Three-address code (1)

Example (4, revisited)

This example will be used as the running example in the ensuing slides

The intermediate representation on the right corresponds to the code below

```
fun int f(int n)
[
  var int r = 1;
  while (n > 0)
  [
    r = r * n;
    n = n - 1;
  ]
  ^ r
]
```

	$t_0 \leftarrow$ i_value 1	
	@$r \leftarrow$ i_lstore t_0	`# r = 1`
l_0:	$t_1 \leftarrow$ i_aload @n	
	$t_2 \leftarrow$ i_value 0	
	$t_3 \leftarrow$ i_lt t_2, t_1	`# 0 < n?`
	cjump t_3, l_1, l_2	
l_1:	$t_4 \leftarrow$ i_lload @r	
	$t_5 \leftarrow$ i_aload @n	
	$t_6 \leftarrow$ i_mul t_4, t_5	`# r * n`
	@$r \leftarrow$ i_lstore t_6	
	$t_7 \leftarrow$ i_aload @n	
	$t_8 \leftarrow$ i_value 1	
	$t_9 \leftarrow$ i_sub t_7, t_8	`# n - 1`
	@$n \leftarrow$ i_astore t_9	
	jump l_0	
l_2:	$t_{10} \leftarrow$ i_lload @r	
	i_return t_{10}	

Three-address code (2)

Remarks

The intermediate representation is not streamlined

- The number of temporaries used is unnecessarily high

  Temporary location t_0 only appears on the first two lines of Example (4, revisited) and could be reused

- It contains redundant operations

  There are two instructions loading the same value to temporaries t_5 and t_7

Basic blocks (1)

A basic block is a maximal sequence of instructions which is always executed from the first instruction to the last

- The first instruction in a function or a procedure is the start of the first basic block
- A label is the start of a new basic block
- Jump, conditional jump and return instructions end the current basic block
- The last instruction in a function or a procedure is the last instruction of the last basic block

Basic blocks (2)

Example (5)

The intermediate representation of Example (4, revisited) contains the following 4 basic blocks

B_1: $t_0 \leftarrow$ i_value 1
$@r \leftarrow$ i_lstore t_0

B_2: l_0: $t_1 \leftarrow$ i_aload $@n$
$t_2 \leftarrow$ i_value 0
$t_3 \leftarrow$ i_lt t_2, t_1
cjump t_3, l_1, l_2

B_3: l_1: $t_4 \leftarrow$ i_lload $@r$
$t_5 \leftarrow$ i_aload $@n$
$t_6 \leftarrow$ i_mul t_4, t_5
$@r \leftarrow$ i_lstore t_6
$t_7 \leftarrow$ i_aload $@n$
$t_8 \leftarrow$ i_value 1
$t_9 \leftarrow$ i_sub t_7, t_8
$@n \leftarrow$ i_astore t_9
jump l_0

B_4: l_2: $t_{10} \leftarrow$ i_lload $@r$
i_return t_{10}

Control flow graph (1)

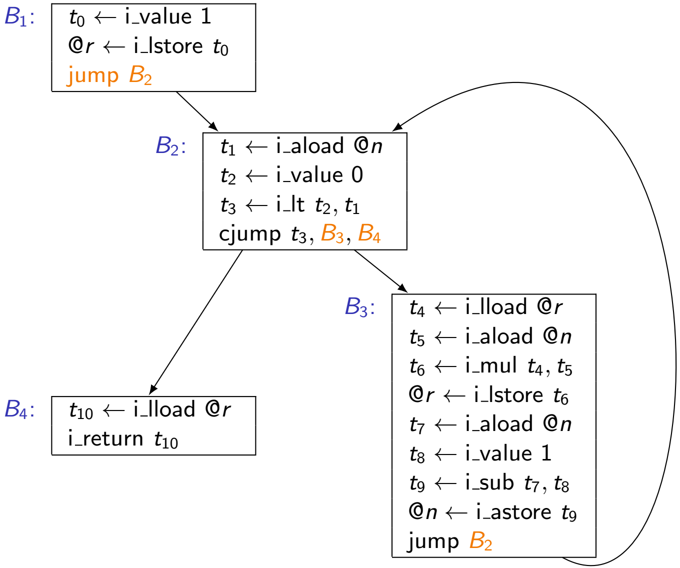
The control flow graph represents the way control flows among the basic blocks of a function or a procedure

