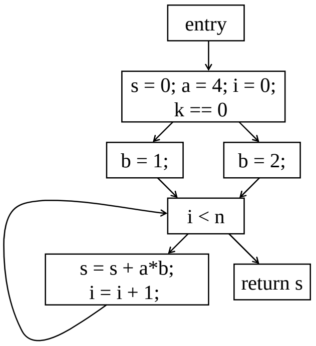
Using Program Analysis for Optimization

Analysis and Optimizations

- Program Analysis
  - Discovers properties of a program
- Optimizations
  - Use analysis results to transform program
  - Goal: improve some aspect of program
    - number of executed instructions, number of cycles
    - cache hit rate
    - memory space (code or data)
    - power consumption
  - Has to be safe: Keep the semantics of the program

Control Flow Graph

```
int add(n, k) {
    s = 0; a = 4; i = 0;
    if (k == 0) b = 1;
    else b = 2;
    while (i < n) {
        s = s + a*b;
        i = i + 1;
    }
    return s;
}
```



Control Flow Graph

- Nodes represent computation
  - Each node is a Basic Block
  - A Basic Block is a sequence of instructions with
    - No branches out of middle of basic block
    - No branches into middle of basic block
    - Basic blocks should be maximal
  - Execution of basic block starts with first instruction
  - Includes all instructions in basic block
- Edges represent control flow

Two Kinds of Variables

- Temporaries introduced by the compiler
  - Transfer values only within basic block
  - Introduced as part of instruction flattening
  - Introduced by optimizations/transformations
- Program variables
  - Declared in original program
  - May transfer values between basic blocks

Basic Block Optimizations

- Common Sub-Expression Elimination
  - a = (x+y)+z; b = x+y;
  - t = x+y; a = t+z; b = t;
- Constant Propagation
  - x = 5; b = x+y;
  - b = 5+y;
- Algebraic Simplification
  - a = x * 1;
  - a = x;
- Copy Propagation
  - a = x+y; b = a; c = b+z;
  - a = x+y; b = a; c = a+z;
- Dead Code Elimination
  - a = x+y; b = a; c = a+z;
  - a = x+y; c = a+z
- Strength Reduction
  - t = i * 4;
  - t = i << 2;

Value Numbering

- Normalize basic block so that all statements are of the form
  - var = var op var (where op is a binary operator)
  - var = op var (where op is a unary operator)
  - var = var
- Simulate execution of basic block
  - Assign a virtual value to each variable
  - Assign a virtual value to each expression
  - Assign a temporary variable to hold value of each computed expression

Value Numbering for CSE

- As we simulate execution of program
- Generate a new version of program
  - Each new value assigned to temporary

    a = x+y; becomes a = x+y; t = a;
  - Temporary preserves value for use later in program even if original variable is rewritten

    a = x+y; a = a+z; b = x+y

    becomes

    a = x+y; t = a; a = a+z; b = t;

CSE Example

- Original

  a = x+y
  b = a+z
  b = b+y
  c = a+z
- After CSE

  a = x+y
  b = a+z
  t = b
  b = b+y
  c = t
- Issues
  - Temporaries store values for use later
  - CSE with different names

    a = x; b = x+y; c = a+y;
  - Excessive generation and use of temporaries

Original Basic Block

a = x+y
b = a+z
b = b+y
c = a+z

New Basic Block

a = x+y
t1 = a
b = a+z
t2 = b
b = b+y
t3 = b
c = t2

Var to Val

x → v1
y → v2
a → v3
z → v4
b → v6
c → v5

Exp to Val

v1+v2 → v3
v3+v4 → v5
v5+v2 → v6

Exp to Tmp

v1+v2 → t1
v3+v4 → t2
v5+v2 → t3

Problems

- Algorithm has a temporary for each new value
  - a = x+y; t1 = a
- Introduces
  - lots of temporaries
  - lots of copy statements to temporaries
- In many cases, temporaries and copy statements are unnecessary
- So we eliminate them with copy propagation and dead code elimination

Copy Propagation

- Once again, simulate execution of program
- If possible, use the original variable instead of a temporary
  - a = x+y; b = x+y;
  - After CSE becomes a = x+y; t = a; b = t;
  - After CP becomes a = x+y; t = a; b = a;
- Key idea: determine when original variables are NOT overwritten between computation of stored value and use of stored value

Copy Propagation Maps

- Maintain two maps
  - tmp to var: tells which variable to use instead of a given temporary variable
  - var to set (inverse of tmp to var): tells which temps are mapped to a given variable by tmp to var

Copy Propagation Example

Original	After CSE	After CSE and Copy Propagation
a = x+y	a = x+y	a = x+y
b = a+z	t1 = a	t1 = a
c = x+y	b = a+z	b = a+z
a = b	t2 = b	t2 = b
	c = t1	c = a
	a = b	a = b

