Data Flow Analysis Based Optimization

Last lecture
- Register allocation

Today
- Dead code elimination
- Common subexpression elimination
- Generalizing data-flow analysis

Logistics
- PA2 has been posted
- Monday the 15th, no class due to LCPC in Oregon

Dead Code Elimination

Remove statements that define only one variable and the variable being defined is not in the live out set.

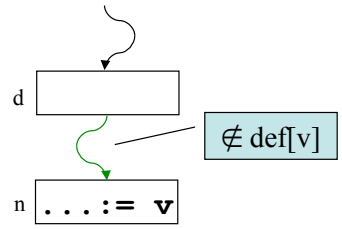
Algorithm

```
1) generate a control flow graph (CFG) from the list of instructions
do {
   2) perform liveness on CFG
   3) for each node in CFG
      if the defs set contains only one temporary
        if the temporary being defined is not in the live out set
                remove the node from the CFG
} while (changes);
4) generate a list of instructions from the modified CFG
```

Recall Liveness Analysis

Definition

- A variable is **live** at a particular point in the program if its value at that point will be used in the future (**dead**, otherwise).



Uses of Liveness

- Register allocation
- Dead-code elimination

If **a** is not live out of statement 1 then statement 1 is dead code.

```
1   a = . . .;
2   b = . . .;
3   ...
4   x = f(b);
```

Data-flow equations

(1) **in[n] = use[n] ∪ (out[n] – def[n])** (3)

out[n] = $\bigcup_{s \in succ[n]}$ **in[s]** (2)

Common Subexpression Elimination

Idea

- Find common subexpressions whose *range* spans the same basic blocks and eliminate unnecessary re-evaluations
- Leverage available expressions

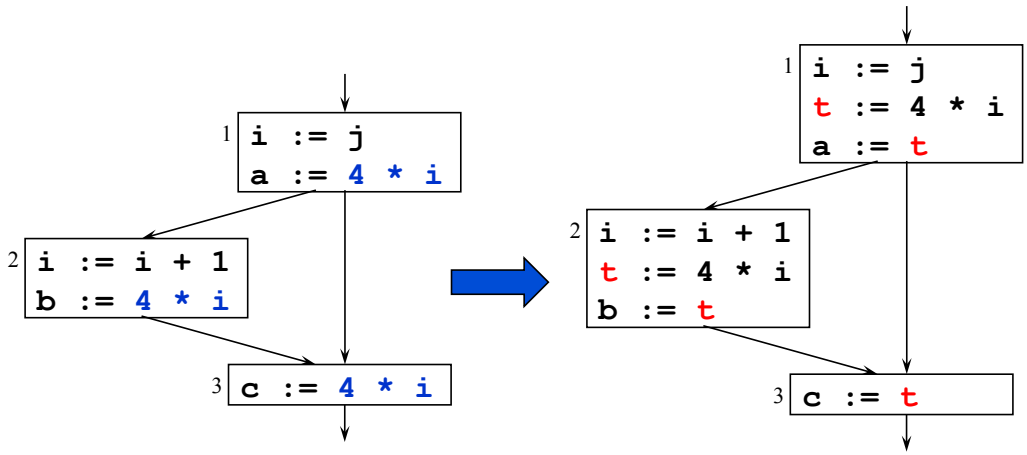
Use available expressions

- An expression (*e.g.,* **x+y**) is **available** at node n if **every** path from the entry node to n evaluates **x+y**, and there are no definitions of **x** or **y** after the last evaluation along that path

Strategy

- If an expression is available at a point where it is evaluated, it need not be recomputed

CSE Example



Another CSE Example

Before CSE

```
c := a + b
d := m & n
e := b + d
f := a + b
g := -b
h := b + a
a := j + a
k := m & n
j := b + d
a := -b
if m & n goto L2
```

Summary
11 instructions
12 variables
9 binary operators

After CSE

```
c := a + b
t1 := c
d := m & n
t2 := d
e := b + d
t3 := e
f := t1
g := -b
h := t1
a := j + a
k := t2
j := t3
a := -b
if t2 goto L2
```

