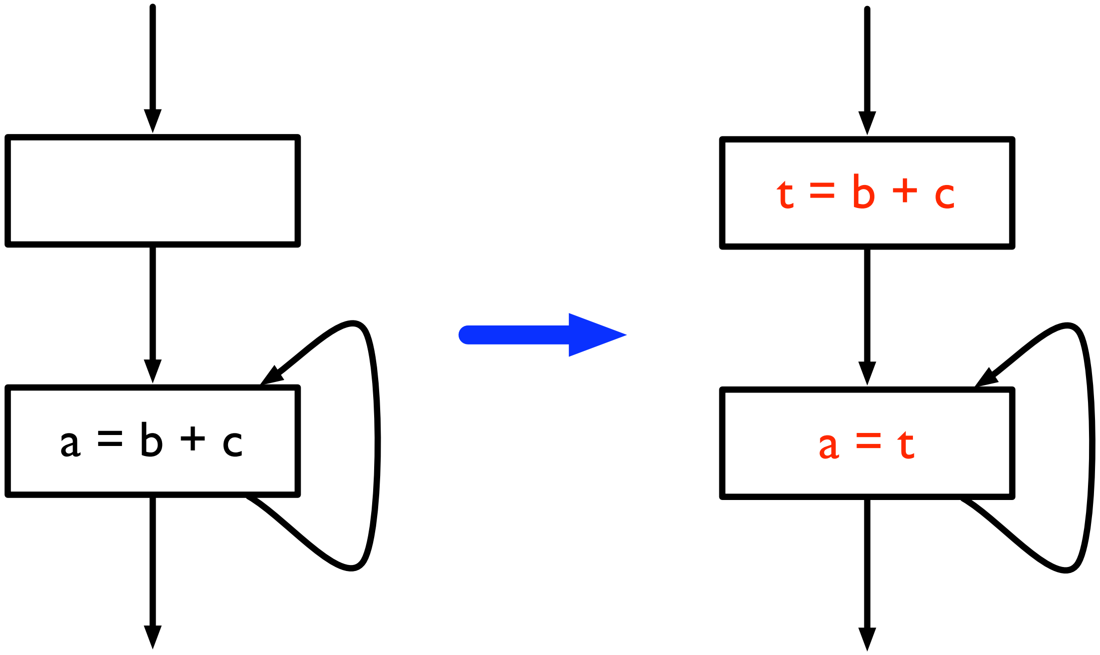
Partial Redundancy Elimination (PRE)