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a

tmp to var	var to set
t1 → a	a →{t1}

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a
b = a+z	b = a+z
t2 = b	t2 = b

tmp to var	var to set
t1 → a	a →{t1}
t2 → b	b →{t2}

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a
b = a+z	b = a+z
t2 = b	t2 = b
c = t1	

tmp to var	var to set
t1 → a	a →{t1}
t2 → b	b →{t2}

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a
b = a+z	b = a+z
t2 = b	t2 = b
c = t1	**c = a**

tmp to var	var to set
t1 → a	a →{t1}
t2 → b	b →{t2}

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a
b = a+z	b = a+z
t2 = b	t2 = b
c = t1	c = a
a = b	a = b

tmp to var	var to set
t1 → a	a →{t1}
t2 → b	b →{t2}

Copy Propagation Example

Basic Block After CSE	Basic Block After CSE and Copy Prop
a = x+y	a = x+y
t1 = a	t1 = a
b = a+z	b = a+z
t2 = b	t2 = b
c = t1	c = a
a = b	a = b

tmp to var	var to set
t1 → t1	a →{}
t2 → b	b →{t2}

Dead Code Elimination

- Copy propagation keeps all temps around
- There may be temps that are never read
- Dead Code Elimination (DCE) removes them

Basic Block After CSE + Copy Prop	Basic Block After CSE + Copy Prop + DCE
a = x+y	a = x+y
t1 = a	b = a+z
b = a+z	c = a
t2 = b	a = b
c = a	
a = b	

Dead Code Elimination

- Basic Idea
  - Process code in *reverse* execution order
  - Maintain a set of variables that are needed later in computation
  - On encountering an assignment to a temporary that is not needed, we remove the assignment

Basic Block After
CSE + Copy Propagation

a = x+y
t1 = a
b = a+z
t2 = b
c = a
⟹ a = b

Assume that initially Needed Set
{a, c}

Basic Block After
CSE + Copy Propagation

a = x+y
t1 = a
b = a+z
t2 = b
⟹ c = a
a = b

Needed Set
{a, b, c}

Basic Block After CSE + Copy Propagation

a = x+y
t1 = a
b = a+z
t2 = b
⇒ c = a
a = b

Needed Set
{b, c}

Basic Block After CSE + Copy Propagation

a = x+y
t1 = a
b = a+z
⇒ t2 = b
c = a
a = b

Needed Set
{a, b}

Basic Block after CSE + Copy Propagation + Dead Code Elimination

a = x+y
t1 = a
b = a+z
⇒
c = a
a = b

Needed Set
{a, b}

Basic Block after CSE + Copy Propagation + Dead Code Elimination

a = x+y
t1 = a
⇒ b = a+z

c = a
a = b

Needed Set
{a, b, z}

Basic Block after CSE + Copy Propagation + Dead Code Elimination

a = x+y
⇒ t1 = a
b = a+z

c = a
a = b

Needed Set
{a, z}

Basic Block after CSE + Copy Propagation + Dead Code Elimination

a = x+y
⇒
b = a+z

c = a
a = b

Needed Set
{a, z}

Basic Block after
CSE + Copy Propagation + Dead Code Elimination

⇒ a = x+y

b = a+z

c = a
a = b

Needed Set
{a, z}

Basic Block after
CSE + Copy Propagation + Dead Code Elimination

a = x+y

b = a+z

c = a
a = b

Needed Set
{z, x, y}

Interesting Properties

- Analysis and Optimization algorithms simulate execution of program
  - CSE and Copy Propagation go forward
  - Dead Code Elimination goes backwards
- Optimizations are stacked
  - Group of basic transformations
  - Work together to get good result
  - Often, one transformation creates inefficient code that is cleaned up by subsequent transformations

Other Basic Block Transformations

- Constant Propagation
- Strength Reduction
  - a << 2 = a * 4;
  - a + a + a = 3 * a;
- Algebraic Simplification
  - a = a * 1;
  - b = b + 0;
- Need a unified transformation framework

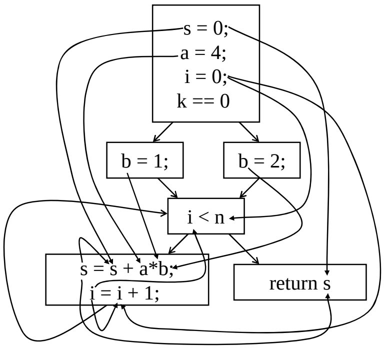
Dataflow Analysis

- Used to determine properties of programs that involve multiple basic blocks
- Typically used to enable transformations
  - common sub-expression elimination
  - constant and copy propagation
  - dead code elimination
- Analysis and transformation often come in pairs

