Compiler Techniques

Lecture 13: Control Flow Analysis
(Not Covered in Exam)

Tianwei Zhang

Outline

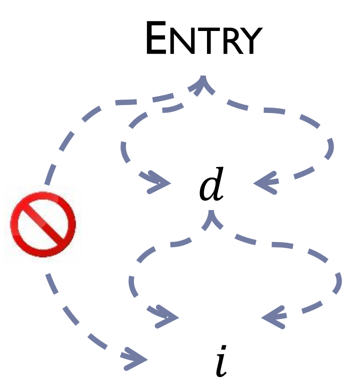
- **Dominance Analysis**
- **Loop Optimization**
- **Static Single Assignment**
- **Inter-Procedural Optimisations**
- **Appendix: Optimisations using Soot**

Outline

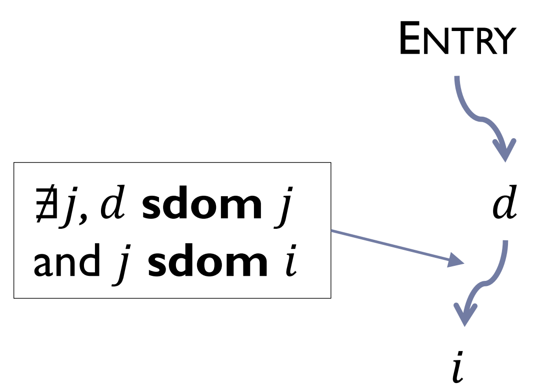
- **Dominance Analysis**
- **Loop Optimization**
- **Static Single Assignment**
- **Inter-Procedural Optimisations**
- **Appendix: Optimisations using Soot**

Dominator Terminology

- Dominator.
  - d is a dominator of i (d **dom** i) if all paths from entry to i include d
- Strict Dominator.
  - d is a strict dominator of i (d **sdom** i) if d **dom** i and $d \neq i$
- Immediate Dominator.
  - d is an immediate dominator of i (d **idom** i) if d **sdom** i and there does not exist a node j such that d **sdom** j and j **sdom** i



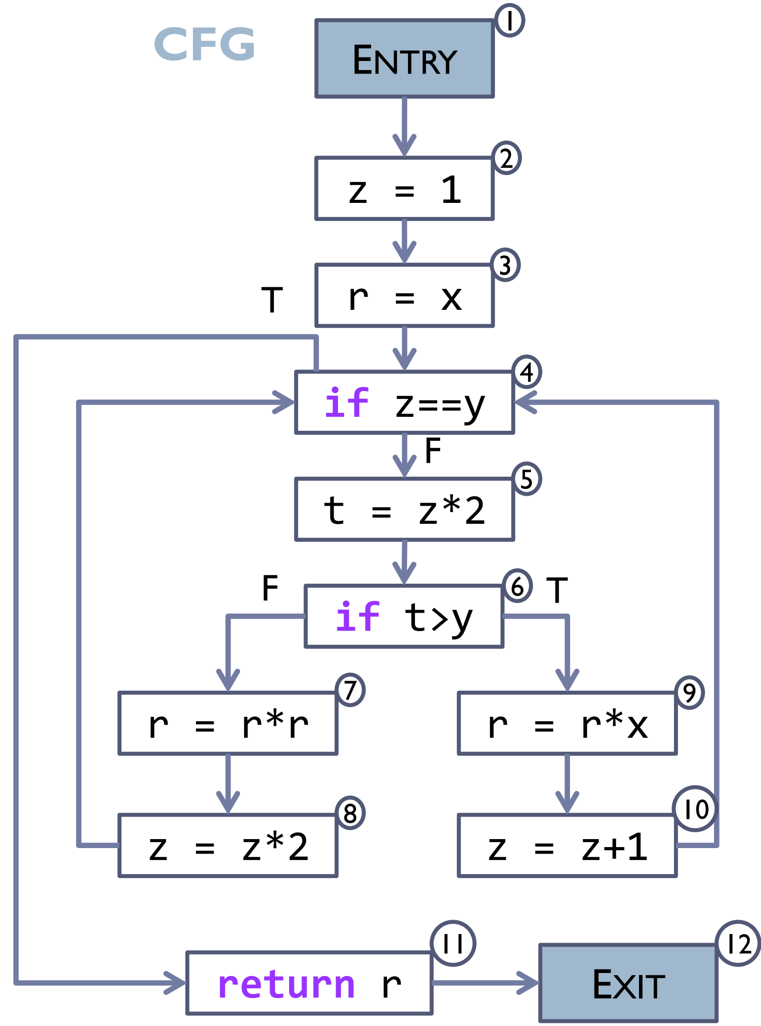
d **dom** i



d **idom** i

Dominator Example

Node	Dom	sdom	idom
1	{1}	∅	∅
2	{1,2}	{1}	{1}
3	{1,2,3}	{1,2}	{2}
4	{1,2,3,4}	{1,2,3}	{3}
5	{1,2,3,4,5}	{1,2,3,4}	{4}
6	{1,2,3,4,5,6}	{1,2,3,4,5}	{5}
7	{1,2,3,4,5,6,7}	{1,2,3,4,5,6}	{6}
8	{1,2,3,4,5,6,7,8}	{1,2,3,4,5,6,7}	{7}
9	{1,2,3,4,5,6,9}	{1,2,3,4,5,6}	{6}
10	{1,2,3,4,5,6,9,10}	{1,2,3,4,5,6,9}	{9}
11	{1,2,3,4,11}	{1,2,3,4}	{4}
12	{1,2,3,4,11,12}	{1,2,3,4,11}	{11}



Dominance analysis

- A node *m* can dominate *n* before or after *n*.

