Compilers - Chapter 7:

Loop optimisations

Part 1: Reaching definitions

- Lecturer:
  - Paul Kelly (p.kelly@imperial.ac.uk)

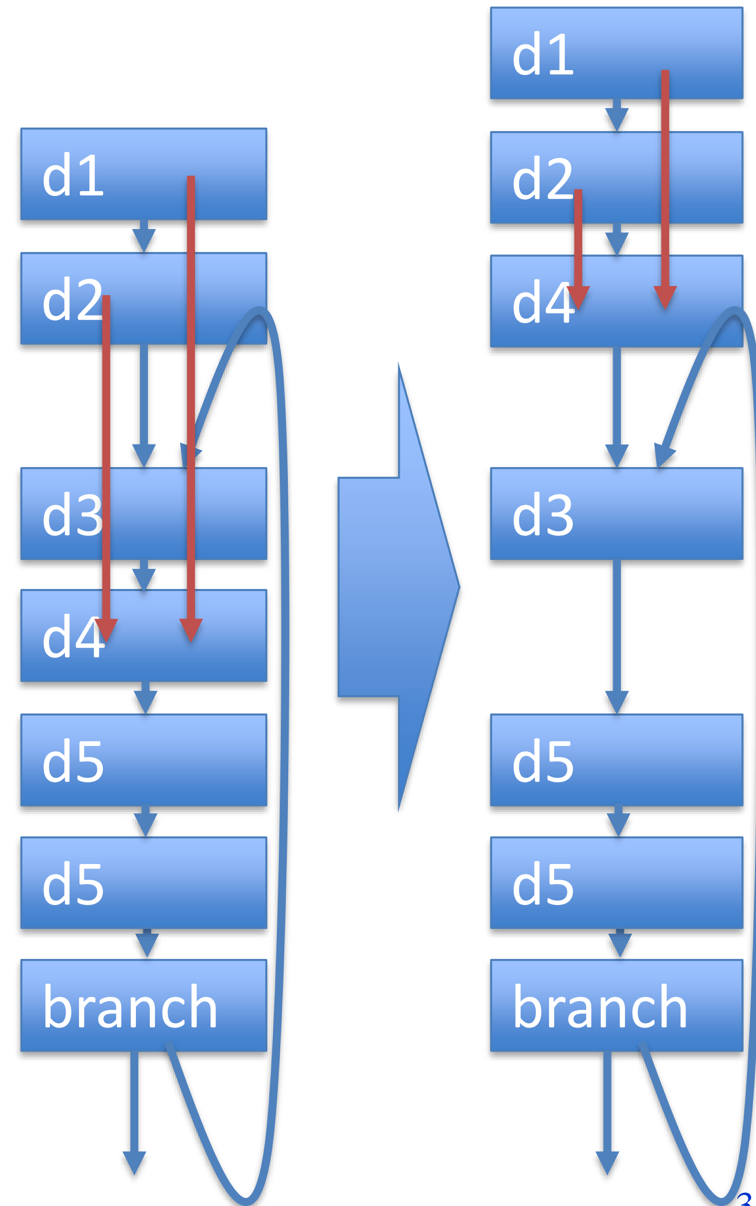
PAGE 3

DEPARTMENT	COURSE	DESCRIPTION	PREREQS
COMPUTER SCIENCE	CPSC 432	INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.	CPSC 432

https://xkcd.com/754/

Loop-invariant code motion

- Definition:
  - An instruction is loop-invariant if its operands can only arrive from outside the loop
- Objective:
  - move (“hoist”) loop-invariant instructions out of loop
- Issues:
  - How can we find out whether operands only arrive from outside loop
  - Where should we move the loop-invariant instructions *to*?
  - Other pitfalls…



Finding loop-invariant instructions

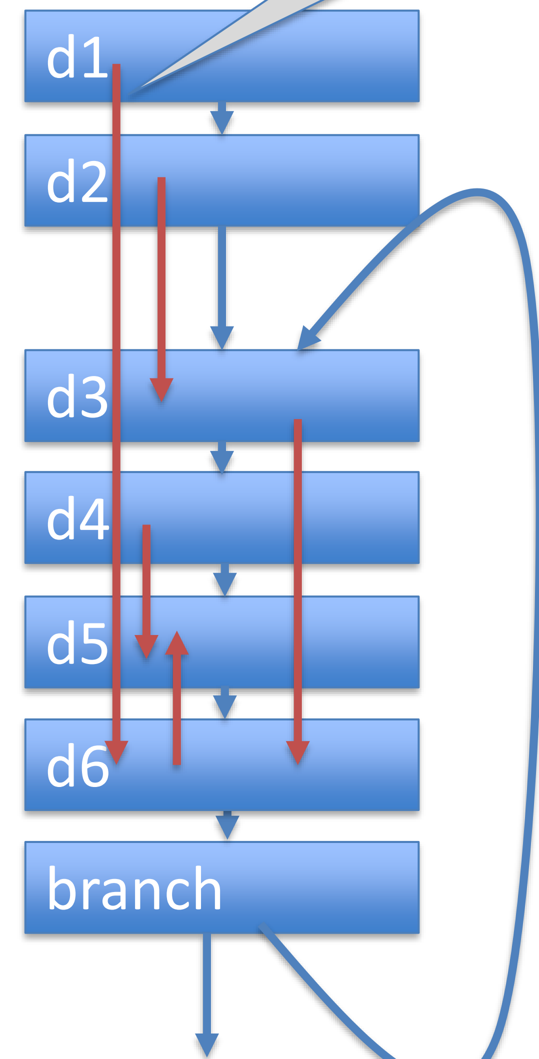
- A CFG node is a definition if it updates a temporary
- In our CFG, an instruction can update at most one temporary, *t*
- Each definition node is labelled with the Node id, *d:*

