Using Program Analysis for Optimization

Advanced Compiler Techniques

2005

Erik Stenman

Virtutech

Reaching Definitions

- Concept of *definition* and *use*
  - `a=x+y`
    - is a definition of a.
    - is a use of x and y.
- A definition reaches a use if value written by definition may be read by use.

Reaching Definitions and Constant Propagation

- Is a use of a variable a constant?
  - Check all reaching definitions.
  - If all assign variable to same constant.
  - Then use is in fact a constant.
- Can replace variable with constant.

Computing Reaching Definitions

- Compute with sets of definitions:
  - Represent sets using bit vectors.
  - Each definition has a position in bit vector.
- At each basic block, compute:
  - Definitions that reach start of block.
  - Definitions that reach end of block.
- Do computation by simulating execution of program until the fixed point is reached.

Formalizing Analysis

- Each basic block has
  - IN - set of definitions that reach beginning of block
  - OUT - set of definitions that reach end of block
  - GEN - set of definitions generated in block
  - KILL - set of definitions killed in the block
- Compiler scans each basic block to derive GEN and KILL sets.

Dataflow Equations

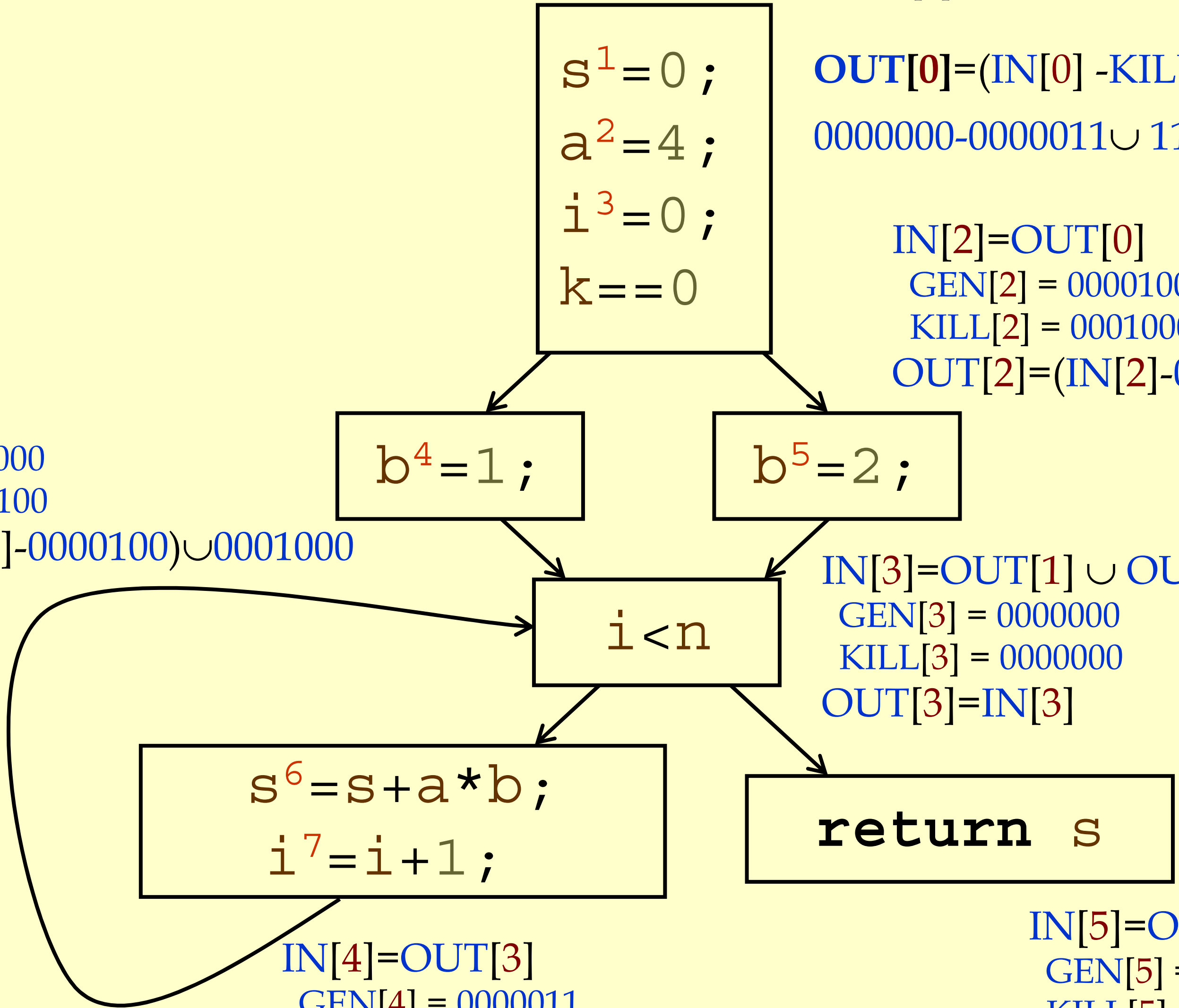
- $IN[b_i] = OUT[b_1] \cup \ldots \cup OUT[b_n]$
  where $b_1, \ldots, b_n$ are predecessors of b_i
- $OUT[b_i] = (IN[b_i] - KILL[b_i]) \cup GEN[b_i]$
- IN[entry] = 0000000
- Result: system of equations.

