CS412/413

Introduction to Compilers

Radu Rugina

Lecture 23: Live Variable Analysis

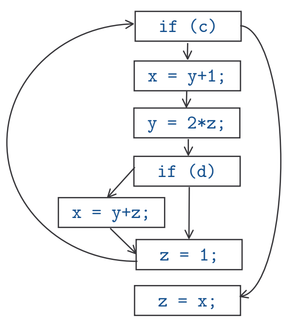
26 Mar 06

Control Flow Graphs

- Control Flow Graph (CFG) = graph representation of computation and control flow in the program
  - framework to statically analyze program control-flow
- In a CFG:
  - Nodes are basic blocks; they represent computation
  - Edges characterize control flow between basic blocks
- Can build the CFG representation either from the high IR or from the low IR

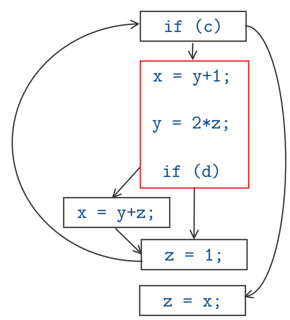
From Three-Address Code to CFG

```
label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y+z;
label L3
z = 1;
jump L1
label L2
z = x;
```



Basic Blocks

```
label L1
fjump c L2
x = y + 1;
y = 2 * z;
fjump d L3
x = y+z;
label L3
z = 1;
jump L1
label L2
z = x;
```



Using CFGs

- Next: use CFG representation to statically extract information about the program
  - Reason at compile-time
  - About the run-time values of variables and expressions in all program executions
- Extracted information example: live variables
- Idea:
  - Define program points in the CFG
  - Reason statically about how the information flows between these program points

Program Points

- Two program points for each instruction:
  - There is a program point before each instruction
  - There is a program point after each instruction

Point before → •

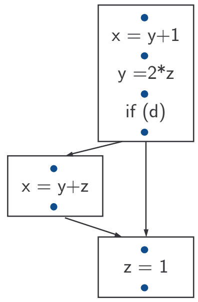
x = y+1

Point after → •

- In a basic block:
  - Program point after an instruction = program point before the successor instruction

Program Points: Example

- Multiple successor CFG nodes:
  - control will flow to one of the successor program points
  - It is not statically known which one
- Similar situation when there are multiple predecessor CFG nodes
- How does information propagate between program points?

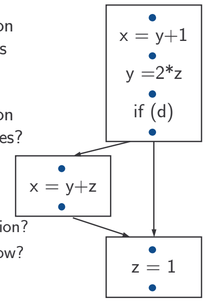


Flow of Extracted Information

- Question 1: how does information flow between the program points before and after an instruction?
- Question 2: how does information flow between different CFG nodes?
- … in other words:

  Q1: what is the effect of computation?

  Q2: what is the effect of control flow?



Using CFGs

- To extract information: reason about how it propagates between program points
- Rest of this lecture: how to use CFGs to compute information at each program point for:
  - Live variable analysis, which computes live variables are live at each program point
  - Copy propagation analysis, which computes the variable copies available at each program point

Live Variable Analysis

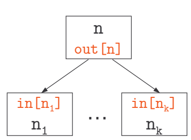
- Using the CFG structure to perform live variable analysis
  - A variable is live if its value might be needed later
  - Goal: compute all live variables at each program point
- For each CFG node n, consider:
  - `in[n]` = live variables at program point before n
  - `out[n]` = live variables at program point after n
- CFG node can be either:
  - An instruction I
  - A basic block B

How to Compute Liveness?

- Answer question 1: for each CFG node `n`, what is the relation between `in[n]` and `out[n]` ?

  `in[n]`
  `n`
  `out[n]`

- Answer question 2: for each CFG node `n` with successors $n_1, \ldots, n_k$, what is the relation between `out[n]` and `in[`n_1`]`, …, `in[`n_k`]`?



Part 1: Analyze Instructions

- Question: what is the relation between sets of live variables before and after a node?

  `in[n]`
  `n`
  `out[n]`

- Examples:

`in[n] = ?`	`in[n] = ?`	`in[n] = ?`
`x = y+z`	`x = y+z`	`x = x+z`
`out[n] = {z}`	`out[n] = {x,t}`	`out[n] = {x,t}`

- … is there a general rule?

