CS 4240: Compilers

Lecture 7: Constant Propagation, Copy Propagation

Instructor: Vivek Sarkar (vsarkar@gatech.edu)

January 30, 2019

REMINDERS

- Homework 1 due TODAY by 11:59pm
  - Must be submitted as PDF file on Canvas

- Project 1 was released on Wednesday (1/16/19) on Piazza
  - Due by 11:59pm on Wednesday, 2/13/19 on Canvas
  - Must be submitted as zip file including instructions on how to build and run your project
  - 100 points total, with an extra credit option for 15 points
    - Extra credit relates to use of copy propagation, which we will study today
  - 5% of course grade
  - We will hold in-class help session for Project 1 on Feb 6th, led by the TAs

- MIDTERM EXAM: Wednesday, March 13, 4:30pm - 5:45pm
- FINAL EXAM: Wednesday, May 1, 2:40 pm - 5:30 pm

Worksheet – 6 Solution

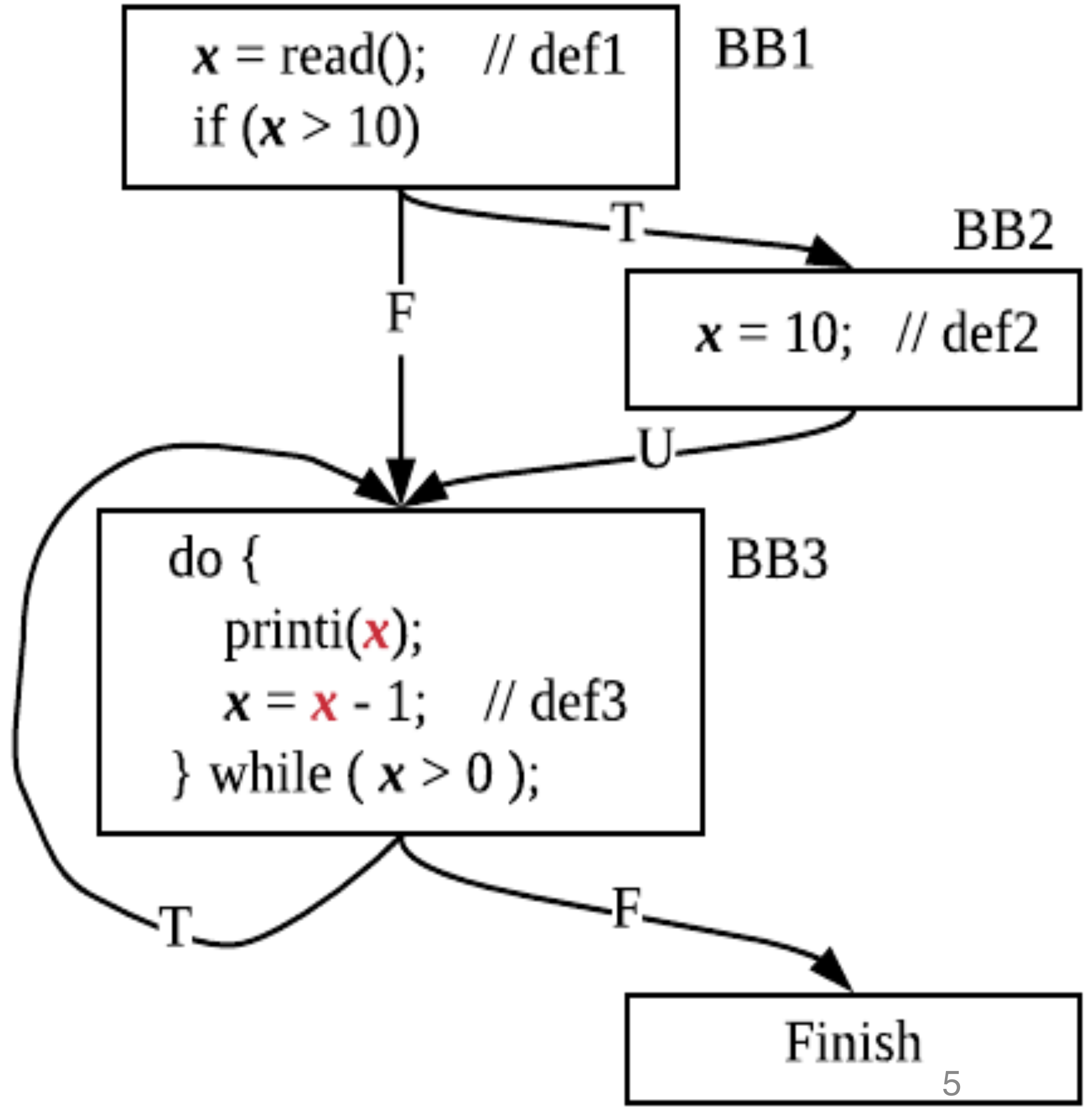
From lecture given on 01/28/2019

1. Convert the following code to **Static Single Assignment (SSA)** form. You can show your transformed code in source code format, IR format, or as a control flow graph with source/IR statements in each basic block.

```
x = read();           // read x from stdin
if (x > 10) x = 10;
do {
    printi(x);        // write x to stdout
    x = x - 1;
} while ( x > 0);
```

Control flow Graph

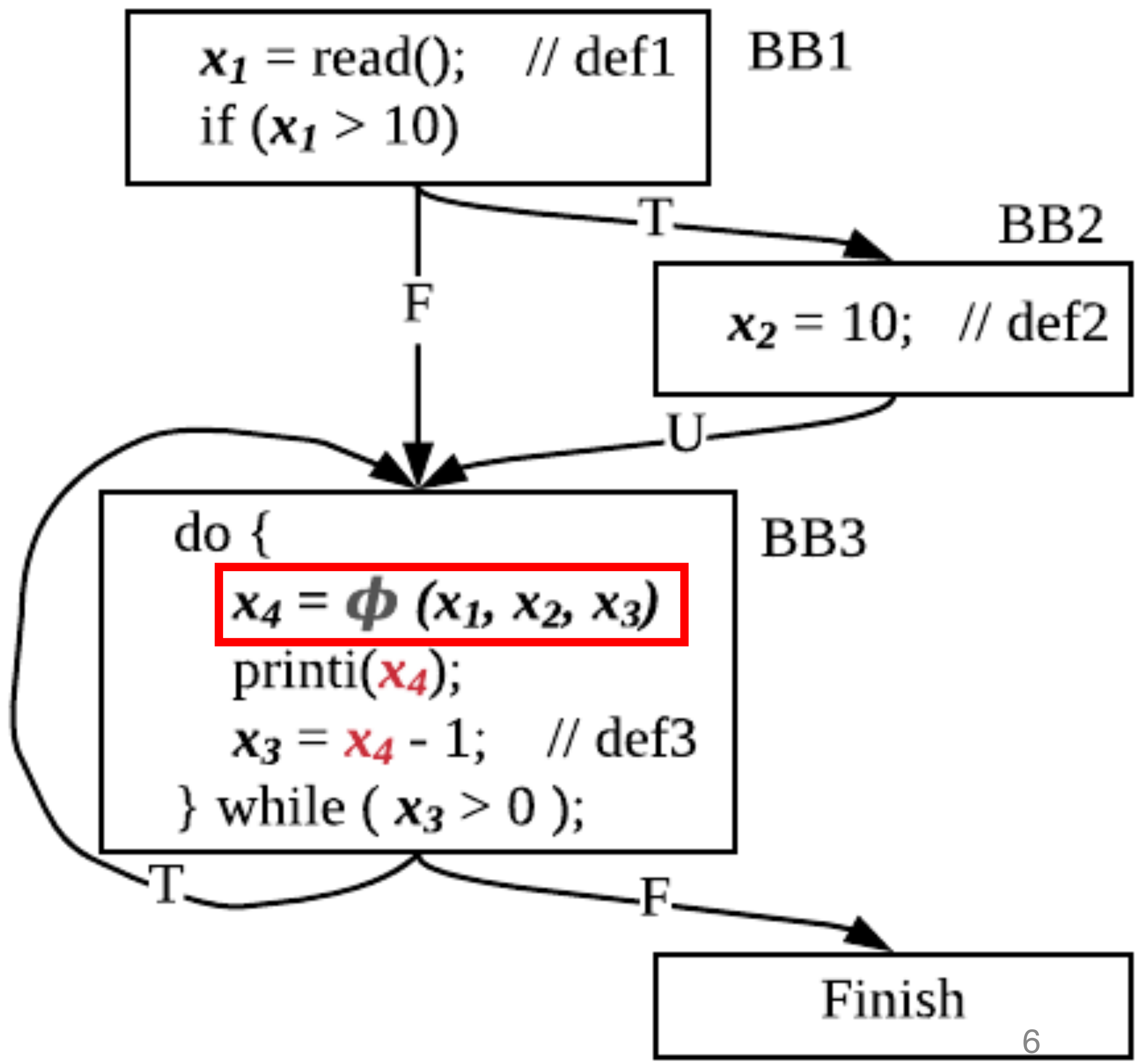
- Observation: def1, def2, def3 all reach uses of x in BB3