IN[0] = 0000000
GEN[0] = 1110000
KILL[0] = 0000011

OUT[0]=(IN[0] -KILL[0])∪GEN[0]=
0000000-0000011∪ 1110000=**1110000**

IN[2]=OUT[0]
GEN[2] = 0000100
KILL[2] = 0001000
OUT[2]=(IN[2]-0001000)∪0000100

IN[1]=OUT[0]
GEN[1] = 0001000
KILL[1] = 0000100
OUT[1]=(IN[1]-0000100)∪0001000



IN[3]=OUT[1] ∪ OUT[2]
GEN[3] = 0000000
KILL[3] = 0000000
OUT[3]=IN[3]

IN[4]=OUT[3]
GEN[4] = 0000011
KILL[4] = 1010000
OUT[4]=(IN[4]-1010000)∪0000011

IN[5]=OUT[3]
GEN[5] = 0000000
KILL[5] = 0000000
OUT[5]=IN[5]

Solving Equations

- Use fix point algorithm.
- Initialize with solution of $OUT[b_i] = 0000000$
- Repeatedly apply equations:
  - $IN[b_i] = OUT[b_1] \cup \ldots \cup OUT[b_n]$
  - $OUT[b_i] = (IN[b_i] - KILL[b_i]) \cup GEN[b_i]$
- Until reach fixed point, i.e., until equation application has no further effect.
- Use a worklist to track which equation applications may have further effect.

Reaching Definitions Algorithm

```
for all nodes n2N
   OUT[n] = ;;                              // Or OUT[n] = GEN[n];
Changed = N;                                // N = all nodes in graph
while (Changed != ;)                        // Until fixed point reached.
  choose n2Changed;                         // Node from worklist
  Changed=Changed-{n};                      // Remove from worklist
   OldOut = OUT[n]                          // Remember old result
   IN[n] = ;;                               // Calculate IN as join
   for all nodes p2predecessors(n)          //    of predecessors.
         IN[n]=IN[n]∪OUT[p];
   OUT[n]=(IN[n]-KILL[n])∪GEN[n];           // Recalculate OUT
   if (OUT[n] != OldOut)                    // If OUT[n] changed
    for all nodes s2successors(n)
         Changed=Changed∪{s};               //Add succs to worklist
```

Questions

- Does the algorithm halt?
  - yes, because transfer function is monotonic.
  - if increase IN, increase OUT.
  - in limit, all bits are 1.
- If bit is 1, is there always an execution in which corresponding definition reaches basic block?
- If bit is 0, does the corresponding definition ever reach basic block?
- Concept of conservative analysis.

