Partial Redundancy Elimination

Motivation

```c
if() {
    a = x + y;
}
b = x + y;
```

Motivation

```c
while(c) {
    a = x + y;
}
```
Assumption

Assume that at any statement of the form \( a = x + y \), the current value of \( x + y \) must be placed in \( a \). That is, the computation of \( x + y \) cannot be deferred until a later point where \( a \) is actually used.
Desired Transformation

Introduce a temporary $t_{x+y}$. Change every statement of the form $a = x + y$ into $a = t_{x+y}$. Insert computations of the form $t_{x+y} = x + y$ at some subset $S$ of program points (nodes and edges) such that the same values are assigned to $a$ as in the original program. [Safe]

Goals for $S$

1. Suppose $S'$ is also safe. No execution path should contain more occurrences of $t_{x+y} = x + y$ in $S$ than in $S'$. [Computationally Optimal]

2. Suppose $S'$ is also safe and computationally optimal. At every program point where $t_{x+y}$ is live under $S$, it should also be live under $S'$. [Lifetime Optimal]
Note: this is not an exhaustive list.

Summary of Properties

- Local properties
  - transparent
  - computed
  - locally anticipable
- Global node properties
  - available
  - anticipable
- Global edge properties
  - earliest
  - later
- Final results
  - insert (on edge)
  - delete (from node)
**Definition**

A basic block $b$ is **transparent** for expression $e$ if none of $e$’s operands are defined in $b$.

**Definition**

An expression $e$ is **computed** (aka downward exposed aka locally available) in basic block $b$ if it contains a computation of $e$, and does not define $e$’s operands after the last computation of $e$.

**Definition**

An expression $e$ is **locally anticipable** (aka upward exposed) in basic block $b$ if it contains a computation of $e$, and does not define $e$’s operands before the first computation of $e$. 
Definition
An expression $e$ is available at program point $p$ if on every path from the start node to $p$, $e$ is computed, and $e$'s operands are not defined after the last computation of $e$.

Compute using dataflow analysis:

1. forward
2. $(\text{Exprs}, \supseteq)$
3. $\cap$
4. $\text{out}(s) = \text{computed}(s) \cup (\text{in}(s) \cap \text{transparent}(s))$
5. empty set
6. $\perp = \text{all expressions}$
Definition

An expression $e$ is **anticipable** at program point $p$ if on every path from $p$ to the end node, $e$ is computed, and $e$’s operands are not defined before the first computation of $e$.

Compute using dataflow analysis:

1. backward
2. $(\text{Exprs}, \supseteq)$
3. $\cap$
4. $\text{in}(s) = \text{locally anticipable}(s) \cup (\text{out}(s) \cap \text{transparent}(s))$
5. empty set
6. $\bot = \text{all expressions}$
Earliest Placement

The edge \((i, j)\) is the earliest point where we should compute the expression \(e\) if

- \(e\) is needed on all paths from \(j\) to the end node,
- \(e\) is not available at the end of \(i\), and
  - a computation before \(i\) would get invalidated in \(i\), or
  - \(e\) is not needed on some other edge out of \(i\).

\[
\text{earliest}(i, j) = \frac{\text{anticipable}_{\text{in}}(j)}{\cap \text{available}_{\text{out}}(i)} \cap \text{transparent}(i) \cup \text{anticipable}_{\text{out}}(i))
\]
A computation of $e$ can be moved from before a block $b$ to after $b$, as long as it needs to be computed on all incoming edges of $b$, and $e$ is not needed in $b$.

$$
\text{later}(i, j) = \text{earliest}(i, j) \\
\quad \cup \left( \text{locally anticipable}(i) \cap \bigcap_{k \in \text{pred}(i)} \text{later}(k, i) \right)
$$
The Transformation

Insert computation as late as possible:

\[
\text{insert}(i,j) = \text{later}(i,j) \cap \bigcap_{k \in \text{pred}(j)} \text{later}(k,j)
\]

\(e \in \text{insert}(i,j)\) means compute \(e\) in edge \((i,j)\).

Remove locally anticipable computations where value is already known:

\[
\text{delete}(j) = \text{locally anticipable}(j) \cap \bigcap_{i \in \text{pred}(j)} \text{later}(i,j)
\]

\(e \in \text{delete}(j)\) means remove first computation of \(e\) from \(j\).