Compiler Techniques

6. Optimisation

Ta N. B. Duong

Overview

- Before producing the final executable code, many code generators perform optimisations to make the code faster or more compact
- Roughly speaking, there are two kinds of optimisations:
  - Platform-independent optimisations: they apply for any target platform (processor and operating system)
  - Platform-specific optimisations: they only apply for one particular target platform
- Here, we will only consider some simple platform-independent optimisations
- Such optimisations are usually implemented on intermediate representations like Jimple

Static Analysis

- Optimisations need to be *behaviour-preserving*: the optimised program should run faster/take less space, but otherwise should behave exactly the same as the original program
- Thus, an optimising compiler has to reason statically (i.e. without running the program) about *all* possible behaviours of a program; this is known as **static analysis**
- It follows from some basic results of computability theory that it is impossible to statically predict precisely the possible behaviours of a program
- So optimisers have to be conservative: only apply an optimisation if we can be absolutely certain that it is behaviour-preserving

Examples of Optimisations

- Some examples of optimisations performed by modern compilers:
  1. *Register allocation*: already discussed
  2. *Common subexpression elimination*: if an expression has already been evaluated, reuse its previously computed value
  3. *Dead code detection*: remove code that cannot be executed
  4. *Loop invariant code motion*: move computation out of a loop body
  5. *Loop fusion*: combine two loops into one
  6. *Reduction in strength*: replace slow operations with faster ones
  7. *Bounds check elimination*: in Java, if an array index can be shown to be in range, we do not need to check it at run-time
  8. *Function inlining*: replace a call to a function by its function body
  9. *Devirtualisation*: turn a virtual method call into a static one
  10. …

Intra-Procedural vs. Inter-Procedural

- Many optimisations only concern the code within a single method or function; they are called *intra-procedural*
- More ambitious optimisations may try to optimise several methods/functions at once; they are called *inter-procedural* or *whole-program* optimisations
- There is a similar distinction between intra-procedural static analysis and inter-procedural (whole-program) static analysis
- Whole-program optimisations have the potential to deliver greater performance improvements, but they are more difficult to apply, and it is harder to reason about their safety

Outline of this chapter

- For intra-procedural optimisations, a *control flow graph* is used to represent potential execution paths within a method/function
- For inter-procedural optimisations, a *procedure call graph* is used to represent potential execution paths between the methods/functions of a program
- We mainly consider intra-procedural optimisations and the analyses on which they are based

1. Intra-Procedural Analysis
2. Intra-Procedural Optimisation using Soot
3. Inter-Procedural Analysis and Optimisation

1. Intra-Procedural Analysis

Control Flow Graphs

- A control flow graph (CFG) is a representation of all instructions in a method and the possible flow of execution through these instructions; it is used by many optimisations to reason about the possible behaviours of a method
- The nodes of the CFG represent the instructions of the method; there is an edge from node n_1 to n_2 if n_2 could be executed immediately after n_1
- Usually, we add distinguished ENTRY and EXIT nodes representing the beginning and end of the method execution
- In Jimple, most instructions only have a single successor, except for conditional jumps, which have two (we ignore exceptions)

Control Flow Graph Example

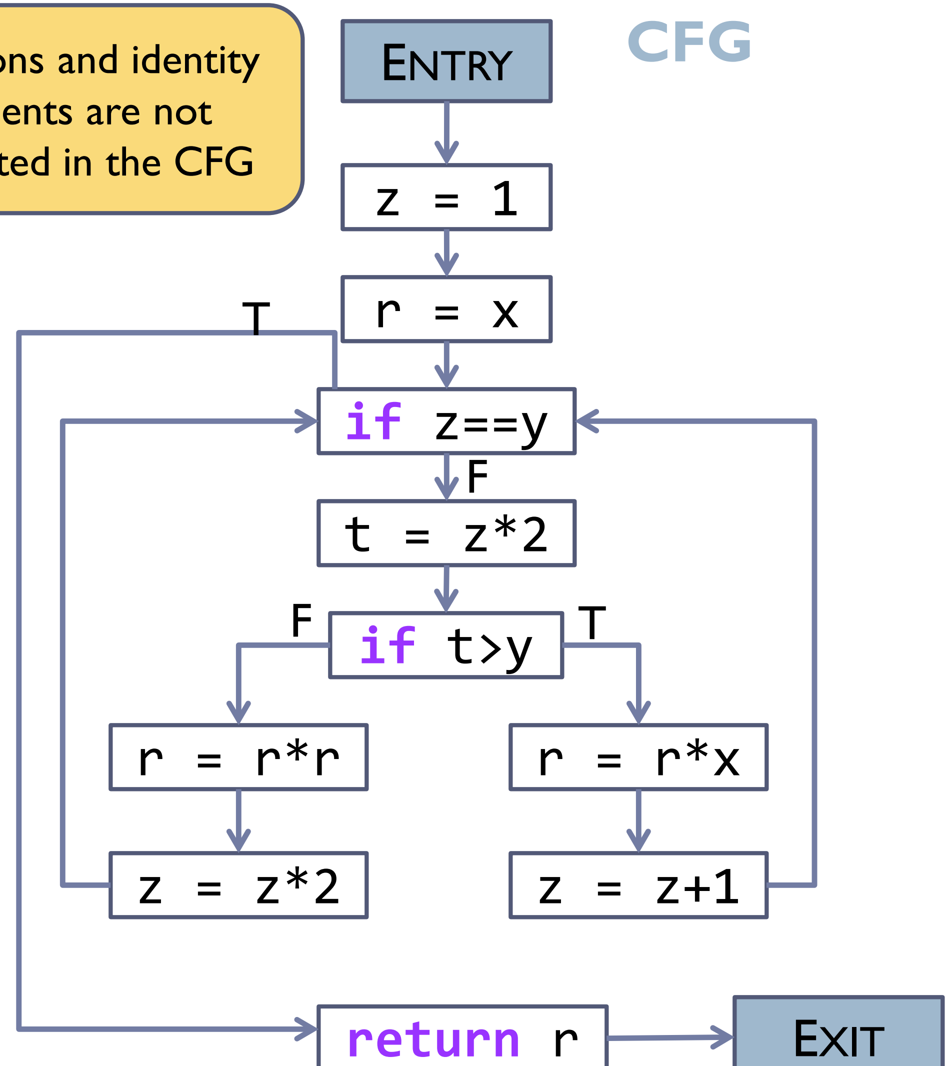
Jimple code

```
 int r, t, x, y, z;
 x:= @parameter0;
 y:= @parameter1;
 z = 1;
 r = x;
l1: if z==y goto l3;
 t = z*2;
 if t>y goto l2;
 r = r*r;
 z = z*2;
 goto l1;
l2:
 r = r*x;
 z = z+1;
 goto l1;
l3: return r;
```