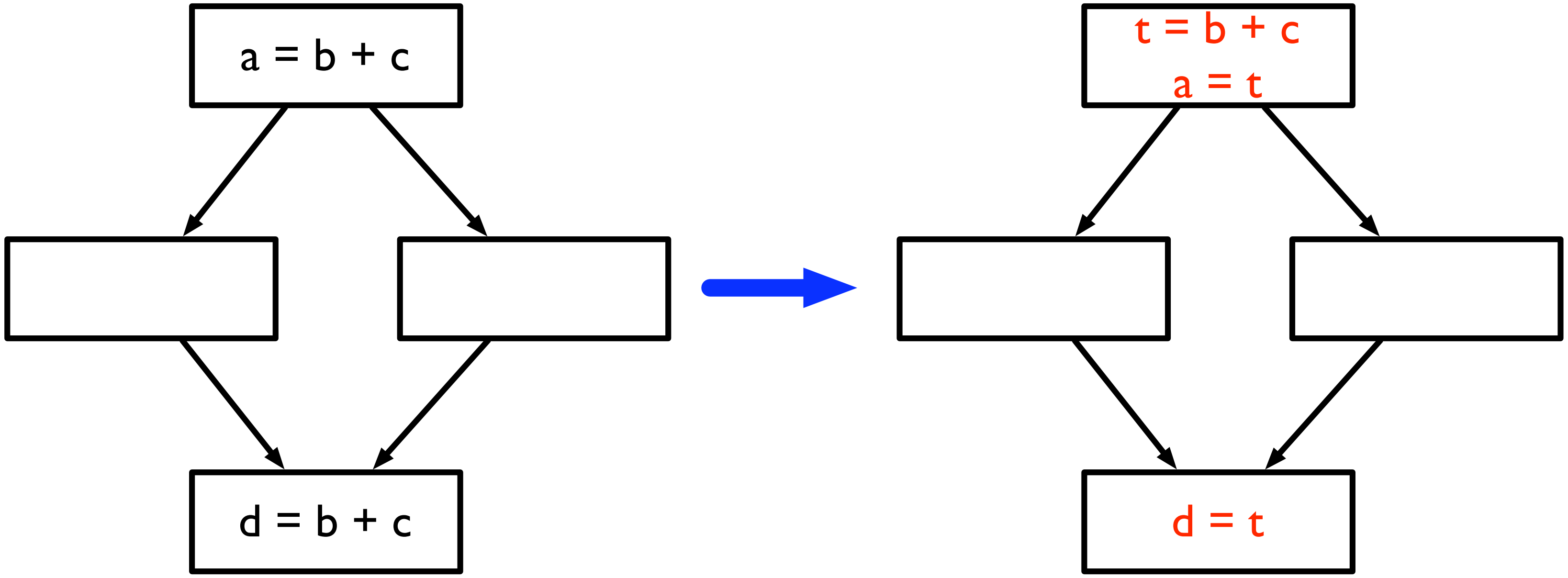
Loop invariant code motion

- Move invariant evaluations of expressions out of loops
  - Identify invariant statements, hoist them out of loop



Common subexpression elimination

- Remove redundant computations of expressions
  - Compute *available* expressions, replace expressions that are available with already-computed expression



a = b + c

d = b + c

t = b + c

a = t

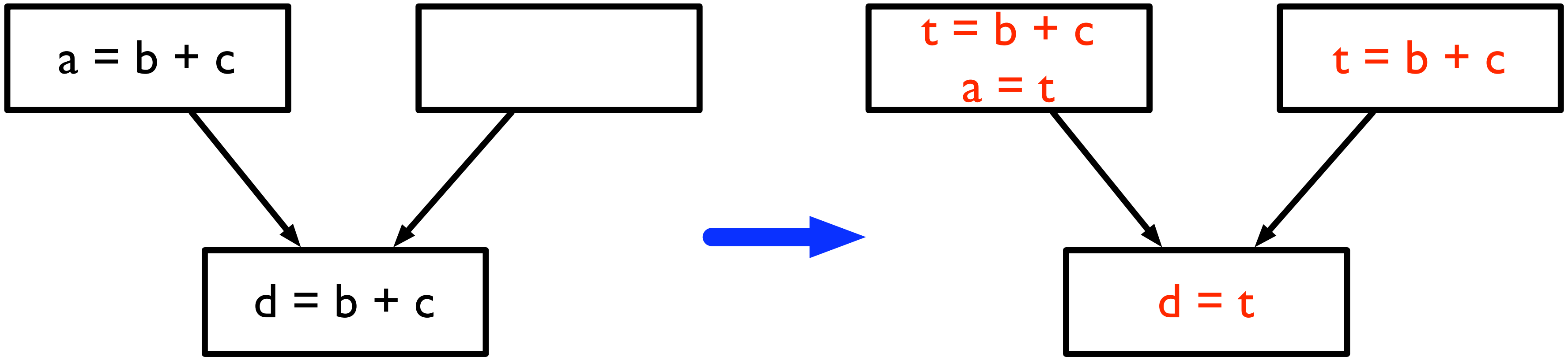
d = t

Removing total redundancies

- Both loop-invariant code motion and common subexpression elimination focus on removing total redundancy
    - Focus on computations which are computed multiple times along every path
    - Are these the only kinds of redundancies?

Partial redundancy

- An expression calculated once along one path, but twice along another
  - Move code to remove *partial* redundancy



One optimization can cover all of these cases

- *Partial redundancy elimination (PRE)*
  - One of the most complex dataflow analyses
  - Subsumes common subexpression elimination and loop invariant code motion
- Originally proposed in 1979 by Morel and Renvoise
  - Used a bi-directional dataflow analysis
- Reformulated by Knoop, Rüthing and Steffen in 1992
  - Uses a backward dataflow analysis followed by a forward analysis
- We will discuss this latter formulation