Sample Solution #1

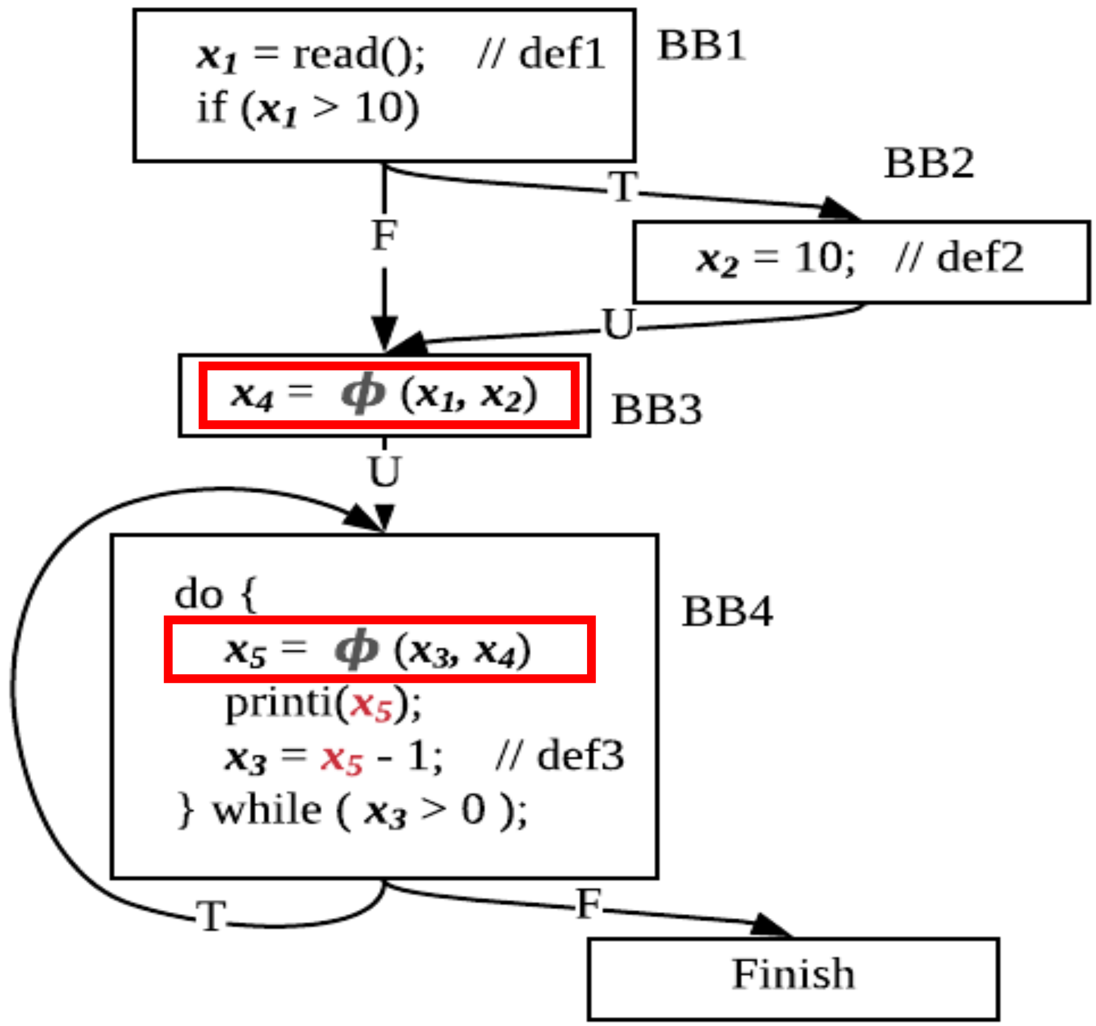
For each def of x, introduce a new variable with an incremented subscript.