declarations and identity statements are not represented in the CFG

goto is not explicitly represented in the CFG

CFG

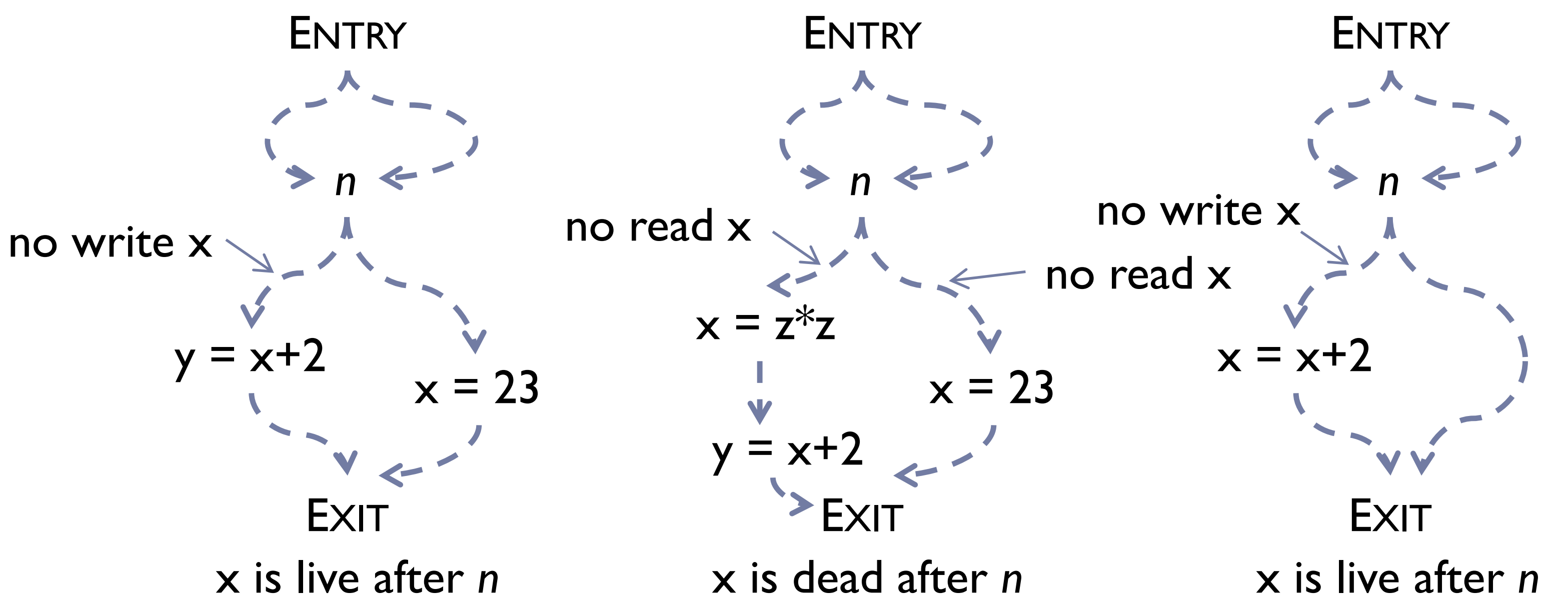


Data Flow Analysis

- Many analyses employed by compiler optimisations are *data flow analyses*: they gather information about the values computed (and assigned to variables) at different points in the program
- Data flow analyses work on the CFG, considering all possible paths from the ENTRY node to a certain node, or from that node to the EXIT node
- A typical example of a data flow analysis is *liveness* analysis, determining which local variables are live at what point in the program (see our previous discussion of register allocation)

Liveness analysis

- Recall: a local variable is live at a program point if its value may be read before it is (re-)assigned
- In terms of CFG: a variable x is live *after* a CFG node *n* if there is a path *p* from *n* to EXIT such that there is a node *r* on *p* that reads x, and *r* is not preceded on *p* by any node that writes x



Liveness analysis (2)

- A variable can be live *before* or *after* a node.
- For a node n, we write $\mathrm{in}_L(n)$ for the set of variables that are live before n, and $\mathrm{out}_L(n)$ for the set of variables that are live after n; these sets are known as *flow sets*
- The goal of liveness analysis is to compute $\mathrm{in}_L(n)$ and $\mathrm{out}_L(n)$ for every CFG node n

Transfer Functions

- Note that if we already know $out_L(n)$ for some node, $in_L(n)$ is easy to compute: note that a variable x is live before n if
  1. either n reads x,
  2. or x is live after n and n does not write x
- By convention, the set of (local) variables a node n reads is written use(n), and the set of variables it writes is def(n)
- So we have

$$in_L(n) = out_L(n) \setminus def(n) \cup use(n)$$

- This equation is known as the *transfer function* for $in_L(n)$
- A data flow analysis where $in_L(n)$ is computed from $out_L(n)$ is called a *backward* flow analysis – an analysis where $out_L(n)$ is computed from $in_L(n)$ is called a *forward* flow analysis

Transfer Functions (2)

So how do we get $out_L(n)$?

- There are two cases:
  1. Node *n* is the EXIT node

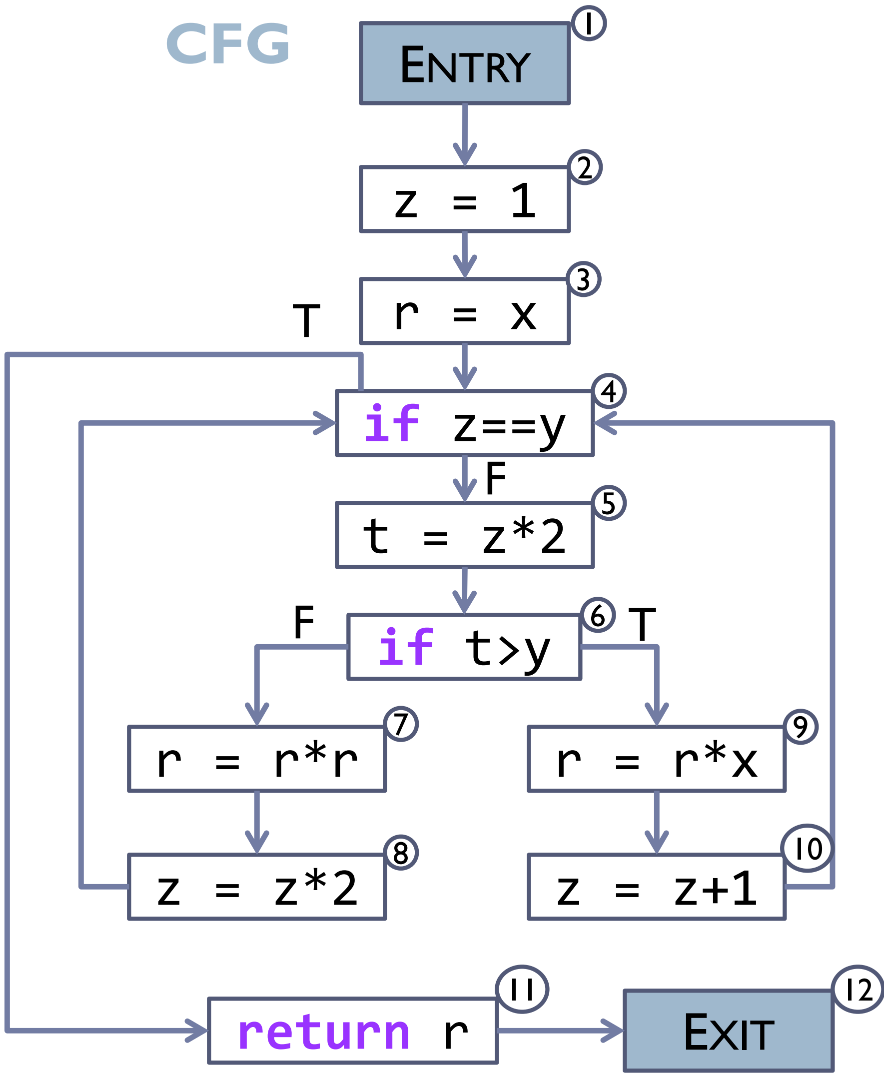
     $out_L(n) = \varnothing$: no variable is live at the end of a method
  2. Node *n* has at least one successor node:

     Let $succ(n)$ be the set of successor nodes of *n*

     Note that a variable x is live after *n* if it is live before any successor of n

     Thus we define $out_L(n) = \bigcup \{ in_L(m) \mid m \in succ(n) \}$
- A data flow analysis where union is used to combine results from successor nodes is called a *may* analysis – if intersection is used, the analysis is a *must* analysis

