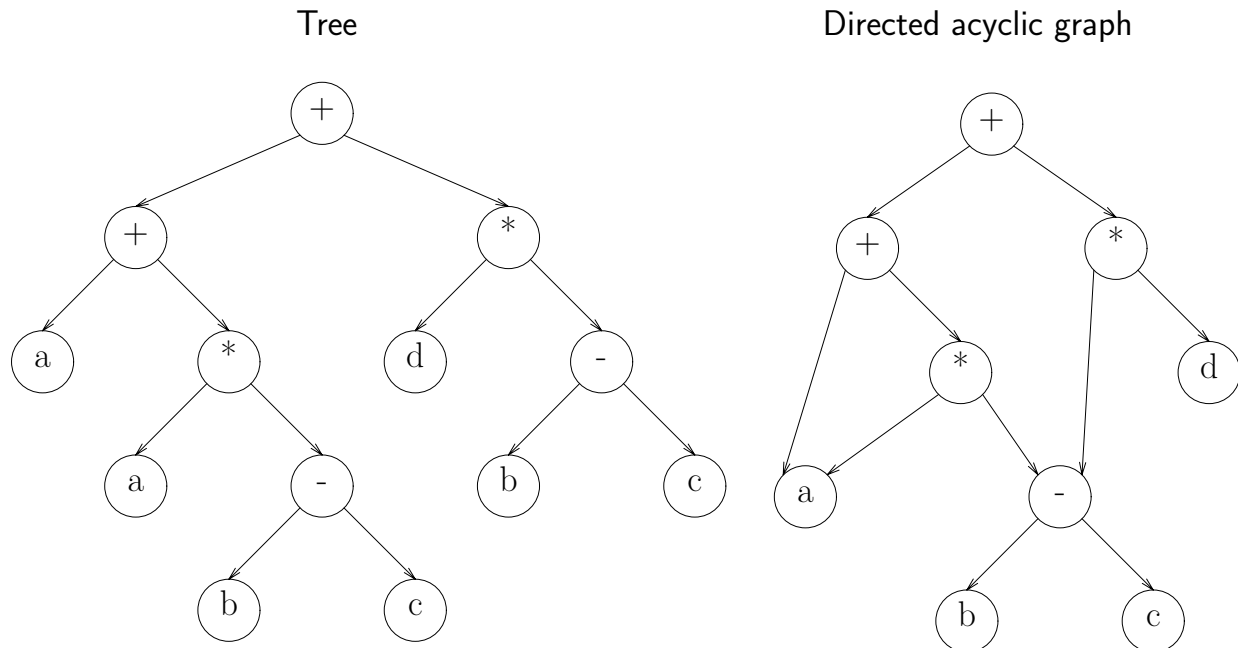


## Local optimizations

---

Consider the expression:  $a + a * ( b - c ) + ( b - c ) * d$



## Local optimizations

---

### Common subexpressions (CSE)

- portion of expressions
- repeated multiple times
- computes same value
- can reuse previously computed value

### Directed acyclic graph (DAG)

- program representation
- nodes can have multiple parents
- no cycles allowed
- exposes common subexpressions

### Building a DAG for an expression

- maintain hash table for leafs, expressions
- unique name for each node — its *value number*
- reuse nodes found in hash table

## Directed acyclic graphs

---

What about *assignment* ?

- complicates detection of common subexpressions
- identical expression  $\rightarrow$  different value
- must ensure each *value* has a unique node

One solution - renaming

- add subscripts to variable names (e.g.,  $x \rightarrow x_i$ )
- increment subscript of name if target (LHS) of assignment
- variables references use new subscript

Example

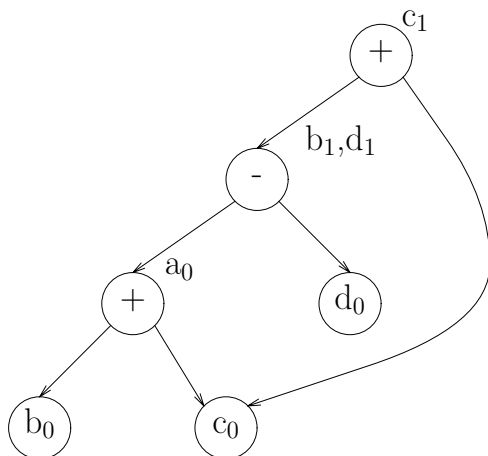
$$a_1 = a_0 + b_0$$

*Can apply to entire basic block*

## Directed acyclic graph example

---

Code	After Renaming
$a = b + c$	$a_0 = b_0 + c_0$
$b = a - d$	$b_1 = a_0 - d_0$
$c = b + c$	$c_1 = b_1 + c_0$
$d = a - d$	$d_1 = a_0 - d_0$



