Farzon Lotfi
CS 6241
Pande

HW 2
Q1 *Consider the problem of detecting available expressions for CSE.*

(a) Develop and reason about the safety condition for the analysis considering the optimization of common subexpression elimination in terms of actual vs found by the analysis.
We need to break this problem into two sets local and global cse. In global cse we look for *availability* a specific form of data flow analysis to find available expressions. If an expression is determined to be available then it does not need to be recomputed. As an an example x+y is considered available at a point p if every path to p has no subsequent assignments to x or y. Every available expression at the end of each basic block forms an outset. An outset of a basic block is:

out[b] = gen[b] U (in[b] - kill[b])
more simply: OUT = GEN + (IN – KILL)

Assuming a transfer function that propagates forward. The kill set is made up of all the assignments of x or y from our example above. The gen set is all the evaluations of x+y.

This is safe b\c the availability algorithm is a conservative solution where “all the expressions found to be available really are available” (Dragon Book 616). To restate that safety is guaranteed for CSE when we check for availability b\c availability has to be evaluated on all paths to our consideration node p in the example above. This is done with an intersect function.
Where the in set of the next block from X from Y equals:
IN(X) = Intersect(OUT(Y)) for all predecessors Y of X

Given the above analysis the actual expressions we can eliminate at any given point will be the sets of a basic block that we did not generate within this block i.e. IN minus KILL. while the found will just be OUT.

Local cse is cse within a single block here we can use value numbering which works like a hashmap for expressions. For the

```
Ex: 1:  r1 = r2 * r3
            …
    n:  r4=r2*r3
```

For the example above instruction 1 and n should have the same opcodes. If instruction 1 is a load then there should be no stores that write to r2 or r3 between instruction 1 and n. The actual vs found concept does not play in as much with local cse b\c we do not build up a set list to check against when there are no edges to analyze, its easy enough to just has a running expression look up. So you could say actual = found for local cse. You can also solve this using just finding your available expressions by checking what expressions that were generated in the block you were looking at make it to your OUT set.

Cite: http://www.cs.cmu.edu/afs/cs/academic/class/15745-f03/public/lectures/L6_handouts.pdf
& Dragon book 9.2.6

(b) In the presence of the pointers, revise the data-flow framework for Available Expressions for all different cases of points-to sets as discussed in class. Using the revised framework, show how you meet the safety conditions developed in (a). Illustrate via an example.

Pointer analysis can be addressed by building up a points to graph for your program. Of the ways to do this a flow insensitive algorithm is prefered b\c it builds up one points to graph data structure for an entire program while a sensitive algorithm would compute a different point to graph at each point of a program.

The prefered algorithms are Steensgaards or Andersen's. The choice between the two is usually decided by precision vs speed with Steensgaards being faster but Andersen's being more precise.

Example:

```
Int a = 0;
Int *b = &a
Int *c = b
Int *d = NULL
Int *e = d
Int *c = e
```

Andersens algorithm

```
c -----> b ---> a          e---->d
|                          |
\__________________________/
```

So we still use availability like before but now we build up this subsequent data structure above. The approach should ignore control-flow and replace all the string updates with weak updates (ie updates where you don't know precisely which variable is being update. That means you need to create a set of vars that point to the same thing. So in the example above c has two sets it is a part of.

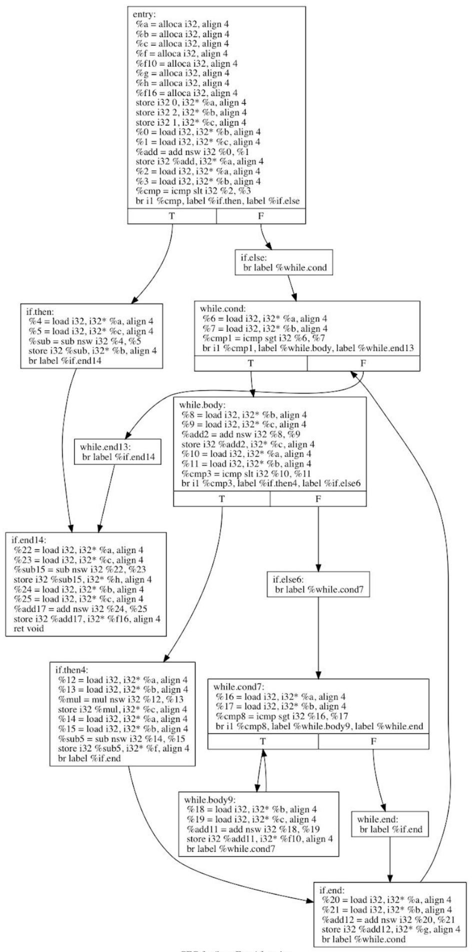
Cite: http://www.cs.cornell.edu/courses/cs711/2005fa/slides/sep13.pdf
https://www.seas.harvard.edu/courses/cs252/2011sp/slides/Lec06-PointerAnalysis.pdf

(c) Modify the available expression analysis to find available expressions on demand inside a given basic block B – this is called demand driven analysis mainly used for

doing CSE only within loops. Note that here you will not perform whole CFG analysis but will only focus on detecting availability of only those expressions which are generated inside loop basic blocks for the purposes of redundancy elimination.