Available Expressions

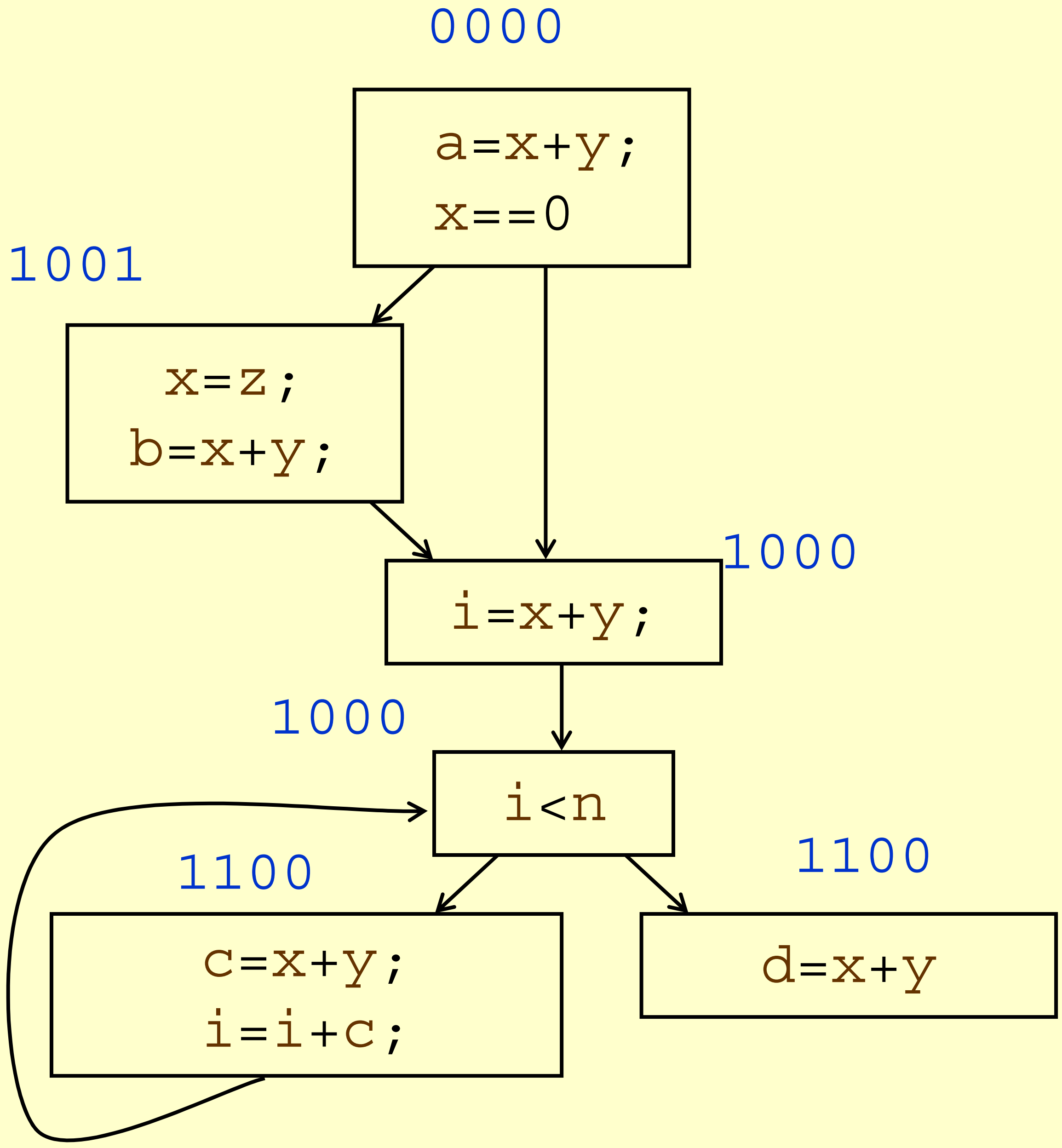
- An expression x+y is available at a point p if
  - **every** path from the initial node to p evaluates x+y before reaching p,
  - and there are **no assignments** to x or y after the evaluation but before p.
- Available Expression information can be used to do global (across basic blocks) CSE.
- If an expression is available at use, there is no need to re-evaluate it.

Computing Available Expressions

- Represent sets of expressions using bit vectors.
- Each expression corresponds to a bit.
- Run dataflow algorithm similar to reaching definitions.
- Big difference:
  - Definition reaches a basic block if it comes from **ANY** predecessor in CFG.
  - Expression is available at a basic block only if it is available from **ALL** predecessors in CFG.

