15-745 Lecture 3

Basic Blocks
Local Opts
Control Flow Graphs

Optimizations

- Register Allocation
- Common subexpression elimination
- Constant Propagation
- Copy propagation
- Dead-code elimination
- Loop optimizations
  - Hoisting
  - Induction variable elimination

Scope of Optimization

- Local
  Within a basic block
- Global
  Within a function, across basic blocks
- Interprocedural
  The entire program, across functions and basic blocks.

Basic Blocks

- What is a basic block?
- How do we create basic blocks?
  - leaders
  - Other definitions of leaders

Local Opts: E.g., CSE

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```


Local Opts: E.g., CSE

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```

```
B5: t6     = 4*i
    x      = a[t6]

    t8     = 4 * j
    t9     = a[t8]
    a[t6]  = t9

    a[t8]  = x
    jump   B2
```

Dags & Stmts

a+a*(b-c)+(b-c)*d

Dags & Blocks

a = b + c;
b = a – d;
c = b + c;
d = a – d;

Dags & Blocks

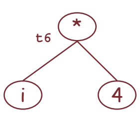
a = b + c;
b = a – d;
c = b + c;
d = a – d;

- Must track assignments
- Must keep track of time

Build a DAG

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```

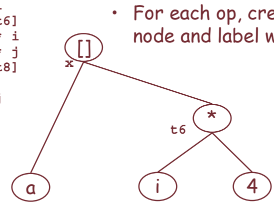
- For each var & constant not seen before create a leaf
- For each op, create a node and label with lvalue



Build a DAG

```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```

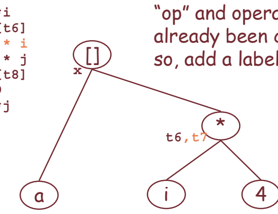
- For each var & constant not seen before create a leaf
- For each op, create a node and label with lvalue



Build a DAG

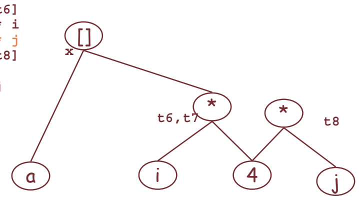
```
B5: t6     = 4*i
    x      = a[t6]
    t7     = 4 * i
    t8     = 4 * j
    t9     = a[t8]
    a[t7]  = t9
    t10    = 4*j
    a[t10] = x
    jump   B2
```

- If you have seen all rvalues before see if an interior node with same "op" and operands has already been created. If so, add a label.



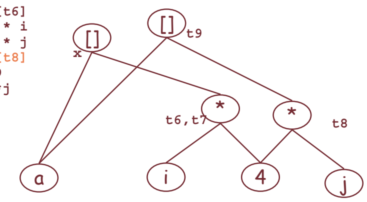
Build a DAG

```
B5: t6      = 4*i
    x       = a[t6]
    t7      = 4 * i
    t8      = 4 * j
    t9      = a[t8]
    a[t7]   = t9
    t10     = 4*j
    a[t10]  = x
    jump    B2
```



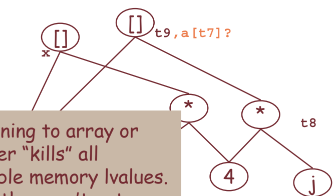
Build a DAG

```
B5: t6      = 4*i
    x       = a[t6]
    t7      = 4 * i
    t8      = 4 * j
    t9      = a[t8]
    a[t7]   = t9
    t10     = 4*j
    a[t10]  = x
    jump    B2
```



Build a DAG

```
B5: t6      = 4*i
    x       = a[t6]
    t7      = 4 * i
    t8      = 4 * j
    t9      = a[t8]
    a[t7]   = t9
    t10     = 4*j
    a[t10]  = x
    jump
```



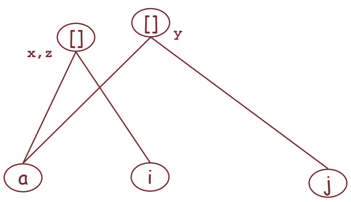
- Assigning to array or pointer "kills" all possible memory lvalues. (I.e., they can't get any more labels.)

Memory References

```
x = a[i]
a[j] = y
z = a[i]
```

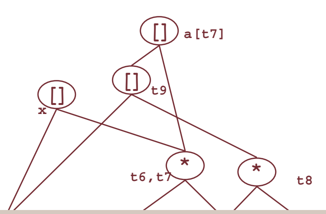
Becomes:

~~x = a[i]~~
~~z = x~~
~~a[j] = y~~



Build a DAG

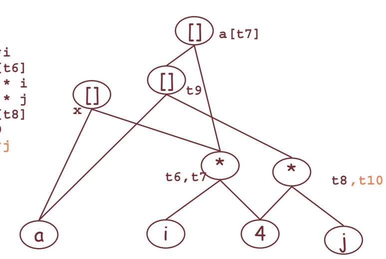
```
B5: t6      = 4*i
    x       = a[t6]
    t7      = 4 * i
    t8      = 4 * j
    t9      = a[t8]
    a[t7]   = t9
    t10     = 4*j
    a[t10]  = x
    jump    B2
```



Assignment to an array creates a new node with children:
- index
- old value of array
- value assigned

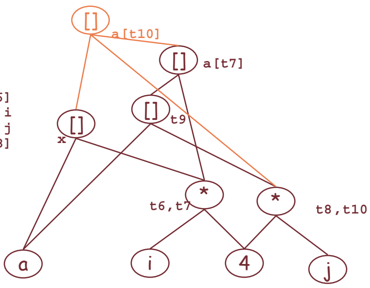
Build a DAG

