Partial Redundancy Elimination

Finding the “Right” Place to Evaluate Expressions
Four Necessary Data-Flow Problems

Jeffrey D. Ullman
Stanford University
Role of PRE

- Generalizes:
  1. Moving loop-invariant computations outside the loop.
  2. Eliminating common subexpressions.
  3. 
     - True partial redundancy: an expression is available along some paths to a point, but not along others.
     - Evaluate it only along paths where it is not yet evaluated, just before it is needed.
Throughout, assume that neither argument of an expression $x+y$ is modified unless we explicitly assign to $x$ or $y$.

And of course, we assume $x+y$ is the only expression anyone would ever want to compute. 😊
Tradeoffs

- We have three competing goals:
  1. Fast execution.
  2. Small code space.
  3. Little use of registers to store values.
- PRE tends to favor (1), but then tries to do its best for (2) and (3).
Allocate a register for $t$. No statement needed for $a = t$. Just have the compiler remember that the value of $a$ is in that register.

An example of the tradeoff: use more registers, get more speed.
Example: Common Subexpression Elimination

\[ a = x+y \]
\[ b = x+y \]
\[ c = x+y \]

\[ t = x+y \]
\[ a = t \]
\[ b = t \]

\[ t = x+y \]
\[ c = t \]
**Example: True Partial Redundancy**

Tradeoff: runs faster along the left path, but more code space and register use.
We could:

1. Add a new block along an edge.
   - Only necessary if the edge enters a block with several predecessors.

2. Duplicate blocks so an expression $x+y$ is evaluated only along paths where it is needed.
Example: Node Splitting

\[ w = x+y \]
\[ z = x+y \]
\[ = x+y \]

\[ t = x+y \]
\[ w = t \]
\[ z = t \]
\[ = t \]

\[ t = x+y \]
\[ z = t \]
Can exponentiate the number of nodes.
Our PRE algorithm can move code to new blocks along edges, but will not split blocks.

**Convention:** All new instructions are either inserted at the beginning of a block or placed in a new block.
The Plan

1. Determine for each expression the earliest place(s) it can be computed while still being sure that it will be used.

2. Postpone the expressions as long as possible without introducing redundancy.
   - Another trade-off: space for speed and low-register use – an expression can be computed in many places, but never if it is already computed and only as late as possible.
   - Guarantee: No expression is computed where its value might have been computed previously.
3. Determine those places where it is really necessary to store $x+y$ in a temporary rather than compute it when needed.

- **Example**: If $x+y$ is used in only one place, then we do not want to compute it early and store it.
We use four data-flow analyses, in succession, plus some set operations on the results of these analyses.

After the first, each analysis uses the results of the previous ones in a role similar to that of Gen (for RD’s) or Use (for LV’s).
Anticipated Expressions

- Expression $x+y$ is *anticipated* at a point if $x+y$ is certain to be evaluated along any path, before any recomputation of $x$ or $y$.
- An example of the fourth kind of DF schema: backwards-intersection.
Example: Anticipated Expressions

\[ t = x + y \]

\[ = x + y \]

\[ = x + y \]

\[ = x + y \]

\[ = x + y \]

\[ = x + y \]

x+y is anticipated here and could be computed now rather than later.

x+y is anticipated here, but is also available. No computation is needed.
Computing Anticipated Expressions

- **Use(B)** = set of expressions $x+y$ evaluated in B before any possible assignment to $x$ or $y$.
  - Analogous to “Use” in live-variables.
- **Def(B)** = set of expressions at least one of whose arguments may be assigned in B.
  - Analogous to “Kill.”
- Direction = backwards.
- Meet = intersection.
- Boundary condition: IN[exit] = $\emptyset$.
- Transfer function:
  \[ IN[B] = (OUT[B] - \text{Def}(B)) \cup \text{Use}(B) \]
Backwards; Intersection; IN[B] = (OUT[B] − Def(B)) ∪ Use(B)
“Available” Expressions

- Modification of the usual AE.
- \(x+y\) is “available” at a point if either:
  1. It is available in the usual sense; i.e., it has been computed and not killed, or
  2. It is anticipated; i.e., it could be available if we chose to precompute it there.
“Available” Expressions