Transfer Functions Example



Transfer Functions

$in_L(1) = out_L(1) \setminus \emptyset \cup \emptyset = out_L(1)$

$in_L(2) = out_L(2) \setminus \{z\} \cup \emptyset = out_L(2) \setminus \{z\}$

$in_L(3) = out_L(3) \setminus \{r\} \cup \{x\}$

$in_L(4) = out_L(4) \setminus \emptyset \cup \{y, z\} = out_L(4) \cup \{y, z\}$

$in_L(5) = out_L(5) \setminus \{t\} \cup \{z\}$

$in_L(6) = out_L(6) \setminus \emptyset \cup \{t,y\} = out_L(6) \cup \{t, y\}$

$in_L(7) = out_L(7) \setminus \{r\} \cup \{r\} = out_L(7) \cup \{r\}$

$in_L(8) = out_L(8) \setminus \{z\} \cup \{z\} = out_L(8) \cup \{z\}$

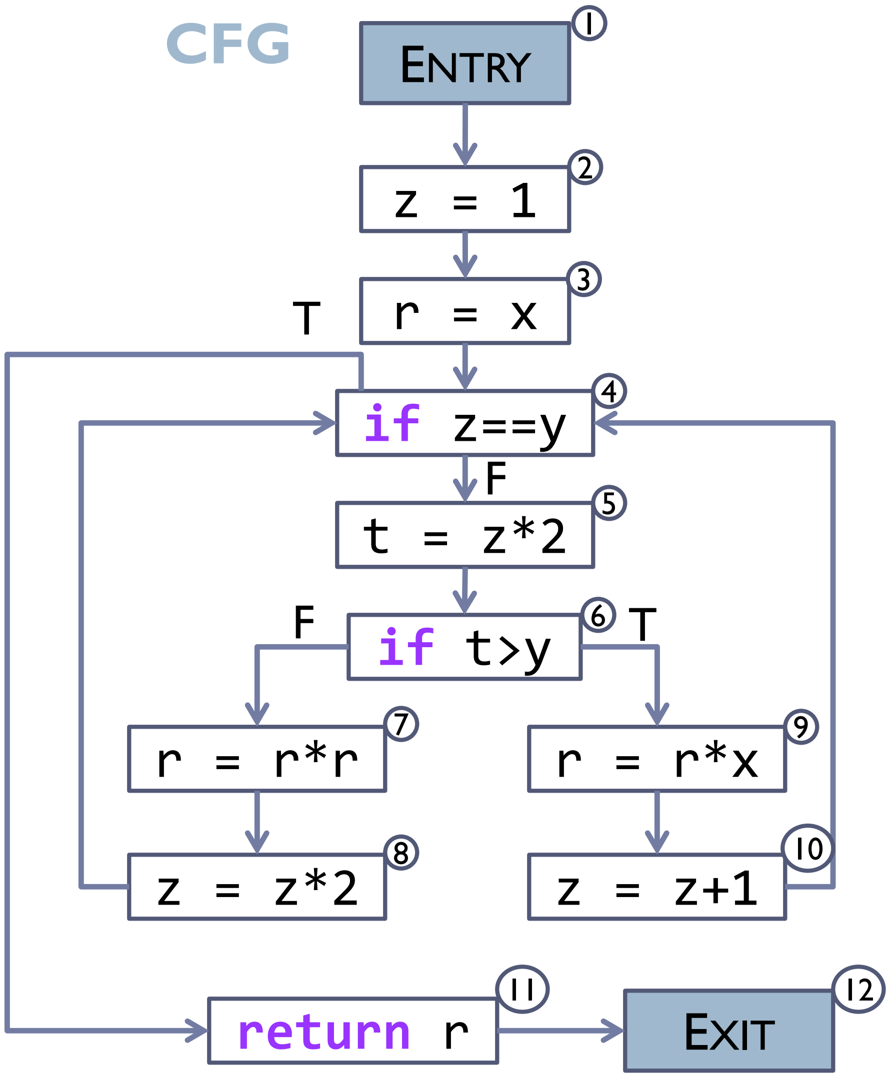
$in_L(9) = out_L(9) \setminus \{r\} \cup \{r, x\} = out_L(9) \cup \{r, x\}$

$in_L(10) = out_L(10) \setminus \{z\} \cup \{z\} = out_L(10) \cup \{z\}$

$in_L(11) = out_L(11) \setminus \emptyset \cup \{r\} = out_L(11) \cup \{r\}$

$in_L(12) = out_L(12) \setminus \emptyset \cup \emptyset = out_L(12)$

Transfer Functions Example (2)



Transfer Functions

$out_L(1) = in_L(2)$

$out_L(2) = in_L(3)$

$out_L(3) = in_L(4)$

$out_L(4) = in_L(11) \cup in_L(5)$

$out_L(5) = in_L(6)$

$out_L(6) = in_L(7) \cup in_L(9)$

$out_L(7) = in_L(8)$

$out_L(8) = in_L(4)$

$out_L(9) = in_L(10)$

$out_L(10) = in_L(4)$

$out_L(11) = in_L(12)$

$out_L(12) = \varnothing$

Computing Transfer Functions

- The system of equations for the transfer functions cannot be solved directly: the definitions are circular!

$$
\begin{aligned}
\mathbf{in_L(4)} \quad &= out_L(4) \cup \{ y, z \} \\
&= in_L(11) \cup in_L(5) \cup \{ y, z \} \\
&= in_L(11) \cup out_L(5) \setminus \{ t \} \cup \{ y, z \} \\
&= in_L(11) \cup in_L(6) \setminus \{ t \} \cup \{ y, z \} \\
&= in_L(11) \cup (out_L(6) \cup \{ t, y \}) \setminus \{ t \} \cup \{ y, z \} \\
&= in_L(11) \cup (in_L(7) \cup in_L(9) \cup \{ t, y \}) \setminus \{ t \} \cup \{ y, z \} \\
&= in_L(11) \cup (out_L(7) \cup in_L(9) \cup \{ r, t, y \}) \setminus \{ t \} \cup \{ y, z \} \\
&= in_L(11) \cup (in_L(8) \cup in_L(9) \cup \{ r, t, y \}) \setminus \{ t \} \cup \{ y, z \} \\
&= in_L(11) \cup (out_L(8) \cup in_L(9) \cup \{ r, t, y, z \}) \setminus \{ t \} \cup \{ y, z \} \\
&= in_L(11) \cup (\mathbf{in_L(4)} \cup in_L(9) \cup \{ r, t, y, z \}) \setminus \{ t \} \cup \{ y, z \} \\
&= \ldots
\end{aligned}
$$

Iterative Solution

- However, the equation system can be solved by iteration:
  1. For all nodes n, set $in_L(n) = out_L(n) = \varnothing$
  2. For every node n, recompute $out_L(n)$ based on the values we have computed in the previous iteration and then recompute $in_L(n)$ from $out_L(n)$
  3. Keep doing step 2 until the values do not change any further
- This algorithm terminates and computes a solution for the equation system (proof of this result is outside the scope of this course)
- As we will see later, this method can be used for many other data flow analyses

Simplifying Transfer Equations

- To make it easier to solve the equations, we substitute away the $out_L(n)$ equations, so we only have to solve for $in_L(n)$

$$
\begin{aligned}
in_L(1) &= in_L(2) \\
in_L(2) &= in_L(3) \setminus \{z\} \\
in_L(3) &= in_L(4) \setminus \{r\} \cup \{x\} \\
in_L(4) &= in_L(11) \cup in_L(5) \cup \{y, z\} \\
in_L(5) &= in_L(6) \setminus \{t\} \cup \{z\} \\
in_L(6) &= in_L(7) \cup in_L(9) \cup \{t,y\} \\
in_L(7) &= in_L(8) \cup \{r\} \\
in_L(8) &= in_L(4) \cup \{z\} \\
in_L(9) &= in_L(10) \cup \{r, x\} \\
in_L(10) &= in_L(4) \cup \{z\} \\
in_L(11) &= in_L(12) \cup \{r\} \\
in_L(12) &= \varnothing
\end{aligned}
$$