  $d: \ \ t := u1 \oplus u2$

  Or simply

  $d: \ \ t := u1$ or $d: \ \ t := \text{constant}$

  (where u1 and u2 are given by the Node's "uses" field)
- This definition is loop-invariant if, for each $u_i \in \text{uses}(d)$,
  - All the definitions of u_i that reach *d* are outside the loop
  - Or only one definition of u_i reaches *d*, and that definition is loop invariant

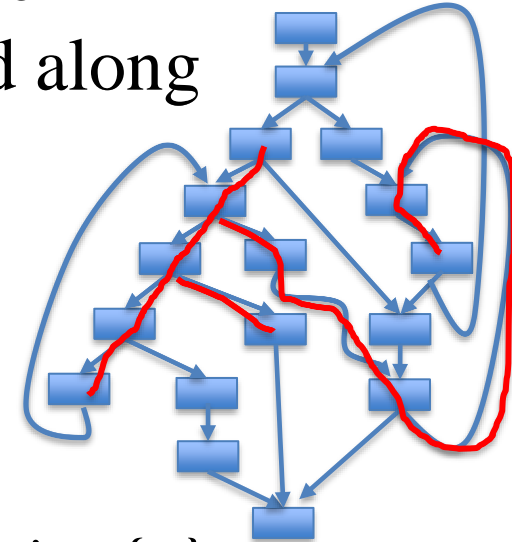


In this example, d3 and d6 are loop-invariant

Finding reaching definitions

- A definition of variable t is a statement which may assign to t
- A definition d **reaches** a program point p if there exists a path from d to p such that d is not killed along that path
- Consider a CFG node

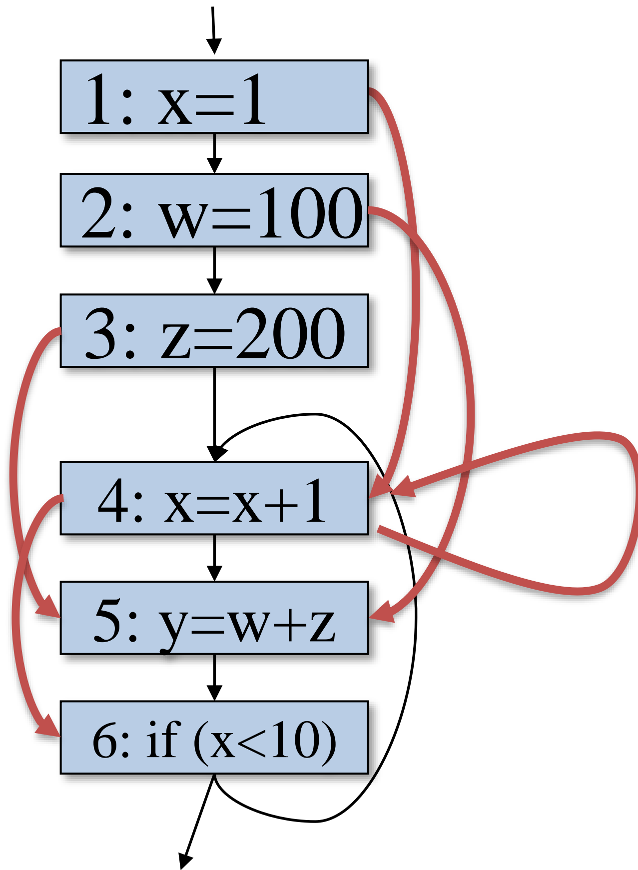
  n: $t := u1 \oplus u2$ (defs(n)={t}, uses(n) = {u1,u2})



Define:

- **Gen(n)** is the set of definitions generated by node n, *i.e. {n}*
- **Kill(n)** is the set of all definitions of t, excluding n
- **ReachIn(n)** is the set of definitions reaching the point before n
- **ReachOut(n)** is the set of definitions reaching the point after n