- \( x+y \) is in \( \text{Kill}(B) \) if \( x \) or \( y \) is defined, and \( x+y \) is not recomputed later in \( B \) (same as for earlier definition of “available expressions”).
- Direction = forward
- Meet = intersection.
- Transfer function:
  \[
  \text{OUT}[B] = (\text{IN}[B] \cup \text{IN}_{\text{ANTICIPATED}}[B]) - \text{Kill}(B)
  \]

Note: This is not a variable of the equations, but a value already computed by the previous DFA.
**Earliest Placement**

- **x+y** is in Earliest[B] if it is anticipated at the beginning of B but not “available” there.
  - That is: when we compute anticipated expressions, **x+y** is in IN[B], but
  - When we compute “available” expressions, **x+y** is not in IN[B].
- I.e., **x+y** is anticipated at B, but not anticipated at OUT of some predecessor.
Example: Available/Earliest

Forward; Intersection; OUT[B] = (IN[B] \cup IN_{\text{ANTICIPATED}}[B]) – \text{Kill}(B)
Postponable Expressions

- Now, we need to delay the evaluation of expressions as long as possible, but ...
  1. Not past a use of the expression.
  2. Not so far that we wind up computing an expression that might already be evaluated.
- **Note trade-off**: Use code space to save register use.
Postponable Expressions – (2)

- \( x+y \) is \textit{postponable} to a point \( p \) if on every path from the entry to \( p \):
  1. There is a block \( B \) for which \( x+y \) is in earliest\([B]\), and
  2. After that block, there is no use of \( x+y \) along that path.
Computed like “available” expressions, with two differences:

1. In place of killing an expression (assigning to one of its arguments): Use(B), the set of expressions used in block B.

2. In place of IN\textsubscript{ANTICIPATED}[B]: earliest[B].
Postponable Expressions – (4)

- Direction = forward.
- Meet = intersection.
- Transfer function:

\[ \text{OUT}[B] = (\text{IN}[B] \cup \text{earliest}[B]) - \text{Use}(B) \]
Example: Postponable Expressions

Forward; Intersection; OUT[B] = (IN[B] ∪ earliest[B]) – Use(B)
We want to postpone evaluating $x+y$ for as long as possible.

- **Question**: Why?
- How do we compute the “winners” – the blocks such that we can postpone no further?
- **Remember** – postponing stops at a use or at a block with another predecessor where $x+y$ is not postponable.
For \( x+y \) to be in latest\([B]\):

1. \( x+y \) is either in earliest\([B]\) or in IN\(_{\text{POSTPONABLE}}\)[B].
   - I.e., we can place the computation at B.

2. \( x+y \) is either used in B or there is some successor of B for which (1) does not hold.
   - I.e., we cannot postpone further along all branches.
Example: Latest

Earliest
Or Postponable to beginning

Latest = gold and red.

Used
Or has a successor not red.
We’re now ready to introduce a temporary $t$ to hold the value of expression $x+y$ everywhere. But there is a small glitch: $t$ may be totally unnecessary.

- E.g., $x+y$ is computed in exactly one place.
Used Expressions – (2)

- used[B] = expressions used along some path from the end of B.
- Direction = backward.
- Meet = union.
- Transfer function:
  \[ \text{IN}[B] = (\text{OUT}[B] \cup \text{e-used}[B]) - \text{Latest}(B) \]
  - e-used = “expression is used in B.”
Recall: Latest

Backwards; Union; $\text{IN}[B] = (\text{OUT}[B] \cup \text{e-used}[B]) - \text{Latest}(B)$
Rules for Introducing Temporaries

1. If $x+y$ is in both $\text{Latest}[B]$ and $\text{OUT}_{\text{USED}}[B]$, introduce $t = x+y$ at the beginning of $B$.
2. If $x+y$ is used in $B$, but either
   1. Is not in $\text{Latest}[B]$ or
   2. Is in $\text{OUT}_{\text{USED}}[B]$, replace the use(s) of $x+y$ by uses of $t$. 
Example: Where is a Temporary Used?

Recall: Latest

= x+y

Recall OUT$_{USED}$

But not here ---

x+y is in Latest and
not in OUT$_{USED}$

Create temporary here

Use it here
Example: Here’s Where $t$ is Used

$= x+y$
$t = x+y$
$t = x+y$
$t = t$

$= t$