At joints where multiple **reaching definitions** of x merge, create a new variable using a Phi - function.



Sample Solution #2

This solution includes two phi functions, and is closer to what a compiler might generate. However, both solutions are equally acceptable.



Constant Propagation

- Goal: Produce an algorithm that will propagate all constants in a procedure, replacing constant expressions with the result of evaluating the expression at compile time
- Strategy:
  - Construct def-use chains to map from definitions to uses within a procedure
  - Propagate constants forward from points of constant definitions along def-use chains
  - Evaluate new constant expressions whenever they are identified
  - Stop when no more constants are available
- Challenges
  - Constructing def-use chains
    - you already know how to do that with reaching definitions!
  - Identifying constant expressions in the presence of multiple reaching definitions

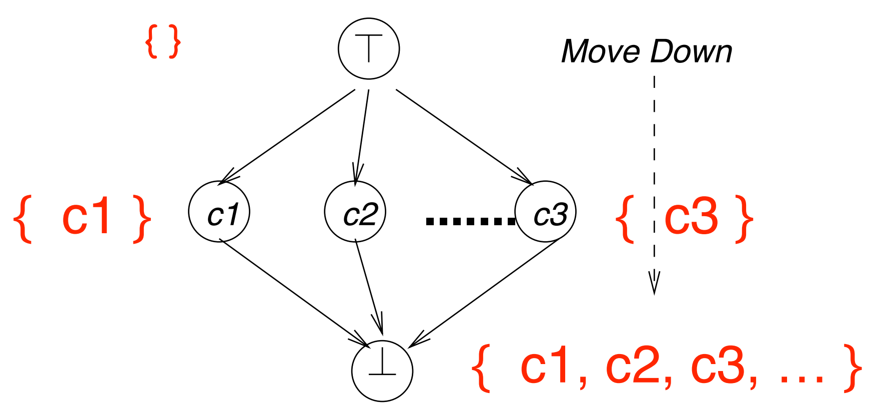
The Rationale

- Expressions that compute constants can be replaced appropriately by assignments and/or eliminated thereby avoiding costly run-time evaluations
- This leads (in an obvious way) to better performance of the compiled code
- Based on analysis of the control variables, code that is never executed can be deleted
- This simplifies the control flow and aids other optimizations

The Lattice Structure

A lattice value denotes a set of possible constant values.
We are interested in the case when the set is a singleton.

- We have a unique symbol $\perp$ representing the fact that a contant value *cannot* be guaranteed
- Several (potentially unbounded number of) constant symbols $\mathcal{C}_i$ that denote the space of all possible constants
- These constants are dominated by a unique $\top$ symbol that represents that fact that the corresponding variable/expression to which it is assigned *may* potentially be reducible to a constant



The Intuition

- We start out with all the nodes being assigned $\top$

- The idea is to move down the lattice towards $\perp$ and see whether the analysis stabilizes at a constant $\mathcal{C}_i$ in between or whether it reaches $\perp$

- The rules for combination are as follows where *Anysymbol* denotes $\top, \perp$ or one of the constants $\mathcal{C}_i$

  1. $Anysymbol \sqcap \top = Anysymbol$
  2. $Anysymbol \sqcap \perp = \perp$
  3. $\mathcal{C}_i \sqcap \mathcal{C}_i = \mathcal{C}_i$
  4. $\mathcal{C}_i \sqcap \mathcal{C}_j = \perp, i \neq j$

The meet operator, $\sqcap$,

corresponds to the set union

operation on sets denoted

by the lattice values. It is performed at a ny point with two or more reaching definitions for the same variable (a phi function in SSA form!)

