CS738: Advanced Compiler Optimizations

SSAPRE: SSA based Partial Redundancy Elimination

Amey Karkare
karkare@cse.iitk.ac.in

http://www.cse.iitk.ac.in/~karkare/cs738
Department of CSE, IIT Kanpur

PRE without SSA

- Based on well known DF analyses

PRE without SSA

- Based on well known DF analyses
  - Availability

PRE without SSA

- Based on well known DF analyses
  - Availability
  - Anticipability

PRE without SSA

- Based on well known DF analyses
  - Availability
  - Anticipability
  - Partial Availability

PRE without SSA

- Based on well known DF analyses
  - Availability
  - Anticipability
  - Partial Availability
  - Partial Anticipability

PRE without SSA

- Based on well known DF analyses
  - Availability
  - Anticipability
  - Partial Availability
  - Partial Anticipability
- Identifies partially redundant computations, make them totally redundant by inserting new computations

PRE without SSA

- Based on well known DF analyses
  - Availability
  - Anticipability
  - Partial Availability
  - Partial Anticipability
- Identifies partially redundant computations, make them totally redundant by inserting new computations
- Remove totally redundant computations (CSE)

PRE without SSA

- Iterative data flow analysis

PRE without SSA

- Iterative data flow analysis
- Operates on control flow graph

PRE without SSA

- Iterative data flow analysis
- Operates on control flow graph
- Computes global and local versions of data flow information

SSAPRE

- Information flow along SSA edges

SSAPRE

- Information flow along SSA edges
- No distinction between global and local information

SSAPRE: Challenge

- SSA form defined for variables

SSAPRE: Challenge

- SSA form defined for variables
- How to identify potentially redundant expressions

SSAPRE: Challenge

- SSA form defined for variables
- How to identify potentially redundant expressions
  - Expressions having different variable versions as operands

SSAPRE: Challenge

- SSA form defined for variables
- How to identify potentially redundant expressions
  - Expressions having different variable versions as operands

$$\begin{tikzcd}
\boxed{a_1 + b_1} \arrow[dr] & & \boxed{a_2 = \cdots} \arrow[dl] \\
& \boxed{a_3 = \phi(a_1, a_2)} \arrow[d] & \\
& \boxed{a_3 + b_1} &
\end{tikzcd}$$

  - Here $a_1 + b_1$ is same as $a_3 + b_1$ when control follows the left branch. Lexically different, but computationally identical

SSAPRE: Key Idea

- Redundancy Class Variables (RCVs)

SSAPRE: Key Idea

- Redundancy Class Variables (RCVs)
  - variable (say h) to represent computation of an expression (say E)

SSAPRE: Key Idea

- Redundancy Class Variables (RCVs)
  - variable (say h) to represent computation of an expression (say E)
- Computation of expression could represent either a *def* or a *use*

SSAPRE: Key Idea

- Redundancy Class Variables (RCVs)
  - variable (say h) to represent computation of an expression (say E)
- Computation of expression could represent either a *def* or a *use*
  - definition of $E \Rightarrow$ store into h

SSAPRE: Key Idea

- Redundancy Class Variables (RCVs)
  - variable (say h) to represent computation of an expression (say E)
- Computation of expression could represent either a *def* or a *use*
  - definition of $E \Rightarrow$ store into h
  - use of $E \Rightarrow$ load from h

SSAPRE: Key Idea

- Redundancy Class Variables (RCVs)
  - variable (say h) to represent computation of an expression (say E)
- Computation of expression could represent either a *def* or a *use*
  - definition of $E \Rightarrow$ store into h
  - use of $E \Rightarrow$ load from h
- PRE on SSA form of RCVs (h) to remove redundancies

SSAPRE: Key Idea

- Redundancy Class Variables (RCVs)
  - variable (say h) to represent computation of an expression (say E)
- Computation of expression could represent either a *def* or a *use*
  - definition of $E \Rightarrow$ store into h
  - use of $E \Rightarrow$ load from h
- PRE on SSA form of RCVs (h) to remove redundancies
- Final program will be in SSA form

SSAPRE: Preparations

- Split all the *critical edges* in the flow graph

SSAPRE: Preparations

- Split all the *critical edges* in the flow graph
  - Edge from a node with more than one successor to a node with more than one predecessor

SSAPRE: Preparations

- Split all the *critical edges* in the flow graph
  - Edge from a node with more than one successor to a node with more than one predecessor
  - WHY is this important?