Partial redundancy elimination

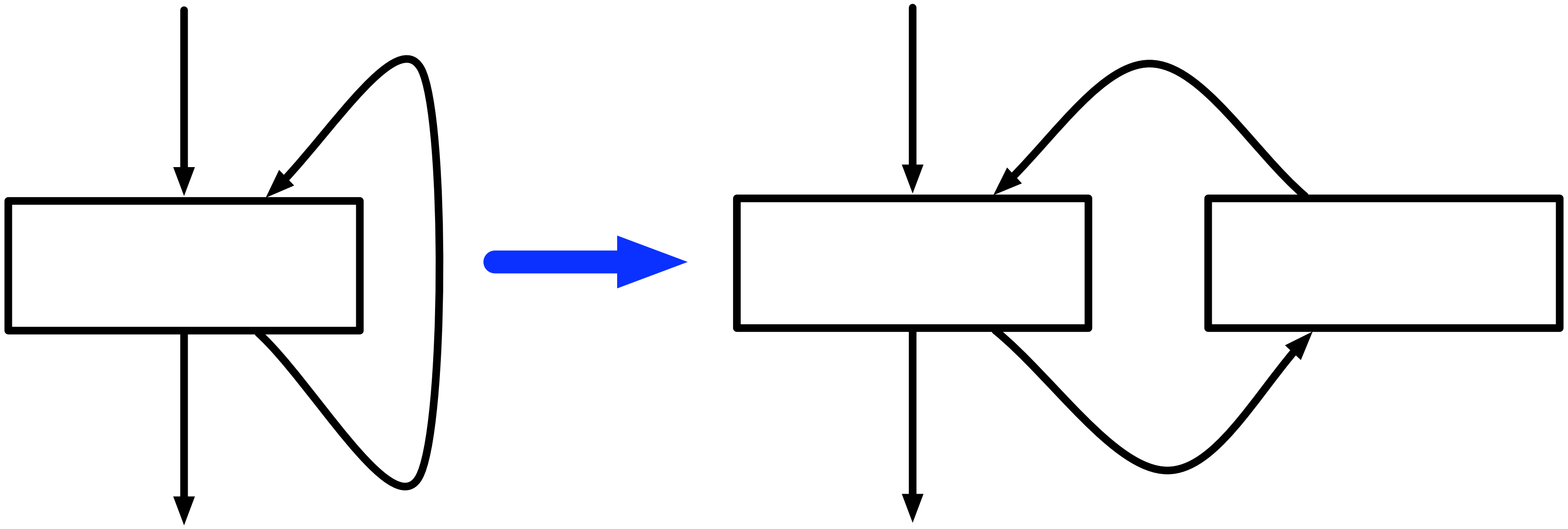
- High level picture:
  - Consider a single expression (b + c)
  - Find CFG nodes where expression will be used before its result is invalidated (*down-safety*)
  - Find CFG nodes where expression has already been evaluated (*up-safety*)
  - Use this information to determine optimal location to evaluate expression

Some particulars

- Will consider just a single expression
    - The flow functions presented operate over a 1-0 lattice
    - Can easily extend this to multiple expressions by using a bit vector lattice
- Only one assignment per CFG node (no aliasing)
- Insert empty blocks before each join node (allowing code to be placed in block)

More particulars

- No edges from branch node directly to join node
  - Must insert empty node



Down-safety

- General idea in PRE: move computation earlier in the program to produce redundancy (which can later be eliminated)
- When can an expression be placed in a node?
  - If expression is calculated on all paths from the node
    - Do not want to evaluate an expression unnecessarily
  - If the operands of the expression are not changed before subsequent uses
    - Do not want to evaluate an expression only to have to re-evaluate it

Down-safety (II)

- *Used(n)* – true if expression (b + c) is calculated in node *n*
- *Transparent(n)* – true if neither b nor c are defined in *n*
- Key insight: if *transparent(n)* and all successors of *n* are down-safe, then *n* is down-safe

$$Dsafe(n) = Used(n) \vee (Transp(n) \wedge \bigwedge_{s \in succ(n)} Dsafe(s))$$

- This can be computed with a straightforward backward dataflow analysis
  - Dsafe(exit) = false

Down-safety (III)

- Called *anticipatable* in the Drechsler and Stadel paper
- Also the same as *very busy* expressions

Very-busy expressions

- An expression is *very busy* at a node if it is computed on every path leading from a node

$$\begin{array}{rcl} IN(s) & = & \mathbf{gen}(s) \cup (OUT(s) - \mathbf{kill}(s)) \\ OUT(s) & = & \bigcap_{t \in succ(s)} IN(t) \end{array}$$

- gen(s): the expressions calculated in a statement
  - Same as *used*
- kill(s): the expressions whose operands are redefined in a statement
  - Same as ¬*transp*
- IN(s) is the same as Dsafe(n)

Up-safety

- Where is it *unnecessary* to recompute an expression?
  - If the expression has already been calculated along every incoming path
  - Should just re-use results of previous computation, rather than re-computing

$$Usafe(n) = \bigwedge_{p \in pred(n)} (Transp(p) \wedge (Used(p) \vee Usafe(p)))$$