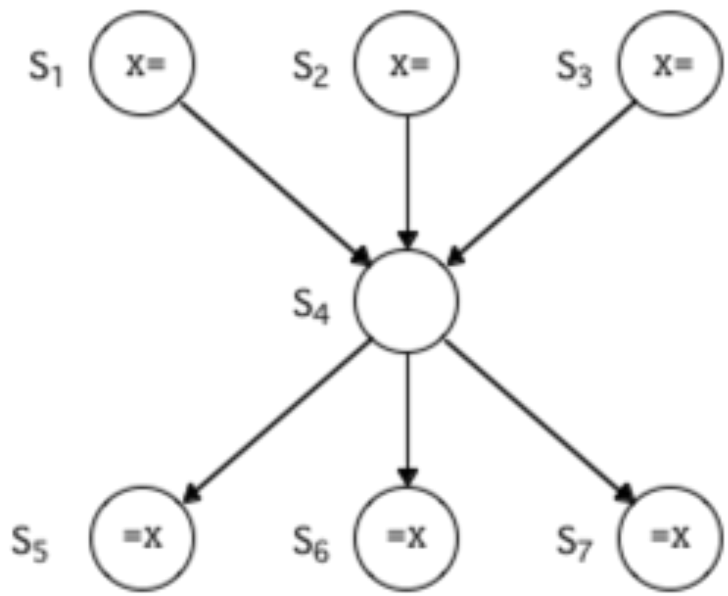
Constant Propagation Algorithm

```
for all statements s in the program do begin
      for each output v of s do valout(v,s) := unknown;
      for each input w of s do
            if w is a variable then valin(w,s) := unknown;
            else valin(w,s) := the constant value of w;
end
worklist := {all statements of constant form, e.g., X = 5};
while worklist ≠ ∅ do begin
      choose and remove an arbitrary statement S from worklist;
      let v denote the output variable for S;
      newval := meet of valin(v,S), for all inputs v to S;
      if newval ≠ valout(v,S) then begin
            valout(v,S) := newval;
            for all (S,S2) ∈ DefUse do begin
                  oldval := valin(v,S2);
                  valin(v,S2) := meet of oldval and valout(v,S1);
                  if valin(v,S2) ≠ oldval then worklist := worklist ∪ {S2};
            end
      end
end
```

Advantages and Disadvantages

- Advantages
  - Linear in the size of the Def-Use graph
    - → Why?