Expressions

1: x+y
2: i<n
3: i+c
4: x==0



Global CSE Transform

Expressions

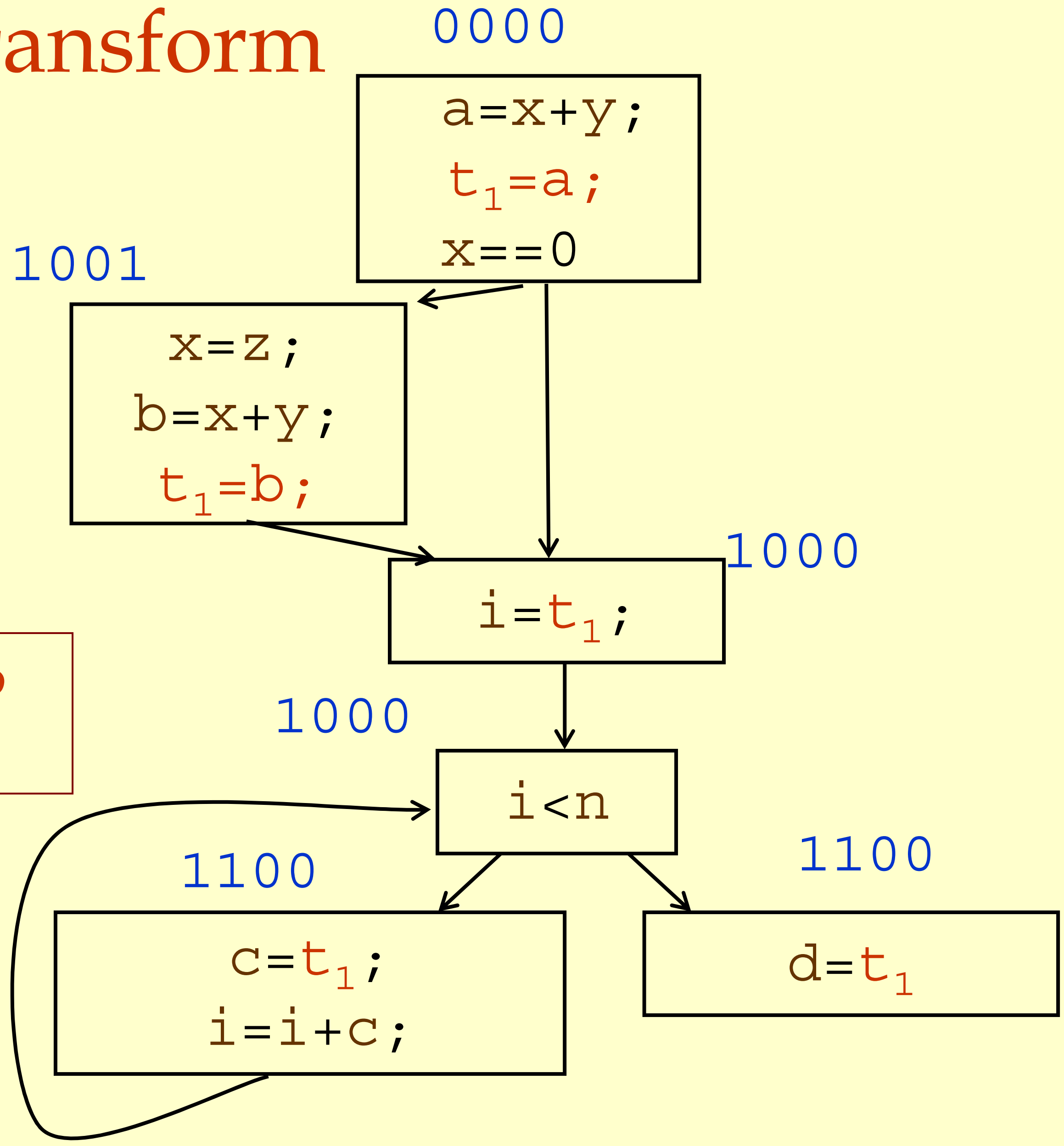
1: x+y

2: i<n

3: i+c

4: x==0

Must use same temp for CSE in all blocks



Formalizing Analysis

- Each basic block has

  IN - set of expressions that reach beginning of block.

  OUT - set of expressions that reach end of block.

  GEN - set of expressions generated in block.

  KILL - set of expressions killed in the block.

- GEN[`x=z; b=x+y`] = 1000
- KILL[`x=z; b=x+y`] = 1001
- Compiler scans each basic block to derive GEN and KILL sets.

Dataflow Equations

- $IN[b_i] = OUT[b_1] \cap \ldots \cap OUT[b_n]$
  - where $b_1, \ldots, b_n$ are predecessors of b_i
- $OUT[b_i] = (IN[b_i] - KILL[b_i]) \cup GEN[b_i]$
- $IN[entry] = 0000$
- Result: system of equations.

Solving Equations

- Use fix point algorithm.
- IN[entry]=0000
- Initialize with solution of $OUT[b_i] = 1111$
- Repeatedly apply equations:
  - $IN[b_i] = OUT[b_1] \cap \ldots \cap OUT[b_n]$
  - $OUT[b_i] = (IN[b_i] - KILL[b_i]) \cup GEN[b_i]$
- Use a worklist to track which equation applications may have further effect.

