Contents of Lecture 7

- What can PRE achieve?
- Partial Redundancy Elimination History
- Key ideas in SSAPRE from SGI
Recall that Partial Redundancy Elimination, or **PRE**, can eliminate both **full** and **partial** redundancies.

- **Full redundancies**: when the expression is available from all predecessor basic blocks.
- **Partial redundancies**: when the expression is only available from some but not all predecessor basic blocks.
- **Partial redundancies** also covers loops, i.e. PRE can move code out from loops.
PRE was invented by Morel and Renvoise in 1979.

Then Fred Chow in his PhD thesis at Stanford from 1983 (with John Hennessy as supervisor) improved it.

In 1992 Knoop et al. published a version of PRE which is optimal in the sense of minimizing register pressure. They called their algorithm Lazy Code Motion.

In 1999 Kennedy and Chow and others at SGI published the SSA formulation of Lazy Code Motion and called it SSAPRE.

We will first study a simpler version of it and then note that there exists an efficient variant of SSAPRE which is much faster.
Limitations of Value Numbering

Both hash-based and global value numbering can optimize the full redundancy in vertex 1.

None of them can optimize the partial redundancy in vertex 3.
The Key Idea of SSAPRE

- We create \( \Phi \)-functions for the hypothetical variable \( h \).
- After SSAPRE, \( \Phi \)-functions become normal \( \phi \)-functions and they are really the same (different notation to distinguish between them only).
- By inserting the expression \( a + b \) at \( \Phi \)-operands with the value \( \bot \) ("bottom"), the partial redundancy in vertex 3 becomes a full redundancy and can be eliminated.
Insert $\Phi$-functions.
Perform SSA-renaming for the variable $h$ and all other variables (again).
Compute $\text{downsafety}$, i.e. where the expression is anticipated.
Compute $\text{can\_be\_avail}$, i.e. where the expression can be available, either because the expression is there or it can replace a $\bot$-operand.
Compute $\text{later}$, i.e. if can be lazy and insert the expression further down in the control flow graph.
Perform $\text{finalize1}$, i.e. modify the code.
Perform $\text{finalize2}$, i.e. clean up various things.
Insertion of $\Phi$-functions

Recall that in SSAPRE every expression assigns to a hypothetical variable $h$.

Where should we then insert $\Phi$-functions for $h$?

1. In the iterated dominance frontiers of all evaluations of the expression, i.e. assignment to $h$.
2. In the iterated dominance frontiers of all assignments to operands in the expression — since they mean $h \leftarrow \bot$.
We have already computed the dominance frontiers of each vertex.

We thus simply have to collect the vertices which contain such an evaluation.
Although we can collect all vertices with assignments to \( a \) or \( b \), and find the iterated dominance frontiers of these, there is a simpler way.

Every vertex for which we will insert a \( \Phi \)-function due to an \( h \leftarrow \perp \) must contain a \( \phi \)-function to any of the variables in the expression, i.e. \( \phi(a) \) or \( \phi(b) \).

So we simply look for \( \phi(a) \) and \( \phi(b) \), and insert \( \Phi(h) \) in the same vertex.

Recall that \( \phi \)-functions are parallel copy statements.
An expression is anticipated at a point \( p \) in the control flow graph if it is certain it will be evaluated with all operands having the same value on all paths from \( p \).

At the end of vertex 0, \( a + b \) is not anticipated since \( a \) might be assigned a new value in vertex 4.

At the end of vertices 1 and 4 the expression is anticipated due to the evaluation in vertex 2 which certainly will be evaluated.

The word "evaluated" here means "executed".
The Main Rule of the Game of PRE

No matter what, PRE may never transform a function so it will execute additional instructions due to PRE.

Should the \( \perp \) in vertex 2 be replaced with \( h \leftarrow a + b \)?

No, it’s not safe to insert the expression since the expression is **not anticipated** by the \( \Phi \)-function.

The path \((0, 7, 2, 3, 4, 5)\) would execute \( a + b \) at the end of vertex 7 (for the \( \Phi \)-operand) without any purpose.

Actually, a \( \Phi \)-operand is regarded as belonging to the predecessor vertex.
There are three main types of so called occurrences of an expression:

1. A **real occurrence**, i.e. the expression \( a + b \),
2. A **Φ-function occurrence**, and
3. A **Φ-operand occurrence**.

Note that Φ-operands are placed in the predecessor basic block.
Each $\Phi$-function has a number of boolean attributes:

- **downsafe** or **ds**
- **can_be_available** or **cba**
- **later**
- **will_be_available** or **wba**

If a $\Phi$-function is downsafe, it’s OK to replace a $\bot$ operand with the expression.

We will soon see how downsafe is computed.

A $\Phi$-operand has the boolean attribute **has_real_use** which is true if the value comes from a real occurrence.
Renaming traverses the dominator tree and links uses with definitions of $h$ variables.

