**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

---

**Common subexpressions**

- Both a and b-c are common subexpressions (CSE)
  - Compute same value
  - Should compute the value once
  - A simple and general form of code improvement

\[ a + a \times ( b - c ) + ( b - c ) \times d \]

The directed acyclic graph is a useful representation for such expressions.

The \textit{dag} clearly exposes the cses
**Directed acyclic graph**

- A directed acyclic graph is a tree with sharing
  - A tree is a directed acyclic graph where each node has at most one parent
  - A dag allows multiple parents for each node
  - Both a tree and a dag have a distinguished root
  - No cycles in the graph!

- To find common subexpressions within a statement
  - Build the dag
  - Generate code from the dag
  - This should lead to faster evaluation

- How do we build a dag for an expression?
  - Use construction primitives for building trees
  - Teach primitives to catch cses
    - Mkleaf() and mknode()
      - Hash on <op, l, r>
  - Unique name for each node – its value number

- Anywhere we build a tree, we could build a dag
  - Initialize hash table for each expression
  - Catch only cses within expressions

- What about assignment?
  - Complicates cse detection
  - Each value has a unique node
  - Add subscripts to variables

- While building the dag, an assignment
  - Creates new nodes for lhs – a new xi
  - Kills all nodes built from xi-1

- Example: a₁ = a₀ + b
  - Can we go beyond a single statement?
    - Use a single dag for an entire basic block

- A dag for a basic block has labeled nodes
  - Leaves are labeled with unique identifiers (either variable names or constants) (Leaves represent values on entry)
  - Interior nodes are labeled with operators
  - Nodes have optional identifier labels
    - Interior nodes represent computed values
    - Identifier label represents assignment
Example

- Code
  \[
  \begin{align*}
  a &= b + c \\
  b &= a - d \\
  c &= b + c \\
  d &= a - d
  \end{align*}
  \]

- After renaming
  \[
  \begin{align*}
  a_0 &= b_0 + c_0 \\
  b_1 &= a_0 - d_0 \\
  c_1 &= b_1 + c_0 \\
  d_1 &= a_0 - d_0
  \end{align*}
  \]

Building a dag

- \text{node( <id> )} \rightarrow \text{current dag for <id>}

1. Set node(y) to undefined for each symbol y
2. for each statement \( x = y \text{ op } z \), repeat steps 3, 4, and 5
3. If node(y) is undefined,
   > create a leaf for y
   > set node(y) to the new node
   > do the same for z
4. if \(<\text{op, node(y), node(z)}>\) doesn't exist, create it and let n point to that node
5. delete x from the list of labels for node(x)
   > append x to the list of labels for n
   > set node(x) to n
Common subexpressions

- Going beyond basic blocks
  - Can no longer build dags
  - Must consider control flow

- Examples
  - Use
    - C = A + B
    - D = A + B
  - Intervening kill
    - C = A + B
    - A = ...
    - D = A + B
  - Possible use
    - C = A + B
    - If (...)
      - D = A + B
  - Possible kill
    - C = A + B
    - If (...)
      - A = ...
      - D = A + B
  - Possible gen
    - If (...)
      - C = A + B
    - D = A + B
  - Multiple gen
    - If (...)
      - C = A + B
      - Else
      - D = A + B

We generalize these conditions as data flow analysis.

Algorithm

- build control flow graph (CFG)
  - initial (local) data gathering
  - propagate information around the graph
  - post-processing (if needed)

- Example control flow graph

```plaintext
a := 1
if (b) then
  c := a+b
else
  b := 1
  c := a+b
  ...
```
**Available expressions**

- 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 and a natural analog to parts of value numbering

---

**Available expressions**

For a block b

- let $\text{AVAIL}(b)$ be the set of expressions available on entry to b.
- let $\text{KILL}(b)$ be the set of expressions killed in b.
- let $\text{GEN}(b)$ be the set of expressions defined in b and not subsequently killed in b.

Note: $\text{GEN}(b)$ can be calculated from the list of live expressions from DAG construction.

- $\text{KILL}(b)$ is harder to construct, since it requires knowledge of all potential expressions in the program.