Summary
14 instructions
15 variables
4 binary operators

Learning CSE

Available expressions data flow analysis

Various approaches to performing CSE

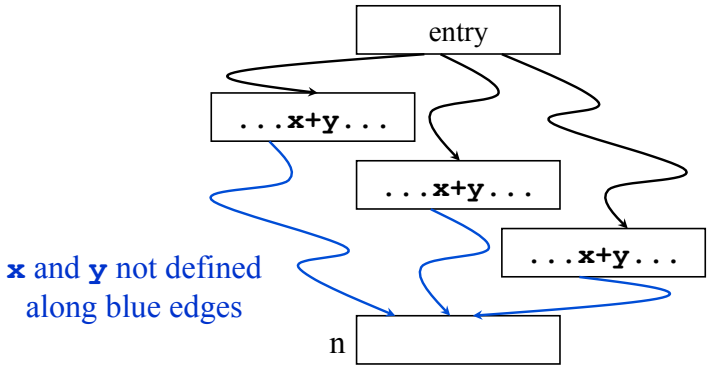
Opportunity to use CSE in mini-cfd code

Performing CSE in SSA and LLVM

Available Expressions

Definition

– An expression, **`x+y`**, is **available** at node n if every path from the entry node to n evaluates **`x+y`**, and there are no definitions of **`x`** or **`y`** after the last evaluation



Available Expressions Iterative Algorithm

Data-Flow Equations

$$in[n] = \bigcap_{p \in pred[n]} out[p]$$
$$out[n] = gen[n] \cup (in[n] - kill[n])$$

Plug it in to our general DFA algorithm

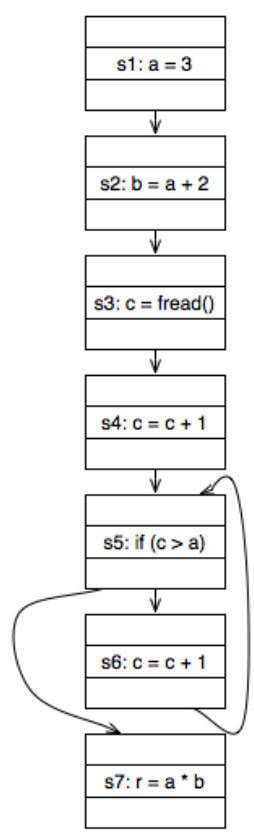
for each node n
  in[n] = U; out[n] = U
repeat
  for each n
    in′[n] = in[n]
    out′[n] = out[n]
    $in[n] = \bigcap_{p \in pred[n]} out[p]$
    $out[n] = gen[n] \cup (in[n] - kill[n])$
until in′[n]=in[n] and out′[n]=out[n] for all n

Available Expressions Example

Is available expressions a forward or backward analysis?

What is the initial guess?

What are the available in and out sets for each statement?



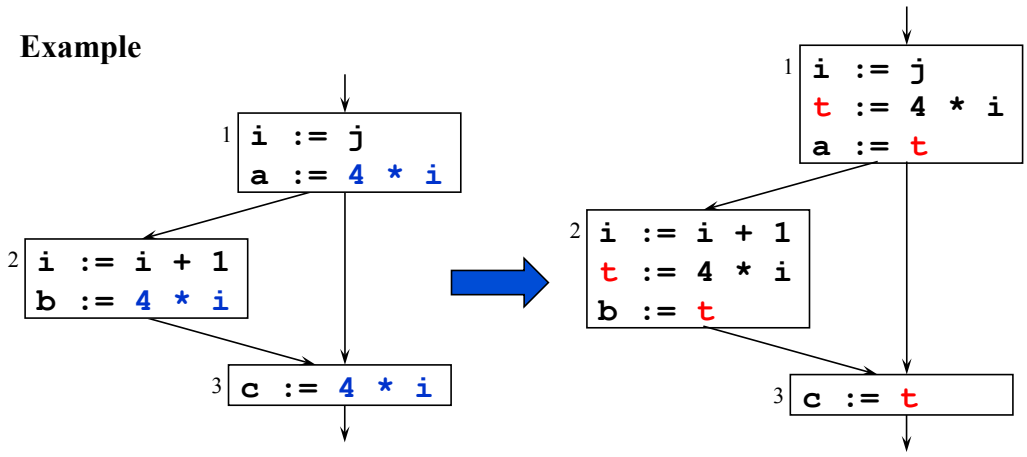
Available Expressions for CSE

How is this information useful?