Iterative Solution Example

$in_L(1) = in_L(2)$

$in_L(2) = in_L(3) \setminus \{z\}$

$in_L(3) = in_L(4) \setminus \{r\} \cup \{x\}$

$in_L(4) = in_L(11) \cup in_L(5) \cup \{y, z\}$

$in_L(5) = in_L(6) \setminus \{t\} \cup \{z\}$

$in_L(6) = in_L(7) \cup in_L(9) \cup \{t, y\}$

$in_L(7) = in_L(8) \cup \{r\}$

$in_L(8) = in_L(4) \cup \{z\}$

$in_L(9) = in_L(10) \cup \{r, x\}$

$in_L(10) = in_L(4) \cup \{z\}$

$in_L(11) = in_L(12) \cup \{r\}$

$in_L(12) = \varnothing$

	0	1	2	3	4	5	6
$in_L(1)$	$\varnothing$						
$in_L(2)$	$\varnothing$						
$in_L(3)$	$\varnothing$						
$in_L(4)$	$\varnothing$						
$in_L(5)$	$\varnothing$						
$in_L(6)$	$\varnothing$						
$in_L(7)$	$\varnothing$						
$in_L(8)$	$\varnothing$						
$in_L(9)$	$\varnothing$						
$in_L(10)$	$\varnothing$						
$in_L(11)$	$\varnothing$						
$in_L(12)$	$\varnothing$						

Iterative Solution Example

$in_L(1) = in_L(2)$

$in_L(2) = in_L(3) \setminus \{z\}$

$in_L(3) = in_L(4) \setminus \{r\} \cup \{x\}$

$in_L(4) = in_L(11) \cup in_L(5) \cup \{y, z\}$

$in_L(5) = in_L(6) \setminus \{t\} \cup \{z\}$

$in_L(6) = in_L(7) \cup in_L(9) \cup \{t, y\}$

$in_L(7) = in_L(8) \cup \{r\}$

$in_L(8) = in_L(4) \cup \{z\}$

$in_L(9) = in_L(10) \cup \{r, x\}$

$in_L(10) = in_L(4) \cup \{z\}$

$in_L(11) = in_L(12) \cup \{r\}$

$in_L(12) = \varnothing$

	0	1	2	3	4	5	6
$in_L(1)$	$\varnothing$	$\varnothing$					
$in_L(2)$	$\varnothing$	$\varnothing$					
$in_L(3)$	$\varnothing$	x					
$in_L(4)$	$\varnothing$	y,z					
$in_L(5)$	$\varnothing$	z					
$in_L(6)$	$\varnothing$	t,y					
$in_L(7)$	$\varnothing$	r					
$in_L(8)$	$\varnothing$	z					
$in_L(9)$	$\varnothing$	r,x					
$in_L(10)$	$\varnothing$	z					
$in_L(11)$	$\varnothing$	r					
$in_L(12)$	$\varnothing$	$\varnothing$					

Iterative Solution Example

$in_L(1) = in_L(2)$

$in_L(2) = in_L(3) \setminus \{z\}$

$in_L(3) = in_L(4) \setminus \{r\} \cup \{x\}$

$in_L(4) = in_L(11) \cup in_L(5) \cup \{y, z\}$

$in_L(5) = in_L(6) \setminus \{t\} \cup \{z\}$

$in_L(6) = in_L(7) \cup in_L(9) \cup \{t,y\}$

$in_L(7) = in_L(8) \cup \{r\}$

$in_L(8) = in_L(4) \cup \{z\}$

$in_L(9) = in_L(10) \cup \{r, x\}$

$in_L(10) = in_L(4) \cup \{z\}$

$in_L(11) = in_L(12) \cup \{r\}$

$in_L(12) = \emptyset$

	0	1	2	3	4	5	6
$in_L(1)$	$\emptyset$	$\emptyset$	$\emptyset$				
$in_L(2)$	$\emptyset$	$\emptyset$	x				
$in_L(3)$	$\emptyset$	x	x,y,z				
$in_L(4)$	$\emptyset$	y,z	r,y,z				
$in_L(5)$	$\emptyset$	z	y,z				
$in_L(6)$	$\emptyset$	t,y	r,t,x,y				
$in_L(7)$	$\emptyset$	r	r,z				
$in_L(8)$	$\emptyset$	z	y,z				
$in_L(9)$	$\emptyset$	r,x	r,x,z				
$in_L(10)$	$\emptyset$	z	y,z				
$in_L(11)$	$\emptyset$	r	r				
$in_L(12)$	$\emptyset$	$\emptyset$	$\emptyset$				

Iterative Solution Example

$in_L(1) = in_L(2)$

$in_L(2) = in_L(3) \setminus \{z\}$

$in_L(3) = in_L(4) \setminus \{r\} \cup \{x\}$

$in_L(4) = in_L(11) \cup in_L(5) \cup \{y, z\}$

$in_L(5) = in_L(6) \setminus \{t\} \cup \{z\}$

$in_L(6) = in_L(7) \cup in_L(9) \cup \{t,y\}$

$in_L(7) = in_L(8) \cup \{r\}$

$in_L(8) = in_L(4) \cup \{z\}$

$in_L(9) = in_L(10) \cup \{r, x\}$

$in_L(10) = in_L(4) \cup \{z\}$

$in_L(11) = in_L(12) \cup \{r\}$

$in_L(12) = \varnothing$

	0	1	2	3	4	5	6
$in_L(1)$	$\varnothing$	$\varnothing$	$\varnothing$	x			
$in_L(2)$	$\varnothing$	$\varnothing$	x	x,y			
$in_L(3)$	$\varnothing$	x	x,y,z	x,y,z			
$in_L(4)$	$\varnothing$	y,z	r,y,z	r,y,z			
$in_L(5)$	$\varnothing$	z	y,z	r,x,y,z			
$in_L(6)$	$\varnothing$	t,y	r,t,x,y	r,t,x,y,z			
$in_L(7)$	$\varnothing$	r	r,z	r,y,z			
$in_L(8)$	$\varnothing$	z	y,z	r,y,z			
$in_L(9)$	$\varnothing$	r,x	r,x,z	r,x,y,z			
$in_L(10)$	$\varnothing$	z	y,z	r,y,z			
$in_L(11)$	$\varnothing$	r	r	r			
$in_L(12)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$			

Iterative Solution Example

$in_L(1) = in_L(2)$
$in_L(2) = in_L(3) \setminus \{z\}$
$in_L(3) = in_L(4) \setminus \{r\} \cup \{x\}$
$in_L(4) = in_L(11) \cup in_L(5) \cup \{y, z\}$
$in_L(5) = in_L(6) \setminus \{t\} \cup \{z\}$
$in_L(6) = in_L(7) \cup in_L(9) \cup \{t, y\}$
$in_L(7) = in_L(8) \cup \{r\}$
$in_L(8) = in_L(4) \cup \{z\}$
$in_L(9) = in_L(10) \cup \{r, x\}$
$in_L(10) = in_L(4) \cup \{z\}$
$in_L(11) = in_L(12) \cup \{r\}$
$in_L(12) = \varnothing$

	0	1	2	3	4	5	6
$in_L(1)$	$\varnothing$	$\varnothing$	$\varnothing$	x	x,y		
$in_L(2)$	$\varnothing$	$\varnothing$	x	x,y	x,y		
$in_L(3)$	$\varnothing$	x	x,y,z	x,y,z	x,y,z		
$in_L(4)$	$\varnothing$	y,z	r,y,z	r,y,z	r,x,y,z		
$in_L(5)$	$\varnothing$	z	y,z	r,x,y,z	r,x,y,z		
$in_L(6)$	$\varnothing$	t,y	r,t,x,y	r,t,x,y,z	r,t,x,y,z		
$in_L(7)$	$\varnothing$	r	r,z	r,y,z	r,y,z		
$in_L(8)$	$\varnothing$	z	y,z	r,y,z	r,y,z		
$in_L(9)$	$\varnothing$	r,x	r,x,z	r,x,y,z	r,x,y,z		
$in_L(10)$	$\varnothing$	z	y,z	r,y,z	r,y,z		
$in_L(11)$	$\varnothing$	r	r	r	r		
$in_L(12)$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$		