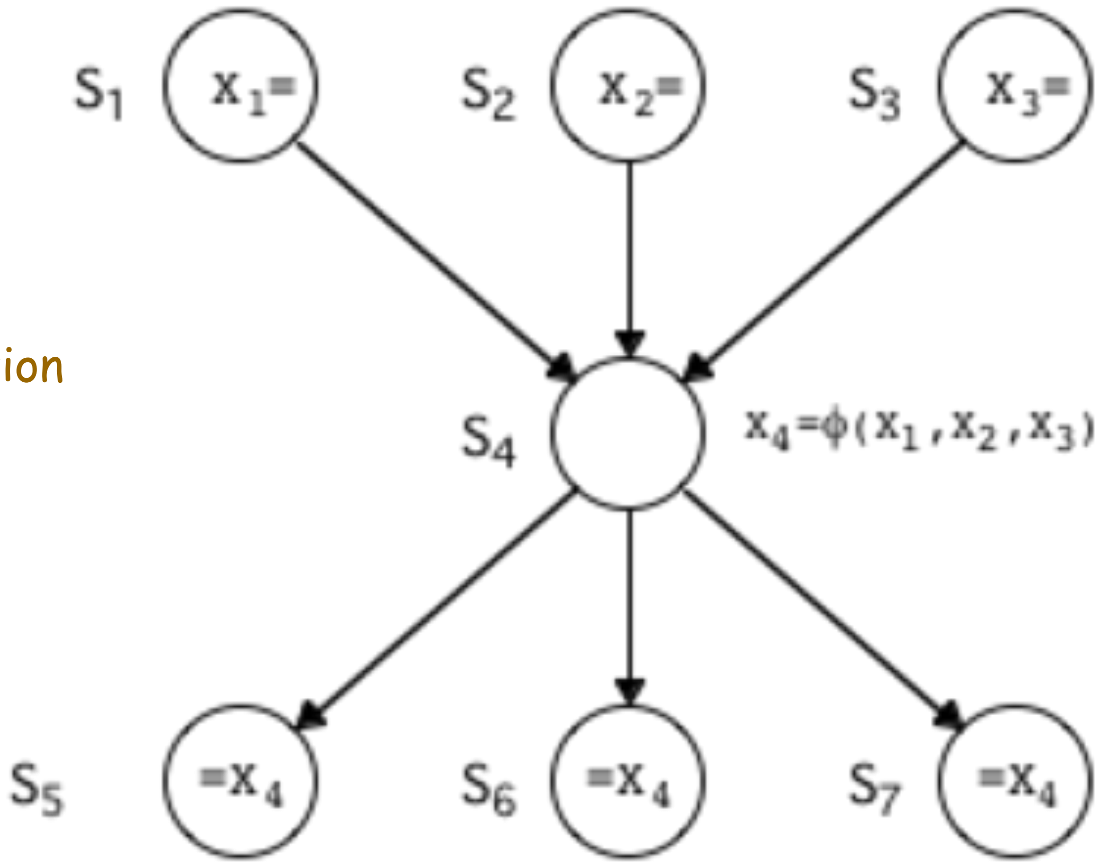
- Disadvantage
  - Def-Use graph could be large



Shrinking the Graph: SSA

» Static Single-Assignment Form

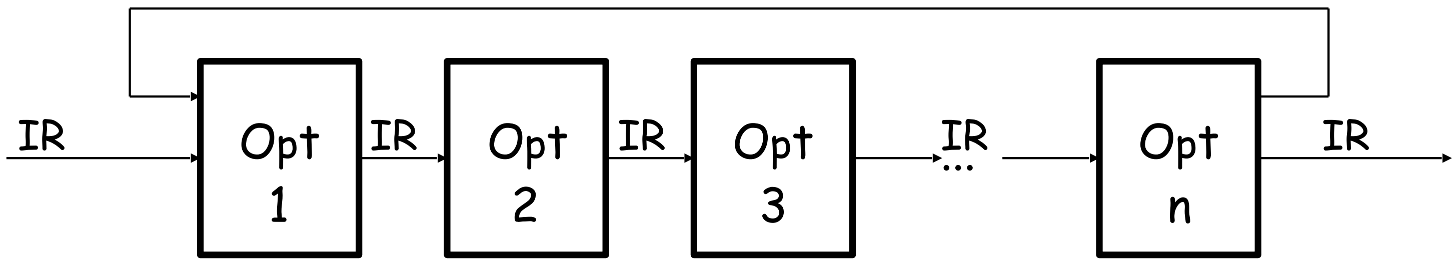
At most one definition reaches each use!



Algorithms on SSA

- Dead code elimination and constant propagation work unchanged, with appropriate assumptions for ϕ-functions;
  - The edge set should be much smaller, so the algorithms should run faster
- Many other algorithms can exploit the single-assignment property
  - What about value numbering?
  - Since each value has a unique name, you can do value numbering in SSA form with general control flow (going beyond available expression analysis)

Modern optimizers are structured as a series of passes



- Middle-end performs multiple optimization passes on an IR
- Combining optimizations to guarantee an optimal fixpoint can be challenging
- The same optimization can be performed multiple times

Typical Transformations

- Discover & propagate some constant value
- Move a computation to a less frequently executed place
- Specialize some computation based on context
- Discover a redundant computation & remove it
- Remove useless or unreachable code
- Encode an idiom in some particularly efficient form

Detecting Unreachable Code

```
I ⟵ 1
∘ ∘
If I = 1
    then j ⟵ 1
    else j ⟵ 2
```

- Note that control variable i is a constant
- Traditional data flow analysis would not catch the fact that
  - j is a constant and
  - $j \leftarrow 2$ is *never executed*; this represents *unreachable code* which is a variety of dead-code

  because it does not factor in the outcome of control instructions such as conditional branches

With care, many data flow analyses can be combined with unreachable code elimination for increased precision. More on this in the next lecture!