Common Subexpression Elimination (CSE)

– If an expression is available at a point where it is evaluated, it need not be recomputed

Example



CSE Approach 1

Notation

– IN_avail(s) is the set of expressions available at statement s
– GEN(s) is the set of expressions generated and not killed at statement s

If we use e and e ∈ IN_avail(s)

– Allocate a new name t
– Search backward from s (in CFG) to find statements (one for each path) that most recently generate e (e ∈ GEN)
– Insert copy to t after generators
– Replace e with t

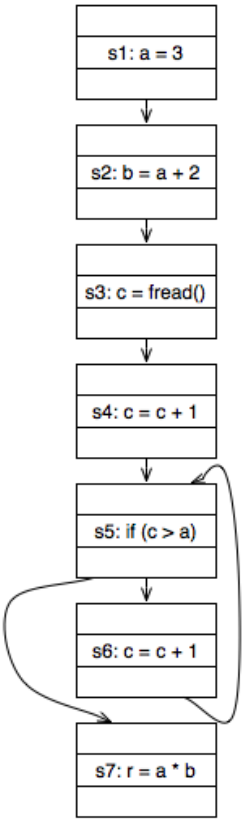
Problems

– Backward search for each use is expensive
– Generates unique name for each use
  – |Uses| > |Avail|
  – Each generator may have many copies

Example

```
 a := b + c
t2 := a
t1 := a
e := b1+ c
f := b2+ c
```

CSE Example



CSE Approach 2

Idea

- Reduce number of copies by assigning a unique name to each unique expression

Summary

- ∀e Name[e] = unassigned
- if we use e and e ∈ Avail(b)
  - if Name[e]=unassigned, allocate new name t and Name[e] = t else t = Name[e]
  - Replace e with t
- In a subsequent traversal of statement s, if e ∈ Gen(s) and Name[e] ≠ unassigned, then insert a copy to Name[e] after the generator of e

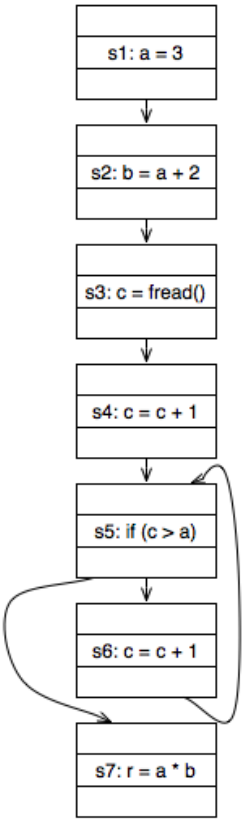
Problem

- May still insert unnecessary copies
- Requires two passes over the code

Example

```
a := b + c
t1 := a
```

CSE Example



CSE Approach 3

Idea

- Don't worry about temporaries
- Create one temporary for each unique expression
- Let subsequent pass eliminate unnecessary temporaries

At an evaluation/generation of e

- Hash e to a name, t, in a table
- Insert an assignment of e to t

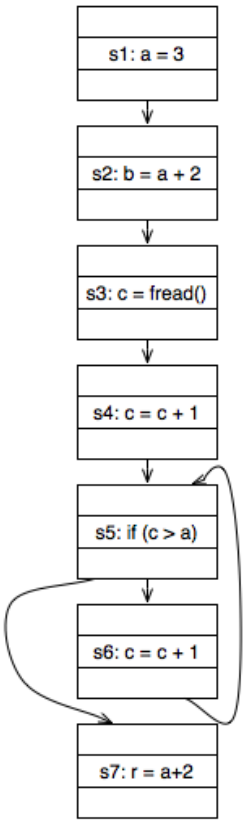
At a use of e in b, if e ∈ Avail(b)

- Lookup e's name in the hash table (call this name t)
- Replace e with t

Problems

- Inserts more copies than approach 2 (but extra copies are dead)
- Still requires two passes (2nd pass is very general)

CSE Example



CSE in SSA and LLVM

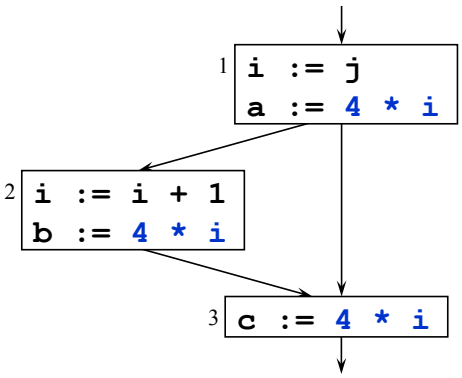
See example in PA2 writeup.