- At a $\Phi$-function occurrence, a new version of $h$ is always created.
- At a $\Phi$-operand occurrence it is noted if the value comes from a real occurrence, in which case `has_real_use` is set to true.
- At a real occurrence, a new version of $h$ is created if the top of stacks of $a$, $b$, and $h$ don’t have the same versions.
- Both real and $\Phi$-function occurrences are pushed on the rename stack of $h$. 

```
0
  a0 = x
  b0 = y
  h0 = a0 + b0

1
  a2 ← x

2
  a1 ← \phi(a0, a2)
  h1 ← \Phi(h0, ⊥)

3

4

5
  h2 ← \Phi(h1, h3)

6
  h3 ← a1 + b0

7
```

```
Initialization of Downsafe

Recall that a $\Phi$-function is downsafe if all paths from it evaluate $a + b$ (with the same variable versions).

Thus, if there is a path from a $\Phi$-function to the exit vertex that $\Phi$-function is not downsafe unless the expression was evaluated.

When renaming comes to the exit vertex, it checks the top of the stack of $h$.

If the top is a $\Phi$-function, it is marked with $ds = 0$. 
After the initialization of downsafety during rename, the downsafety is computed for all $\Phi$-functions.

What should be done?

- A $\Phi$-function with $ds = 0$ should tell other $\Phi$-functions that also they are not downsafe!

- A $\Phi$-function with $ds = 0$ and with a $\Phi$-operand that is defined by a $\Phi$-function and for which $\text{has\_real\_use} = 0$, should reset its downsafety and continue the recursion.

In this example both $\Phi$-functions have $ds = 0$. 

\begin{align*}
a_0 &= x \\
b_0 &= y \\
h_0 &= a_0 + b_0 \\
a_1 &\leftarrow \phi(a_0, a_2) \\
h_1 &\leftarrow \Phi(h_0, \bot) \\
a_2 &\leftarrow x \\
h_2 &\leftarrow \Phi(h_1, h_3) \\
h_3 &\leftarrow a_1 + b_0 \\
&\text{set to } ds = 0 \end{align*}
procedure reset_downsafe(x)
    if (has_real_use(x) or def(x) is not a Φ)
        return
    f ← def(x)
    if not down_safe(f))
        return
    down_safe(f) ← false
    for each operand ω of f do
        reset_downsafe(ω)

procedure downsafety
    for each f ∈ F do
        if not down_safe(f))
            for each operand ω of f do
                reset_downsafe(ω)
procedure compute_can_be_avail
    for each $f \in \mathcal{F}$ in the program do
        $can\_be\_avail(f) \leftarrow \text{true}$
    end
    for each $f \in \mathcal{F}$ in the program do
        if (not $\text{down\_safe}(f)$
            and $can\_be\_avail(f)$
            and $\exists$ an operand of $f$ that is $\bot$)
            $reset\_can\_be\_avail(f)$
        end
end
procedure reset_can_be_avail(g)
    can_be_avail(g) ← false
    for each $f \in \mathcal{F}$ with operand $\omega$ with $g = \text{def}(\omega)$ do
        if (not has_real_use($\omega$)
        and not downsafe($f$)
        and can_be_avail($f$))
            reset_can_be_avail($f$)
    end
procedure reset\_later(g)
  \( \text{later}(g) \leftarrow \text{false} \)
  for each \( f \in \mathcal{F} \) with operand \( \omega \) with \( g = \text{def}(\omega) \) do
    if (\( \text{later}(f) \))
      reset\_later(f)
  end

procedure compute\_later
  for each \( f \in \mathcal{F} \) do
    \( \text{later}(f) \leftarrow \text{can\_be\_avail}(f) \)
    for each \( f \in \mathcal{F} \) do
      if (\( \text{later}(f) \)) and
        \( \exists \) an operand \( \omega \) of \( f \) such that \( \text{def}(\omega) \neq \bot \) and \( \text{has\_real\_use}(\omega) \))
        reset\_later(f)
    end
end

procedure will\_be\_avail
  compute\_can\_be\_avail
  compute\_later
end
procedure finalize1(g)
    let E ← the current expression
    for each redundancy class x of E do
        avail_def[x] = ⊥
    for each occurrence ψ of E in preorder DT traversal order do
        x ← class(ψ)
        if (ψ is a Φ occurrence) {
            if (will_be_avail(ψ))
                avail_def[x] = ψ
        } else if (ψ is a real occurrence) {
            if (avail_def[x] is ⊥ or avail_def[x] does not dominate ψ)
                reload(ψ) ← false
                avail_def[x] = ψ
            } else {
                reload(ψ) ← true
                def(ψ) ← avail_def[x]
            }
        } else {
            /* ψ is a Φ operand occurrence. */
            let f be the Φ in the successor vertex of this operand
            if (will be avail(f)) {
                if (ψ satisfies insert) {
                    insert E at the end of the vertex containing ψ
                    def(ψ) ← inserted occurrence
                } else
                    def(ψ) ← avail_def[x]
            }
        }
end