

Generalizing Data-flow Analysis

Announcements

- Read Section 9.3 in the book

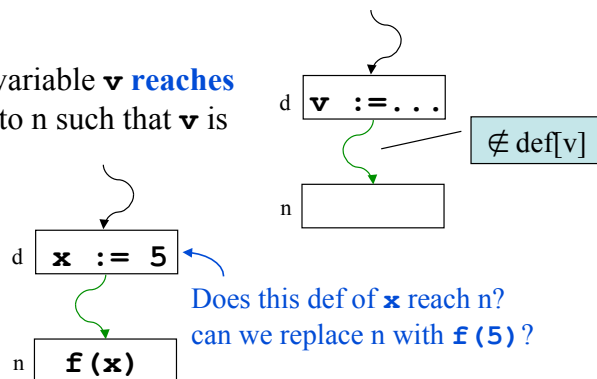
Today

- Other types of data-flow analysis
 - Reaching definitions, available expressions, 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

```
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

<u>statement</u>	<u>Gen[s]</u>	<u>Kill[s]</u>	<u>statement</u>	<u>Gen[s]</u>	<u>Kill[s]</u>
s: t = b op c	{s}	def[t] – {s}	s: goto L	{}	{}
s: t = M[b]	{s}	def[t] – {s}	s: L:	{}	{}
s: M[a] = b	{}	{}	s: f(a,...)	{}	{}
s: if a op b goto L	{}	{}	s: t=f(a, ...)	{s}	def[t] – {s}

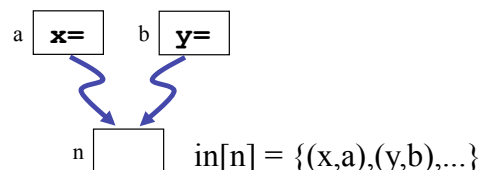
A Better Formulation of Reaching Definitions

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

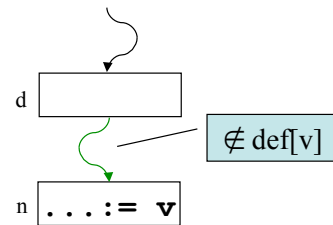
- Reformulate to include variable
e.g., Use a set of (var, def) pairs



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

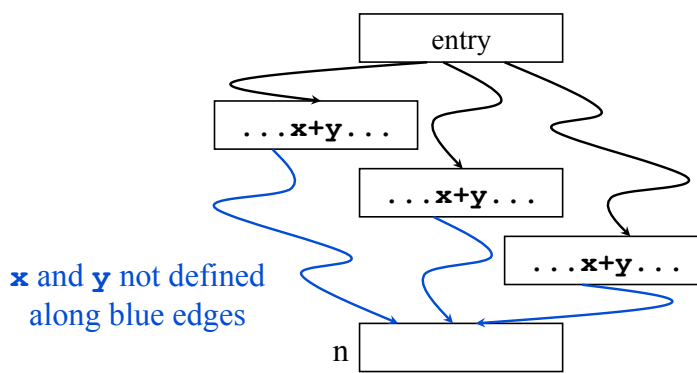
```

1  a = . . . ; ← If a is not live out of statement 1 then
2  b = . . . ;   statement 1 is dead code.
3  ...
4  x = f(b) ;
  
```

Available Expressions

Definition

- An expression, $\mathbf{x+y}$, is **available** at node n if every path from the entry node to n evaluates $\mathbf{x+y}$, and there are no definitions of \mathbf{x} or \mathbf{y} after the last evaluation



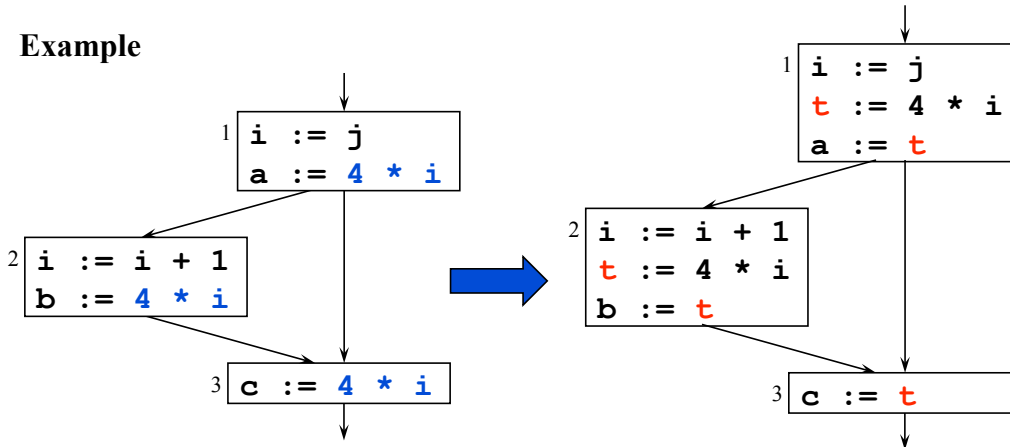
Available Expressions for CSE

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



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

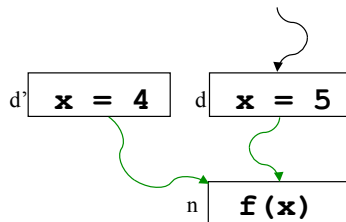
CS553 Lecture

Generalizing Data-flow Analysis

9

Reaching Definitions: Must or May Analysis?

Consider uses of reaching definitions



We need to know if d' might reach node n

CS553 Lecture

Generalizing Data-flow Analysis

10

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

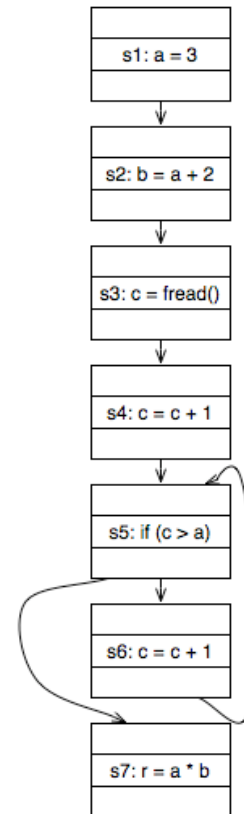
- Effect of node $n \quad \mathbf{x} = \mathbf{c}$
 - $\text{kill}[n] = \{(x,d) \mid \forall d\}$
 - $\text{gen}[n] = \{(x,c)\}$
- Effect of node $n \quad \mathbf{x} = \mathbf{y} + \mathbf{z}$
 - $\text{kill}[n] = \{(x,c) \mid \forall c\}$
 - $\text{gen}[n] = \{(x,c) \mid c = \text{val}(y) + \text{val}(z), (y, \text{val}(y)) \in \text{in}[n], (z, \text{val}(z)) \in \text{in}[n]\}$

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

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

Next Time

Lecture

- Lattice theoretic foundation for data-flow analysis

Suggested Exercises

- exercises from book: 9.2.1, 9.2.2, 9.2.6