Available Expression Analysis

- Don't need to compute kills due to SSA.

Creating temporaries

- Can't assign to same temp name in multiple locations due to SSA.
- Can't put in copies however. %t1 = %v doesn't work.
- Ideas?



Concepts

Common subexpression elimination

- Uses the results of available expressions
- Many ways to implement
- SSA and LLVM representation lead to yet another way

Generalizing Data-flow Analysis

Types of data-flow analysis

- liveness analysis
- available expressions
- reaching definitions
- reaching constants

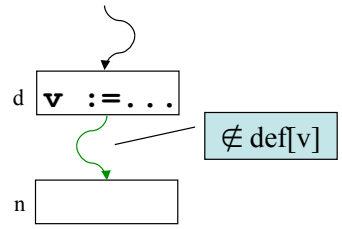
Abstracting data-flow analysis

- What's common among the different analyses?

Reaching Definitions

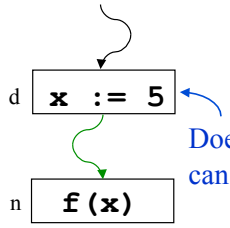
Definition

- A definition (statement) d of a variable **v** **reaches** node n if there is a path from d to n such that **v** is not redefined along that path



Uses of reaching definitions

- Build use/def chains
- Constant propagation
- Loop invariant code motion



Does this def of **x** reach n?
can we replace n with **f(5)**?

```
1   a = . . .;
2   b = . . .;
3   for (. . .) {
4       x = a + b;
5       . . .
6   }
```

Reaching definitions of **a** and **b**

To determine whether it's legal to move statement 4 out of the loop, we need to ensure that there are no reaching definitions of **a** or **b** inside the loop

Computing Reaching Definitions

Assumption

- At most one definition per node
- We can refer to definitions by their node "number"

Gen[n]: Definitions that are generated by node n (at most one)

Kill[n]: Definitions that are killed by node n

Defining Gen and Kill for various statement types

statement	Gen[s]	Kill[s]
s: t = b op c	{s}	def[t] – {s}
s: t = M[b]	{s}	def[t] – {s}
s: M[a] = b	{}	{}
s: if a op b goto L	{}	{}

statement	Gen[s]	Kill[s]
s: goto L	{}	{}
s: L:	{}	{}
s: f(a,…)	{}	{}
s: t=f(a, …)	{s}	def[t] – {s}

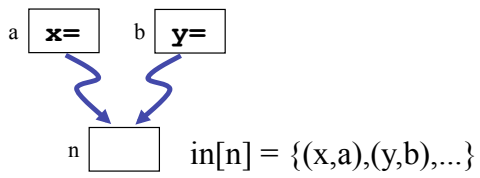
A Better Formulation of Reaching Definitions

Problem

- Reaching definitions gives you a set of definitions (nodes)
- Doesn't tell you what variable is defined
- Expensive to find definitions of variable v

Solution

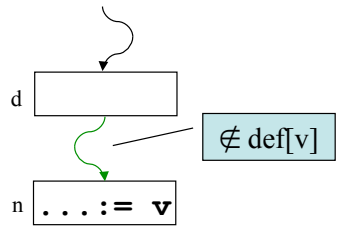
- Reformulate to include variable
  e.g., Use a set of (var, def) pairs



Recall Liveness Analysis

Definition

- A variable is **live** at a particular point in the program if its value at that point will be used in the future (**dead**, otherwise).



Uses of Liveness

- Register allocation
- Dead-code elimination

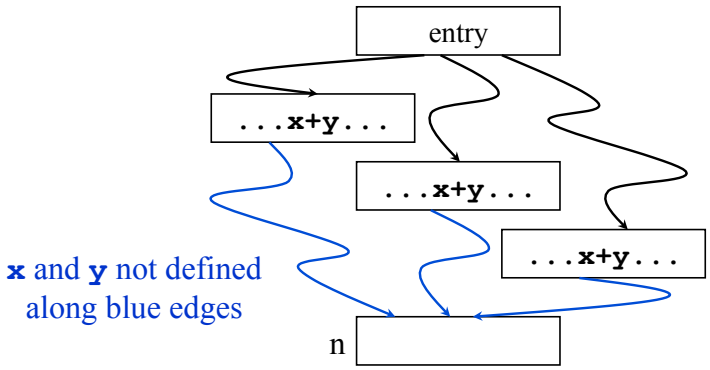
```
1   a = . . .;
2   b = . . .;
3   ...
4   x = f(b);
```

If **a** is not live out of statement 1 then statement 1 is dead code.