This sounds very similar to invariant code removal. We have to make sure that our operation if a load store is either not modified in the loop body or if it is modified it is only modified by itself. Further we should executed the variable on every iteration, but if its not it should not alter the state of the program when it does by either causing an exception or interrupt. Finally we should make sure if thi is a memory or pointer operation that the address does not change either (ie not just the value). Probably can gurantee that last step with one of the flow insensitive algorithm mentioned in part b.

Q2 A very busy expression e is defined as the one that is anticipatable at a program point p ie, there is an evaluation of e on every path that begins at p before the end of the function or redefinition of its operands. It is proposed to hoist e at a program point p so that it eliminates original expressions maximally leading to code size reduction.
(a) Show an example which shows when is it legal to hoist very busy expressions up and when is it not. Also show the case of maximal hoisting when hoisting is legal.


```
entry:
 %a = alloca i32, align 4
 %b = alloca i32, align 4
 %c = alloca i32, align 4
 %f = alloca i32, align 4
 %f10 = alloca i32, align 4
 %g = alloca i32, align 4
 %h = alloca i32, align 4
 %f16 = alloca i32, align 4
 store i32 0, i32* %a, align 4
 store i32 2, i32* %b, align 4
 store i32 1, i32* %c, align 4
 %0 = load i32, i32* %b, align 4
 %1 = load i32, i32* %c, align 4
 %add = add nsw i32 %0, %1
 store i32 %add, i32* %a, align 4
 %2 = load i32, i32* %a, align 4
 %3 = load i32, i32* %b, align 4
 %cmp = icmp slt i32 %2, %3
 br i1 %cmp, label %if.then, label %if.else
```
```
if.else:
 br label %while.cond
```
```
if.then:
 %4 = load i32, i32* %a, align 4
 %5 = load i32, i32* %c, align 4
 %sub = sub nsw i32 %4, %5
 store i32 %sub, i32* %b, align 4
 br label %if.end14
```
```
while.cond:
 %6 = load i32, i32* %a, align 4
 %7 = load i32, i32* %b, align 4
 %cmp1 = icmp sgt i32 %6, %7
 br i1 %cmp1, label %while.body, label %while.end13
```
```
while.body:
 %8 = load i32, i32* %b, align 4
 %9 = load i32, i32* %c, align 4
 %add2 = add nsw i32 %8, %9
 store i32 %add2, i32* %c, align 4
 %10 = load i32, i32* %a, align 4
 %11 = load i32, i32* %b, align 4
 %cmp3 = icmp slt i32 %10, %11
 br i1 %cmp3, label %if.then4, label %if.else6
```
```
while.end13:
 br label %if.end14
```
```
if.end14:
 %22 = load i32, i32* %a, align 4
 %23 = load i32, i32* %c, align 4
 %sub15 = sub nsw i32 %22, %23
 store i32 %sub15, i32* %h, align 4
 %24 = load i32, i32* %b, align 4
 %25 = load i32, i32* %c, align 4
 %add17 = add nsw i32 %24, %25
 store i32 %add17, i32* %f16, align 4
 ret void
```
```
if.else6:
 br label %while.cond7
```
```
if.then4:
 %12 = load i32, i32* %a, align 4
 %13 = load i32, i32* %b, align 4
 %mul = mul nsw i32 %12, %13
 store i32 %mul, i32* %c, align 4
 %14 = load i32, i32* %a, align 4
 %15 = load i32, i32* %b, align 4
 %sub5 = sub nsw i32 %14, %15
 store i32 %sub5, i32* %f, align 4
 br label %if.end
```
```
while.cond7:
 %16 = load i32, i32* %a, align 4
 %17 = load i32, i32* %b, align 4
 %cmp8 = icmp sgt i32 %16, %17
 br i1 %cmp8, label %while.body9, label %while.end
```
```
while.body9:
 %18 = load i32, i32* %b, align 4
 %19 = load i32, i32* %c, align 4
 %add11 = add nsw i32 %18, %19
 store i32 %add11, i32* %f10, align 4
 br label %while.cond7
```
```
while.end:
 br label %if.end
```
```
if.end:
 %20 = load i32, i32* %a, align 4
 %21 = load i32, i32* %b, align 4
 %add12 = add nsw i32 %20, %21
 store i32 %add12, i32* %g, align 4
 br label %while.cond
```