- Flow sets:
  - $in_D(n)$: the set of nodes that can dominate before *n*
  - $out_D(n)$: the set of nodes that can dominate after *n*

- Goal of dominance analysis: compute $in_D(n)$ and $out_D(n)$ for every CFG node *n*

Transfer Functions

- If we already know $in_D(n)$, it is easy to compute $out_D(n)$ (forward analysis):

$$out_D(n) = in_D(n) \cup \{n\}$$

- Calculate $in_D(n)$ (must analysis)
  1. Node n is the ENTRY node

$$in_D(\text{ENTRY}) = \varnothing$$

  2. Node n has at least one predecessor nodes:
     - $pred(n)$ be the set of predecessor nodes of n
     - A node can dominate before n if it can dominate after all predecessors of n

$$in_D(n) = \cap\{\, out_D(m) \mid m \in pred(n) \,\}$$

- Can be addressed by iterative solution or worklist algorithm

Outline

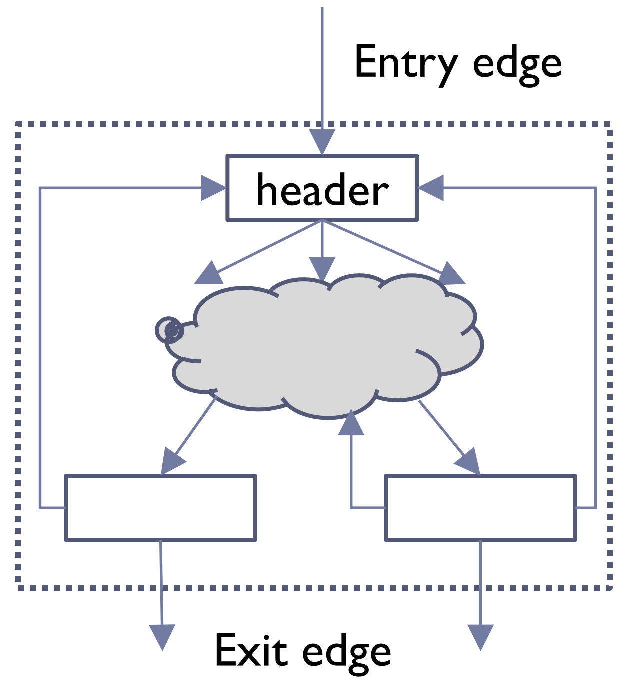
- Dominance Analysis
- **Loop Optimization**
- Static Single Assignment
- Inter-Procedural Optimisations
- Appendix: Optimisations using Soot

Loops

- Motivation
  - Most execution time is spent in loops.
  - Optimizing loops can give huge benefit.
- Loop optimization
  - Loop-invariant code hoisting: hoisting expressions out of the loop to avoid re-computation
  - Strength reduction: convert complex operations to simple operations
  - Remove useless variables: delete variables that are never used in the loops
  - …..
- Loop identification
  - Interval analysis
  - Structural analysis
  - Dominator-based

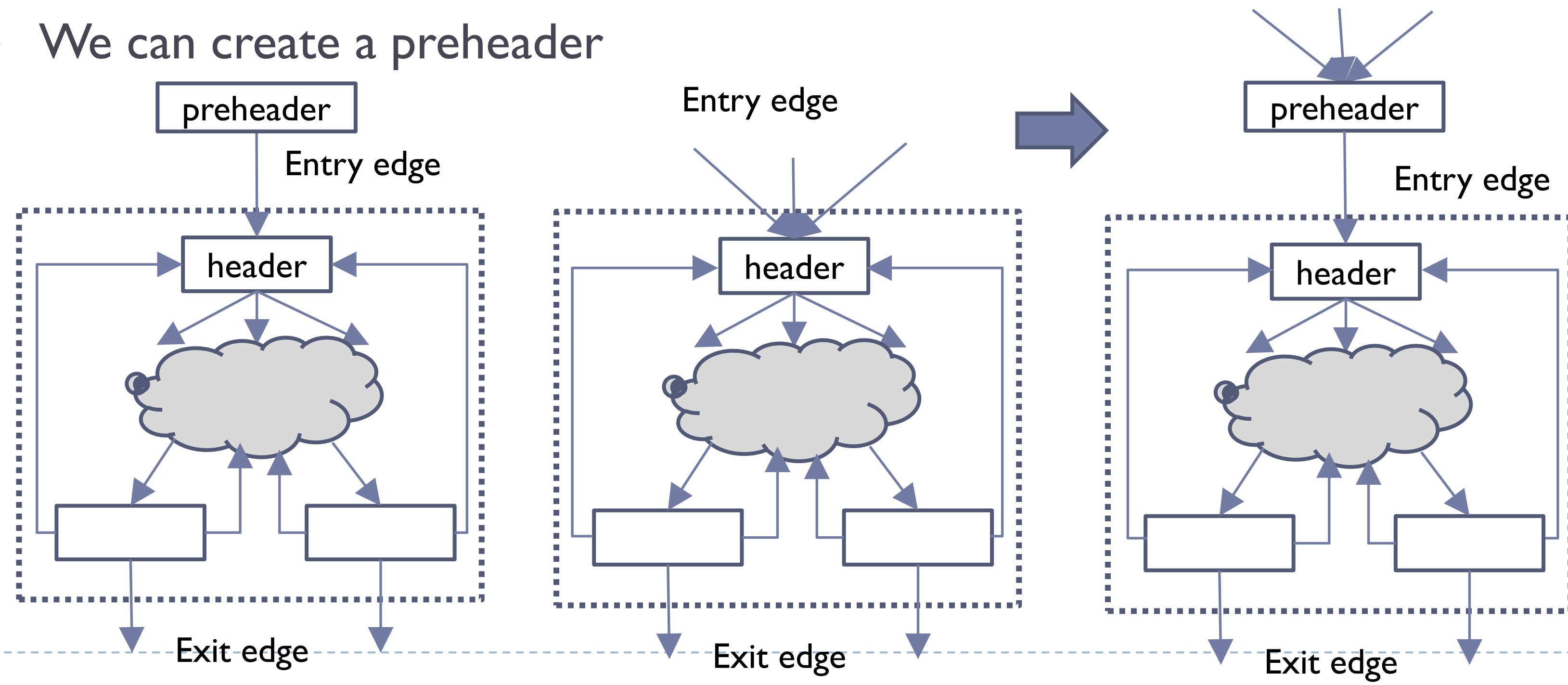
What is a Loop?

- A loop is a set of nodes S in a control flow graph such that:
  - There is a header node h that dominates all nodes in S, *i.e.*, there exists a path from h to any node inside S
  - h is the only node in S with predecessors not in S
  - For any node in S, there exists a path from it to h
- Entry edge: an edge whose source is outside of the loop and target is inside the loop
- Exit edge: an edge whose source is inside the loop and target is outside of the loop
- Nested loop: a loop whose header is inside another loop.



