Control Flow Analysis (https://goo.gl/0QIqeI)

```
module ControlFlowAnalysis

/**
  Returns the node(s) in the CFG that has an edge pointing to node 'n'.

  @param n the CFG target
  @return the CFG source(s)
 */
def cfg(n : ICFGNode) : ICFGNode = {
  s := cfg0(n)
  assert s not instanceOf ReturnStatement
  return s
}

/**
  Helper function for function cfg. This function does not
  filter out return statements as illegal CFG sources.

  @param n the CFG target
  @return the CFG source(s)
 */
private def cfg0(n : ICFGNode) : ICFGNode = {
  assert undef cannotActAsEdgeTarget(n)
  return cfg_before(n)
} alt {
  assert undef cannotActAsEdgeTarget(n)
  return cfg_testFirstControl(n)
} alt {
  return cfg_testAfterControl(n)
} alt {
  return cfg_special(n)
} alt {
  return cfg_continue(n)
} alt {
  return cfg_break(n)
} alt {
  return cfg_section(n)
}

/**
  CFG edges related to continue statements
  (1) node 'n' is the first loop ancestor of a continue statement 's'

  @param n the CFG target loop
  @return the CFG source continue statement
 */
private def cfg_continue(n : ICFGNode) : ContinueStatement = {
  assert s instanceOf ContinueStatement
  assert n == firstLoopAncestor(s)
  return s
}

/**
  CFG edges related to break statements
  (1) there exists a break statement 's', its first loop ancestor is 'l'
      and the first CFG node after 'l' is 'n': 's' -> 'n'

  @param n the CFG target
  @return the CFG source break statement
 */
private def cfg_break(n : ICFGNode) : BreakStatement = {
  assert s instanceOf BreakStatement
  l := firstLoopAncestor(s)
  a := directlyAfter(l)
  assert n == firstNodes(a)
```

```
  return s
}

/**
  CFG edges related to else if parts and switch branches.

  @param n the CFG target
  @return the CFG source
 */
private def cfg_special(n : ICFGNode) : ICFGNode = {
  return cfg_elseIf(n)
} alt {
  return cfg_switchCase(n)
}

/**
  CFG edges related to test first control statements.

  @param n the CFG target
  @return the CFG source
 */
private def cfg_testFirstControl(n : ICFGNode) : ICFGNode = {
  s := cfg_while(n)
  assert s not instanceOf ContinueStatement
  return s
} alt {
  s := cfg_for(n)
  assert s not instanceOf ContinueStatement
  return s
} alt {
  s := cfg_if(n)
  assert s not instanceOf ContinueStatement
  return s
} alt {
  s := cfg_switch(n)
  assert s not instanceOf ContinueStatement
  return s
}

/**
  CFG edges related to test after control statements.

  @param n the CFG target
  @return the CFG source
 */
private def cfg_testAfterControl(n : ICFGNode) : ICFGNode = {
  s := cfg_doWhile(n)
  assert s not instanceOf ContinueStatement
  return s
}

/**
  CFG edges produced through the syntactic precedence relationship.
  (1) node 'b' is directly before node 's' and node 'l' is a last CFG node of 'b': 'l' -> 's'

  @param s the CFG target
  @return the CFG source
 */
private def cfg_before(s : ICFGNode) : ICFGNode = {
  b := directlyBefore(s)
  l := lastNodes(b)
  assert l not instanceOf ContinueStatement
  return l
}

/**
  We call a statement headless if the very first CFG node in the context of the statement
  is not the statement itself, rather some other contained CFG node.
```

```
  The function returns the headless parent of a statement 's', if any.

  @param s the contained statement
  @return the container headless statement
 */
private def cfg_headless_parent(s : Statement) : ICFGNode = {
  assert undef directlyBefore(s)
  p := parent(s)
  assert p instanceOf StatementList
  d := parent(p)
  assert d instanceOf DoWhileStatement
  return d
} alt {
  assert undef directlyBefore(s)
  l := parent(s)
  assert l instanceOf StatementList
  p := parent(l)
  assert eval(!CFGHelper.hasStatementList(p))
  return l
}

/**
  Returns the CFG source(s) of statement 's' after skipping all
  headless ancestors.