SSAPRE: Preparations

- Split all the *critical edges* in the flow graph
  - Edge from a node with more than one successor to a node with more than one predecessor
  - WHY is this important?
- Single pass to identify identical expressions

SSAPRE: Preparations

- Split all the *critical edges* in the flow graph
  - Edge from a node with more than one successor to a node with more than one predecessor
  - WHY is this important?
- Single pass to identify identical expressions
  - Ignoring the version number of the operands

SSAPRE: Preparations

- Split all the *critical edges* in the flow graph
  - Edge from a node with more than one successor to a node with more than one predecessor
  - WHY is this important?
- Single pass to identify identical expressions
  - Ignoring the version number of the operands
  - In the earlier example, $a_3 + b_1$ and $a_1 + b_1$ could be identical

SSAPRE Steps

- Six step algorithm

SSAPRE Steps

- Six step algorithm
  1. Φ-insertion

SSAPRE Steps

- Six step algorithm
  1. Φ-insertion
  2. Renaming

SSAPRE Steps

- Six step algorithm
  1. Φ-insertion
  2. Renaming
  3. Down-safety computation

SSAPRE Steps

- Six step algorithm
  1. Φ-insertion
  2. Renaming
  3. Down-safety computation
  4. WillBeAvail computation

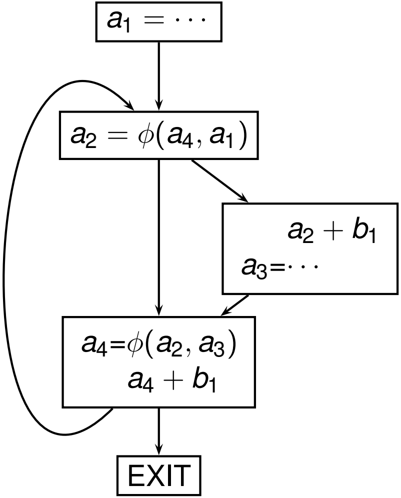
SSAPRE Steps

- Six step algorithm
  1. Φ-insertion
  2. Renaming
  3. Down-safety computation
  4. WillBeAvail computation
  5. Finalization

SSAPRE Steps

- Six step algorithm
  1. Φ-insertion
  2. Renaming
  3. Down-safety computation
  4. WillBeAvail computation
  5. Finalization
  6. Code Motion

Running Example



Φ-insertion

- Φ for an expression E is required where two potentially different values of an expression merge

Φ-insertion

- Φ for an expression E is required where two potentially different values of an expression merge
- At iterated dominance frontiers of occurrences of E

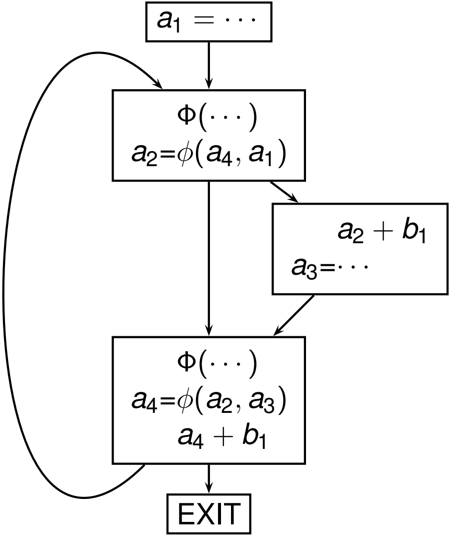
Φ-insertion

- Φ for an expression E is required where two potentially different values of an expression merge
- At iterated dominance frontiers of occurrences of E
- At each block having a ϕ for some argument of E

Φ-insertion

- Φ for an expression E is required where two potentially different values of an expression merge
- At iterated dominance frontiers of occurrences of E
- At each block having a ϕ for some argument of E
  - Potential change in the expression's value

Φ-insertion



Rename

- Similar to SSA variable renaming

Rename

- Similar to SSA variable renaming
- Stack of every expression is maintained

Rename

- Similar to SSA variable renaming
- Stack of every expression is maintained
- Three kinds of occurrences of E

Rename

- Similar to SSA variable renaming
- Stack of every expression is maintained
- Three kinds of occurrences of E
  - Real occurrences (present in original program)

Rename

- Similar to SSA variable renaming
- Stack of every expression is maintained
- Three kinds of occurrences of E
  - Real occurrences (present in original program)
  - Results of Φ operators inserted

Rename

- Similar to SSA variable renaming
- Stack of every expression is maintained
- Three kinds of occurrences of E
  - Real occurrences (present in original program)
  - Results of Φ operators inserted
  - Operands of inserted Φ

Rename

- Similar to SSA variable renaming
- Stack of every expression is maintained
- Three kinds of occurrences of E
  - Real occurrences (present in original program)
  - Results of Φ operators inserted
  - Operands of inserted Φ
- After renaming

Rename

- Similar to SSA variable renaming
- Stack of every expression is maintained
- Three kinds of occurrences of E
  - Real occurrences (present in original program)
  - Results of Φ operators inserted
  - Operands of inserted Φ
- After renaming
  - Identical SSA instances of h represent identical values of E