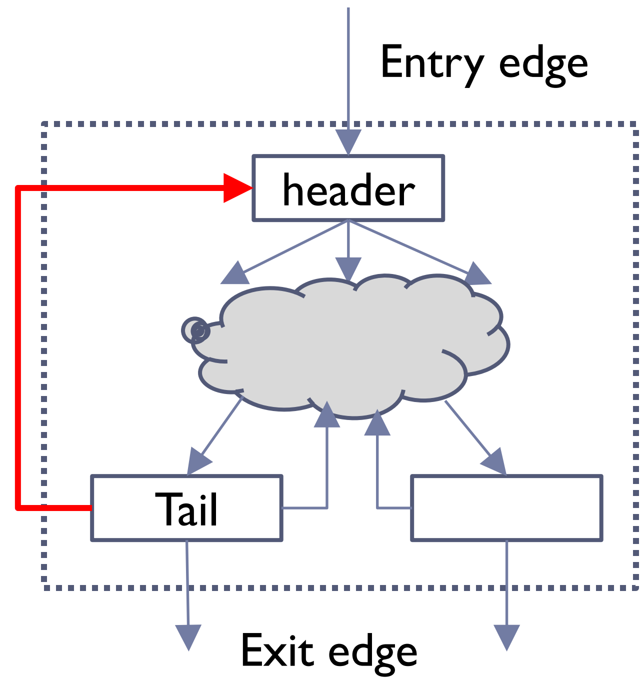
Loop Preheader

- Preheader: a single node who is the source of the entry edge.
  - There is only one entry edge
- A loop may have no preheader node,
  - When there are multiple entry edges
  - We can create a preheader



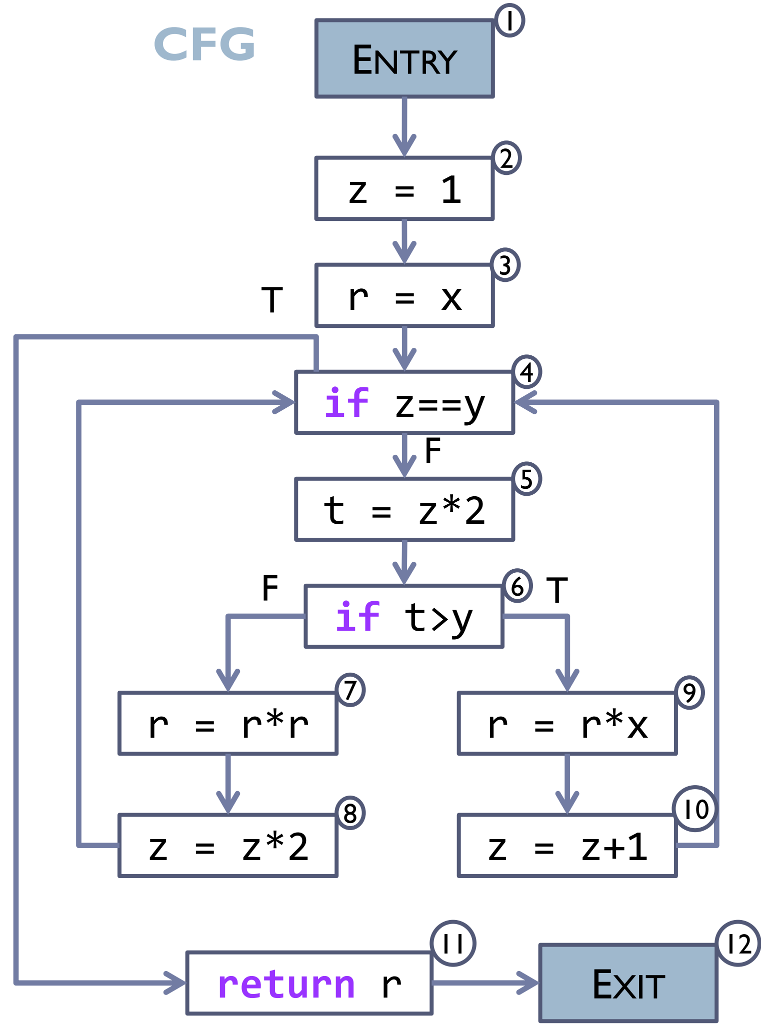
Natural Loop

- Back edge: a connection whose target node dominates its source node
  - In a loop, the target of a back edge is the header,
  - The source is a node inside the loop (tail node)
- Natural loop: associated with a back edge. It is the set of nodes x dominated by the header, and with path from x to the tail without containing the header.



Natural Loop Example

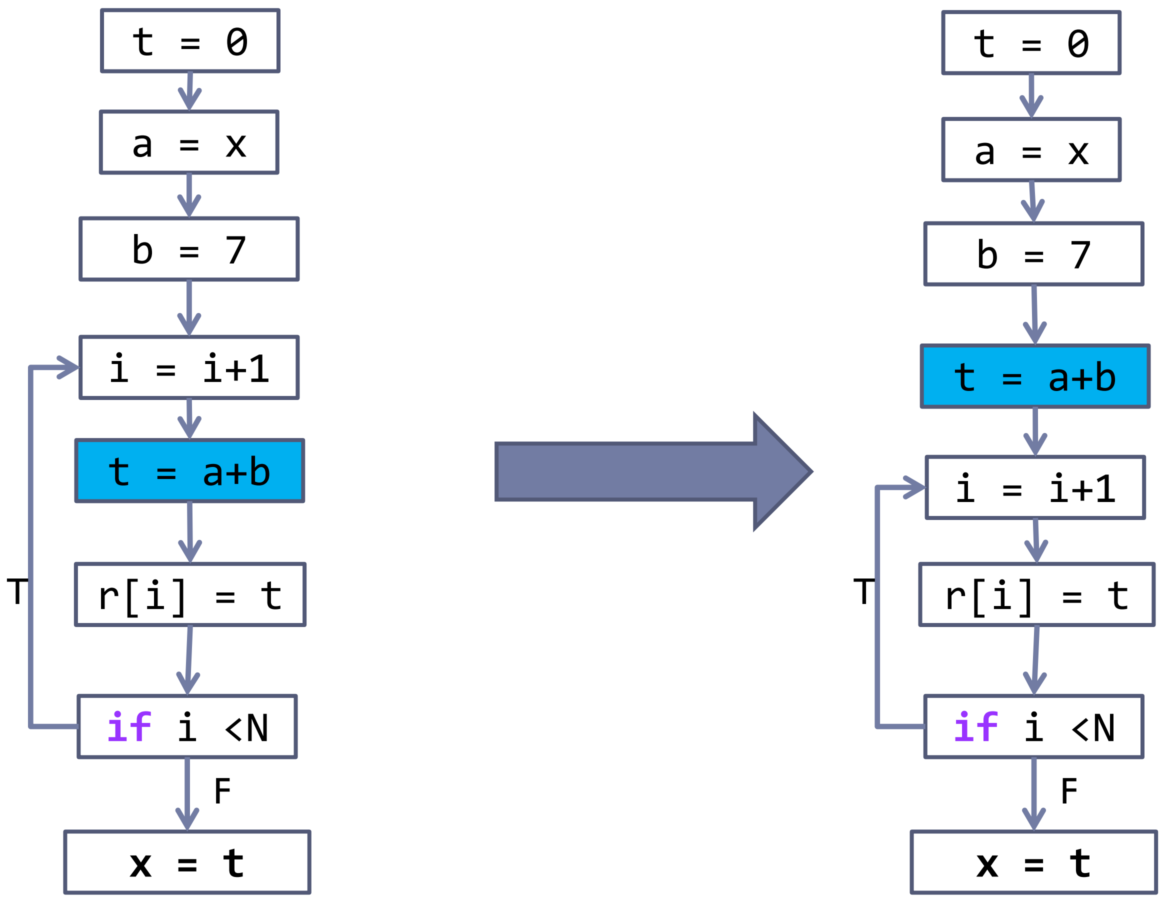
- To discover all the natural loops inside a CFG, we can apply dominance analysis to identify back edges, and then identify the corresponding natural loops.
- Back edge
  - 8 → 4
  - 10 → 4
- Natural loop
  - {4, 5, 6, 7, 8}
  - {4, 5, 6, 9, 10}