  @param s the CFG target
  @return the CFG source
 */
private def cfg_headless_traversal(s : Statement) : ICFGNode = {
  p := cfg_headless_parent(s)
  return cfg_before(p)
} alt {
  p := cfg_headless_parent(s)
  return cfg_testFirstControl(p)
} alt {
  p := cfg_headless_parent(s)
  return cfg_headless_traversal(p)
}

/**
  CFG edges related to sections.

  @param s the CFG target
  @return the CFG source
 */
private def cfg_section(s : Statement) : ICFGNode = {
  assert undef cannotActAsEdgeTarget(s)
  return cfg_headless_traversal(s)
}

/**
  CFG edges related to do while statements
  (1) there is a do while statement 'd' whose first CFG node is 's': 'd' -> 's'
  (2) CFG source(s) obtained after skipping all headless ancestors.
  (3) 's' is a do while statement and 'l' is a last node in 's': 'l' -> 's'

  @param s the statement in the context of a do while statement
  @return the CFG source
 */
private def cfg_doWhile(s : Statement) : ICFGNode = {
  assert undef cannotActAsEdgeTarget(s)
  assert d instanceOf DoWhileStatement
  assert s == firstNodes(d)
  return d
} alt {
  assert undef cannotActAsEdgeTarget(s)
  return cfg_headless_traversal(s)
} alt {
```

```
  assert s instanceOf DoWhileStatement
  i := lastStatementInDoWhile(s)
  l := lastNodes(i)
  assert l not instanceOf BreakStatement
  return l
}

/**
  CFG edges related to if statements
  (1) statement 's' is the first statement in the body of an if statement 'e' or
      in the body of an else if part 'e': 'e' -> 's'
  (2) statement 's' is the first statement in the body of an else part 'e' and the container
      if statement's last else if part is 'f': 'f' -> 's'
  (3) statement 's' is the first statement in the body of an else part 'e' and the container
      if statement 'i' has no else if part: 'i' -> 's'

  @param s the statement in the context of a switch statement
  @return the CFG source
 */
private def cfg_if(s : Statement) : ICFGNode = {
  assert undef directlyBefore(s)
  p := parent(s)
  assert p instanceOf StatementList
  e := parent(p)
  assert eval(e.isInstanceOf(IfStatement) || e.isInstanceOf(ElseIfPart))
  return e
} alt {
  assert undef directlyBefore(s)
  p := parent(s)
  assert p instanceOf StatementList
  e := parent(p)
  assert e instanceOf ElsePart
  i := parent(e)
  assert i instanceOf IfStatement
  return lastElseIfInIfStatement(i)
} alt {
  assert undef directlyBefore(s)
  p := parent(s)
  assert p instanceOf StatementList
  e := parent(p)
  assert e instanceOf ElsePart
  i := parent(e)
  assert i instanceOf IfStatement
  assert undef lastElseIfInIfStatement(i)
  return i
}

/**
  CFG edges related to switch statements
  (1) statement 's' is the first statement in the body of a case branch 'c': 'f' -> 's'
  (2) statement 's' is the first statement in the body of a default branch 'd' and the container
      switch statement is 'c': 'c' -> 's'
  (3) statement 's' is the first statement in the body of a branch and the control
      falls through from the last statement(s) 'l' of the previous branch: 'l' -> 's'

  @param s the statement in the context of a switch statement
  @return the CFG source
 */
private def cfg_switch(s : Statement) : ICFGNode = {
  assert undef directlyBefore(s)
  p := parent(s)
  assert p instanceOf StatementList
  c := parent(p)
  assert c instanceOf SwitchCase
  return c
} alt {
  assert undef directlyBefore(s)
  p := parent(s)
```

```
  assert p instanceOf StatementList
  d := parent(p)
  assert d instanceOf SwitchDefault
  c := parent(d)
  assert c instanceOf SwitchStatement
  return c
} alt {
  assert undef directlyBefore(s)
  p := parent(s)
  assert p instanceOf StatementList
  m := parent(p)
  assert m instanceOf SwitchMember
  pm := directlyBefore(m)
  i := lastStatementInSwitchCase(pm)
  l := lastNodes(i)
  assert l not instanceOf BreakStatement
  return l
}

/**
  CFG edges related to while statements
  (1) statement 's' is the first statement in the body of a while statement 'f': 'f' -> 's'
  (2) 's' is a while statement and 'l' is a last statement in its body: 'l' -> 's'

  @param s the statement in the context of a while statement
  @return the CFG source
 */
private def cfg_while(s : Statement) : ICFGNode = {
  assert undef directlyBefore(s)
  p := parent(s)
  assert p instanceOf StatementList
  f := parent(p)
  assert f instanceOf WhileStatement
  return f
} alt {
  assert s instanceOf WhileStatement
  i := lastStatementInFor(s)
  l := lastNodes(i)
  assert l not instanceOf BreakStatement
  return l
}

/**
  CFG edges related to for statements.