The control flow graph is a directed graph whose nodes are the basic blocks and whose edges correspond to the control transitions between blocks

- The graph contains an edge from node B_i to node B_j if and only if the execution of basic block B_i may be (immediately) followed by that of basic block B_j

Control flow graph (2)

Example (6)



Control flow graph (3)

Remarks

The targets of jumps are basic blocks, instead of labels

If controls falls through from a basic block into another, then a jump to the latter is added at the end of the former (as was done with block B_1)

It may be useful to add an artificial initial block, with no incident edges and only one outgoing edge to the first block, and an artificial final block, with no outgoing edges and only one incident edge coming from the last block

Notice that in languages where a function may return from any point in its body, like in C, there may be more than one "last" block

Control flow graph (4)

Through the analysis of the control flow graph, dead code may be identified and removed, and blocks may be reordered to obtain “better” code

Example (7)

With the blocks reordered as shown on the right, only one jump is executed in each iteration of the loop

Since B_2 follows B_3, no jump between them is needed

A new jump is required, to skip over the loop body, from B_1 to B_2

B_1: $t_0 \leftarrow$ i_value 1
  @r $\leftarrow$ i_lstore t_0
  jump B_2
B_3: $t_4 \leftarrow$ i_lload @r
  $t_5 \leftarrow$ i_aload @n
  $t_6 \leftarrow$ i_mul t_4, t_5
  @r $\leftarrow$ i_lstore t_6
  $t_7 \leftarrow$ i_aload @n
  $t_8 \leftarrow$ i_value 1
  $t_9 \leftarrow$ i_sub t_7, t_8
  @n $\leftarrow$ i_astore t_9
  jump B_2
B_2: $t_1 \leftarrow$ i_aload @n
  $t_2 \leftarrow$ i_value 0
  $t_3 \leftarrow$ i_lt t_2, t_1
  cjump t_3, B_3, B_4
B_4: $t_{10} \leftarrow$ i_lload @r
  i_return t_{10}

Traces (1)

A trace is a sequence of (distinct) basic blocks that may be executed in sequence

Equivalently, a trace may be defined as an acyclic path in the control flow graph

Example (8)

Sequence $B_1 B_2 B_4$ is a trace for the intermediate representation of Example (5)

The only jump needed in the intermediate code of the trace is marked below

B_1
B_2
cjump B_3, B_4
B_4

Traces (2)

Processors' jump and branch instructions are expensive

Traces are used as the base for code generation, and help reducing and simplifying the flow of control code

If code is generated by following a trace, jumps from the end of one block to the first instruction in the next block do not have to be generated

Since conditional jumps in real machines only have one label, the generation of code for traces where the false label of a cjump follows the cjump gives rise to simpler code

Traces (3)

Trace coverage

A set of traces covers an intermediate representation if every basic block appears in one and only one trace in the set

The generation of code for a function or a procedure follows some set of traces that covers its intermediate representation

Traces (4)

Algorithm for generating a set of traces covering the IR

$U \leftarrow \{$ all basic blocks $\}$
$S \leftarrow \emptyset$ # set of traces
while $U \neq \emptyset$ **do**
  $T \leftarrow \lambda$ # empty trace
  $B \leftarrow$ one block from U
  while $B \in U$ **do**
    $T \leftarrow T \parallel B$ # append B to trace T
    $U \leftarrow U - \{B\}$
    if B has a successor $C \in U$ **then**
      $B \leftarrow C$
  $S \leftarrow S \cup \{T\}$

Traces (5)

Example (9)

The following sets of traces cover the intermediate representation from Example (5)

Set 1	Set 2	Set 3
$B_1 B_2 B_4$	$B_1 B_2 B_3$	B_1
B_3	B_4	$B_3 B_2 B_4$

(These are not the only sets of traces that cover that IR)

Traces (6)

Example (9, cont)