- Similar to *available expressions*

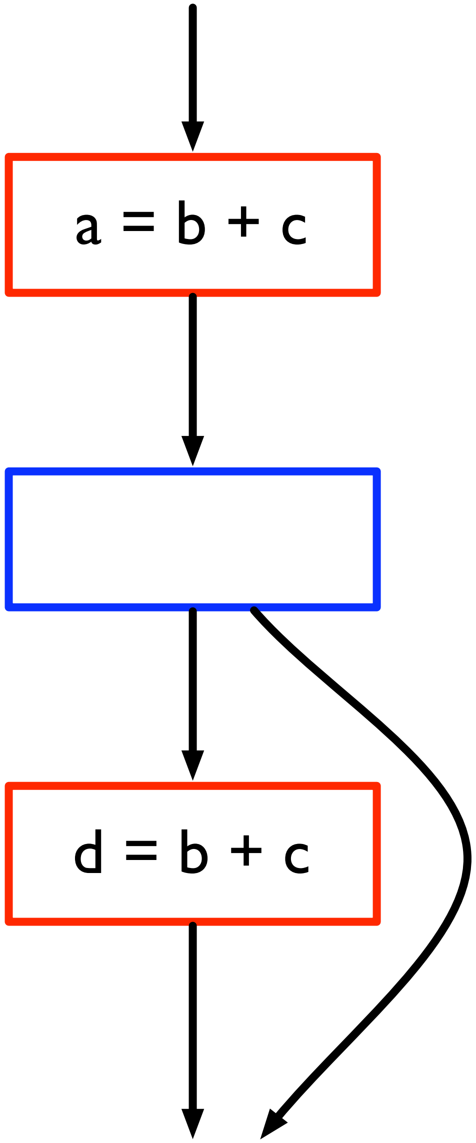
$$\begin{aligned} IN(s) &= \textstyle\bigcap_{t \in pred(s)} OUT(t) \\ OUT(s) &= (IN(s) \cup \mathbf{gen}(s)) - \mathbf{kill}(s) \end{aligned}$$

Where to place expressions?

- Any downsafe node is a valid place for an expression
  - But clearly do not want to place expressions in *all* downsafe nodes
  - Want to minimize number of times expression is evaluated
  - Place expression in *earliest* downsafe position
- Intuition
  - Definitely earliest if it's the start node
  - Earliest if a predecessor isn't transparent
    - Need to recalculate expression along that path
  - Earliest if has a predecessor that is not downsafe
    - Predecessor isn't a valid place to place expression
  - Predecessor should also not be upsafe
    - Why?

Why no upsafety?

- Consider the example
- Red nodes are downsafe
- Blue node is upsafe
  - Shouldn’t place expression in bottom node because the expression has already been calculated by the first node



Earliest downsafe node

- Equation to capture conditions

$$\begin{aligned} Earliest(n) = \quad & Dsafe(n) \wedge \\ & \bigvee_{pred(n)} (\neg Transp(p) \vee (\neg Usafe(p) \wedge \neg Dsafe(p))) \end{aligned}$$

  - Note: not recursive, so no need for fixpoint computation
- Can now transform code:
  - Place expression t = b + c at all nodes marked *earliest*
  - Replace all other uses of b + c with t

Delaying placement

- May want to place expressions later than *earliest*
  - Why? To minimize live ranges of temporaries
- Calculate *Delay(n)* to determine if placement can be delayed to this node

$$
\begin{aligned}
Delay(n) &= Earliest(n) \vee \\
& \bigwedge_{p \in pred(n)} (\neg Used(p) \wedge Delay(p))
\end{aligned}
$$

- Obviously can delay if the node is earliest
- Can also delay if expression is not used in any predecessor and can be delayed to all predecessors

Latest

- Find the latest node to which we can delay placement:

$$Latest(n) = Delay(n) \wedge (Used(n) \vee \bigvee_{s \in succ(n)} \neg Delay(s))$$

  - Note: not recursive

- What is the purpose of each clause?

