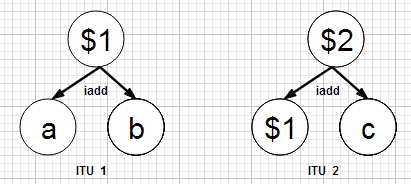
Definition (Instruction Tree Unit): An ITU is an intermediate representation of Java bytecode instruction. Each ITU is a binary tree that consists of three nodes, one parent node and two child nodes, as well as an operator (i.e., instruction) between the two children. The childnodes are the explicit named operands.The root is a named intermediate result of the operation. An ITU can be simply represented by a four-tuple(instruction, root, leftNode, rightNode) i.e. *u(b,p,l,r).*Here, *b* represents the bytecode instruction, *p* is parent root representing intermediate results of executing the instruction, *l* is left child node representing……, and *r* is right child node.

One of the essential characteristics is that ITU are restricted to the least number of operands (2 in most cases, such as for arithmetic and logic), and these operands must be either constants or locals.

For example, for a given Java expression statement x=a+b+c, the corresponding twoarithmetic ITUs are shown in Fig.1, where $1 and $2 local intermediate variables that represent (a+b) and ($1+c) respectively. Variables with a “$” sign are intermediate local variables.



Definition(Variable Dependency Tree): A VDT is a tree structure used for tracking dependency relations among all variables in the bytecode of the program. A VDT consists of 0 or more set of ITUs. All the leaves of the VDT are the bytecode input variables.

A set of ITUs can be represented by {u0, u1…uj)and U is extracted from a given p and VDT *ti*can be represented by *ti(Ui,r).*The values of *r*represents the desired outcome of VDT (i.e., true or false) if the VDT represents a predicate or (none) if the VDT does not represent predicate. The desired outcome drives the execution to the reachable statement. A set of VDTs*{t0(U0,r), t1(U1,r)…tm(Um,r)}T*can be used to represent a given path*P*of the program.

v(u0)={u1,u2,…un}, where u0 is a predicate ITU, u1-un are expression ITUs, and existing a set of R={(ui, uj) | (li=pj or ri=pj) and uiU and uj U}.

Attributes of the VDT : All leaves of the VDT are test bytecode input variables.

Definition(Edge): common leaves cl(vi, vj)=leaf (vi)leaf (vj)

Definition (Predicate Relation Graph): A Predicate Relation Graph (PRG) is an [ordered pair](http://en.wikipedia.org/wiki/Ordered_pair) *G = (T, E)* comprising a [set](http://en.wikipedia.org/wiki/Set_(mathematics)) *T* of vertices or nodes togetherwith a set *E* of edges, where *Ti*is a VDTwhich is represented by *ti(Ui,r)*derived from a give path *P* and edge e *E* is a variable dependency. Edge is represented as *e={Ti,Tj},*where*TiT* and*TjT*.

Algorithm: Constructing a PRG from a given tagged path

Input: A tagged path <T0(t), 1(t), …sn(t)>, t {none, true, false}

Output: A PRG

Declare: E=, V=

1. begin
2. VDT<s0(t), s1(t), …sn(t)>
3. VVDT
4. for each pair of *{vi,vj,}*
5. e leaf (vi)leaf (vj)
6. E=Ee
7. end
8. //
9. end

Input: PRG of a given path *p*

Output: Aboolean value representing if *p* feasible

Declare:

1. begin
2. VDT<s0(t), s1(t), …sn(t)>
3. VVDT
4. for each pair of *{vi,vj,}*
5. e leaf (vi)leaf (vj)
6. E=Ee
7. end
8. for each edge *e={vi,vj}*
9. end

* Build RPG
* Rules to identify infeasible path
  + Empty nodes
  + Circular Dependency
  + Contradiction variables

//step 1

PRG generatePRG(path){

}

//step 2: connect

PRG[] getSubPRGs(PRG){

}

//step 3 Main function

isFeasiblePath(path){

PRG prg=generatePRG(path)

RPGprgs[] getSubPRGs(prg)

Fofr(each element p inprgs)

If(isEmpty(p)&&isCyclic(p) &&isContr(p)){

Return true

}

Return false;

}

// help functionsuse dependency

Boolean isContradiction(PRG subPRG){

}

Boolean isCyclicPRG(PRG subPRG){

}

use dependency

Boolean isEMpty(PRG subPRG){

}

**Code documentations:**

1. Get all the relations for the test path and store in “relations” object
2. Separate the relations pertaining to “if” statement
3. Extract the expected results pertaining to if statements from “nodes”
4. Attach the expected result from step 3 to relations of “if” statement in step 2, i.e. create myRelations object
5. Create object InfeasiblePathAnalyzer for analysing the test path, by passing relations and myRelations objects.
6. Call the object’s (step 5) isFeasiblePath method to check if path is feasible.
   1. Generate the main PRG
   2. Get sub PRG from the main PRG
   3. For each of the sub PRG of the main PRG, get cycles(including self loops)
   4. For each cycle of the sub PRG, find if all nodes in that cycle are empty
      1. When creating the edge between nodes while creating the main PRG, at that time the elements common to nodes had their “used” flag set to true
      2. So for nodes, it was checked if all elements belonging to each node in the cycle have their used flag set to true
      3. If any of the flag is set to false in any of the node that means the cycle is not empty
   5. For each cycle of the sub PRG, find if nodes are in conflict with each other i.e. if cycle is contradictory i.e. any single edge in the cycle is having ALL parameters in edgeData with conflicting trends, also pass the relations object as it is been used to apply trends via rules to the parameters.
      1. Trends or Directions are applied by calling method applyRule to source and target nodes of each edge of the cycle
         1. There are 4 cases which are covered while applying rules. This is done to avoid setting multiple directions too many times. These 4 cases are:
            1. >= is converted to < and expected result is flipped
            2. <= is converted to > and expected result is flipped
            3. ==

with expected result True is kept same i.e. not converted to any other operator

with expected result False is further split into 2 sub-cases(i.e. !=, True means either > or < is True)

>, True

<,True

(In this case, the element’s direction is set to both Inc and Dec as both possibilities hold true.)

* + - * 1. !=

with expected result True is further split into 2 sub-cases(similar to 3.b.)

with expected result False is converted to == with expected result True(similar to 3.a.)

* + 1. For each edge, get the edge data i.e. parameter list on the edge
    2. For each parameter, check the trends by comparing source node parameter trend and target node parameter trend
    3. Store the result of the trend comparison for each parameter in a list
    4. If the result store in the list is all T, or T+F, edge is valid, if all result are F then the edge is invalid or in conflict and hence the cycle is in conflict