## Common subexpressions

---

### Going beyond basic blocks

- can no longer build DAGs
- must consider control flow

### Examples

- possible kill

```
c = a+b
if (...)
    a = ...
d = a+b
```

- possible gen

```
if (...)
    c = a+b
d = a+b
```

We handle these conditions using data-flow analysis

## Data-flow analysis

---

### Data-flow analysis

- *compile-time* reasoning about the *run-time* flow of values in the program
- represent facts about run-time behavior
- represent effect of executing each basic block
- propagate facts around control flow graph

### Formulated as a set of simultaneous equations

- sets attached to the nodes and edges
- lattice to describe relation between values
- usually represented as bit or bit vector

### Limitations

- answers must be conservative
- often need to approximate information
- assume all possible paths can be taken

## Data-flow analysis

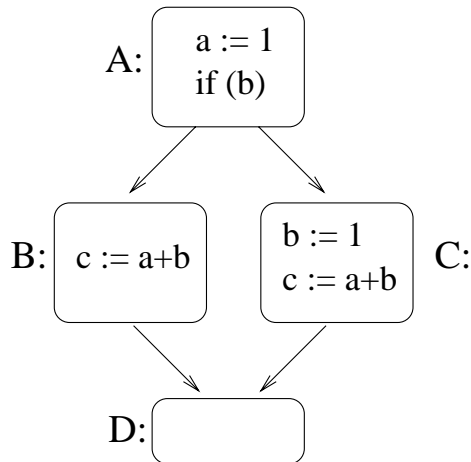
---

### Algorithm

1. build control flow graph (CFG)
2. initial (local) data gathering
3. propagate information around the graph
4. post-processing (*if needed*)

### Example control flow graph

```
a = 1
if (b) then
  c = a+b
else
  b = 1
  c = a+b
...
```



## Available expressions

---

### Definition

- An expression is *defined* at point  $p$  if its value is computed at  $p$ .
- An expression is *killed* at a point  $p$  if one of its argument variables is defined at  $p$ .
- an expression  $e$  is *available* at a point  $p$  in a procedure if every path leading to  $p$  contains a prior definition of  $e$  that is not killed between its definition and  $p$ .

### Global common subexpression elimination

- If, at some definition point for  $p = e$ ,  $e$  is available with name  $x$ , we can replace the evaluation with a reference to  $x$ .
- requires a global naming scheme
- natural analog to parts of value numbering

## Available expressions

---

For a block  $b$

- let  $GEN(b)$  be the set of expressions defined in  $b$  and not subsequently killed in  $b$ .
- let  $KILL(b)$  be the set of expressions killed in  $b$ .
- let  $IN(b)$  be the set of expressions available on entry to  $b$ .
- let  $OUT(b)$  be the set of expressions available on exit to  $b$ .

IN and OUT represent global information and can be calculated as:

$$OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$$

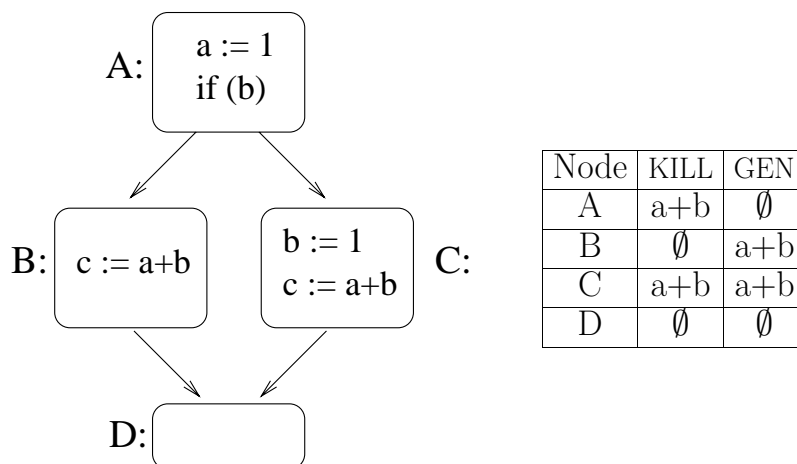
$$IN(b) = \bigcap_{x \in pred(b)} (OUT(x))$$