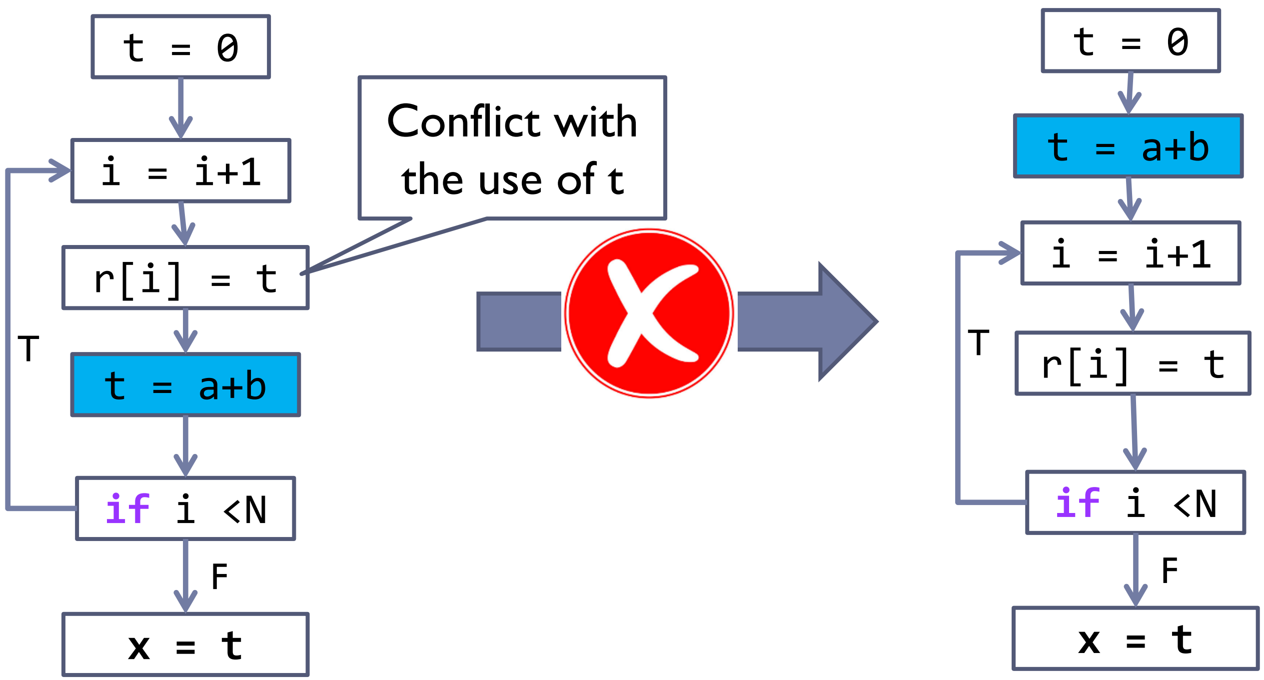
Optimization: Loop-invariant code hoisting

- An assignment $x = v_1 \text{ op } v_2$ is invariant for a loop if for each operand v_1 and v_2 either
  - the operand is constant, or
  - all the definitions that reach the assignment are outside the loop, or
  - only one definition reaches the assignment, and it is a loop invariant
- We can hoist loop-invariant code.

Code Hoisting Example



Invalid Code Hoisting Example



Conditions for Safe Hoisting

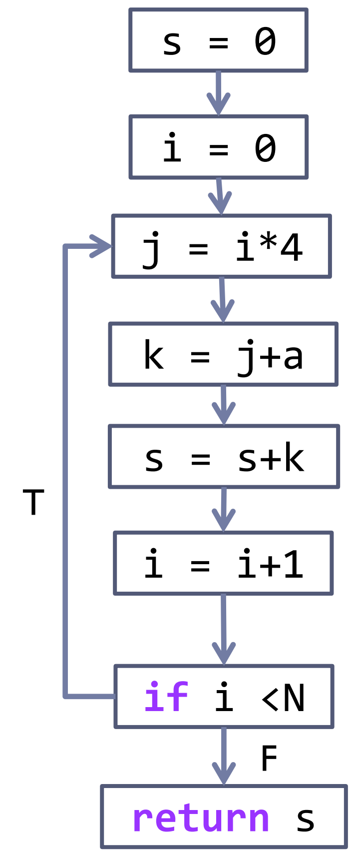
- An invariant assignment d: $x = v_1 \text{ op } v_2$ is safe to hoist if:
  - d dominates all loop exits at which x is live and
  - there is only one definition of x in the loop, and
  - x is not live at the entry point for the loop (the preheader)