CFG for 'busyFunc' function

```
void busyFunc() {
 int a = 0;
 int b = 2;
 int c = 1;
 a = b + c;
 if (a < b) {
      b = a - c;
 } else {
      while (a > b) {
          c = b + c;
          if (a < b) {
            c = a * b;
            int f = a - b;
          }
          else
          {
            while(a > b) {
               int d = a + b;
```

```
        int f = b + c;
      }
    }
    int g = a + b;
```

```
            }
        }
        int h = a - c;
        int f = b + c;
      }
```

Notice the expression a + b highlighted in yellow. In both while loops a and b are invariant thus a + b is invariant. This makes the expression a+b a good candidate to move out of the loops using global variable migration (je hoisting). If we were to use a liveness data flow analysis with a transfer function that propagates backwards from the use of the expression then we would **wrongly** consider moving this expression to the basic block highlighted in green (ie basic block one). Anticipatable expression analysis would have caught this. We can still hoist but to the top of the first else block. This is the maximal hoisting we can do that's legal. For a legal example take this same example but don't do a store on b (the red highlighted section).

(b) Assuming that the Anticipatable_IN[B] and Anticipatable_OUT[B] sets are already computed, write a condition for determining legal hoisting points for a set of Very Busy Expressions. Also write a condition for determining maximal hoisting (for maximally covering very busy expressions) – write an algorithm that uses both of these conditions to maximally and legally hoist very busy expressions. The algorithm must reject illegal hoistings.

An expression should only be hoisted to a basic blocks its In set if it can be hoisted out into its predecessor. This means that for an expression to be hoisted at the start of a hoisting path must insert itself in the OUTset of the first block it moves to rather than the entry. So to safely hoist to an Out set an expression should only be hoisted if for ever successor of our basic block we can hoist to the entry. This can be captured in following equation

In[b] = (Out[b] - Kill[b]) U gen[b]
Out[b] ={BI b is start block ∩ s ∈ SUCC(b) In[s]}

Cite: **Data Flow Analysis: Theory and Practice** By Uday Khedker, Amitabha Sanyal, Bageshri Sathe

(c) Show on the example CFG, how maximal hoisting is performed and how the algorithm rejects illegal hoistings.

Note first two in bit vector are a and b the third i is whatever var is getting the expression a+b
The example given in the HW is not a great one b\c there are no redefinitions that we know of.

Start with Node 2 it generates blah = a+b . assign a +b a temporary and propagate that temp down to its successors. When we hit Node 9, 15 and 14 we will just replace the expression with the temp.

Block	Local Info	Global info

	Gen	Kill	Out	In
Node 16	000	000	110	111
Node 15	001	000	111	110
Node 14	001	000	111	110
Node 13	000	000	110	110
Node 12	000	000	110	110
Node 11	000	000	110	110
Node 10	000	000	110	111
Node 9	000	000	111	111
Node 8	000	000	110	110
Node 7	000	000	110	110
Node 6	000	000	110	110
Node 5	000	000	110	110
Node 4	000	000	110	110
Node 3	000	000	110	110
Node 2	001	000	110	110
Node 1	000	000	110	110

Cite: https://www.cs.umd.edu/class/spring2014/cmsc430/lectures/lec20.pdf pg 9-11

3)
This did not go as intended. I started out with a naive approach count all the stores and put those values in a set. Then check all loads against the set. If a load is not found then the variable has not been initialized. But then I wanted to test cases were on one side of a conditional the variable was initialized but on the other side it was not

```
void func3() {
 int i;
 int n = 0;
 if (n < 1)
 {
   i = 0;
 }
 int j = i + 1;
}

void func4() {
 int i;
 int n = 0;
 if (n > 1)
 {
   i = 0;
 }
 int j = i + 1;
}
```

So i knew I needed to enumerate the paths. So I took my reachability code from last hw. The solution never got to a functioning state so my submission is just the naive implementation.

4) **On the given CFG shown below (Figure 1), perform PRE showing all the steps.**
Need to find expressions available on some, but not all, paths. PRE will insert code to make all paths fully redundant. Great example Node2 and Node 3 are Node 4 successors. Node 2 has blah = a+b. Node 3 does not. Add the expression blah = a+b to Node 3. We do this to make the code fully redundant. If we can make the predecessors full redundant then we can remove them. When we do this we need to watch out for critical edges which are defined as an edge from a node with multiple successors to a node with multiple predecessors. Node 4 is a good example of this. To solve this we can split the critical edges by inserting in empty basic blocks. We continue along like this until we have propagate the redundancy on the first pass. On the second pass we clean then all up as far as they can go.

The Knoop Lazy code motion paper

Does PRE framework discussed in the class (Drechler and Knoop's papers) completely remove partial redundancy? Show an example where partial redundancy is not completely removed.

Yeah PRE can find a+ b all day and mark it redundant throughout the graph but won't do the same for b + a (usually something more complex), it has to do something called global reassociation to achieve this. This reassociation reorders expressions to reveal constants and loop invariant terms. This is actually a downside of using something like GVN. instead a good idea is to use an algorithm that identifies equivalent variables. An algorithm that could group by associativity would know that a + b and b + a are the same.