Iterative Solution Example

$in_L(1) = in_L(2)$

$in_L(2) = in_L(3) \setminus \{z\}$

$in_L(3) = in_L(4) \setminus \{r\} \cup \{x\}$

$in_L(4) = in_L(11) \cup in_L(5) \cup \{y, z\}$

$in_L(5) = in_L(6) \setminus \{t\} \cup \{z\}$

$in_L(6) = in_L(7) \cup in_L(9) \cup \{t, y\}$

$in_L(7) = in_L(8) \cup \{r\}$

$in_L(8) = in_L(4) \cup \{z\}$

$in_L(9) = in_L(10) \cup \{r, x\}$

$in_L(10) = in_L(4) \cup \{z\}$

$in_L(11) = in_L(12) \cup \{r\}$

$in_L(12) = \varnothing$

	0	1	2	3	4	5	6
$in_L(1)$	∅	∅	∅	x	x,y	x,y	
$in_L(2)$	∅	∅	x	x,y	x,y	x,y	
$in_L(3)$	∅	x	x,y,z	x,y,z	x,y,z	x,y,z	
$in_L(4)$	∅	y,z	r,y,z	r,y,z	r,x,y,z	r,x,y,z	
$in_L(5)$	∅	z	y,z	r,x,y,z	r,x,y,z	r,x,y,z	
$in_L(6)$	∅	t,y	r,t,x,y	r,t,x,y,z	r,t,x,y,z	r,t,x,y,z	
$in_L(7)$	∅	r	r,z	r,y,z	r,y,z	r,y,z	
$in_L(8)$	∅	z	y,z	r,y,z	r,y,z	r,x,y,z	
$in_L(9)$	∅	r,x	r,x,z	r,x,y,z	r,x,y,z	r,x,y,z	
$in_L(10)$	∅	z	y,z	r,y,z	r,y,z	r,x,y,z	
$in_L(11)$	∅	r	r	r	r	r	
$in_L(12)$	∅	∅	∅	∅	∅	∅	

Iterative Solution Example

No further changes after iteration 6

$in_L(1) = in_L(2)$

$in_L(2) = in_L(3) \setminus \{z\}$

$in_L(3) = in_L(4) \setminus \{r\} \cup \{x\}$

$in_L(4) = in_L(11) \cup in_L(5) \cup \{y, z\}$

$in_L(5) = in_L(6) \setminus \{t\} \cup \{z\}$

$in_L(6) = in_L(7) \cup in_L(9) \cup \{t, y\}$

$in_L(7) = in_L(8) \cup \{r\}$

$in_L(8) = in_L(4) \cup \{z\}$

$in_L(9) = in_L(10) \cup \{r, x\}$

$in_L(10) = in_L(4) \cup \{z\}$

$in_L(11) = in_L(12) \cup \{r\}$

$in_L(12) = \varnothing$

	0	1	2	3	4	5	6
$in_L(1)$	∅	∅	∅	x	x,y	x,y	x,y
$in_L(2)$	∅	∅	x	x,y	x,y	x,y	x,y
$in_L(3)$	∅	x	x,y,z	x,y,z	x,y,z	x,y,z	x,y,z
$in_L(4)$	∅	y,z	r,y,z	r,y,z	r,x,y,z	r,x,y,z	r,x,y,z
$in_L(5)$	∅	z	y,z	r,x,y,z	r,x,y,z	r,x,y,z	r,x,y,z
$in_L(6)$	∅	t,y	r,t,x,y	r,t,x,y,z	r,t,x,y,z	r,t,x,y,z	r,t,x,y,z
$in_L(7)$	∅	r	r,z	r,y,z	r,y,z	r,y,z	r,x,y,z
$in_L(8)$	∅	z	y,z	r,y,z	r,y,z	r,x,y,z	r,x,y,z
$in_L(9)$	∅	r,x	r,x,z	r,x,y,z	r,x,y,z	r,x,y,z	r,x,y,z
$in_L(10)$	∅	z	y,z	r,y,z	r,y,z	r,x,y,z	r,x,y,z
$in_L(11)$	∅	r	r	r	r	r	r
$in_L(12)$	∅	∅	∅	∅	∅	∅	∅

Common Subexpression Elimination

- Useful optimisation: identify expressions that have been computed before, reuse previously computed result

```
 int r, t, x, y, z;
 x:= @param0; y:= @param1;
 z = 1;
 r = x;
l1: if z==y goto l3;
 t = z*2;
 if t>y goto l2;
 r = r*r;
 z = z*2;
 goto l1;
l2:
 r = r*x;
 z = z+1;
 goto l1;
l3: return r;
```

e.g. this value:
has been computed here:
So we can replace the assignment with z = t

Correctness Conditions

- There are two conditions that need to hold before we can eliminate a common subexpression:
  1. The expression must have been computed previously on *every* possible execution path, not just on one
  2. None of the variables involved in computing the expression may have been updated in the meantime
- Hence, we cannot eliminate subexpressions in this example:

```
x = y + z
y = y + 1
r = y + z
```

- Nor in this:

```
    if z > 0 goto l
    x = y + z
l:  r = y + z
```

Available Expressions

- We formalise these conditions in terms of the CFG via the concept of an *available expression*:

  An expression e is available before a CFG node *n* if

  1. e is computed on every path from ENTRY to *n*, and
  2. no variable in e is overwritten between the computation of e and *n*

- We can compute the set $\text{in}_A(n)$ of available expressions before a node *n* using a data flow analysis
- The set of possible expressions is infinite, but we only need to consider expressions that actually occur in the method (which is a finite set)
- For simplicity, we only consider expressions computed from local variables using arithmetic and logical operators

Available Expressions Analysis

- Clearly, an expression e is available after a node *n* if
  1. Node *n* computes e, or
  2. e is available before *n* and *n* does not write to any variable in e
- Let us write vars(e) for the variables in e, and comp(*n*) for the expressions computed by *n* (either empty or a singleton set)
- Recall def(*n*) is the set of variables node *n* writes
- Then we have

$$\mathrm{out}_A(n) = \mathrm{in}_A(n) \setminus \{e \mid \mathrm{vars}(e) \cap \mathrm{def}(n) \neq \emptyset\} \cup \mathrm{comp}\,(n)$$