Optimization: Strength Reduction

- Replace expensive operation (multiplication) with cheaper one (addition)
- Basic induction variable: a variable i in a loop if the only definition of i in this loop is in the form: $i = i + c$ or $i = i - c$, where c is loop-invariant.
- Derived induction variable: a variable k in a loop if the only definition of k in this loop can be derived as $k = a * i + b$, where a, b is loop-invariant, and i is a basic induction variable

Induction Variable Example

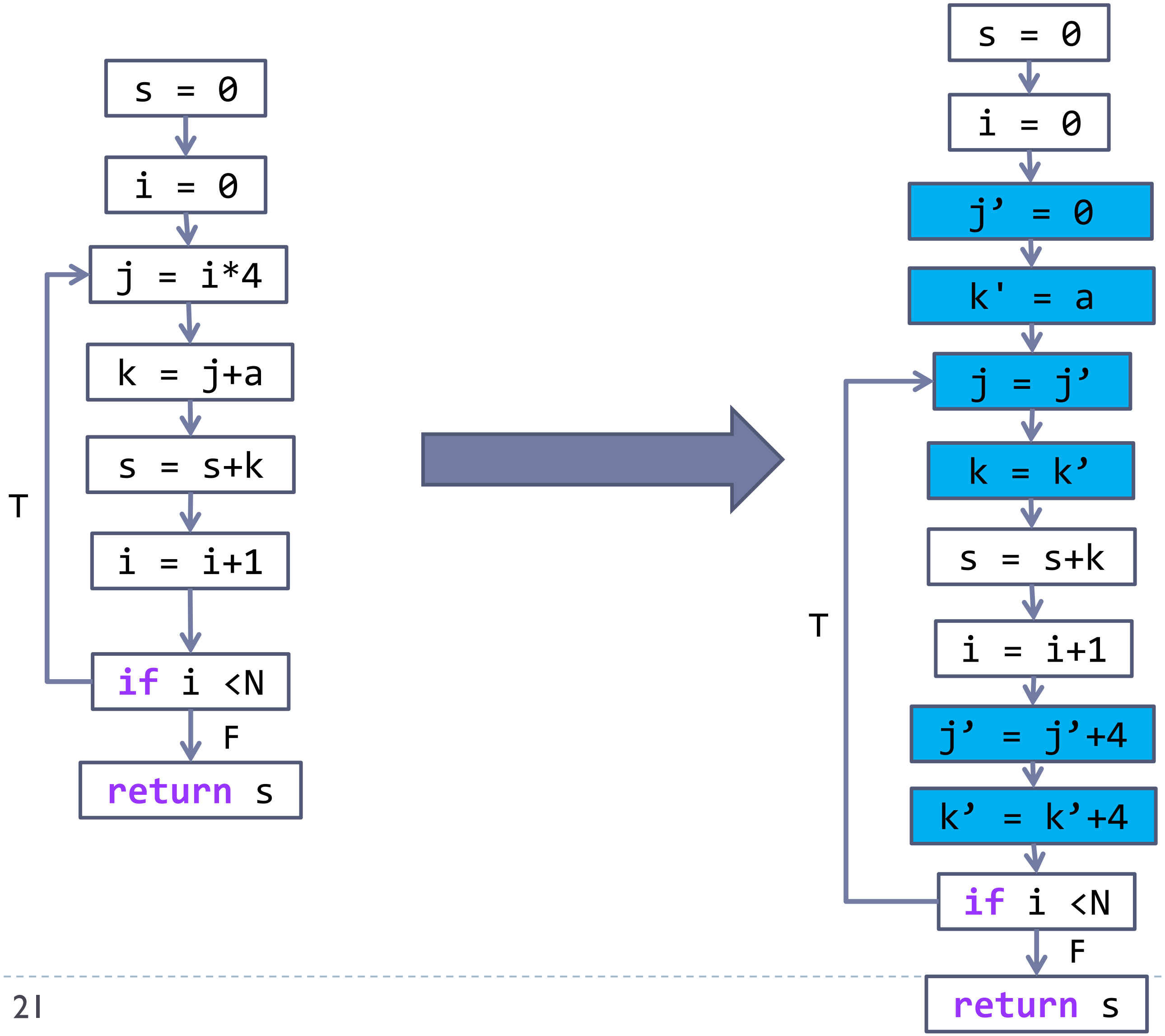
- i = i + 1: basic induction variable
- j = i * 4: derivable induction variable
- k = i * 4 + a: derivable induction variable



Strength Reduction

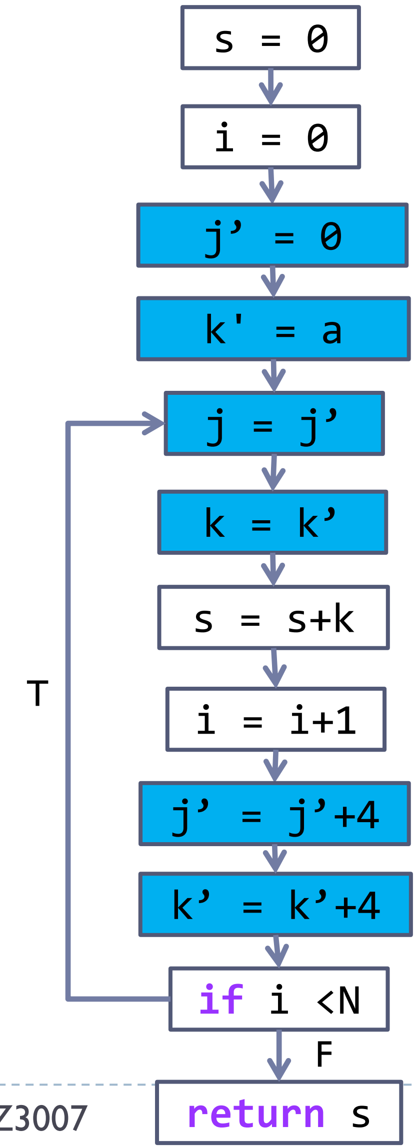
- For each derived induction variable $k = a * i + b$, make a fresh temp k'
- At the loop pre-header, initialize k' to b
- After each $i = i + c$, define $k' = k' + a * c$ (note $a * c$ can be computed in the loop preheader for only once.)
- Replace the unique assignment of k in the loop with $k = k'$

Strength Reduction Example



Optimization: Removing Useless Variables

- A variable x is useless for the loop if it is dead at all the exit nodes, and its only use is a definition of itself
- We can delete useless variables
- j' is useless and can be deleted