Available Expressions

Definition

– An expression, **`x+y`**, is **available** at node n if every path from the entry node to n evaluates **`x+y`**, and there are no definitions of **`x`** or **`y`** after the last evaluation



Aspects of Data-flow Analysis

Must or may Information	guaranteed or possible
Direction	forward or backward
Flow values	variables, definitions, ...
Initial guess	universal or empty set
Kill	due to semantics of stmt what is removed from set
Gen	due to semantics of stmt what is added to set
Merge	how sets from two control paths compose

Must vs. May Information

Must information

- Implies a guarantee

May information

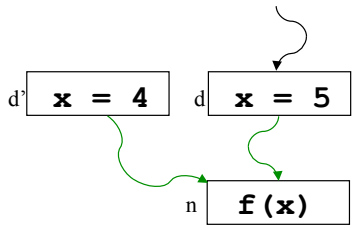
- Identifies possibilities

Liveness? Available expressions?

	May	Must
safe	overly large set	overly small set
desired information	small set	large set
Gen	add everything that might be true	add only facts that are guaranteed to be true
Kill	remove only facts that are guaranteed to be true	remove everything that might be false
merge	union	intersection
initial guess	empty set	universal set

Reaching Definitions: Must or May Analysis?

Consider uses of reaching definitions



We need to know if d' might reach node n

Defining Available Expressions Analysis

Must or may Information?	Must
Direction?	Forward
Flow values?	Sets of expressions
Initial guess?	Universal set
Kill?	Set of expressions killed by statement s
Gen?	Set of expressions evaluated by s
Merge?	Intersection

Reaching Constants (aka Constant Propagation)

Goal

- Compute value of each variable at each program point (if possible)

Flow values

- Set of (variable,constant) pairs

Merge function

- Intersection

Data-flow equations

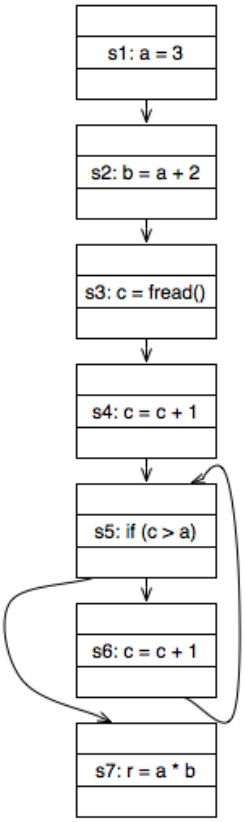
- Effect of node n `x = c`
  - kill[n] = {(x,d)| ∀d}
  - gen[n] = {(x,c)}
- Effect of node n `x = y + z`
  - kill[n] = {(x,c)| ∀c}
  - gen[n] = {(x,c) | c=val(y)+valz, (y, valy) ∈ in[n], (z, valz) ∈ in[n]}

Reaching Constants Example

Must or may info?

Direction?

Initial guess?



Reality Check!

Some definitions and uses are ambiguous

- We can't tell whether or what variable is involved
  e.g., `*p = x;  /* what variable are we assigning?! */`
- Unambiguous assignments are called **strong updates**
- Ambiguous assignments are called **weak updates**

Solutions

- Be conservative
  - Sometimes we assume that it could be everything
    e.g., Defining `*p` (generating reaching definitions)
  - Sometimes we assume that it is nothing
    e.g., Defining `*p` (killing reaching definitions)
- Try to figure it out: alias/pointer analysis (more later)

Side Effects

What happens at function calls?

- For example, the call foo(&x) might use or define
  - any local or heap variable x that has been passed by address/reference
  - any global variable

Solution

- How do we handle this for liveness used for register allocation?
- In general
  - Be conservative: assume all globals and all vars passed by address/reference may be used and/or modified
  - Or Figure it out: calculate side effects (example of an interprocedural analysis)

Concepts

Common subexpression elimination

- Uses the results of available expressions
- Many ways to implement
- SSA and LLVM representation lead to yet another way

Data-flow analyses are distinguished by

- Flow values (initial guess, type)
- May/must
- Direction
- Gen
- Kill
- Merge

Complication

- Ambiguous references (strong/weak updates)
- Side effects