AVAIL is simply IN. Its calculation can be combined as:

$$AVAIL(b) = \bigcap_{x \in pred(b)} (GEN(x) \cup (AVAIL(x) - KILL(x)))$$

## Available expressions example

---



$$AVAIL(A) = \emptyset$$

$$AVAIL(B) = GEN(A) \cup (AVAIL(A) - KILL(A)) = \emptyset \cup (\emptyset - \{ a+b \}) = \emptyset$$

$$AVAIL(C) = GEN(A) \cup (AVAIL(A) - KILL(A)) = \emptyset \cup (\emptyset - \{ a+b \}) = \emptyset$$

$$\begin{aligned} AVAIL(D) &= (GEN(B) \cup (AVAIL(B) - KILL(B))) \cap \\ &\quad (GEN(C) \cup (AVAIL(C) - KILL(C))) \\ &= (\{ a+b \} \cup (\emptyset - \emptyset)) \cap (\{ a+b \} \cup (\emptyset - \{ a+b \})) = \{ a+b \} \end{aligned}$$

### Iterative algorithm

```
change = true;
while (change)
  change = false;
  for each basic block // faster in reverse PostOrder:
    solve data-flow equations for b
    if (old  $\neq$  new)
      change = true;
  end for
end while
```

### Speed of solution

- node may change only if some predecessor changes
- try to visit node after all its predecessors
- reverse PostOrder propagates information quickly
- programs usually converge after 3–4 passes
- use bitvectors for more efficiency

## PostOrder and reverse PostOrder

---

### Step1: PostOrder

```
main()
  count = 1;
  Visit (root);

Visit(n)
  mark  $n$  as visited
  for each successor  $s$  of  $n$  not yet visited
    Visit( $s$ );
  PostOrder( $n$ ) = count;
  count = count + 1;
```

### Step 2: Reverse PostOrder (rPostorder)

```
for each node  $n$ 
  rPostOrder( $n$ ) = NumNodes - PostOrder( $n$ )
```

Depth-first search  $\approx$  rPostOrder

## Reaching definitions

---

- **The problem:** What are the assignments (or definitions) of a variable  $x$  that may reach a particular reference to  $x$ ?
- **Why is this useful?**

Constant propagation:

```
          a = 1
a = 2      a = 2
           b = 3
           = a
           = b
```

Loop invariant code motion:

```
L:
  a = a + 4
  b = 20
  c = b + a
  if (...) goto L
```

## Reaching definitions

---

- A **definition** of a variable  $x$  is a statement that assigns, or may assign, a value to  $x$ .
- A definition  $d$  **reaches** a program point  $p$  if **there exists** a path from the point immediately following  $d$  to  $p$  such that  $d$  is not killed along that path.
- $\text{REACH}(b)$  is the set of definitions reaching the entry of basic block  $b$
- $\text{DEF}(b)$  is the set of *local definitions* in  $b$  that reach the end of  $b$
- $\text{KILL}(b)$  is the set of variables killed by  $b$
- **Equations:**

$$\text{REACH}(b) = \bigcup_{x \in \text{pred}(b)} (\text{DEF}(x) \cup (\text{REACH}(x) - \text{KILL}(x)))$$

Best case for  $\text{REACH}(b) = \emptyset$

Worse case for  $\text{REACH}(b) = \{ \text{all definitions} \}$

## Live variables

---

### Definition:

- A definition  $d$  is **live** at program point  $p$  if the variable  $v$  defined by  $d$  may be used along some path in the program starting at  $p$  without being redefined between  $d$  and  $p$ .
- Otherwise, the definition is **dead**

### Why is this useful?

- global analysis to locate dead assignments.

```
          a =
          b =
a =
b =          b =
          = a
          = b
```

## Live variables

---

- Slightly different, since information at basic block is based on what happens later in the program.
- A *backward* data-flow problem.
  
- $LIVE(b)$  is the set of definitions live on exit from block  $b$ .
- $KILL(b)$  is as before.
- $USE(b)$  is the set of locally exposed uses
- $succ(b)$  is the set of basic blocks that are immediate successors of  $b$  in the control flow graph.
  
- Equations:

$$LIVE(b) = \bigcup_{x \in succ(b)} (USE(x) \cup (LIVE(x) - KILL(x)))$$

Best case for  $LIVE(b) = \emptyset$

Worse case for  $LIVE(b) = \{ \text{all definitions} \}$

## What do these have in common?

---

$$AVAIL(b) = \bigcap_{x \in pred(b)} (GEN(x) \cup (AVAIL(x) - KILL(x)))$$

$$REACH(b) = \bigcup_{x \in pred(b)} (DEF(x) \cup (REACH(x) - KILL(x)))$$

$$LIVE(b) = \bigcup_{x \in succ(b)} (USE(x) \cup (LIVE(x) - KILL(x)))$$

- **Confluence Operator or Meet Function:** union or intersection
- **Behavior for block:** GEN and KILL
- **A direction:** forward (confluence over predecessors) or backward (over successors)
- **Best case set value:**  $\top$
- **Worst case set value:**  $\perp$

General equations:

$$IN(b) = \bigwedge_{p \in pred(b)} OUT(p)^\dagger$$

$$OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$$

† Reverse graph for backward problem.