The previous sets of traces with the jump instructions needed explicitly shown

Set 1	Set 2	Set 3
B_1	B_1	B_1
B_2	B_2	jump B_2
cjump B_3, B_4	cjump B_3, B_4	
B_4	B_3	B_3
	jump B_2	B_2
B_3		cjump B_3, B_4
jump B_2	B_4	B_4

The traces from Set 3 are the basis for the basic block ordering obtained in Example (7)

Sample IR (1)

Using the IR on the next slide

1. Identify the basic blocks
2. Draw the control flow graph
3. Define 3 sets of traces covering the IR
4. For each of the sets defined, determine the number of jumps (conditional and unconditional) needed if code is generated in the order induced by the traces, and how many of those jumps will reside in a loop

Sample IR (2)

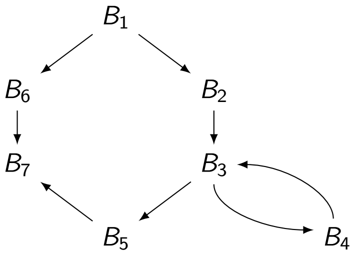
```
function @fact
        t0 <- i_aload @n
        t1 <- i_value 1
        t2 <- i_value 0
        t2 <- i_lt t0, t2
        cjump t2, l1, l0
l0:     t3 <- i_value 1
l3:     t4 <- i_copy t0
        t5 <- i_value 0
        t6 <- i_lt t5, t4
        cjump t6, l4, l5
l4:     t3 <- i_mul t3, t0
        t0 <- i_sub t0, t1
        jump l3
l5:     jump l2
l1:     t3 <- i_value 1
        t3 <- i_inv t3
l2:     i_return t3
```

Sample IR (3)

```
B1:          t0 <- i_aload @n
             t1 <- i_value 1
             t2 <- i_value 0
             t2 <- i_lt t0, t2
             cjump t2, l1, l0

B2: l0:      t3 <- i_value 1

B3: l3:      t4 <- i_copy t0
             t5 <- i_value 0
             t6 <- i_lt t5, t4
             cjump t6, l4, l5

B4: l4:      t3 <- i_mul t3, t0
             t0 <- i_sub t0, t1
             jump l3

B5: l5:      jump l2

B6: l1:      t3 <- i_value 1
             t3 <- i_inv t3

B7: l2:      i_return t3
```

Sample IR (4)

Control flow graph



	Sets of traces covering the IR	cjumps and jumps: Total	cjumps and jumps: Inside a loop
1	$\{B_1 B_6 B_7, B_2 B_3 B_4, B_5\}$	$2+2$	$1+1$
2	$\{B_1 B_2 B_3 B_4, B_5 B_7, B_6\}$	$2+2$	$1+1$
3	$\{B_1 B_2 B_3 B_5 B_7, B_4, B_6\}$	$2+2$	$1+1$
4	$\{B_1 B_2, B_4 B_3 B_5 B_7, B_6\}$	$2+2$	$1+0$

Sample IR (5)

Using set number 4

B_1
cjump B_6, B_2
B_2
jump B_3

B_4
B_3
cjump B_4, B_5
B_5
B_7

B_6
jump B_7

Using set number 1

B_1
cjump B_6, B_2
B_6
B_7

B_2
B_3
cjump B_4, B_5
B_4
jump B_3

B_5
jump B_7

Liveness analysis (1)

Liveness analysis analyses the use of values (variables and temporaries)

Its uses include finding uses of uninitialised variables and computing the resources needed by the code, such as the number of temporaries

Liveness analysis is a global analysis, performed on the control flow graph

There is also a local dimension to liveness analysis, when applied to a single basic block

Liveness analysis (2)

A value is live if control may flow to a point in the code where it will be used, otherwise it is dead

A value used in a graph node is live on entry to the node

A value that is live on entry to a node, is live on exit from its predecessors in the graph

Liveness analysis (3)

The values live on exit from node n are

$$\textsc{LiveOut}(n) = \bigcup_{m \in succ(n)} \textsc{UEVar}(m) \cup (\textsc{LiveOut}(m) - \textsc{VarKill}(m))$$

where

$m \in succ(n)$ are all the successors of node n, *i.e.*, the control flow graph nodes corresponding to the basic blocks where execution may continue when it leaves n