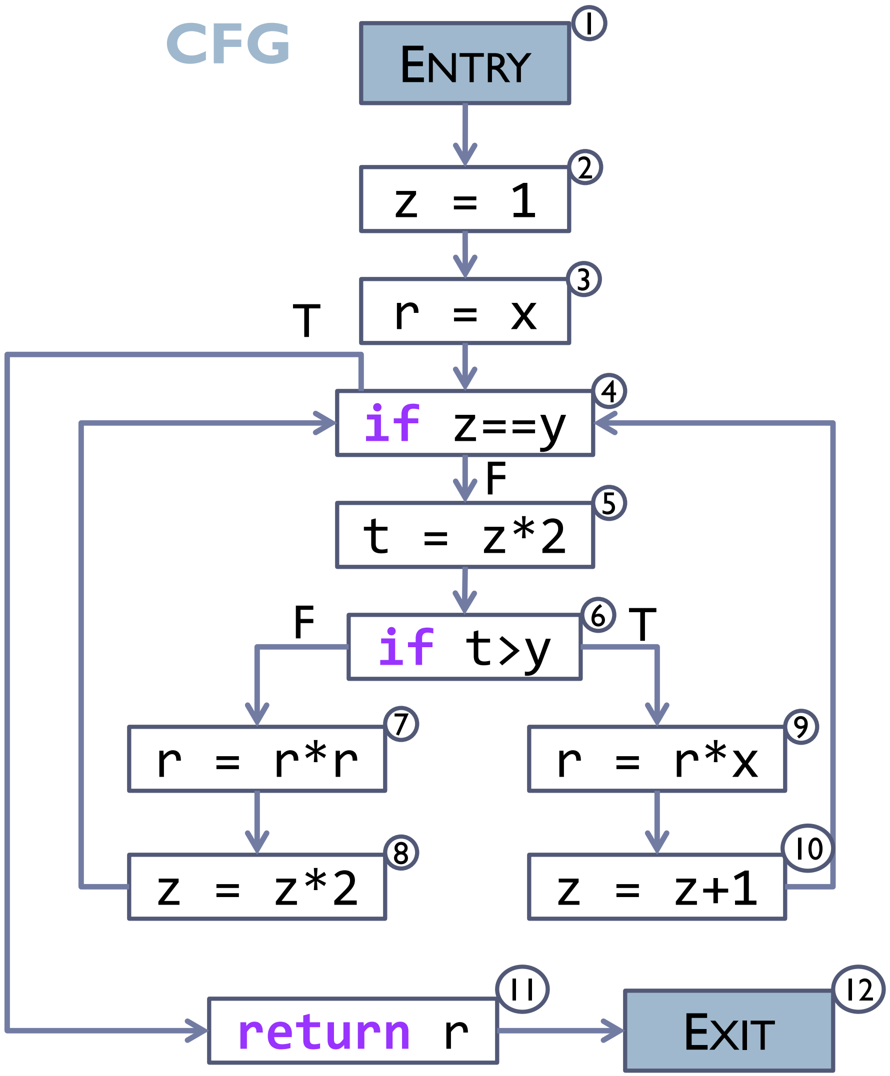
- An expression is available before *n* if it is available after every predecessor *m* of *n*; no expression is available before ENTRY:

$$\mathrm{in}_A(n) = \bigcap \{\, \mathrm{out}_A(m) \mid m \in \mathrm{pred}(n) \,\}$$

$$\mathrm{in}_A(\text{ENTRY}) = \emptyset$$

Available Expressions Example

CFG



Transfer Functions

$out_A(1) = in_A(1)$

$out_A(2) = in_A(2) \setminus \{ z*2, z+1 \}$

$out_A(3) = in_A(3) \setminus \{ r*r, r*x \}$

$out_A(4) = in_A(4)$

$out_A(5) = in_A(5) \cup \{ z*2 \}$

$out_A(6) = in_A(6)$

$out_A(7) = in_A(7) \setminus \{ r*r, r*x \} \cup \{\}$

$out_A(8) = in_A(8) \setminus \{ z*2, z+1 \} \cup \{\}$

$out_A(9) = in_A(9) \setminus \{ r*r, r*x \} \cup \{\}$

$out_A(10) = in_A(10) \setminus \{ z*2, z+1 \} \cup \{\}$

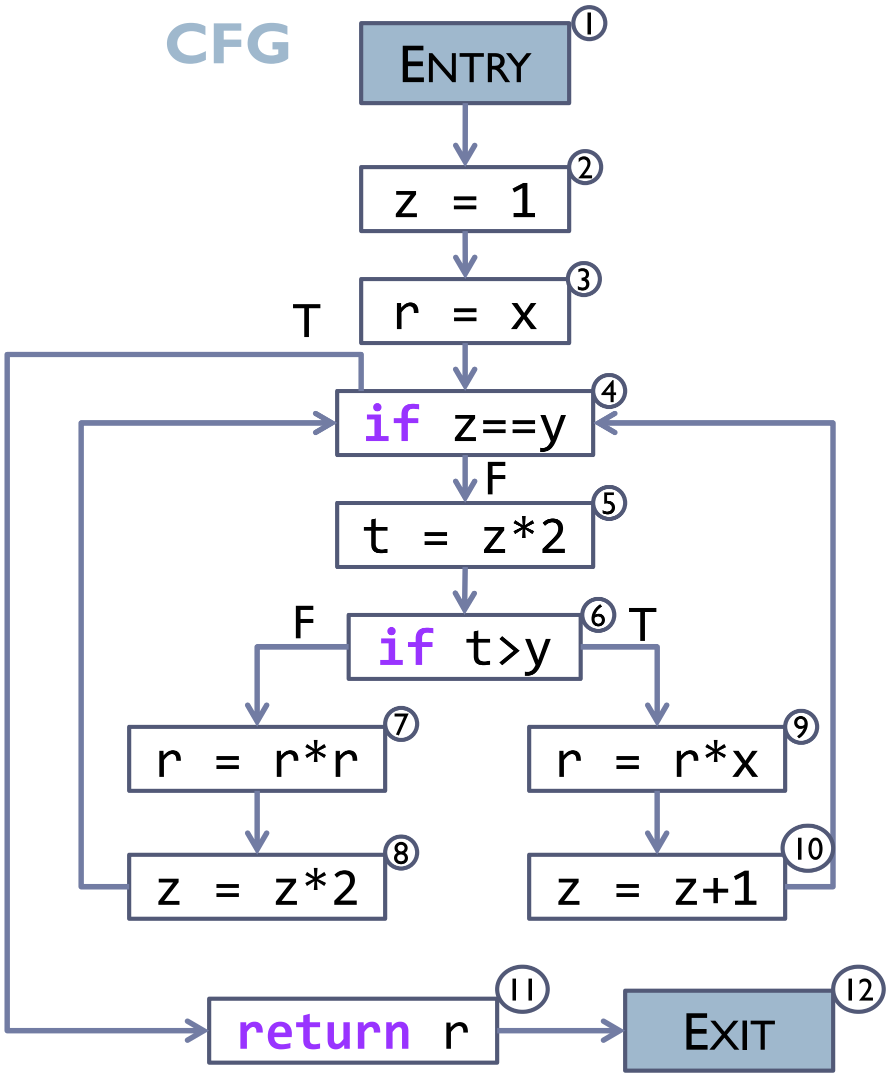
$out_A(11) = in_A(11)$

$out_A(12) = in_A(12)$

r*r should not be included in comp(7) as the value of r has been changed immediately by the statement in node 7. The same can be said for node 8, 9, and 10.

Available Expressions Example (2)

CFG



Transfer Functions

$in_A(1) = \varnothing$

$in_A(2) = out_A(1)$

$in_A(3) = out_A(2)$

$in_A(4) = out_A(3) \cap out_A(8) \cap out_A(10)$

$in_A(5) = out_A(4)$

$in_A(6) = out_A(5)$

$in_A(7) = out_A(6)$

$in_A(8) = out_A(7)$

$in_A(9) = out_A(6)$

$in_A(10) = out_A(9)$

$in_A(11) = out_A(4)$

$in_A(12) = out_A(11)$

Available Expressions Example (3)

To make it easier to solve the equations, we substitute away in_A

$out_A(1) = \varnothing$
$out_A(2) = out_A(1) \setminus \{ z*2, z+1 \}$
$out_A(3) = out_A(2) \setminus \{ r*r, r*x \}$
$out_A(4) = out_A(3) \cap out_A(8) \cap out_A(10)$
$out_A(5) = out_A(4) \cup \{ z*2 \}$
$out_A(6) = out_A(5)$
$out_A(7) = out_A(6) \setminus \{ r*r, r*x \}$
$out_A(8) = out_A(7) \setminus \{ z*2, z+1 \}$
$out_A(9) = out_A(6) \setminus \{ r*r, r*x \}$
$out_A(10) = out_A(9) \setminus \{ z*2, z+1 \}$
$out_A(11) = out_A(4)$
$out_A(12) = out_A(11)$

- Again, we can solve iteratively
- As we use intersection to compute $in_A(n)$, we initialise $out_A(n) = in_A(n) = U$ for all nodes n except ENTRY, *where* U is the set of *all* expressions in the method
- $out_A(\text{ENTRY}) = in_A(\text{ENTRY}) = \varnothing$