SSAPRE

A sparse version of PRE

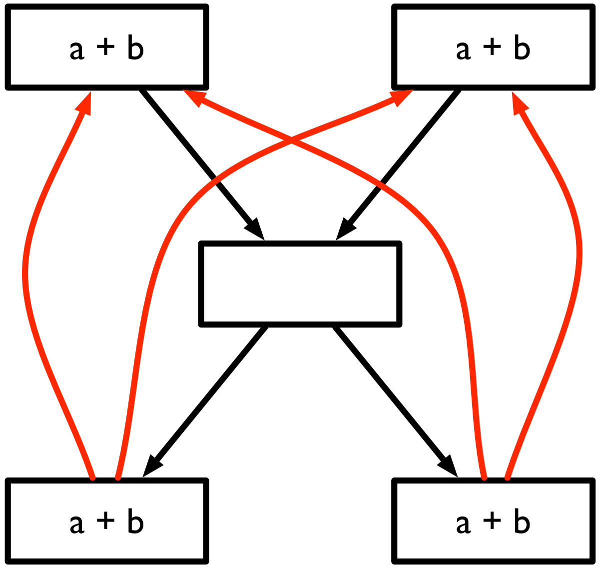
- PRE as presented operates over the CFG
  - Calculate downsafety and upsafety by looking at predecessors and successors in CFG
- Can we calculate PRE in a sparse manner, as we did for CP?
- Solution: SSAPRE
  - "Partial Redundancy Elimination in SSA Form," Kennedy *et al.*

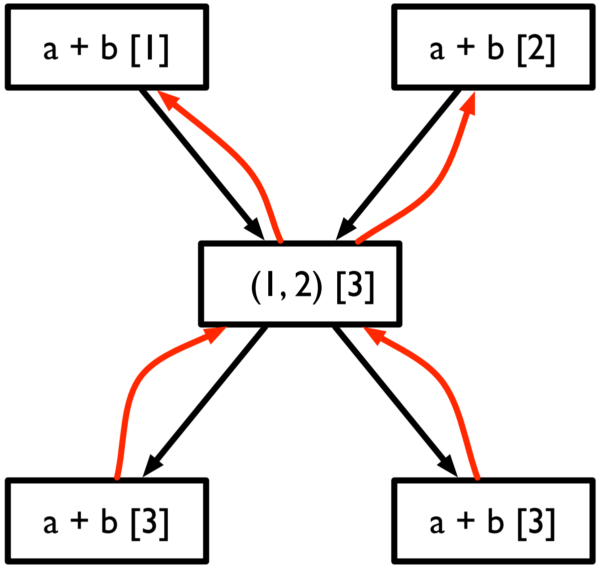
Factored Redundancy Graph

- Sparse representation that captures redundancy between expressions
  - Intuition: like SSA form for expressions
  - Problem: no notion of "uses" and "defs" for expressions
  - Instead, track computations of expression *E*
  - *E* is "defined" when it is computed
  - *E* is "used" when it is computed in a redundant way
    - There is a path leading from a previous computation to this one where the operands of *E* are not redefined

Factored Redundancy Graph

- Can construct "redundancy graph"
  - Nodes for each computation of expression *E*
  - *Redundancy edge* from node x to node y if computation in x is redundant with respect to y
- Factored redundancy graph is like SSA for redundancy relation
  - Φ-node for each merge point where two computations of *E* come together
  - Also insert Φ-nodes where *E* only computed along one incoming path. Set other operand to ⊥
  - Edges (called "upward edges") from a node to the computation-node or Φ-node that dominates it





Central insight

- Suppose we perform optimal PRE for an expression *E*, inserting computations of temporary *t* at some sites and replacing other computations with uses of *t*
- Every use-def relation for *t* corresponds directly to a redundancy edge for *E*
- If a redundancy edge is not captured by a use-def edge of *t*, then this means either
    - Redundancy could not be safely exploited or
    - Expression has same value on both sides of redundancy edge (so no need to recalculate)
- Goal of SSEPRE: figure out which redundancy edges for *E* should turn into use-def edges for *t*

Constructing FRG

- Insert Φ nodes
  - Just like in SSA
- Rename expressions
  - A “def” in the FRG and its corresponding “uses” represents a *redundancy class*
  - Give each redundancy class a unique name
- Perform PRE over FRG