Analyze Instructions

- Yes: knowing variables live after `n`, we can compute variables live before `n` :
  - All variables live after `n` are also live before `n`, unless `n` defines (writes) them
  - All variables that `n` uses (reads) are also live before instruction `n`

`in[n]`
`n`
`out[n]`

- Mathematically:

`in[n] = ( out[n] – def[n] ) ∪ use[n]`

where:
- `def[n]` = variables defined (written) by node `n`
- `use[n]` = variables used (read) by node `n`

Computing Use/Def

- Compute `use[n]` and `def[n]` for each instruction:

```
if n is x = y OP z :  use[n] = {y, z}  def[n] = {x}
if n is x = OP y   :  use[n] = {y}     def[n] = {x}
if n is x = y      :  use[n] = {y}     def[n] = {x}
if n is if (x)     :  use[n] = {x}     def[n] = {}
if n is return x   :  use[n] = {x}     def[n] = {}
if n is x = f(y1,…, yn) :
                 use[I] = {y1,…, yn}   def[I] = {x}
```

(For now, ignore load and store instructions)

Example

- Example: three consecutive instructions `I1, I2, I3`:

```
Live1 = in[I1]
Live2 = out[I1] = in[I2]
Live3 = out[I2] = in[I3]
Live4 = out[I3]
```

```
      Live1
I1  x = y+1
      Live2
I2  y =2*z
      Live3
I3  if (d)
      Live4
```

- Relation between Live sets:

```
Live1 = (Live2-{x}) ∪ {y}
Live2 = (Live3-{y}) ∪ {z}
Live3 = (Live4-{} ) ∪ {d}
```


Backward Flow

- Relation:

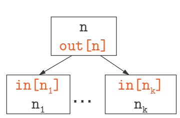
`in[I] = ( out[I] – def[I] ) ∪ use[I]`

- Can compute `in[I]` if we know `out[I]`
- The information flows backward

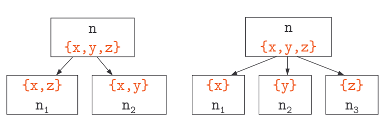
`in[I]`
`I`
`out[I]`

Part 2: Analyze Control Flow

- Question: for each node `n` with successor blocks n_1, …, n_k, what is the relation between `out[n]` and `in[n1]`, …, `in[nk]`?



- Examples:

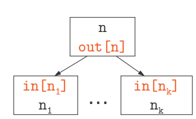


- What is the general rule?

Analyze Control Flow

- Rule: A variables is live at a program point if it is live at one of the successor points
- Characterizes all possible program executions
- Mathematically:

$$\texttt{out[n]} = \bigcup_{\texttt{n'} \in \texttt{succ(n)}} \texttt{in[n']}$$



- Again, information flows backward: from successors n' of n, to n itself

Constraint System

- Put parts together: start with CFG and derive a system of constraints between live variable sets:

```
in[n] = ( out[n] – def[n] ) ∪ use[n]    for each node n
out[n] =      ∪       in[n']
          n' ∈ succ(n)
```

- Solve constraints:
  - Start with empty sets of live variables
  - Iteratively apply constraints
  - Stop when we reach a fixed point

Constraint Solving Algorithm

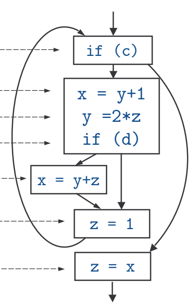
```
For all CFG nodes n: in[n] = out[n] = ∅
worklist = all “return” (exit) nodes

while (worklist is not empty)
    remove n from worklist
    in[n] = ( out[n] – def[n] ) ∪ use[n]

    for each predecessor n':
        out[n'] = out[n'] ∪ in[n]
        if (out[n'] changed)
            add n' to worklist
```


Example: Live Variable Analysis

```
def = {},   use = {c}
def = {x},  use = {y}
def = {y},  use = {z}
def = {},   use = {d}
def = {x},  use = {y,z}
def = {x},  use = {}
def = {z},  use = {x}
```



Copy Propagation

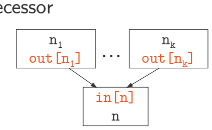
- Goal: determine copies available at each program point
- Information: set of copies <x=y> at each point
- For each CFG node:
  - in[n] = copies available at program point before n
  - out[n] = copies available at program point after n

Copy Propagation Analysis

- Knowing in[n], we can compute out[n]:
  - Remove from in[n] all copies <u=v> if variable u or v is written by n
  - Keep all other copies from in[n]
  - If n is of the form x=y, add <x=y> to out[n]

in[n]
n
out[n]

- A copy is available at point before n if it is available at the end of all predecessor program points



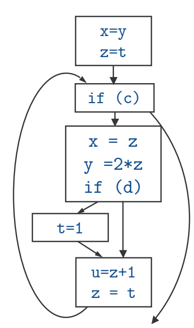
Example: Copy Propagation

- What are the available copies at the end of the program?

x=y?

z=t?

x=z?



Summary

- Extracting information about live variables, available copies:
  - Define the required information
  - Build constraints for instructions/control flow
  - Solve constraints to get needed information
- …is there a general framework?
  - Yes: dataflow analysis!