Optimization: Other Loop Operations

- Loop fusion: combine two loops into one
- Loop fission: split one loop into two
- Loop unrolling: make copies of loop body
- Loop interchange: change order of loop iteration variables
- Loop peeling: split the first (or last) iterations from the loop and perform them separately

Outline

- Dominance Analysis
- Loop Optimization
- **Static Single Assignment**
- Inter-Procedural Optimisations
- Appendix: Optimisations using Soot

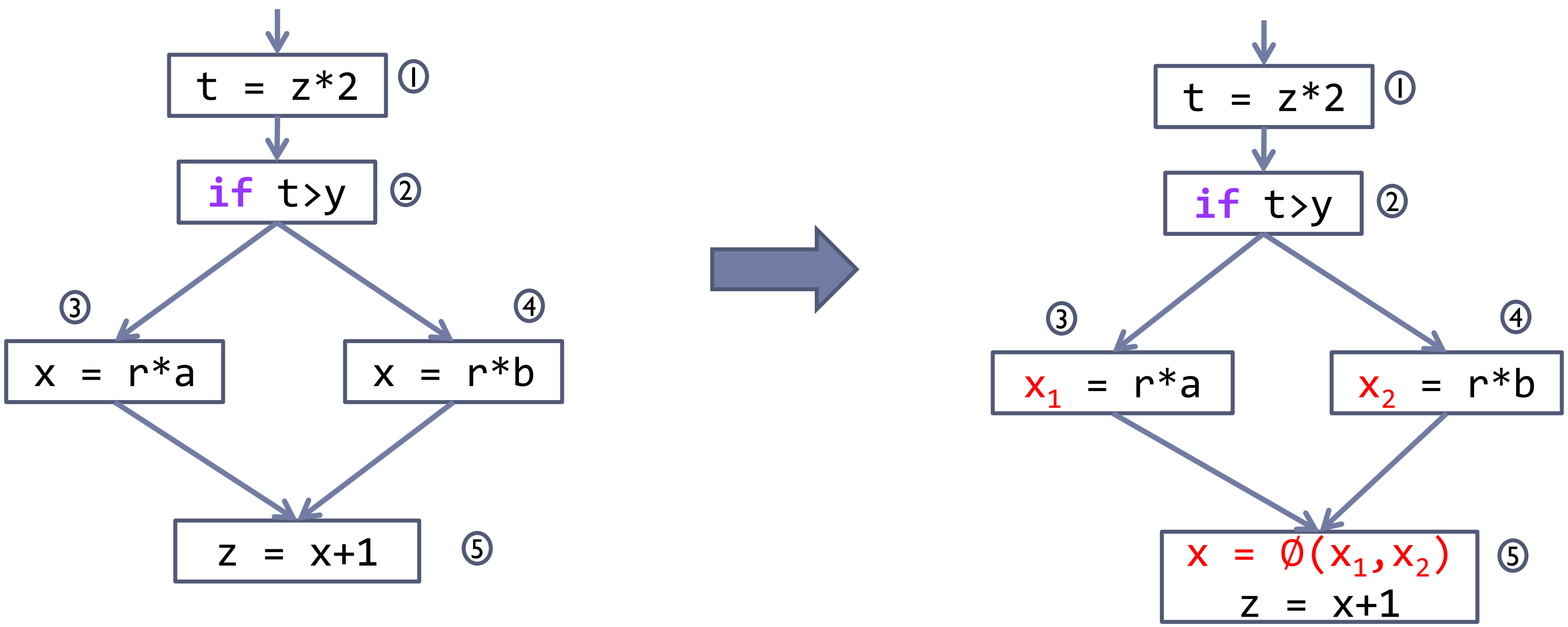
Static Single Assignment

- SSA Form.
  - Each variable has only one static definition
  - Simplify and improve the results of many optimization techniques
    - Constant propagation
    - Value range propagation
    - …
- SSA Conversion
  - Rename each definition
  - Rename all uses reached by that assignment

$x = a + b$	$x_1 = a + b$
$y = x * 2$	$y = x_1 * 2$
$x = a - b$	$x_2 = a - b$
$z = x * 3$	$z = x_2 * 3$

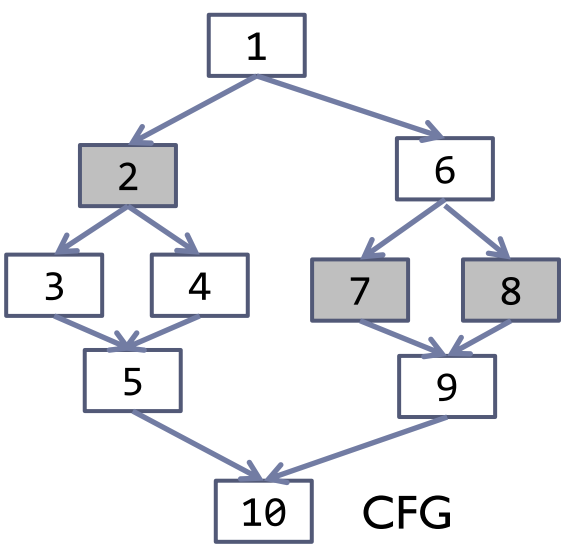
SSA Conversion with Control Flow

- Problem
  - A use may be reached by several definitions in different branches
- Merge definition
  - Introduce ∅-function to merge multiple reaching definitions

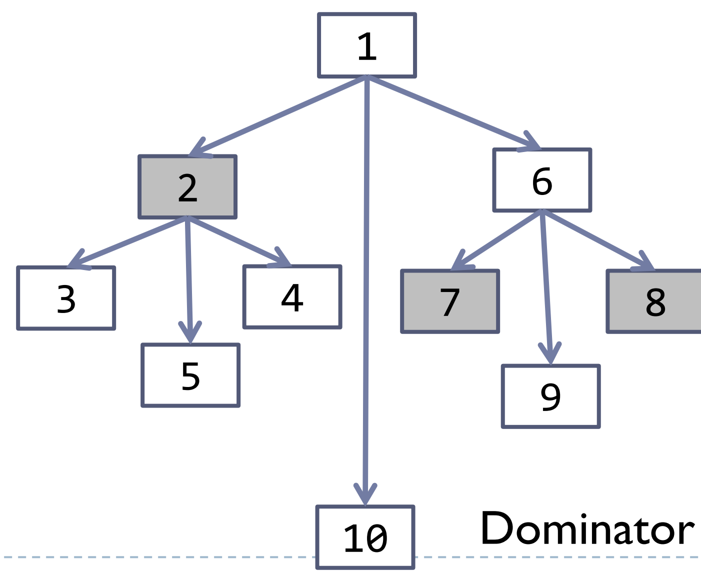


More Dominator Terminology

- Dominator Tree
  - A tree where the children of each node are those it immediately dominates
- Dominance Frontier
  - $DF(n)$ is a set of nodes d, such that
    - n dominates a predecessor of d
    - n does not strictly dominate d