  (1) statement 's' is the first statement in the body of a for statement 'f': 'f' -> 's'
  (2) 's' is a for statement and 'l' is a last statement in its body: 'l' -> 's'

  @param s the statement in the context of a for statement
  @return the CFG source
 */
private def cfg_for(s : Statement) : ICFGNode = {
  assert undef directlyBefore(s)
  p := parent(s)
  assert p instanceOf StatementList
  f := parent(p)
  assert f instanceOf ForStatement
  return f
} alt {
  assert s instanceOf ForStatement
  i := lastStatementInFor(s)
  l := lastNodes(i)
  assert l not instanceOf BreakStatement
  return l
}

/**
  Returns the CFG source of an else if part.
  (1) the direct predecessor else if part
```

```
   (2) the container if statement if (1) is rejected
   @param e the else if part
   @return the CFG source
 */
private def cfg_elseIf(e : ElseIfPart) : ICFGNode = {
  b := directlyBefore(e)
  assert b instanceOf ElseIfPart
  return b
} alt {
  assert undef directlyBefore(e)
  s := parent(e)
  assert s instanceOf IfStatement
  return s
}

/**
  Returns the CFG source of a switch case branch.
  The CFG source if the cotainer switch statement.

  @param c the switch case branch
  @return the CFG source
 */
private def cfg_switchCase(c : SwitchCase) : ICFGNode = {
  s := parent(c)
  assert s instanceOf SwitchStatement
  return s
}

/**
  Returns the CFG node which can act as the first node in a given CFG node.
  First means that in the context of a given node this node represents
  the first control flow point during execution.

  @param n the CFG node
  @return the first node
 */
private def firstNodes(n : ICFGNode) : ICFGNode = {
  assert eval(!n.isInstanceOf(DoWhileStatement))
  return n
} alt {
  assert n instanceOf DoWhileStatement
  f := firstStatementInDoWhile(n)
  return firstNodes(f)
}

/**
  Returns the CFG node(s) which can act as the last node(s) in a given CFG node.
  Last means that in the context of a given node these nodes represent
  the final control flow points during execution.

  @param n the CFG node
  @return the last node(s)
 */
private def lastNodes(n : ICFGNode) : ICFGNode = {
  // if statement is a last statement itself if it has neither else if part(s) nor else part
  assert n instanceOf IfStatement
  assert undef lastElseIfInIfStatement(n)
  assert undef elsePartOfIf(n)
  return n
} alt {
  // last elseif part of if statement, if it has no else part
  assert n instanceOf IfStatement
  assert undef elsePartOfIf(n)
  elseIf := lastElseIfInIfStatement(n)
  return elseIf
} alt {
  // last statements of if statement
```

```
  assert n instanceOf IfStatement
  l := lastStatementInIf(n)
  return lastNodes(l)
} alt {
  // last statements of section
  assert n instanceOf StatementList
  l := lastStatementInStatementList(n)
  return lastNodes(l)
} alt {
  // while statement itself is a last statement
  assert n instanceOf WhileStatement
  return n
} alt {
  // for statement itself is a last statement
  assert n instanceOf ForStatement
  return n
} alt {
  // do while statement itself is a last statement
  assert n instanceOf DoWhileStatement
  return n
} alt {
  // primitive statement itself is a last statement
  assert eval(CFGHelper.isPrimitiveStatement(n))
  return n
} alt {
  // last statements of switch statement
  assert n instanceOf SwitchStatement
  l := lastStatementInSwitch(n)
  return lastNodes(l)
}

// HELPER FUNCTIONS

/**
  Enumerates CFG nodes which cannot act as CFG edge targets
  from syntcatically before the node.

  @param n the node
 */
def cannotActAsEdgeTarget(n : ICFGNode) : << ... >> = {
  assert n instanceOf DoWhileStatement
} alt {
  assert n instanceOf StatementList
}

/**
  Returns the first loop-like ancestor of a node.