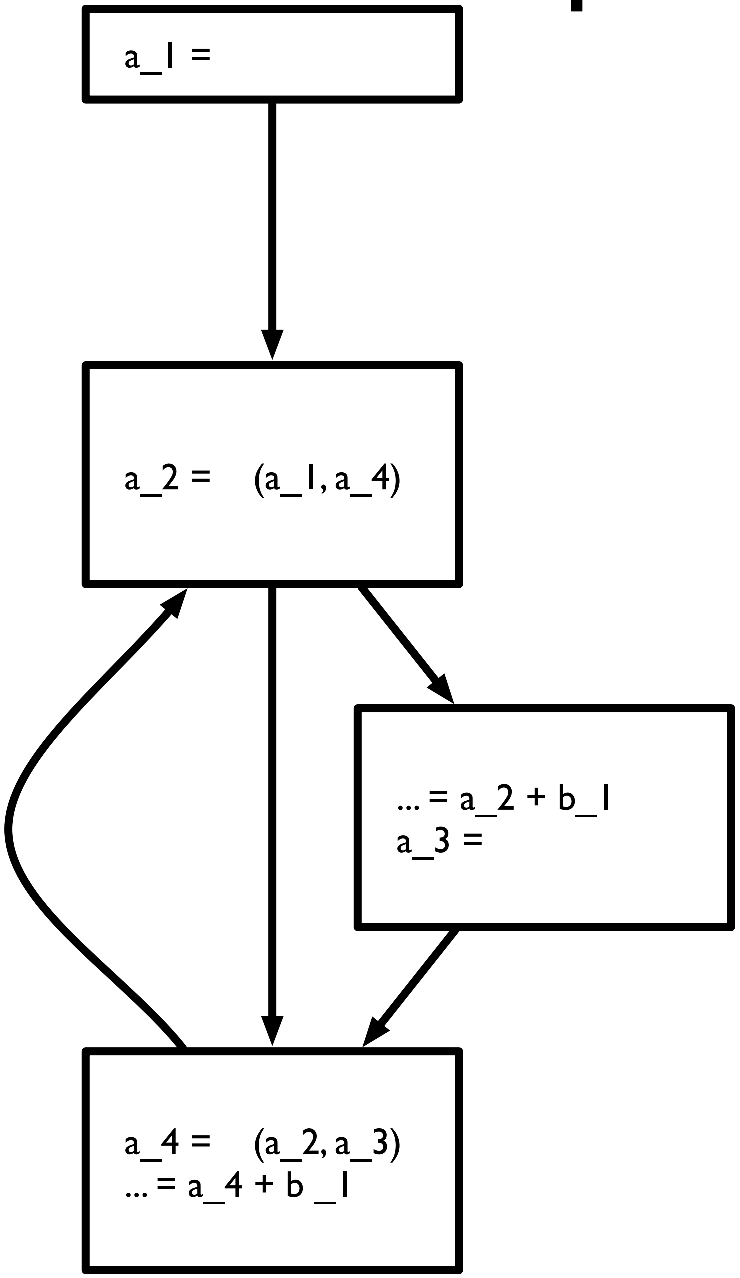
Φ-insertion

- Insert a Φ node at the iterated dominance frontier of each occurrence of *E*
  - Because each occurrence of *E* represents a potential definition of *t*
- Insert a Φ node at every block where there is a φ-node for one of the expression's operands
  - Existence of φ-node indicates result of *E* has changed by this merge point, and so may need to be recalculated

Renaming step

- Give each occurrence of *E* a name (similar to naming versions of variables in SSA)
- Three occurrences
  - Φ-*node*: give occurrence a new class number
  - *Real* (original) occurrence: if current operands of *E* match versions of operands in previous use of *E*, use appropriate class number, otherwise generate new one
  - *Operand of* Φ-*node*: if current operands of *E* match versions of operands in previous use of *E*, use appropriate class number, otherwise, use ⊥
- Invariant: if two occurrences of *E* have same class number, they produce the same result. If not, then there must be an intervening redefinition of operand, or a Φ-node