Copy Propagation

- What does it mean?
  - Given an assignment x = y, replace later uses of x with uses of y, provided there are no intervening assignments to x or y.
- When is it performed?
  - Can be performed as a clean-up pass after each optimization that introduces one or more copy statements
- What is the result?
  - Smaller code, possibly more efficient code depending on interaction with other optimizations

- Slides source: https://www.cs.northwestern.edu/academics/courses/322/notes/14.ppt

Local Copy Propagation

- Local copy propagation
  - Performed within basic blocks
  - Algorithm sketch:
    - traverse BB from top to bottom
    - maintain table of copies encountered so far
    - modify applicable instructions as you go

Local Copy Propagation (Example)

Example: Local copy propagation on basic block:

b = a
c = b + 1
d = b
b = d + c
b = d

step	instruction	updated instruction	table contents
1	b = a	b = a	{(b,a)}
2	c = b + 1	c = a + 1	{(b,a)}
3	d = b	d = a	{(b,a), (d,a)}
4	b = d + c	b = a + c	{(d,a)}
5	b = d	b = a	{(d,a), (b,a)}

Note: if there was a definition of 'a' between 3 and 4, then we would have to remove (b,a) and (d,a) from the table. As a result, we wouldn't be able to perform local copy propagation at instructions 4 and 5. However, this will be taken care of when we perform global copy propagation.

Local Copy Propagation (Algorithm)

- Algorithm sketch for a basic block containing instructions i_1, i_2, ..., i_n

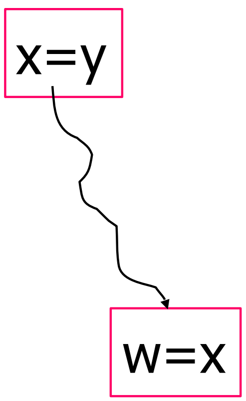
```
for instr = i1  to  in
    if  instr is of the form 'res = opd1  op  opd2'
        opd1 = REPLACE(opd1, copytable)
        opd2 = REPLACE(opd2, copytable)
    else if instr is of the form 'res = var'
        var = REPLACE(var, copytable)
    if instr has a lhs res,
         REMOVE from copytable all pairs involving res.
    if instr is of the form 'res = var' /* i.e. a copy */
              insert {(res, var2)} in the copytable
endfor
```

copytable is table containing copy pairs
e.g. if there's an assignment x := a, then
copytable should contain
(x,a)

```
REPLACE(opd, copytable)
    if you find (opd, x) in copytable /* use hashing for faster access */
         return x
    else return opd
```

Copy Propagation

- <u>Global</u> copy propagation
  - Performed on flow graph.
  - Given copy statement **x=y** and use **w=x**, we can replace **w=x** with **w=y** only if the following conditions are met:
    1. **x=y** must be the only definition of **x** reaching **w=x**
       - This can be determined by the output of reaching definitions (or by using SSA form)
    2. There may be no definitions of **y** on any path from **x=y** to **w=x**.
       - Use iterative data flow analysis to solve this.
       - Can be combined with reaching definitions analysis



Copy Propagation

- Data flow analysis to determine which instructions are candidates for global copy propagation
  - **gen**[Bi] = {(**x**,**y**,i,p) | p is the position of **x**=**y** in block Bi and neither **x** nor **y** is assigned a value after p}
  - **kill**[Bi] = {(**x**,**y**,j,p) | **x**=**y**, located at position p in block Bj≠Bi, is killed due to a definition of **x** or **y** in Bi }
  - **in**[B]=∩**out**[P] over all predecessors P of B
  - Initialize **in**[B1]=∅, **in**[B]= Universal set, for B≠B1

```
p: x = y
...
...
...
```

generate x=y if no definitions of x or y in this area

```
...
q: x = z
...
s: y = w
```

kill all other definitions of x

kill all other definitions of y

Copy Propagation reveals opportunities for Dead Code Elimination