Reaching Definitions

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

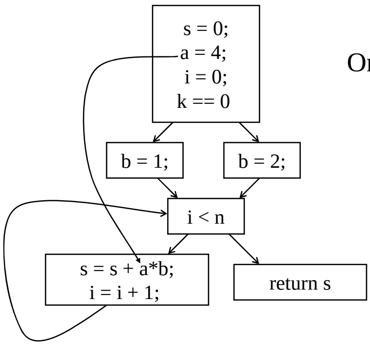
Reaching Definitions



Reaching Definitions and Constant Propagation

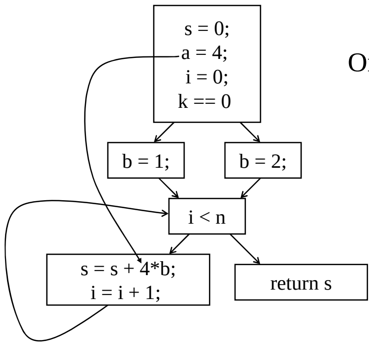
- Is a use of a variable a constant?
  - Check all reaching definitions
  - If all assign variable to same constant
  - Then use is in fact a constant
- Can replace variable with constant

Is a constant in s = s+a*b?



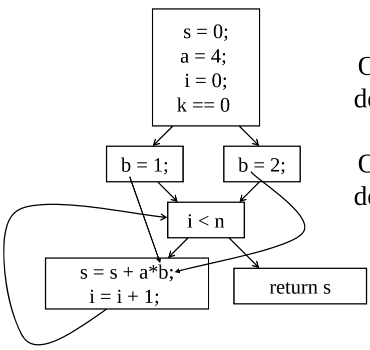
Yes!
On all reaching definitions
a = 4

Constant Propagation Transform



Yes!
On all reaching definitions
a = 4

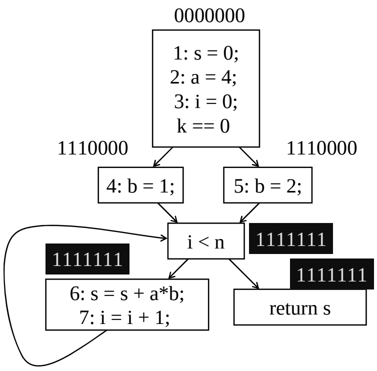
Is b constant in s = s+a*b?



No!
One reaching definition with
b = 1
One reaching definition with
b = 2

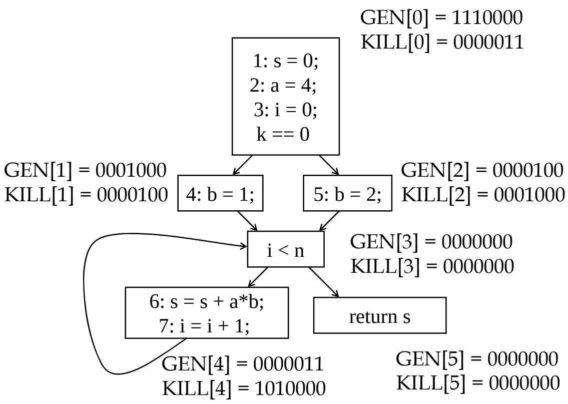
Computing Reaching Definitions

- 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
- GEN[s = s + a*b; i = i + 1;] = 0000011
- KILL[s = s + a*b; i = i + 1;] = 1010000
- Compiler scans each basic block to derive GEN and KILL sets



Dataflow Equations

- IN[b] = OUT[b1] ∪ ... ∪ OUT[bn]
  - where b1, ..., bn are predecessors of b in CFG
- OUT[b] = (IN[b] - KILL[b]) ∪ GEN[b]
- IN[entry] = 0000000
- Result: system of equations

Solving Equations

- Use fixed point algorithm
- Initialize with solution of OUT[b] = 0000000
- Repeatedly apply equations
  - IN[b] = OUT[b1] ∪ ... ∪ OUT[bn]
  - OUT[b] = (IN[b] - KILL[b]) ∪ GEN[b]
- Until reaching fixed point
  - I.e., until equation application has no further effect
- Use a worklist to track which equation applications may have a further effect

Reaching Definitions Algorithm

```
for all nodes n in N OUT[n] = ∅;      // OUT[n] = GEN[n];
Worklist = N;                          // N = all nodes in graph
while (Worklist != ∅)
  choose a node n in Worklist;
  Worklist = Worklist - { n };
  IN[n] = ∅;
  for all nodes p in predecessors(n)  IN[n] = IN[n] ∪ OUT[p];
  OUT[n] = (IN[n] - KILL[n]) ∪ GEN[n];
  if (OUT[n] changed)
    for all nodes s in successors(n) Worklist = Worklist ∪ { s };
```