## Data-flow analysis frameworks

---

Use same framework for all data-flow problems

- given local information GEN, KILL
- start with some initial values for sets IN, OUT
- iterate through nodes in the flow graph, recompute transfer functions until sets *stabilize*

Framework has three components

- Domain of values:  $L$
- Operator for combining values:  $\wedge$
- A set of transfer functions ( $L \rightarrow L$ ):  $\mathcal{F}$

Usefulness of unified framework

- Defines a collection of properties that guarantee correctness, convergence;
- Can describe speed of convergence and precision of result for a family of analysis problems
- Can re-use code to solve new analysis problems

## Data-flow lattices

---

### Definitions

1. a *lattice* is a set  $L$  and a meet operation  $\wedge$  such that,  $\forall a, b, c \in L$ 
  - (a)  $a \wedge a = a$  [idempotent]
  - (b)  $a \wedge b = b \wedge a$  [commutative]
  - (c)  $a \wedge (b \wedge c) = (a \wedge b) \wedge c$  [associative]
2.  $\wedge$  imposes a partial order on  $L$ ,  $\forall a, b \in L$ 
  - (a)  $a \geq b \Leftrightarrow a \wedge b = b$
  - (b)  $a > b \Leftrightarrow a \geq b$  and  $a \neq b$
3. a lattice may have a *bottom* element  $\perp$ 
  - (a)  $\forall a \in L, \perp \wedge a = \perp$
  - (b)  $\forall a \in L, a \geq \perp$
4. a lattice may have a *top* element  $\top$ 
  - (a)  $\forall a \in L, \top \wedge a = a$
  - (b)  $\forall a \in L, \top \geq a$

## Data-flow lattices

---

Available expressions example:

let  $D = \{x \mid x \subseteq \{e_1, e_2, e_3\}\}$ ,  $\wedge = \cap$

$\top =$

$\perp =$

Partial ordering  $\{e_1, e_2\}$  vs.  $\{e_3\}$

Single lattice vs. one for each variable

$\top =$

$\perp =$

## Data-flow lattices

---

How does this relate to data-flow analysis?

- choose a semi-lattice  $L$  to represent facts
- attach to each element of  $L$  a *meaning*  
each  $a \in L$  is a distinct set of known facts
- with each node  $n$ , associate a *transfer function*  
 $f_n : L \rightarrow L$  to model behavior of  $n$
- propagate facts around the graph

Example – AVAIL

- semi-lattice is  $2^E$ , where  $E$  is the set of all expressions computed  
 $\wedge$  is  $\cap$ ,  $\perp$  is  $\emptyset$ ,  $\top$  is  $E$
- for a node  $n$ ,  $f_n$  has the form  $f_n(x) = D_n \cup (x - N_n)$   
where  $D_n = \text{GEN}_n$  and  $N_n = \text{KILL}_n$
- the underlying graph is the flow graph  $G = (N, E, n_0)$   
 $n_0$  is the entry node

## Iterative algorithm

---

What about loops?

- circular dependencies between blocks
- can initialize solutions, then solve repeatedly

Example

```
c = a+b
```

L:

```
d = a+b
```

```
a = ...
```

```
if (...) goto L
```

Termination

- goal is for solutions to converge to a *fixed point*
- can stop once solutions stop changing
- is this guaranteed?

## Monotonicity

---

- A framework  $(D, \wedge, F)$  is monotone iff
$$x \leq y \text{ implies } f(x) \leq f(y)$$

i.e., a “smaller or equal” input to the same function will always give a “smaller or equal” output
- Equivalently, monotone iff
$$f(x \wedge y) \leq f(x) \wedge f(y)$$

i.e., if merge input, then apply  $f$ , result is “smaller or equal” to applying  $f$  individually and merging result
- Intuitively, monotonicity means “smaller” input will not yield “larger” output.
- monotone frameworks are guaranteed to converge and terminate (if lattice elements can only drop in information a finite number of times)

## Quality of solution

---

### Possible solutions

- perfect solution = meet over *real* paths taken during program execution
- meet-over-all-paths (MOP) = meet over *potential* paths in control flow graph
- maximal-fixed-point (MFP) = solution from iterative framework

### Properties

- in general,  $MFP \leq MOP \leq \text{Perfect Solution}$
- in some sense, MOP is best feasible solution
- MFP is unique, regardless of order of propagation
- a framework is *distributive* if  $f(x \wedge y) = f(x) \wedge f(y)$
- for a distributive framework,  $MFP = MOP$