Rename

- Similar to SSA variable renaming
- Stack of every expression is maintained
- Three kinds of occurrences of E
  - Real occurrences (present in original program)
  - Results of Φ operators inserted
  - Operands of inserted Φ
- After renaming
  - Identical SSA instances of h represent identical values of E
  - A control flow path with two different instances of h has to cross either an assignment to an operand of E or a Φ of h

Rename Algorithm

- Runs with variable renaming

Rename Algorithm

- Runs with variable renaming
- When an E is encountered

Rename Algorithm

- Runs with variable renaming
- When an E is encountered
  - if E is result of Φ, assign a new version to h and push it on E stack

Rename Algorithm

- Runs with variable renaming
- When an E is encountered
  - if E is result of Φ, assign a new version to h and push it on E stack
  - if E is the real occurrence

Rename Algorithm

- Runs with variable renaming
- When an E is encountered
  - if E is result of Φ, assign a new version to h and push it on E stack
  - if E is the real occurrence
    - for each operand, compare the version of operand with the top of the rename stack for operand

Rename Algorithm

- Runs with variable renaming
- When an E is encountered
  - if E is result of Φ, assign a new version to h and push it on E stack
  - if E is the real occurrence
    - for each operand, compare the version of operand with the top of the rename stack for operand
    - If all match, h gets same version as the top of E stack

Rename Algorithm

- Runs with variable renaming
- When an E is encountered
  - if E is result of Φ, assign a new version to h and push it on E stack
  - if E is the real occurrence
    - for each operand, compare the version of operand with the top of the rename stack for operand
    - If all match, h gets same version as the top of E stack
    - If any mismatch, assign a new version to h and push it on E stack

Rename Algorithm

- Runs with variable renaming
- When an E is encountered
  - if E is result of Φ, assign a new version to h and push it on E stack
  - if E is the real occurrence
    - for each operand, compare the version of operand with the top of the rename stack for operand
    - If all match, h gets same version as the top of E stack
    - If any mismatch, assign a new version to h and push it on E stack
  - if E is operand of Φ, in the corresponding predecessor block

Rename Algorithm

- Runs with variable renaming
- When an E is encountered
  - if E is result of Φ, assign a new version to h and push it on E stack
  - if E is the real occurrence
    - for each operand, compare the version of operand with the top of the rename stack for operand
    - If all match, h gets same version as the top of E stack
    - If any mismatch, assign a new version to h and push it on E stack
  - if E is operand of Φ, in the corresponding predecessor block
    - for each operand of E, compare the version of operand with the top of the rename stack for operand

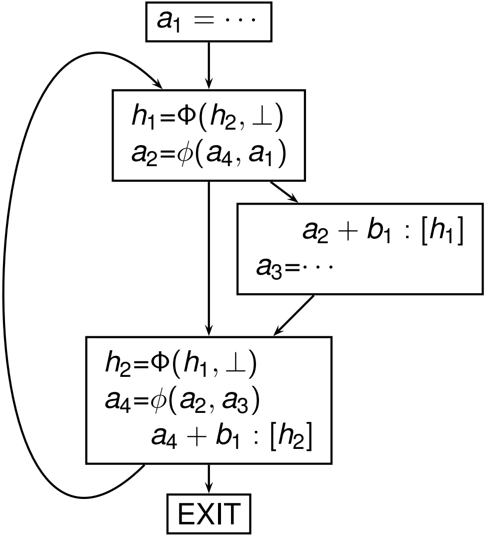
Rename Algorithm

- Runs with variable renaming
- When an E is encountered
  - if E is result of Φ, assign a new version to h and push it on E stack
  - if E is the real occurrence
    - for each operand, compare the version of operand with the top of the rename stack for operand
    - If all match, h gets same version as the top of E stack
    - If any mismatch, assign a new version to h and push it on E stack
  - if E is operand of Φ, in the corresponding predecessor block
    - for each operand of E, compare the version of operand with the top of the rename stack for operand
    - If all match, h gets same version as the top of E stack

Rename Algorithm

- Runs with variable renaming
- When an E is encountered
  - if E is result of Φ, assign a new version to h and push it on E stack
  - if E is the real occurrence
    - for each operand, compare the version of operand with the top of the rename stack for operand
    - If all match, h gets same version as the top of E stack
    - If any mismatch, assign a new version to h and push it on E stack
  - if E is operand of Φ, in the corresponding predecessor block
    - for each operand of E, compare the version of operand with the top of the rename stack for operand
    - If all match, h gets same version as the top of E stack
    - If any mismatch, replace E by $\perp$ in the operand push it on E stack (WHY?)