Other Data Flow Analyses

- Liveness is a *backward-may* analysis: whether a variable is live before a node depends on whether it is live after the node (but not vice versa), and a variable is live after a node if it is live before *any* successor node
- Available Expressions is a *forward-must* analysis: whether an expression is available after a node depends on whether it is available before the node (but not vice versa), and an expression is available before a node if it is available after *every* predecessor node
- There are also *backward-must* analyses (e.g. very busy expressions) and *forward-may* analyses (e.g. reaching definitions)

General Characteristics

- Clearly, Liveness and Available Expressions (and many other data flow analyses) share some basic similarities:
  - They compute flow sets in(n) and out(n) for before and after every node n in the CFG
  - The analysis results are always sets of *data flow facts*: in the case of Liveness, sets of variables; in the case of Available Expressions, sets of expressions
  - For every node, there is a transfer function that computes either in(n) from out(n) or out(n) from in(n), depending on whether the analysis is backward or forward
  - There is also a merge function that computes either out(n) from in(m) for the successors m of n, or in(n) from out(m) for the predecessors m of n, again depending on whether the analysis is backward or forward

Gen and Kill Sets

- Compare the transfer functions for Liveness and Available Expressions:

$$in_L(n) = out_L(n) \setminus def(n) \cup use(n)$$

$$out_A(n) = in_A(n) \setminus \{e \mid vars(e) \cap def(n) \neq \emptyset\} \cup comp\,(n)$$

- They are both of the shape

$$after(n) = before(n) \setminus kill(n) \cup gen(n)$$

where

  - after/before are in/out depending on whether it is a backward or forward analysis
  - kill(n) is the set of analysis facts *killed* (i.e. removed from the solution) by node n
  - gen(n) is the set of new analysis facts *generated* by node n

Kildall-style Data Flow Analysis

- In general, a *Kildall-style* (named after Gary Kildall) *forward analysis* is described by:
  1. A set of *U* possible data flow facts
  2. Gen and kill sets for every node *n*
  3. A join operator, which is either set union or intersection, for computing in(*n*) from out(*m*) for every predecessor *m* of *n*
  4. A set of data flow facts in(ENTRY) defining the result of the analysis before ENTRY
- A *Kildall-style backward analysis* is similar, but with in(*n*) and out(*n*) reversed
  1. A join operator, which is either set union or intersection, for computing out(*n*) from in(*m*) for every successor *m* of *n*
  2. A set of data flow facts out(EXIT) defining the result of the analysis after EXIT

Iterative Data Flow Analysis (Forward)

- Any Kildall-style forward analysis can be computed by the iterative approach introduced earlier:
  1. Initialise in(n): the value of in(ENTRY) is given by the analysis description; for all other nodes, if the join operator is union, then in(n) = $\varnothing$, otherwise in(n) = U
  2. Now compute out(n) from in(n) for every node n
  3. Recompute in(n) based on the previous value of out(m) for every predecessor m of n
  4. Repeat steps 2 and 3 until in(n) and out(n) no longer change
- To make it easier to solve the equations, we can substitute away in(n) and solve only for out(n)
  - We initialise out(ENTRY) = in(ENTRY) and for all other nodes, if the join operator is union, then out(n) = $\varnothing$, otherwise out(n) = U

Worklist Algorithm (Forward)

- For forward analyses, out(*n*) can only change if in(*n*) changes, and in(*n*) only if out(*m*) changes for some predecessor *m* of *n*
- We can avoid unnecessary recomputation by keeping a *worklist* of nodes for which in(*n*) has changed:

worklist = [all nodes]
while worklist != empty **do**
    m = removeFirst(worklist)
    recompute out(*m*)
    if out(*m*) has changed **then**
        for each successor *n* of *m*
            compute in(*n*)
            if in(*n*) has changed **then**
                put *n* into worklist (if not already in worklist)

Worklist Algorithm (Forward): Example

- An example of available expression analysis using the worklist algorithm.
- We assume that the worklist is initialized to the set of all nodes, in increasing order of node number.
- We initialize the worklist to contain all nodes in the CFG to ensure that each node is evaluated at least once
- For all nodes, $out_A(n)$ and $in_A(n)$ are initialized to U, the set of all expressions in the method, except $in_A(1) = \varnothing$.
- In our example, $U = \{z*2, r*r, r*x, z+1\}$.

Example

worklist = [all nodes]

while worklist != empty do

  m = removeFirst(worklist)

  recompute $out_A(m)$

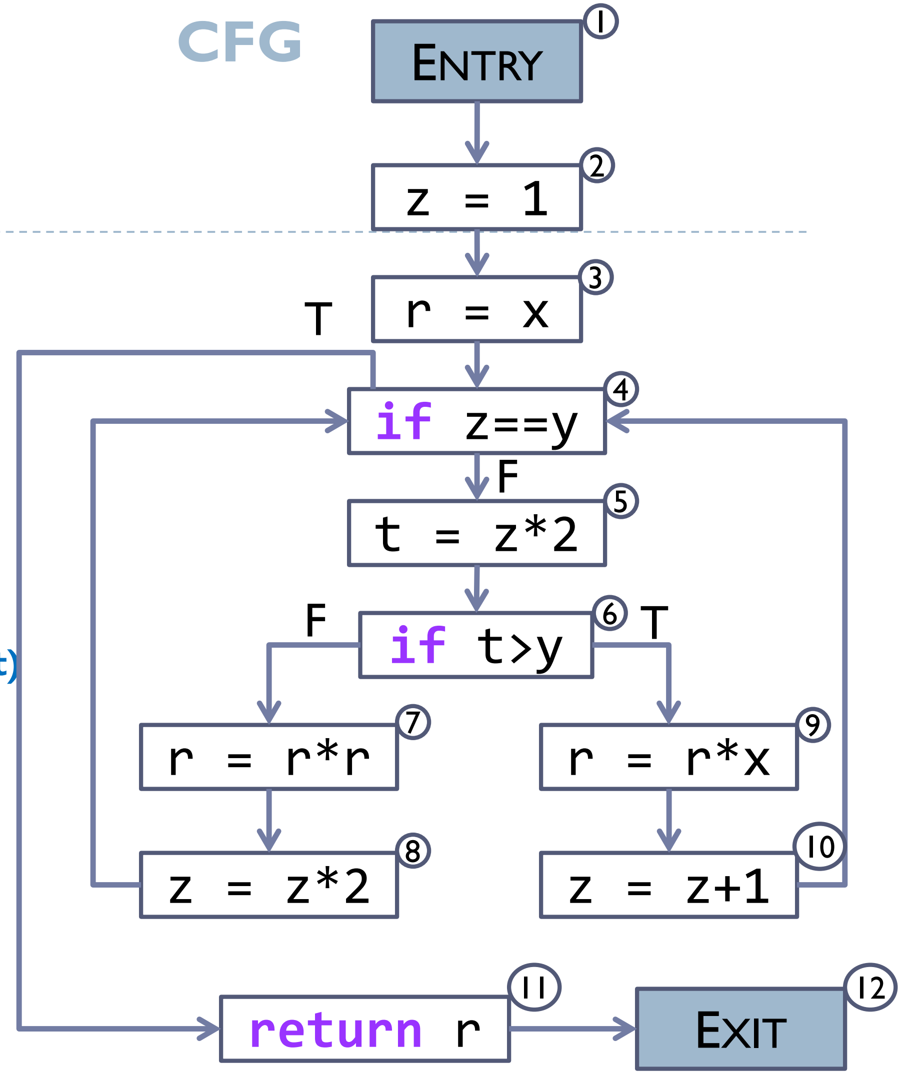
  if $out_A(m)$ has changed then

    for each successor *n* of *m*

      compute $in_A(n)$

      if $in_A(n)$ has changed then put *n* into worklist (if not already in worklist)