FRG example



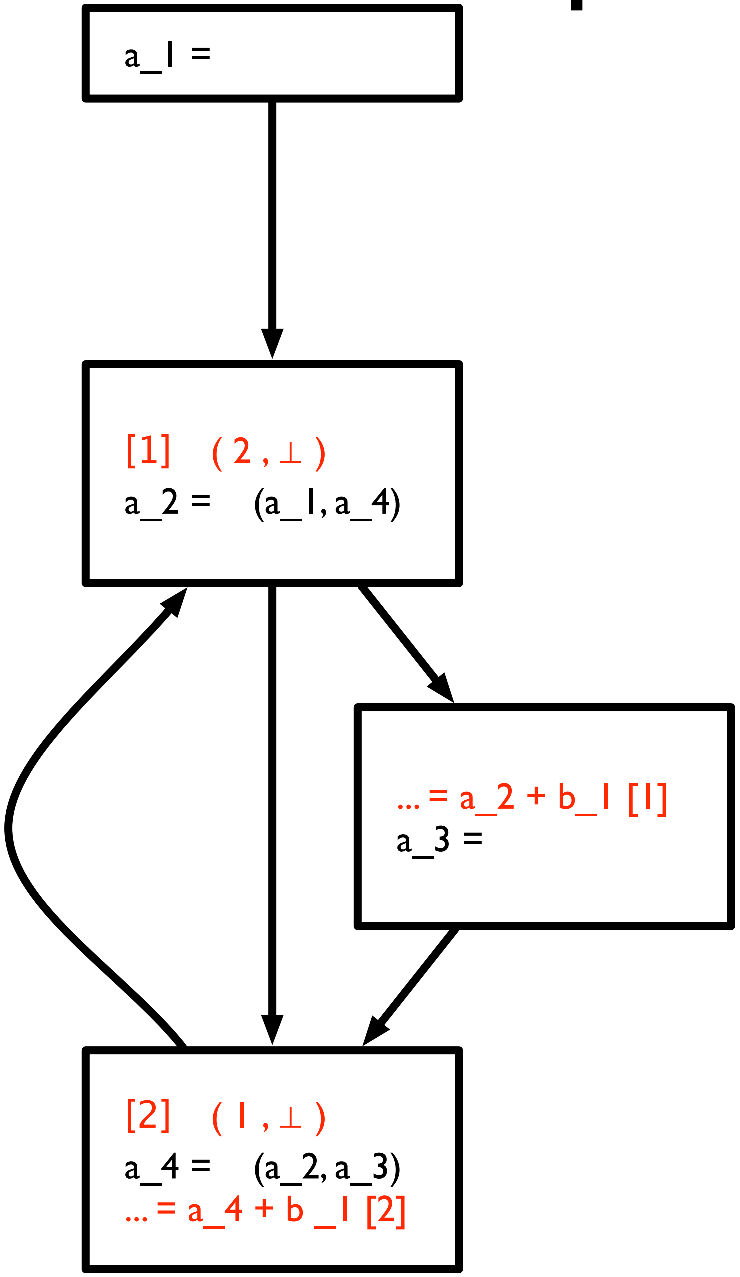
a_1 =

a_2 =   (a_1, a_4)

... = a_2 + b_1
a_3 =

a_4 =   (a_2, a_3)
... = a_4 + b _1

FRG example



a_1 =

[1] (2 , ⊥)

a_2 = (a_1, a_4)

... = a_2 + b_1 [1]

a_3 =

[2] (1 , ⊥)

a_4 = (a_2, a_3)

... = a_4 + b _1 [2]

Calculating down-safety

- Trick: Insertions of computation only necessary at Φ-nodes, so only need to consider downsafety there
- a Φ-node *isn't* downsafe if one of two cases is true
  - There is a path to the exit where Φ-node's redundancy class does not appear (which means expression is not calculated before the exit)
  - There is a path from Φ-node to another Φ-node which is not downsafe *and* there is no real occurrence of redundancy class (which means that expression is not actually calculated before we get to a non-downsafe node)
- All downsafe Φ-nodes are valid places to calculate an expression (i.e., by evaluating expression in predecessors)