```
B5: t6      = 4*i
    x       = a[t6]
    t7      = 4 * i
    t8      = 4 * j
    t9      = a[t8]
    a[t7]   = t9
    t10     = 4*j
    a[t10]  = x
    jump    B2
```



Build a DAG

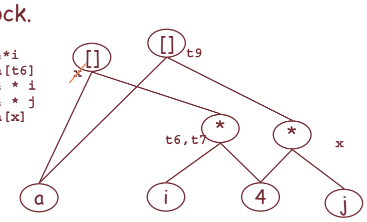
```
B5: t6      = 4*i
    x       = a[t6]
    t7      = 4 * i
    t8      = 4 * j
    t9      = a[t8]
    a[t7]   = t9
    t10     = 4*j
    a[t10]  = x
    jump    B2
```



Other uses for DAGs

- Can determine those variables that *can* be live at end of a block.
- Can determine those variables that are live at start of block.

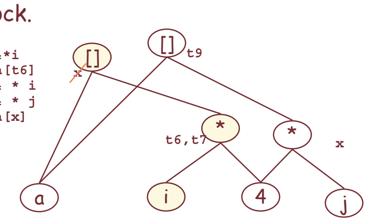
```
B8: t6      = 4*i
    x       = a[t6]
    t7      = 4 * i
    x       = 4 * j
    t9      = a[x]
    …
```



Dead code too?

- Can determine those variables that *can* be live at end of a block.
- Can determine those variables that are live at start of block.

```
B8: t6      = 4*i
    x       = a[t6]
    t7      = 4 * i
    x       = 4 * j
    t9      = a[x]
    …
```



Using the DAG to recreate blocks

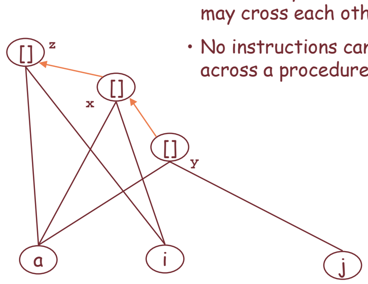
- Order of evaluation is any topological sort
- We pick a node. Assign it to ONE of the labels (hopefully one needed later in the program)
- If we end up with identifiers that are needed after this block, insert move statements.
- If a node has no identifiers, make up a new one.
- Caveats:
  - Procedure calls kill nodes
  - A[] = and *p = kill nodes

Memory References

```
x = a[i]
a[j] = y
z = a[i]
```

In general,

- No memory references may cross each other
- No instructions can move across a procedure call



Value Numbering

- Don't actually build DAG
- Track value of variable in time as a "value number"
- compute valueOf(var) and valueOf(val, op, val)
- Scan stmts: d <- a op b, computing
  - a_n = valueOf(a)
  - b_n = valueOf(b)
  - op_n = valueOf(a_n, op, b_n)
  - set valueOf(d) = op_n

VN example

a = b + c;
b = a – d;
c = b + c;
d = a – d;

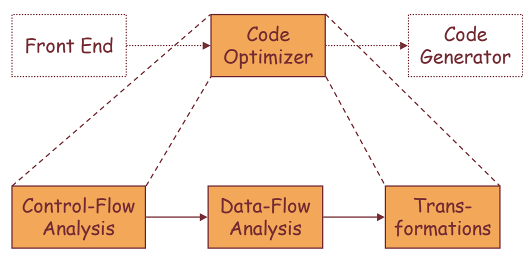
VN uses

- Same as DAGs (live in, live out, CSE)
- How about constant folding?
  a = 1
  b = 2
  c = a + b

Scope of Optimization

- Local
  Within a basic block
- Global
  Within a function, across basic blocks
- Interprocedural
  The entire program, across functions and basic blocks.

The Optimizer

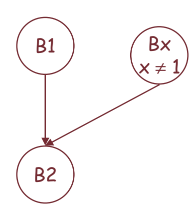


Control Flow Graph

- Each BB is a node in the graph
- Distinguished nodes: Entry & Exit
- Edge between B1 & B2 iff
  - B2 can follow B1 in some execution of the program
    - B2 is a target of a jump/branch at end of B1
    - B2 follows B1 and B1 does not end with an unconditional jump
  - B1 is Entry and B2 is first instruction
  - B2 is Exit and B1 can exit procedure

Terms

- B1 is Predecessor of B2
- B2 is successor of B1
- B2 is a join node



Unreachable Code Elimination

Straigtening