CFG



Worklist	**$out_A(m)$ &**	**$in_A(n)$**
1, …, 12	$out_A(1) = \varnothing$	$in_A(2) = \varnothing$
2, …, 12	$out_A(2) = \varnothing$	$in_A(3) = \varnothing$
3, …, 12	$out_A(3) = \varnothing$	$in_A(4) = \varnothing$
4, …, 12	$out_A(4) = \varnothing$	$in_A(5) = \varnothing$
		$in_A(11) = \varnothing$
5, …, 12	$out_A(5) = \{z*2\}$	$in_A(6) = \{z*2\}$
6, …, 12	$out_A(6) = \{z*2\}$	$in_A(7) = \{z*2\}$
		$in_A(9) = \{z*2\}$

Example

worklist = [all nodes]

while worklist != empty do

    m = removeFirst(worklist)

    recompute $out_A(m)$

    if $out_A(m)$ has changed then

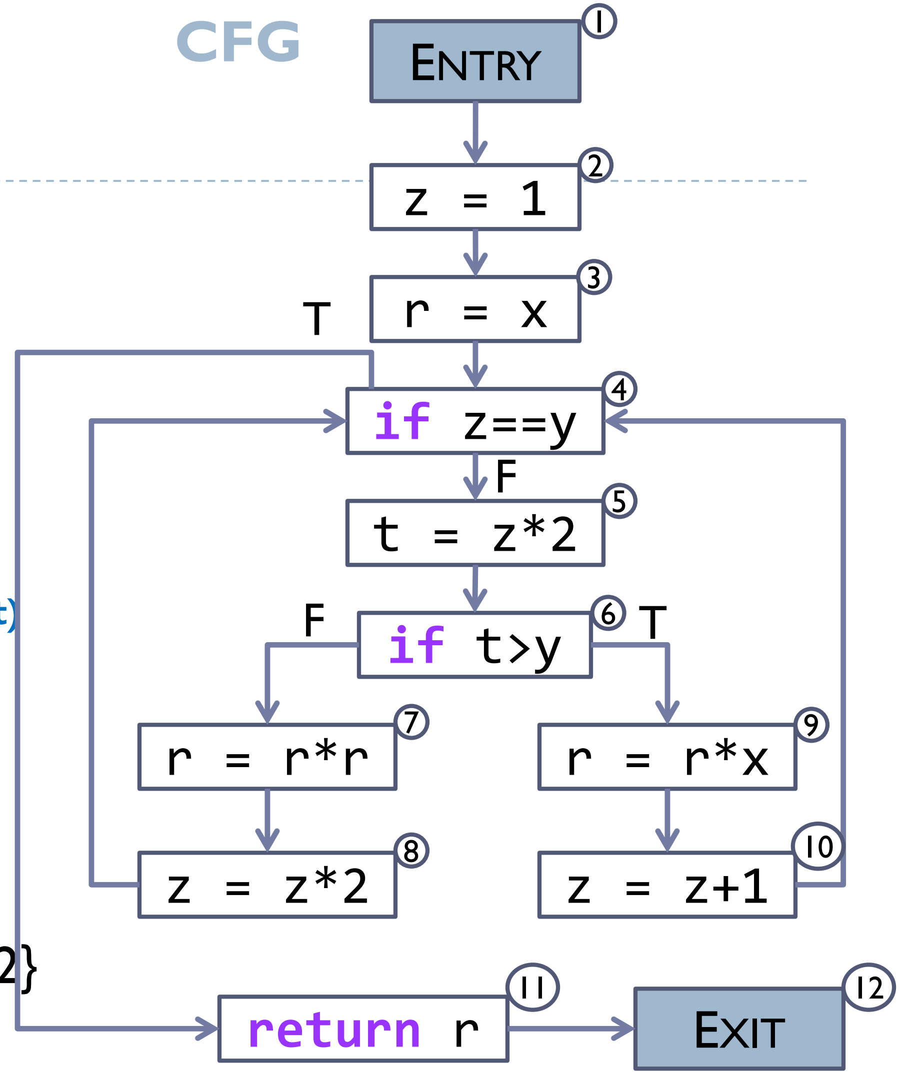
        for each successor n of m

            compute $in_A(n)$

            if $in_A(n)$ has changed then put n into worklist

                (if not already in worklist)

CFG



Worklist	**$out_A(m)$ &**	**$in_A(n)$**
7, …, 12	$out_A(7) = \{z*2\}$	$in_A(8) = \{z*2\}$
8, …, 12	$out_A(8) = \varnothing$	
	$in_A(4) = \varnothing$ (no change)	
9, …, 12	$out_A(9) = \{z*2\}$	$in_A(10) = \{z*2\}$
10, …, 12	$out_A(10) = \varnothing$	
	$in_A(4) = \varnothing$ (no change)	
11, 12	$out_A(11) = \varnothing$	$in_A(12) = \varnothing$
12	$out_A(12) = \varnothing$	

Iterative Data Flow Analysis (Backward)

- Similarly, any Kildall-style backward analysis can be computed by the following iterative approach:
  1. Initialise out(n): the value of out(EXIT) is given by the analysis description; for all other nodes, if the join operator is union, then out(n) = ∅, otherwise out(n) = U
  2. Now compute in(n) from out(n) for every node n
  3. Recompute out(n) based on the previous value for in(m) for every successor m of n
  4. Repeat steps 2 and 3 until out(n) and in(n) no longer change
- To make it easier to solve the equations, we can substitute away out(n) and solve only for in(n)
  - We initialise in(EXIT) = out(EXIT) and for all other nodes, if the join operator is union, then in(n) = ∅, otherwise in(n) = U

Worklist Algorithm (Backward)

- For backward analyses, in(*n*) can only change if out(*n*) changes, and out(*n*) only if in(*m*) changes for some successor *m* of *n*
- We can avoid unnecessary recomputation by keeping a *worklist* of nodes for which out(*n*) has changed:

worklist = [all nodes]
while worklist != empty **do**
    m = removeFirst(worklist)
    recompute in(*m*)
    if in(*m*) has changed **then**
        for each predecessor *n* of *m*
            compute out(*n*)
            if out(*n*) has changed **then**
                put *n* into worklist (if not already in worklist)

Worklist Algorithm Initialisation

- Forward Analysis
  - For all nodes, if the join operator is union, then out(n) = in(n) = $\emptyset$, otherwise U, except in(ENTRY) which is defined by the analysis
  - We initialise the worklist to contain all nodes in the CFG to ensure that each node is evaluated at least once
  - The order in which the worklist is initialised is important:
    - As far as possible the nodes should be ordered so that a node is only evaluated after all its predecessors have updated their values (but this can be difficult when the CFG has cycles)
    - A good order can generally be obtained by computing a depth first numbering with the ENTRY node as root
- Backward Analysis
  - This is similar except the roles of in(n) and out(n), ENTRY and EXIT, predecessor and successor are reversed

Conflicts between Optimisations

- Sometimes different optimisations may be in conflict with each other
- For instance, common subexpression elimination increases the live range of local variables (i.e. the number of nodes where they are live)
- This makes register allocation harder, so the performance gain of avoiding recomputation may be lost because more memory accesses are needed
- Deciding which optimisations to apply and when to apply them is difficult, and there is no general recipe

3. Inter-Procedural Analysis and Optimisation

Inter-Procedural Optimisations

- Finally, we consider two inter-procedural optimisations:
  - **Inlining**: Function calls incur overhead due to parameter passing and pipeline stalls; inlining replaces a call with the body of the invoked function, which avoids overhead, but increases code size
  - **Devirtualisation**: In Java, the method invoked by an expression `e.m()` often depends on the runtime type of `e`: if `e` has type C at compile time, then the object it evaluates to at runtime could have a more specific type D, so the JVM needs to look up `m` on the runtime type D in order to work out which method to invoke
    - This is known as a *virtual call* – the aim of this optimisation is to 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 (2)

```
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

Appendix: Intra and Inter-Procedural Optimisation 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)`
  - $c = a \setminus \{v\}$ 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);
  }
}
```