Will be available

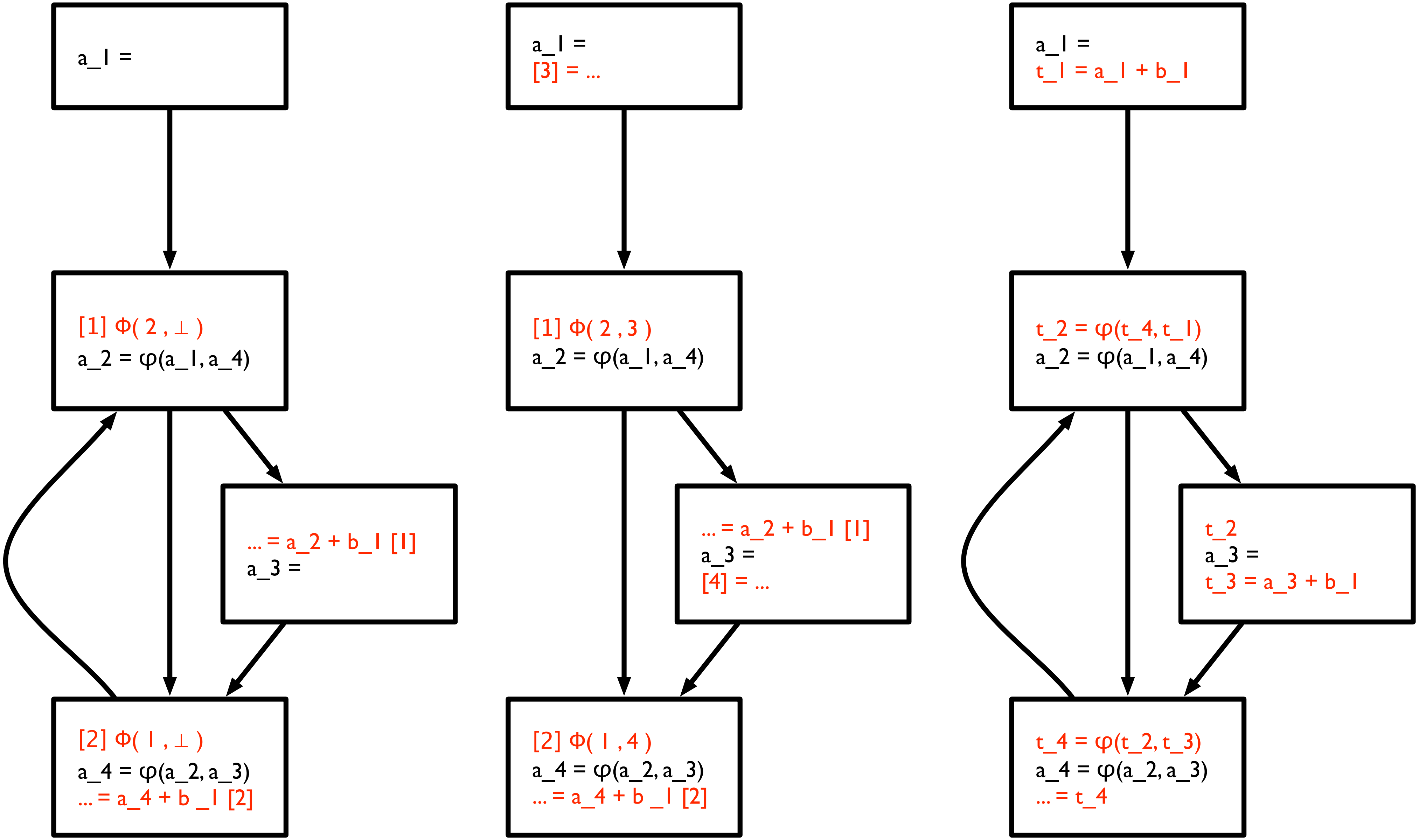
- Φ-nodes where expression will be available *after* PRE has happened are labeled WillBeAvailable
- Intuition:
  - WillBeAvailable is true if *E* can be made available (because there is some downsafe set of nodes which will make *E* available here) and *E* cannot be computed later instead

Inserting computation

- Insert additional evaluations of *E* to produce operands of Φ nodes where WillBeAvailable is true and:
    - operand is ⊥ (*E* hasn't been calculated yet) or
    - no actual computation of *E* on path to operand but Φ node leading to operand does not satisfy WillBeAvailable (*E* isn't calculated along path *and E* won't be available already)
- Some occurrences of *E* will be *reloaded* from temporary
    - If *E* is dominated by a computation of *E* (incl. Φ nodes)
- Other occurrences of *E* will be *saved* to the temporary
    - If *E* is the *inserted* operand of a Φ-node (but not other operands)
    - If *E* dominates a *reloaded E*

Generating code

- Walk over FRG
- At a real occurrence of *E*
  - If *save* is true, compute expression, save in new version of *t*
  - If *reload* is true, load result from appropriate *t* (from the computation of *E* that dominates this occurrence)
  - If *insert* is true, compute expression, save in new version of *t*
- At Φ-node
  - Replace with φ-node for *t*



a_1 =

[1] Φ(2 , ⊥)
a_2 = φ(a_1, a_4)

... = a_2 + b_1 [1]
a_3 =

[2] Φ(1 , ⊥)
a_4 = φ(a_2, a_3)
... = a_4 + b _1 [2]

a_1 =
[3] = ...

[1] Φ(2 , 3)
a_2 = φ(a_1, a_4)

... = a_2 + b_1 [1]
a_3 =
[4] = ...

[2] Φ(1 , 4)
a_4 = φ(a_2, a_3)
... = a_4 + b _1 [2]

a_1 =
t_1 = a_1 + b_1

t_2 = φ(t_4, t_1)
a_2 = φ(a_1, a_4)

t_2
a_3 =
t_3 = a_3 + b_1

t_4 = φ(t_2, t_3)
a_4 = φ(a_2, a_3)
... = t_4