Available Expressions Algorithm

```
for all nodes n2N                    // E is set of all expressions.
   OUT[n] = E;                       // OUT[n] =E -KILL[n];
Changed = N;                         // N = all nodes in graph
while (Changed != ;)
   choose n2Changed;
   Changed=Changed-{n};
   IN[n] = E ;
   OldOut = OUT[n]
   for all nodes p2predecessors(n)
        IN[n]=IN[n]∩OUT[p];
   OUT[n]=(IN[n]-KILL[n])∪GEN[n];
   if (OUT[n] != OldOut)
    for all nodes s2successors(n) Changed=Changed∪{s};
```

Questions

- Does algorithm always halt?
- If expression is available in some execution, is it always marked as available in analysis?
- If expression is not available in some execution, can it be marked as available in analysis?
- In what sense is the algorithm conservative?

Duality In Two Algorithms

- Reaching definitions
  - Confluence operation is set **union.**
  - OUT[b] initialized to **empty set**.
- Available expressions
  - Confluence operation is set **intersection.**
  - OUT[b] initialized to **set of available expressions.**
- General framework for dataflow algorithms.
- Build parameterized dataflow analyzer once, use for all dataflow problems.

Liveness Analysis

- A variable v is live at point p if
  - v is used along some path starting at p, and
  - no definition of v along the path before the use.
- When is a variable v dead at point p?
  - No use of v on any path from p to exit node, or
  - If all paths from p, redefine v before using v.

What Use is Liveness Information?

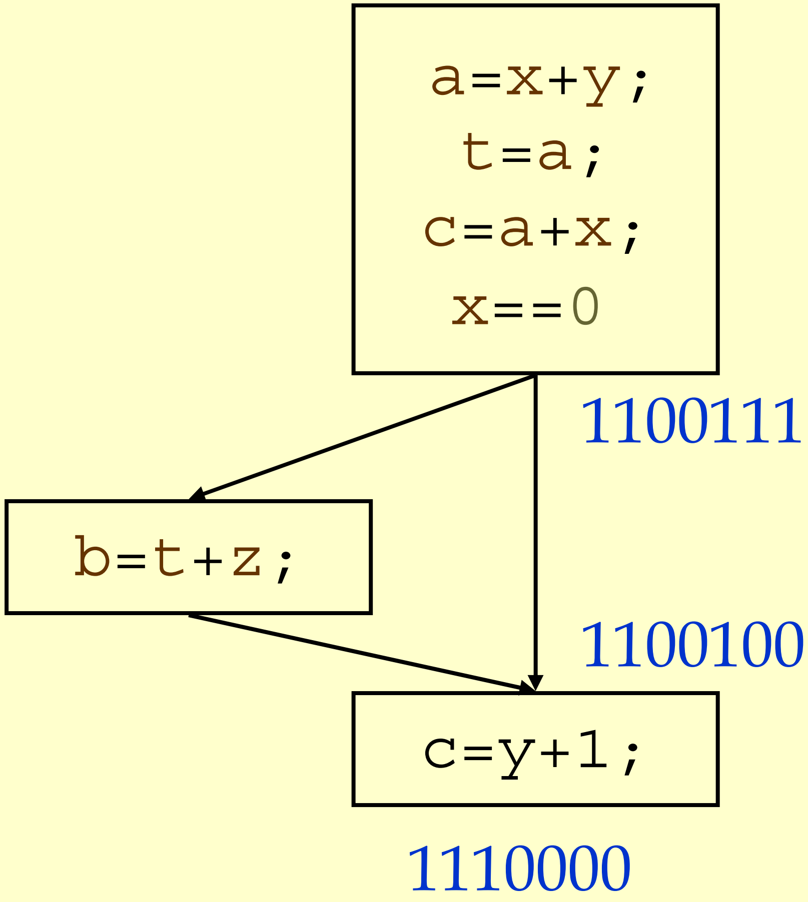
- Register allocation.
  - If a variable is dead, we can reassign its register.
- Dead code elimination.
  - Eliminate assignments to variables not read later.
  - But must not eliminate last assignment to variable (such as instance variable) visible outside CFG.
  - Can eliminate other dead assignments.
  - Handle by making all externally visible variables live on exit from CFG.

Conceptual Idea of Analysis

- Simulate execution.
- But start from exit and go **backwards** in CFG.
- Compute liveness information from end to beginning of basic blocks.