- Now, AVAIL can be defined as:
  \[
  \text{AVAIL}(b) = \bigcap_{x \in \text{pred}(b)} (\text{GEN}(x) \cup (\text{AVAIL}(x) - \text{KILL}(x)))
  \]
- Note: initializations must be conservative.
Available expression example

- \( \text{AVAIL}(A) = \emptyset \)
- \( \text{AVAIL}(B) = \text{GEN}(A) \cup ( \text{AVAIL}(A) - \text{KILL}(A)) = \emptyset \cup (\emptyset - \{a+b\}) = \emptyset \)
- \( \text{AVAIL}(C) = \text{GEN}(A) \cup ( \text{AVAIL}(A) - \text{KILL}(A)) = \emptyset \cup (\emptyset - \{a+b\}) = \emptyset \)
- \( \text{AVAIL}(D) = (\text{GEN}(B) \cup ( \text{AVAIL}(B) - \text{KILL}(B))) \cap (\text{GEN}(C) \cup ( \text{AVAIL}(C) - \text{KILL}(C))) \)
  \[ = (\{a+b\} \cup (\emptyset - \emptyset)) \cap (\{a+b\} \cup (\emptyset - \{a+b\})) = \{a+b\} \]

Partial redundancy elimination

- **Partial redundancy elimination** (PRE) is an optimization that
  - Discovers partially redundant expressions
  - Converts them to fully redundant expressions
  - Removes redundant expressions

- Intuition
  - PRE moves computation back (against the control flow) as far as possible to make their effects universal as possible,

- How does it work?
  - Anticipability \( \rightarrow \) expression can be precomputed at point \( p \)
  - Use data-flow analysis to find availability and anticipability
  - Solve a data-flow problem to discover where to insert code
  - Insert the code and remove redundant expression
**Redundant expressions**

- An expression $\epsilon$ is redundant at point $p$ if every path to $p$:
  - $\epsilon$ is evaluated before reaching $p$, and
  - None of the constituent values of $\epsilon$ are redefined before $p$.

```
a ← b+c
```

**Partially redundant expressions**

- An expression is partially redundant at $p$ if it is available on some, but not all paths reaching $p$.

```
b ← b+1
```
PRE equations

\[ \text{avon}(b) = \begin{cases} \text{false} & \text{if } b \text{ is an entry block} \\ \cap_{w \in \text{pred}(b)} \text{avon}(w) & \text{otherwise} \end{cases} \]

\[ \text{avout}(b) = \cap_{w \in \text{pred}(b)} \cup (\text{avout}(b) \cap \text{transp}(b)) \]

\[ \text{pavon}(b) = \begin{cases} \text{false} & \text{if } b \text{ is an entry block} \\ \cup_{w \in \text{pred}(b)} \text{pavon}(w) & \text{otherwise} \end{cases} \]

\[ \text{pavout}(b) = \cup_{w \in \text{pred}(b)} \cup (\text{pavout}(b) \cap \text{transp}(b)) \]

\[ \text{antout}(b) = \begin{cases} \text{false} & \text{if } b \text{ is an exit block} \\ \setminus_{\text{succ}(b)} \text{avout}(x) & \text{otherwise} \end{cases} \]

\[ \text{antin}(b) = \text{antin}(b) \cup (\text{out}(b) \cap \text{transp}(b)) \]

\[ \text{ppout}(b) = \begin{cases} \text{false} & \text{if } b \text{ is an exit block} \\ \setminus_{\text{succ}(b)} \text{ppin}(x) & \text{otherwise} \end{cases} \]

\[ \text{ppin}(b) = \text{antin}(b) \cap \text{pavin}(b) \cap (\text{ppout}(b) \cup (\text{ppin}(b) \cap \text{transp}(b)))) \]

\[ \text{insert}(b) = (\text{ppin}(b) \cap \text{transp}(b)) \cap \text{ppout}(b) \cap \text{avout}(b) \]

\[ \text{delete}(b) = \text{avon}(b) \cap \text{antin}(b) \]

Solving data flow equations

- Iterative algorithm
  
  change = true;
  while (change)
    change = false;
    for each basic block b  // faster in reverse PostOrder
      solve data-flow equations for b
      if old != new then 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 bit vectors for more efficiency
**PostOrder and reverse PostOrder**

- **Step 1: 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 ~ rPostOrder

**Data-flow analysis framework**

- Use same framework for all data-flow problems
  - Given local information Gen, Kill
  - Start with some initial values for In, Out
  - Iterate through nodes in the flow graph, recompute transfer functions until sets stabilize

- Framework has 3 components
  - Domain of values: L
  - Operator for combining values: Λ
  - A set of transfer functions (L→L): F

- Usefulness of unified framework
  - Defines a collection of properties that guarantee correctness and convergence
  - Can describe speed of convergence and precision of result for a family of analysis problems
  - Can reuse code to solve new analysis problems
**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 (x = f(x))
  - Can stop once solution stops changing
  - Is this guaranteed?
    → If system is monotone (i.e., f(x ∧ y) ≤ f(x) ∧ f(y))