CFG



Dominator Tree

DF(2) = {10}

DF(7) = {9}

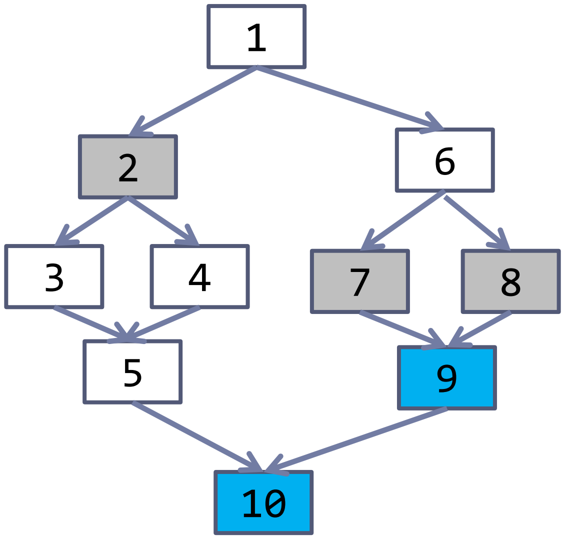
DF(8) = {9}

Inserting ∅-function

- Insights
  - If two distinct paths $x \rightarrow z$ and $y \rightarrow z$ converge at z, and nodes x and y contain definitions of variable v, then a ∅-function for v is inserted at z
- Let S be the set of CGF nodes that define variable v, then $DF_{\infty}(S)$ is the set of nodes that require ∅-functions for v.
  - $DF_1(S)$=$DF(S)$
  - $DF_{i+1}(S)$=$DF(S \cup DF_i(S))$
  - $DF_{\infty}(S)$

Example

- Insert place:
  - $DF(2) = \{10\}, DF(7) = \{9\}, DF(8) = \{9\}$
  - $DF_1(S) = \{9, 10\}$
  - $DF(9) = \{10\}, DF(10) = \emptyset$
  - $DF_2(S) = DF(\{2, 7, 8, 9, 10\}) = \{9, 10\}$
  - $DF_\infty(S)=\{9,10\}$



Outline

- Dominance Analysis
- Loop Optimization
- Static Single Assignment
- **Inter-Procedural Optimisations**
- Appendix: Optimisations using Soot

Inter-Procedural Optimisations

- Consider two inter-procedural optimisations:
  - **Inlining**: replaces a call with the body of the invoked function, which avoids overhead, but increases code size
  - **Devirtualisation**: the method invoked by an expression e.m() often depends on the runtime type of e. This is known as a *virtual call*. JVM needs to look up m on the runtime type in order to identify which method to invoke
    - Determine at compile time which method could be invoked, turning it into a *static call*
- Both optimisations need a *call graph* indicating for each call all possible call targets

Call Graph Example

```
interface Shape { double area(); }
class Rectangle implements Shape {
  // …
  public double area() { return width*height; }
}
class Circle implements Shape {
  // …
  public double area() { return Math.PI*radius*radius;}
}

class Test {
  public static void main(String[] args) {
    Shape[] shapes = { new Rectangle(), new Circle() };
    for(Shape shape : shapes) shape.area();
  }
}
```

Virtual call: could invoke either `Rectangle.area()` or `Circle.area()`; cannot be inlined

Call Graph Example

```
interface Shape { double area(); }
class Rectangle implements Shape {
  // …
  public double area() { return width*height; }
}
class Circle implements Shape {
  // …
  public double area() { return Math.PI*radius*radius;}
}

class Test {
  public static void main(String[] args) {
    Shape[] shapes = { new Rectangle() };
    for(Shape shape : shapes) shape.area();
  }
}
```

Static call: Definitely invokes `Rectangle.area()`, can be inlined

Computing Call Graphs

- A call graph is a set of *call edges* (*c*, *m*), where *c* is a *call site* (i.e. a method call) and *m* is a *call target* (i.e. a method)
- This means that, call site *c* may invoke call target *m* at runtime (one call site may have multiple call targets)
- Language features like method overriding and function pointers make call graph computation difficult; in fact, computing a *precise* call graph is impossible (undecidable)
- We can, however, compute an *overapproximate* call graph: if, at runtime, *c* may, in fact, invoke *m*, then the call graph contains the edge (*c*, *m*)
- On the other hand, the call graph may contain edges (*c*', *m*') where call site *c*' can never actually invoke *m*'; this is called a *spurious call edge*

Call Graph Algorithms

- For object-oriented programming languages, there are three popular call graph construction algorithms; all yield overapproximate call graphs:
  1. Class Hierarchy Analysis (CHA)
  2. Rapid Type Analysis (RTA)
  3. Control Flow Analysis (CFA), also known as Pointer Analysis
- CHA is the fastest of these algorithms, but it yields the least precise call graphs (many spurious edges); CFA gives the best call graphs (few spurious edges), but is quite slow in practice
- CFA is also applicable to other languages
  - A lot of research has gone into making CFA faster, and most modern compilers now use CFA-like analyses for inter-procedural optimisation

Class Hierarchy Analysis (CHA)

- The idea of CHA is very simple:

  1. For a call `e.m(...)`, determine the static type `C` of `e`
  2. Then look up method `m` in class `C` or its ancestors; this yields some method definition *md*
  3. The possible call targets of the call are *md* (unless it is abstract) and any (non-abstract) methods *md*' that override *md*

- In our earlier examples, CHA would determine that the call targets of `shape.area()` are `Rectangle.area()` and `Circle.area()`, which is imprecise for the second example Thus, CHA could not be used for inlining that call

Rapid Type Analysis (RTA)