Reaching definitions



- Reaching definitions link each use of a variable back to where its value could have been generated
- Loops and conditionals result in multiple reaching definitions

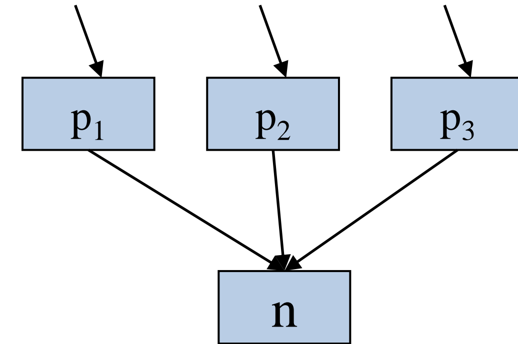
- Gen(4) = {4}
- Gen(5) = {5}
- Gen(6) = {}

- kill(4) = {1}
- kill(5) = {}

- ((In the worst case, the number of reaching definitions could be quite large))

Reaching definitions – another data flow analysis

- Dataflow equations:



$$\mathbf{ReachIn}(n) = \bigcup_{p \in \mathrm{pred}(n)} \mathbf{ReachOut}(p)$$

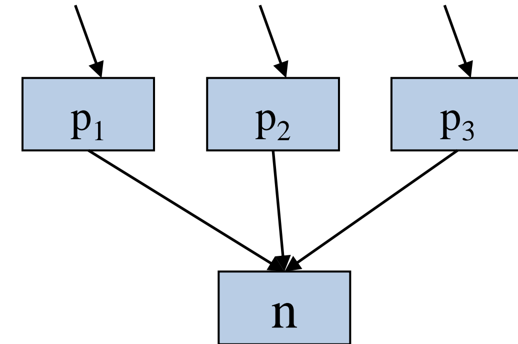
$$\mathbf{ReachOut}(n) = \mathbf{Gen}(n) \cup (\mathbf{ReachIn}(n) - \mathbf{Kill}(n))$$

("The Gen(n) + whatever survives")

- Many dataflow problems have "gen" and "kill"
- In the case of ReachOut(n):
  – gen(n) is usually just its own id, {n}
  – But if node n defines no value (eg it's a jump), it will never reach anything – so gen(n) = { }

Reaching definitions – another data flow analysis

- Dataflow equations:



$$\textbf{ReachIn}(n) = \bigcup_{p \in \mathrm{pred}(n)} \textbf{ReachOut}(p)$$

$$\textbf{ReachOut}(n) = \textbf{Gen}(n) \cup (\textbf{ReachIn}(n) - \textbf{Kill}(n))$$

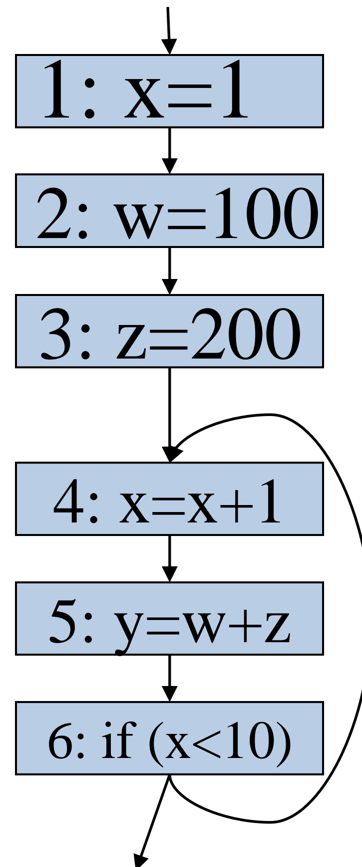
("The Gen(n) + whatever survives")

- Solve in the usual way:
  - Initialise **ReachIn**(*n*) and **ReachOut**(*n*) to { }
  - Iterate, repeatedly updating **ReachIn**(*n*) and **ReachOut**(*n*) using definitions above
  - Until convergence
  - At each step, the sets increase in size

Use reaching definitions to find loop invariant instructions

- Find the set of definitions of variables used by this node
- An instruction is loop invariant if the definitions of all the values it uses are outside the loop
- Example:

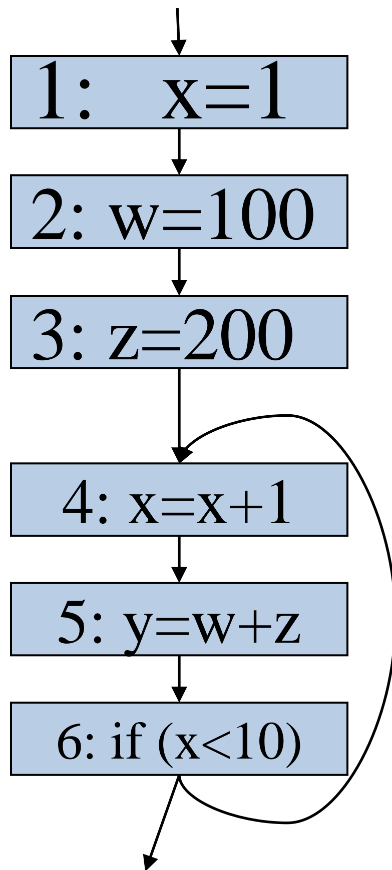
```
1    x = 1
2    w=100
3    z=200
  Here:
4    x = x+1
5    y=w+z
6    if (x<10) goto Here
```



- Reaching definitions (ReachIn):
- 1: []
- 2: [1]
- 3: [1,2]
- 4: [1,2,3,4,5]
- 5: [2,3,4,5]
- 6: [2,3,4,5]

Use reaching definitions to find loop invariant instructions

- Find the definitions which reach this node which are ***relevant*** – that is, which generate the values this node uses:



- Reaching definitions:
- 1: []
- 2: [1]
- 3: [1,2]
- 4: [1,2,3,4,5]
- 5: [2,3,4,5]
- 6: [2,3,4,5]

- "***Relevant***" Reaching definitions:
- 1: []
- 2: []
- 3: []
- 4: [1,4]
- 5: [2,3] ← All the definitions of the values used by node 5 lie outside the loop
- 6: [4]

Summary

- We can find loop-invariant instructions
  - We will use this shortly to actually optimise the code
- We used another dataflow analysis: “reaching definitions”
  - ReachIn and ReachOut are sets of *definitions*
  - This is a *forward* analysis

Piazza question:

“Reaching definitions for globals?”

Should the definition of a global variable be considered a reaching definition for all the nodes of the CFG of a procedure?

For example:

```
int a = 3;
int b() {
   return a;
}
```

Is "int a = 3" a reaching definition for "return a;"? This seems a bit problematic as, for example, if we have:

```
int a = 3;

int b() {
  return a;
}

int c() {
  a = 2;
  return b();
}
```

In this case, the line "a = 2" would be a reaching definition as well? But this would make it very hard to compute all the reaching definitions, as this would require interprocedural analysis, which we haven't discussed as it seems it would be quite slow and problematic.

In this course, we confine our attention to reaching definitions of local variables, not global ones - for the reason illustrated in this example.

Piazza question: "Interprocedural dominator analysis?"

When finding dominators of a line, do we only consider the lines in the same procedure or do calls to other functions mean that we should consider those lines as well? In this example:

```
1: int a() {
2:  return 1;
   }
3: int b() {
4:     int x = a();
5:     return x;
   }
```

Does line 2 dominate line 5? It is always executed before line 5, but building a CFG which takes this into consideration would be somewhat cumbersome, so are we only expected to consider intraprocedural control flow graphs, as in the lectures? (So in the examples the dominators of line 5 would be lines 3, 4, 5)

In this course we stick to intraprocedural analysis.

You could of course inline a() to make the issue intraprocedural.

What makes the interprocedural case hard is that there might be another call to a() somewhere else.

[[This is called the "infeasible paths" problem: naively, your interprocedural CFG would have edges 3->1 and 2->4. But if a() is called from somewhere some other function similar to b():

```
6: : int c() {
7:   int x = a();
8:   return x;
   }
```

Now we would have additional edges 6->1 and 2->7. But the path 3-1-2-7-8 is infeasible.

]]

- Fine print:
  - For efficiency, it is better for the CFG to consist of basic blocks instead of individual instructions – so many compilers compute the gen and kill sets for basic blocks before solving iteratively
  - For loop optimisations, we would do the data-flow analysis on the IR *before* instruction selection – on the three-address-code representation. It's simpler, and means the optimization is machine-independent – it doesn't depend on the target machine's instruction set.
  - See Appel pg388
- Credits: the primary source for these slides (and parts 2 and 3) was Appel's book. I also found it very useful to study the course notes of Liz White (George Mason University), Laurie Hendren (McGill University) and Chau-Wen Tseng (University of Maryland)

Feeding curiosity

- You can represent a control-flow graph in Datalog, and then you can express reaching definitions analysis as a Datalog query:

```
Reach(d,x,j) :- Reach(d,x,i),
                StatementAt(i,s),
                !Assign(s,x),
                Follows(i,j).
Reach(s,x,j) :- StatementAt(i,s),
                Assign(s,x),
                Follows(i,j).
```

- Now the challenge is to take this abstract specification and turn it into an efficient implementation.
- This is a compilation problem!
- See John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. **Using datalog with binary decision diagrams for program analysis**. In *Proceedings of the Third Asian conference on Programming Languages and Systems (APLAS'05).* https://doi.org/10.1007/11575467_8