$\textsc{UEVar}(m)$ are those values used in m before being defined in m (the *upward-exposed variables*)

$\textsc{VarKill}(m)$ are the values defined in m

Liveness analysis (4)

Liveness analysis is a backward data-flow problem

- Which values are live on exit from a node depend on
  - Which values its successors use
  - Which values are live on exit from its successors
- The live on exit values are computed following the flow of control in the reverse direction

The values live on entry to n are computed from the values live on exit from n

$$\textsc{LiveIn}(n) = \textsc{UEVar}(n) \cup (\textsc{LiveOut}(n) - \textsc{VarKill}(n))$$

Liveness analysis (5)

Example (10)

Instruction $t_6 \leftarrow \text{i\_mul } t_4, t_5$ uses t_4 and t_5 and defines t_6

$$\textsc{UEVar}(t_6 \leftarrow \text{i\_mul } t_4, t_5) = \{t_4, t_5\}$$

(The values of t_4 and t_5 are used when the instruction is executed)

$$\textsc{VarKill}(t_6 \leftarrow \text{i\_mul } t_4, t_5) = \{t_6\}$$

(The former value of t_6 is destroyed by the instruction)

Example (11)

Instruction $t_{11} \leftarrow \text{i\_sub } t_{11}, t_{12}$ uses t_{11} and t_{12} and defines t_{11}

$$\textsc{UEVar}(t_{11} \leftarrow \text{i\_sub } t_{11}, t_{12}) = \{t_{11}, t_{12}\}$$
$$\textsc{VarKill}(t_{11} \leftarrow \text{i\_sub } t_{11}, t_{12}) = \{t_{11}\}$$

Liveness analysis (6)

In the liveness analysis within a basic block

- Graph nodes are the block instructions
- Control flows linearly, from the first to the last instruction of the block
- The analysis is done in a single pass through the graph, starting at the last instruction and ending at the first
- Values live on entry to an instruction are computed after computing the live on exit values

Liveness analysis (7)

Example (12)

Liveness analysis for block B_3

		UEVAR	VARKILL	LIVEOUT	LIVEIN
1	$t_4 \leftarrow$ i_lload @r	@r	t_4	t_4, @n	@r, @n
2	$t_5 \leftarrow$ i_aload @n	@n	t_5	t_4, t_5, @n	t_4, @n
3	$t_6 \leftarrow$ i_mul t_4, t_5	t_4, t_5	t_6	t_6, @n	t_4, t_5, @n
4	@$r \leftarrow$ i_lstore t_6	t_6	@r	@n	t_6, @n
5	$t_7 \leftarrow$ i_aload @n	@n	t_7	t_7	@n
6	$t_8 \leftarrow$ i_value 1	—	t_8	t_7, t_8	t_7
7	$t_9 \leftarrow$ i_sub t_7, t_8	t_7, t_8	t_9	t_9	t_7, t_8
8	@$n \leftarrow$ i_astore t_9	t_9	@n	—	t_9
9	jump B_2	—	—	—	—

In the analysis of a single block, the last instruction has no successors and no values live on exit from it

Values live on entry to the first instruction of a block must be defined outside the block

Liveness analysis (8)

For block B_i we have

$\text{UEVAR}(B_i)$ $=$ LIVEIN of the first instruction of B_i

$\text{VARKILL}(B_i)$ $=$ union of the VARKILL for the instructions of B_i

Example (13)

Having performed liveness analysis for B_3, we get

$$\text{UEVAR}(B_3) = \text{LIVEIN}(t_4 \leftarrow \text{i\_lload } @r) = \{@r, @n\}$$

$$\text{VARKILL}(B_3) = \{t_4, t_5, t_6, @r, t_7, t_8, t_9, @n\}$$

Liveness analysis (9)

Example (14)

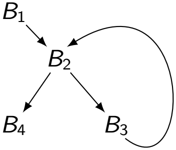
From liveness analysis of all the blocks, we get

