Points-to Analysis (https://goo.gl/0QIqeI)

```
module PointsToAnalysis import ControlFlowAnalysis

/**
  Returns the points-to tuple targets for a variable before the execution of CFG node 'n'.

  @param n the CFG node
  @param u the pointer variable
  @return the pointed-to variable
 */
def pointsToBefore(n : ICFGNode, u : IVariableDeclaration) : IVariableDeclaration = {
  b := cfg(n)
  return pointsToAfter(b, u)
}

/**
  Returns the points-to tuple targets for a variable after the execution of CFG node 'n'.

  @param n the CFG node
  @param u the pointer variable
  @return the pointed-to variable
 */
def pointsToAfter(n : ICFGNode, u : IVariableDeclaration) : IVariableDeclaration = {
  // no new binding at the current node 'n'
  assert undef pointsToAt(n)
  return pointsToBefore(n, u)
} alt {
  // there is a binding at the current node but it does not affect 'u'
  (x, y) := pointsToAt(n)
  assert x != u
  return pointsToBefore(n, u)
} alt {
  // there is a binding at the current node and it affects 'u'
  // and the binding does not point to the null literal
  (x, y) := pointsToAt(n)
  assert x == u
  assert y not instanceOf NullExpression
  return variableInAssignmentSide_right(y)
}

/**
  Returns the points-to binding at a CFG node.
  The pointed-to element of the tuple has type IAssignmentSide, because
  pointsToAfter needs to handle the null assignment as well.

  @param n the CFG node
  @return the current binding
 */
private def pointsToAt(n : ICFGNode) : (IVariableDeclaration, IAssignmentSide) = {
  (l, r) := extractSides(n)
  u := variableInAssignmentSide_left(l)
  return (u, r)
}

/**
  Returns the potential sides of assignments at a CFG node.

  @param n the CFG node
  @return the sides of assignment(s)
 */
private def extractSides(n : ICFGNode) : (IAssignmentSide, IAssignmentSide) = {
  assert n instanceOf LocalVariableDeclaration
  return (n, n.init)
} alt {
  e := extractExpression(n)
  c := extractAssignment(e)
  return (c.left, c.right)
}

/**
  Returns the expressions at a CFG node n.

  @param n the CFG node
  @return the expressions at the node
 */
private def extractExpression(n : ICFGNode) : Expression = {
  assert n instanceOf ExpressionStatement
  return n.expr
} alt {
  assert n instanceOf ForStatement
  return n.incr
} alt {
  assert n instanceOf WhileStatement
  return n.condition
} alt {
  assert n instanceOf DoWhileStatement
  return n.condition
} alt {
  assert n instanceOf IfStatement
  return n.condition
} alt {
  assert n instanceOf ElseIfPart
  return n.condition
} alt {
  assert n instanceOf SwitchStatement
  return n.expression
} alt {
  assert n instanceOf LocalVariableDeclaration
  return n.init
}

/**
  Returns the assignment expression(s) in an expression node.

  @param e the expression
  @return the assignment expression(s)
 */
private def extractAssignment(e : Expression) : AssignmentExpr = {
  assert e instanceOf AssignmentExpr
  return e
} alt {
  assert e instanceOf ParensExpression
  return extractAssignment(e.expression)
} alt {
  assert e instanceOf ExpressionList
  return extractAssignment(e.expressions)
} alt {
  assert e instanceOf BinaryExpression
  return extractAssignment(e.left)
} alt {
  assert e instanceOf BinaryExpression
  return extractAssignment(e.right)
} alt {
  assert e instanceOf TernaryExpression
  return extractAssignment(e.condition)
} alt {
  assert e instanceOf TernaryExpression
  return extractAssignment(e.thenExpr)
} alt {
  assert e instanceOf TernaryExpression
  return extractAssignment(e.elseExpr)
}

/**
```

```
  Returns the pointer variable from an assignmet left hand side
  based on Andersen's rules.

  @param s the left hand side of an assignment
  @return the pointer variable
 */
private def variableInAssignmentSide_left(s : IAssignmentSide) : IVariableDeclaration = {
  assert s instanceOf DerefExpr
  u := variableInAssignmentSide_left(s.expression)
  n := eval(s.ancestor<concept = ICFGNode>)
  v := pointsToBefore(n, u)
  return v
} alt {
  assert s instanceOf ParensExpression
  return variableInAssignmentSide_left(s.expression)
} alt {
  return variableInAssignmentSide_primitive(s)
}

/**
  Returns the pointed-to variable from an assignmet right hand side
  based on Andersen's rules.

  @param s the right hand side of an assignment
  @return the pointed-to variable
 */
private def variableInAssignmentSide_right(s : IAssignmentSide) : IVariableDeclaration = {
  assert s instanceOf ReferenceExpr
  return variableInAssignmentSide_primitive(s.expression)
} alt {
  assert s instanceOf DerefExpr
  u := variableInAssignmentSide_right(s.expression)
  n := eval(s.ancestor<concept = ICFGNode>)
  v := pointsToBefore(n, u)
  return v
} alt {
  assert s instanceOf ParensExpression
  return variableInAssignmentSide_right(s.expression)
} alt {
  u := variableInAssignmentSide_primitive(s)
  n := eval(s.ancestor<concept = ICFGNode>)
  v := pointsToBefore(n, u)
  return v
}

/**
  Returns the variable in an assignment side.

  @param s the assignment side (left or right)
  @return the variable
 */
private def variableInAssignmentSide_primitive(s : IAssignmentSide) : IVariableDeclaration = {
  assert s instanceOf GlobalVarRef
  return s.var
} alt {
  assert s instanceOf LocalVarRef
  return s.var
} alt {
  assert s instanceOf ArgumentRef
  return s.arg
} alt {
  assert s instanceOf LocalVariableDeclaration
  return s
}
```