[INTEGER];value : INTEGER) : ARRAY[INTEGER]
  ... (Omitted)

Where:
Precondition(Q) :
    not_empty : (a.count > 0)
Postcondition(R) :
    case1 : exists k1 | ((a[k1] = value) => exists s1 | (Result[s1] = k1))
Implementation(S) :
... (Omitted)
```

Part 1: Similarly, in this part, the original program details and implementations are omitted.

Output Result-Part 2:

```
Correctness conditions :
1. Given precondition Q, the initialization step Sinit establishes LI I : {Q} Sinit {I}
  ((a.count > 0) => (1 <= 1))

2. At the end of Sbody, if not yet to exit, LI I is maintained : {I and (not B)} Sbody {I}
  (((j <= i) and (not (i > a.upper))) => (((a[i] = value) => ((j + 1) <= (i + 1))) and ((not (a[i] = value)) => (j <= (i + 1)))))

3. If ready to exit and LI I maintained, postcondition R is established : I and B => R
  (((j <= i) and (i > a.upper)) => exists k1 | ((a[k1] = value) => exists s1 | (Result[s1] = k1)))

4. Given LI I, and not yet to exit, Sbody maintains LV V as non-negative : {I and (not B)} Sbody {V >= 0}
  (((j <= i) and (not (i > a.upper))) => (((a[i] = value) => (((a.upper - (i + 1)) + 1) >= 0)) and ((not (a[i] = value)) => (((a.upper - (i + 1)) + 1) >= 0))))

5. Given LI I, and not yet to exit, Sbody decrements LV V : {I and (not B)} Sbody {V < V0}
  (((j <= i) and (not (i > a.upper))) => (((a[i] = value) => (((a.upper - (i + 1)) + 1) < ((a.upper - i) + 1))) and ((not (a[i] = value)) => (((a.upper - (i + 1)) + 1) < ((a.upper - i) + 1)))))
```

Part 2: For programs that contain loop, there are five correctness conditions. Therefore, in this part, the result will include the description of these five conditions along with the calculated boolean predicate *precondition(Q)* ⇒ *wp* for each condition.

Output Result-Part 3:

```
Condition 1 is discharged.
Condition 2 is discharged.
Condition 3 is discharged.
Condition 4 is discharged.
Condition 5 is discharged.
```

Part 3: At the end, the result will indicate if these five conditions are discharged separately.

3 RELATED WORKS

Related Works here...