Block	UEVAR	VARKILL
B_1	$-$	$t_0, @r$
B_2	$@n$	t_1, t_2, t_3
B_3	$@r, @n$	$t_4, t_5, t_6, @r, t_7, t_8, t_9, @n$
B_4	$@r$	t_{10}

Liveness analysis (10)

Global liveness analysis

Global liveness analysis takes into account all basic blocks



Unlike what happens inside a block, the flow of control among blocks is not linear

Values live on exit from the blocks are computed iteratively

The algorithm halts when there is no change in the computed sets

Liveness analysis (11)

Algorithm for global liveness analysis

foreach block B_i **do**
  $\textsc{LiveOut}'(B_i) \leftarrow \emptyset$
repeat
  foreach block B_i **do**
    $\textsc{LiveOut}(B_i) \leftarrow \textsc{LiveOut}'(B_i)$
  foreach block B_i **do**
    $\textsc{LiveOut}'(B_i) \leftarrow$

$$\bigcup_{B_j \in succ(B_i)} \textsc{UEVar}(B_j) \cup (\textsc{LiveOut}(B_j) - \textsc{VarKill}(B_j))$$

until $(\forall_{B_i})\ \textsc{LiveOut}'(B_i) = \textsc{LiveOut}(B_i)$
foreach block B_i **do**
  $\textsc{LiveIn}(B_i) \leftarrow \textsc{UEVar}(B_i) \cup (\textsc{LiveOut}(B_i) - \textsc{VarKill}(B_i))$

Liveness analysis (12)

Example (15)

For the running example, the algorithm would halt after the third iteration (of the **repeat** loop)

Block	LiveOut				LiveIn
B_1	—	$@n$	$@r, @n$	$@r, @n$	$@n$
B_2	—	$@r, @n$	$@r, @n$	$@r, @n$	$@r, @n$
B_3	—	$@n$	$@r, @n$	$@r, @n$	$@r, @n$
B_4	—	—	—	—	$@r$
	↑ 0^{th}	↑ 1^{st}	↑ 2^{nd}	↑ 3^{rd}	

Iterations

Sample IR (1)

Using the IR on the next slide

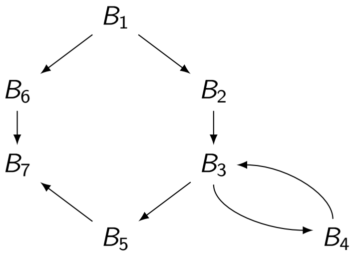
1. Perform local liveness analysis for every block
2. Perform global liveness analysis

Sample IR (2)

```
B1:          t0 <- i_aload @n
             t1 <- i_value 1
             t2 <- i_value 0
             t2 <- i_lt t0, t2
             cjump t2, l1, l0

B2: l0:      t3 <- i_value 1

B3: l3:      t4 <- i_copy t0
             t5 <- i_value 0
             t6 <- i_lt t5, t4
             cjump t6, l4, l5

B4: l4:      t3 <- i_mul t3, t0
             t0 <- i_sub t0, t1
             jump l3

B5: l5:      jump l2

B6: l1:      t3 <- i_value 1
             t3 <- i_inv t3

B7: l2:      i_return t3
```

Sample IR (3)

Control flow graph



Sample IR (4)

Block	LiveOut					LiveIn
B_1	—	—	t_0	t_0, t_1	t_0, t_1	@n
B_2	—	t_0	t_0, t_1, t_3	t_0, t_1, t_3	t_0, t_1, t_3	t_0, t_1
B_3	—	t_0, t_1, t_3	t_0, t_1, t_3	t_0, t_1, t_3	t_0, t_1, t_3	t_0, t_1, t_3
B_4	—	t_0	t_0, t_1, t_3	t_0, t_1, t_3	t_0, t_1, t_3	t_0, t_1, t_3
B_5	—	t_3	t_3	t_3	t_3	t_3
B_6	—	t_3	t_3	t_3	t_3	—
B_7	—	—	—	—	—	t_3
	↑ 0^{th}	↑ 1^{st}	↑ 2^{nd}	↑ 3^{rd}	↑ 4^{th}	

Iterations