- Note that in the second example, class `Circle` is never instantiated, so clearly `Circle.area()` can never be invoked
- RTA improves on CHA by keeping track of which classes are instantiated somewhere in the program; call these *live classes*
- If a method is neither declared in a live class nor inherited by a live class, then it clearly can never be a call target
- RTA thus can build precise call graphs for both examples

Control Flow Analysis (CFA)

- RTA is easily fooled; consider this example:

```
class Test {
  public static void main(String[] args) {
    new Circle();
    Shape[] shapes = { new Rectangle() };
    for(Shape shape : shapes) shape.area();
  }
}
```

- RTA sees that Circle is live, so it thinks `shape.area()` could invoke `Circle.area`, but this is clearly not possible
- CFA flow analysis keeps track of the possible runtime types of every variable; it can tell that elements of the shapes array can only be of type `Rectangle`, hence shape must be of type `Rectangle`, yielding a precise call graph

The End

Outline

- Dominance Analysis
- Loop Optimization
- Static Single Assignment
- Inter-Procedural Optimisations
- **Appendix: Optimisations using Soot**

Data Flow Analysis in Soot

- Data flow analysis is a key part of the Soot framework
- Intra-procedural data flow analyses can be implemented in Soot by extending class `ForwardFlowAnalysis<N, A>` or `BackwardFlowAnalysis<N, A>`, respectively
  - A forward analysis computes out(*n*) from in(*n*)
  - A backward analysis computes in(*n*) from out(*n*)
- Parameter `N` is the type of the CFG nodes (usually `Unit`), `A` is the type of the flow set (usually `ArraySparseSet`)
- The classes construct the analysis from a directed graph representation of the method body using a worklist algorithm
- Tutorial: http://www.bodden.de/2008/09/22/soot-intra/

Data Flow Analysis in Soot (2)

- Extend class `ForwardFlowAnalysis<N, A>` or `BackwardFlowAnalysis<N, A>` depending on whether a forward or backward analysis is required
- Methods to implement for forward analysis:
  - `copy`(*a*, *b*): copy flow set *a* into flow set *b*
  - `merge`(*a*, *b*, *c*): merge flow sets *a* and *b*, and store the result in *c*
  - `flowThrough`(*a*, *n*, *b*): compute out(*n*) from *a*, which is in(*n*), and store it in *b*
  - `entryInitialFlow`: return initial value for in(ENTRY)
  - `newInitialFlow`: return initial value for in(*n*) for other nodes
  - Constructor must call `doAnalysis()`
- Similarly for backward analysis, except that the roles of in(*n*) and out(*n*) and ENTRY and EXIT are reversed

Flow Set

- The flow set provides implementations of set intersection, set union, copy, etc.
  - c = a $\cap$ b where a and b are flow sets: `a.intersection(b, c)`
  - c = a $\cup$ b where a and b are flow sets: `a.union(b, c)`
  - c = a where a and c are flow sets: `a.copy(c)`
  - c = a $\cup$ {v} where a is a flow set and v is a flow item: `a.add(v)`
  - c = a \ {v} where a is a flow set: and v is a flow item: `a.remove(v)`
- There are different implementations of flow sets
  - ArraySparseSet is the simplest and is usually sufficient
- The `copy()` and `merge()` methods are implemented using the appropriate flow set operations
  - A may analysis uses set union
  - A must analysis uses set intersection

Example: Liveness

- Extend BackwardFlowAnalysis

```
Class LiveVariableAnalysis extends
BackwardFlowAnalysis<Unit, ArraySparseSet>
```

- If a node *n* has only one successor, `copy()` is used to copy in(*m*) to out(*n*) where *m* is the successor of *n*

```
void copy(Object src, Object dest) {
    FlowSet s = (FlowSet) src, d = (FlowSet) dest;
    s.copy(d);
}
```

- If a node *n* has two successors, `merge()` is used to merge in(*m*) for nodes $m \in$ succ(*n*)

```
void merge(Object src1, Object src2, Object dest) {
    // cast src1, src2 and dest to FlowSet s1, s2 and d
    s1.union(s2, d);  // may analysis
}
```

Example: Liveness (2)

- The `flowThrough()` method computes in(n) from out(n) using the kill (def) and gen (use) sets
- Soot provides a method for obtaining the def set and use set for a unit `u` and returning it as a `ValueBox`

```
void flowThrough(Object src, Object ut, Object dest) {
    // cast src and dest to FlowSet s and d as before
    Unit u = (Unit) ut;
    s.copy(d);  // copy source to destination
    for (ValueBox box : u.getDefBoxes() { // kill set
        Value v = box.getValue();
        if (v instanceof Local) d.remove(v);
    }
    for (ValueBox box : u.getUseBoxes() { // gen set
        Value v = box.getValue();
        if (v instanceof Local) d.add(v);
    }
}
```

Example: Liveness (3)

- Create Initial Sets – for Liveness these are initialised to the empty set

```
Object newInitialFlow() {
    return new ArraySparseSet();
}
Object entryInitialFlow() {
    return new ArraySparseSet();
}
```

- Implement Constructor
  - Call `doAnalysis()` to compute flow sets

```
LiveVariableAnalysis(UnitGraph g)
    super(g);
    doAnalysis();
}
```

Example: Liveness (4)

- Obtain Unit Graph for method – this is a directed graph representation of the body of the method
  - `UnitGraph g = new UnitGraph(body);`
- Create new instance of LiveVariableAnalysis
  - `LiveVariableAnalysis lv =`
  - `    new LiveVariableAnalysis(g);`
- The constructor calls `doAnalysis()` which computes the flow sets using a worklist algorithm
- Get results of in(*n*) and out(*n*) for any node (unit) *n* using
  - `lv.getFlowBefore(n);`
  - `lv.getFlowAfter(n);`
- These return SparseArraySets of the live variables before and after a node

Inter-Procedural Optimisation using Soot

- The Soot framework supports inter-procedural (whole-program) optimisation
- It can generate a call graph using CHA or more precise methods
- It also provides methods for querying the call graph, e.g.
  - `edgesOutOf(Unit)` returns an iterator over edges with a given source statement

```
void mayCall(Unit src) {
  CallGraph cg = Scene.v().getCallGraph();
  Iterator targets = new Targets(cg.edgesOutOf(src));
  // Targets adapts an iterator over edges to be
  // an iterator over the target methods of the edges
  while(targets.hasNext()) {
    SootMethod tgt = (SootMethod) targets.next();
    System.out.println(src + " maycall " + tgt);
  }
}
```