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

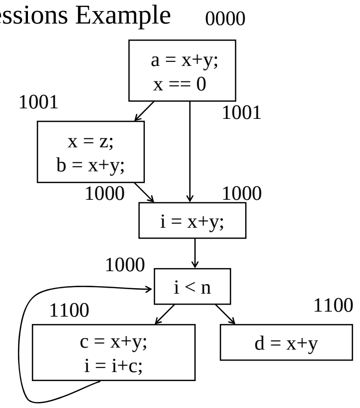
- An expression x+y is available at a point p if
  - every path from the initial node to p evaluates x+y before reaching p
  - and there are no assignments to x or y after the evaluation but before p
- Available Expression information can be used to do global (across basic blocks) CSE
- If an expression 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:
  - A definition reaches a basic block if it comes from ANY predecessor in CFG
  - An expression is available at a basic block only if it is available from ALL predecessors in CFG

Available Expressions Example

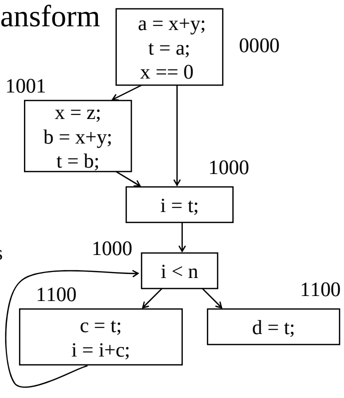
Expressions
1: x+y
2: i < n
3: i+c
4: x == 0



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 available at start of block
  - OUT - set of expressions available at end of block
  - GEN - set of expressions computed 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] = OUT[b1] ∩ ... ∩ OUT[bn]
  - where b1, ..., bn are predecessors of b in CFG
- OUT[b] = (IN[b] - KILL[b]) ∪ GEN[b]
- IN[entry] = 0000
- Result: system of equations

Solving Equations

- Use fixed point algorithm
- IN[entry] = 0000
- Initialize OUT[b] = 1111
- Repeatedly apply equations
  - IN[b] = OUT[b1] ∩ ... ∩ OUT[bn]
  - OUT[b] = (IN[b] - KILL[b]) ∪ GEN[b]
- Use a worklist algorithm to track which equation applications may have further effect

Available Expressions Algorithm

```
for all nodes n in N OUT[n] = E;          // OUT[n] = E - KILL[n];
IN[Entry] = ∅; OUT[Entry] = GEN[Entry];
Worklist = N - { Entry };                 // N = all nodes in graph
 while (Worklist != ∅)
   choose a node n in Worklist;
   Worklist = Worklist - { n };
   IN[n] = E;                             // E is set of all expressions
   for all nodes p in predecessors(n)
        IN[n] = IN[n] ∩ OUT[p];
   OUT[n] = (IN[n] - KILL[n]) ∪ GEN[n];
   if (OUT[n] changed)
     for all nodes s in successors(n) Worklist = Worklist ∪ { 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 analysis 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?

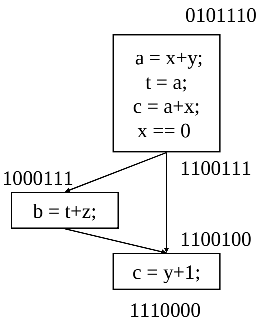
- Register allocation
  - If a variable is dead, we can reassign its register
- Dead code elimination
  - Eliminate assignments to variables not read later
  - But must not eliminate last assignment to variable (such as instance variable) visible outside CFG
  - Can eliminate other dead assignments
  - 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

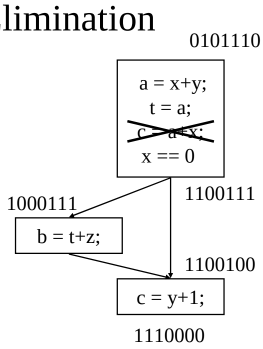
Liveness Example

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



Using Liveness Information for Dead Code Elimination

- Assume a,b,c visible outside function and thus are live on exit
- Assume x,y,z,t are not visible on exit
- 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
  - DEF - set of variables defined in block
- USE[x = z; x = x+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 n in N - { Exit } IN[n] = ∅;
Worklist = N - { Exit };
 while (Worklist != ∅)
   choose a node n in Worklist;
   Worklist = Worklist - { n };
   OUT[n] = ∅;
   for all nodes s in successors(n) OUT[n] = OUT[n] ∪ IN[s];
    IN[n] = USE[n] ∪ (OUT[n] - DEF[n]);
   if (IN[n] changed)
     for all nodes p in predecessors(n) Worklist = Worklist ∪{ 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

- Basic blocks and basic block optimizations
  - Copy and constant propagation
  - Common sub-expression elimination
  - Dead code elimination
- Dataflow Analysis
  - Control flow graph
  - IN[b], OUT[b], transfer functions, join points
- Paired of analyses and transformations
  - Reaching definitions/constant propagation
  - Available expressions/common sub-expression elimination
  - Liveness analysis/Dead code elimination