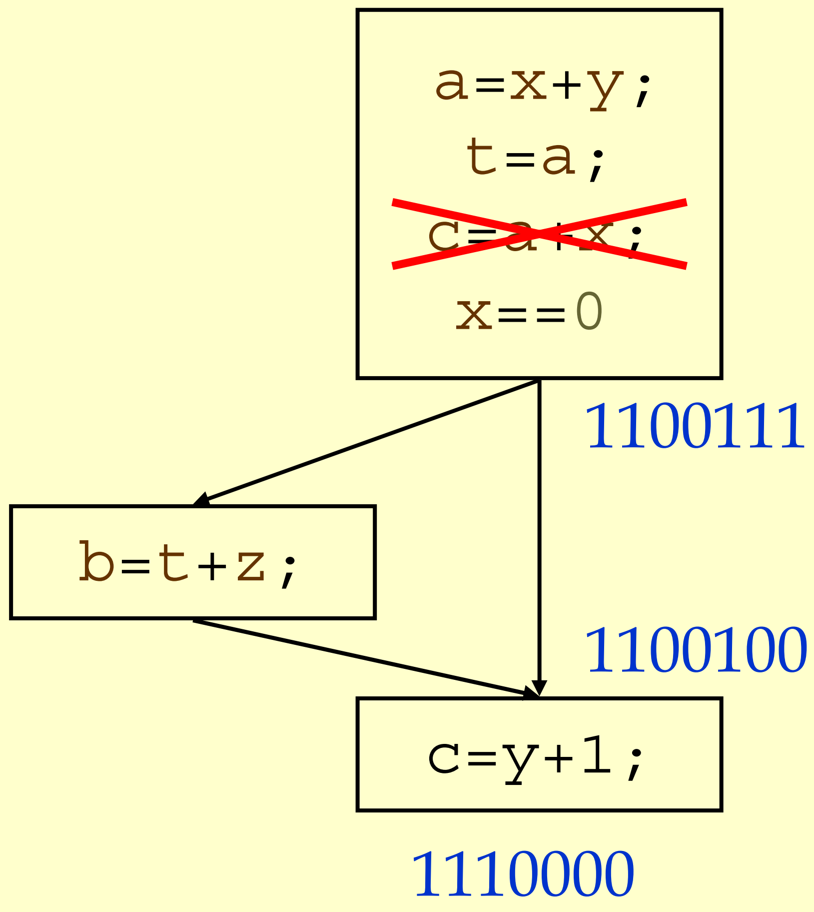
Liveness Example

- Assume `a,b,c` visible outside function. They are live on exit.
- Assume `x,y,z,t` are not visible.
- Represent liveness using a bit vector:
  order is `abcxyzt`.



Using Liveness Information for Dead Code Elimination

- Assume `a,b,c` visible outside function. They are live on exit.
- Assume `x,y,z,t` are not visible.
- Represent liveness using a bit vector:
  order is `abcxyzt`.



Formalizing Analysis

- Each basic block has
  - IN - set of variables live at start of block.
  - OUT - set of variables live at end of block.
  - USE - set of variables with upwards exposed uses in block. (**GEN**)
  - DEF - set of variables defined in block. (**KILL**)
- USE[`x=z;x=x+1;y=1;`] = {z} (x not in USE)
- DEF[`x=z;x=x+1;y=1;`] = {x, y}
- Compiler scans each basic block to derive USE and DEF sets.

Algorithm

```
OUT[Exit] = ;;
IN[Exit] = USE[n];
 for all nodes n2N-{Exit}
    IN[n] = ;;
Changed = N-{Exit};
 while (Changed != ;)
   choose n 2 Changed;
    Changed = Changed-{n};
    OldIn=IN[n]
    OUT[n] = ;;
    for all nodes s 2 successors(n) OUT[n] = OUT[n] ∪ IN[p];
    IN[n] = USE[n] ∪ (OUT[n] - DEF[n]);
    if (IN[n] != OldIn)
      for all nodes p 2 predecessors(n) Changed=Changed∪{p};
```

Similar to Other Dataflow Algorithms

- Backwards analysis, not forwards.
- Still have transfer functions.
- Still have confluence operators.
- Can generalize framework to work for both forwards and backwards analyses.

Analysis Information Inside Basic Blocks

- One detail:
  - Given dataflow information at IN and OUT of node.
  - Also need to compute information at each statement of basic block.
  - Simple propagation algorithm usually works fine.
  - Can be viewed as restricted case of dataflow analysis.

Summary

- Dataflow Analysis
  - Control flow graph.
  - IN[b], OUT[b], transfer functions, join points.
- Pairs of analyses and transformations:
  - Reaching definitions/constant propagation.
  - Available expressions/common sub-expression elimination.
  - Liveness analysis/Dead code elimination.