  @param child the child node
  @return the loop ancestor
 */
def firstLoopAncestor(child : IWithParentPointer) : IParentPointerTarget = {
  parent := parent(child)
  assert eval(CFGHelper.isLoopStatement(parent))
  return parent
} alt {
  parent := parent(child)
  assert eval(!CFGHelper.isLoopStatement(parent))
  return firstLoopAncestor(parent)
}

/**
  Returns the last statement(s) of an if statement.

  @param s the if statement
  @return the last statement(s)
 */
private def lastStatementInIf(s : IfStatement) : Statement = {
```

```
  b := s.thenPart
  return lastStatementInStatementList(b)
} alt {
  b := s.elseIfs.body
  return lastStatementInStatementList(b)
} alt {
  b := s.elsePart.body
  return lastStatementInStatementList(b)
}

/**
  Returns the last statement of a do while statement.

  @param s the do while statement
  @return the last statement
 */
private def lastStatementInDoWhile(s : DoWhileStatement) : Statement = {
  b := s.body
  return lastStatementInStatementList(b)
}

/**
  Returns the first statement of a do while statement.

  @param s the do while statement
  @return the first statement
 */
private def firstStatementInDoWhile(s : DoWhileStatement) : Statement = {
  b := s.body
  return firstStatementInStatementList(b)
}

/**
  Returns the last statement of a while statement.

  @param s the while statement
  @return the last statement
 */
private def lastStatementInWhile(s : WhileStatement) : Statement = {
  b := s.body
  return lastStatementInStatementList(b)
}

/**
  Returns the last statement of a for statement.

  @param s the for statement
  @return the last statement
 */
private def lastStatementInFor(s : ForStatement) : Statement = {
  b := s.body
  return lastStatementInStatementList(b)
}

/**
  Returns the last statement(s) of a switch statement.

  @param s the switch statement
  @return the last statement(s)
 */
private def lastStatementInSwitch(s : SwitchStatement) : Statement = {
  c := s.cases
  assert c instanceOf SwitchCase
  return lastStatementInSwitchCase(c)
} alt {
  d := s.cases
  assert d instanceOf SwitchDefault
  return lastStatementInSwitchDefault(d)
```

```
}

/**
  Returns the last statement of a switch case branch.

  @param c the case branch
  @return the last statement
 */
private def lastStatementInSwitchCase(c : SwitchCase) : Statement = {
  b := c.body
  return lastStatementInStatementList(b)
}

/**
  Returns the last statement of a switch default branch.

  @param d the default branch
  @return the last statement
 */
private def lastStatementInSwitchDefault(d : SwitchDefault) : Statement = {
  b := d.body
  return lastStatementInStatementList(b)
}

/**
  Returns the else part of an if statement.

  @param s the if statement
  @return the else part
 */
private def elsePartOfIf(s : IfStatement) : ElsePart = {
  return s.elsePart
}

/**
  Returns the last statement in a statement list.

  @param l the statement list
  @return the last statement
 */
private def lastStatementInStatementList(l : StatementList) : Statement = {
  s := l.statements
  assert undef directlyAfter(s)
  return s
}

/**
  Returns the first statement in a statement list.

  @param l the statement list
  @return the first statement
 */
private def firstStatementInStatementList(l : StatementList) : Statement = {
  s := l.statements
  assert undef directlyBefore(s)
  return s
}

/**
  Returns the last else if part of an if statement.

  @param s the if statement
  @return the last else if part
 */
private def lastElseIfInIfStatement(s : IfStatement) : ElseIfPart = {
  elseIf := s.elseIfs
  assert undef directlyAfter(elseIf)
  return elseIf
```

```
}

/**
  Returns the parent node of a child node.

  @param child the child node
  @return the parent node
 */
private def parent(child : IWithParentPointer) : IParentPointerTarget = {
  return child.parent_wb
}

/**
  Returns the node which is syntactically before a given node.

  @param trg the successor node
  @return the predecessor node
 */
private def directlyBefore(trg : INextPointerTarget) : IWithNextPointer = {
  assert src instanceOf IWithNextPointer
  assert trg == src.next_wb
  return src
}

/**
  Returns the node which is syntactically after a given node.

  @param src the predecessor node
  @return the successor node
 */
def directlyAfter(src : IWithNextPointer) : INextPointerTarget = {
  return src.next_wb
}
```