Rename


$a_1 = \cdots$
$h_1 = \Phi(h_2, \perp)$
$a_2 = \phi(a_4, a_1)$
$a_2 + b_1 : [h_1]$
$a_3 = \cdots$
$h_2 = \Phi(h_1, \perp)$
$a_4 = \phi(a_2, a_3)$
$a_4 + b_1 : [h_2]$
EXIT

Down-safety

- Down-safety is same as very-busy (anticipability) property of expressions

Down-safety

- Down-safety is same as very-busy (anticipability) property of expressions
  - Do not want to introduce new computation of E

Down-safety

- Down-safety is same as very-busy (anticipability) property of expressions
  - Do not want to introduce new computation of E
- We only need to compute down-safety for inserted Φ-operators

Down-safety

- Down-safety is same as very-busy (anticipability) property of expressions
  - Do not want to introduce new computation of E
- We only need to compute down-safety for inserted Φ-operators
- A Φ computation is **NOT** down-safe if

Down-safety

- Down-safety is same as very-busy (anticipability) property of expressions
  - Do not want to introduce new computation of E
- We only need to compute down-safety for inserted Φ-operators
- A Φ computation is **NOT** down-safe if
- there is a path to EXIT from Φ along which the result of Φ is

Down-safety

- Down-safety is same as very-busy (anticipability) property of expressions
  - Do not want to introduce new computation of E
- We only need to compute down-safety for inserted Φ-operators
- A Φ computation is **NOT** down-safe if
- there is a path to EXIT from Φ along which the result of Φ is
  - either not used

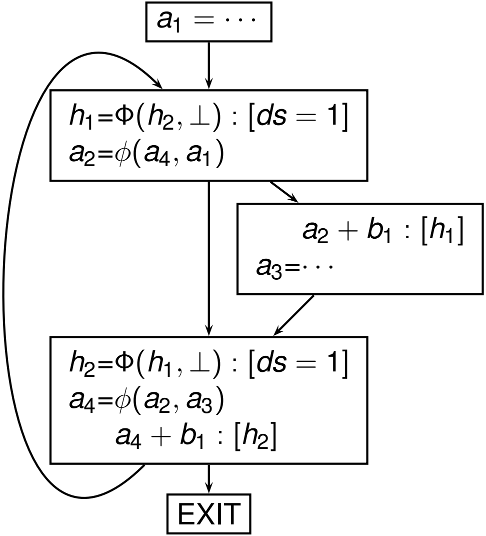
Down-safety

- Down-safety is same as very-busy (anticipability) property of expressions
  - Do not want to introduce new computation of E
- We only need to compute down-safety for inserted Φ-operators
- A Φ computation is **NOT** down-safe if
- there is a path to EXIT from Φ along which the result of Φ is
  - either not used
  - used only as an operand of another Φ that itself is **NOT** down-safe

Down-safety

- Down-safety is same as very-busy (anticipability) property of expressions
  - Do not want to introduce new computation of E
- We only need to compute down-safety for inserted Φ-operators
- A Φ computation is **NOT** down-safe if
- there is a path to EXIT from Φ along which the result of Φ is
  - either not used
  - used only as an operand of another Φ that itself is **NOT** down-safe
- *HasRealUse*: Real occurrence of an expression

Down-safety (ds = $\cdots$)



WillBeAvail

- The set of Φs where the expression must be available in any computationally optimal placement

WillBeAvail

- The set of Φs where the expression must be available in any computationally optimal placement
- Computation of *two forward* properties:

WillBeAvail

- The set of Φs where the expression must be available in any computationally optimal placement
- Computation of *two forward* properties:
  - *CanBeAvail*: Φs for which E is either available or anticipable or both

WillBeAvail

- The set of Φs where the expression must be available in any computationally optimal placement
- Computation of *two forward* properties:
  - *CanBeAvail*: Φs for which E is either available or anticipable or both
  - *Later*: Φs beyond which insertion can not be postponed without introducing new redundancy

$$\mathit{WillBeAvail} = \mathit{CanBeAvail} \land \neg \mathit{Later}$$

CanBeAvail

- Initialized to *true* for all Φs

CanBeAvail

- Initialized to *true* for all Φs
- Boundary Φs:

CanBeAvail

- Initialized to *true* for all Φs
- Boundary Φs:
  - Not Down-safe, and

CanBeAvail

- Initialized to *true* for all Φs
- Boundary Φs:
  - Not Down-safe, and
  - At least one argument is $\perp$

CanBeAvail

- Initialized to *true* for all Φs
- Boundary Φs:
  - Not Down-safe, and
  - At least one argument is $\perp$
- Set *false* for boundary Φs

CanBeAvail

- Initialized to *true* for all Φs
- Boundary Φs:
  - Not Down-safe, and
  - At least one argument is $\perp$
- Set *false* for boundary Φs
- Propagate *false* value along the chain of def-use to other Φs

CanBeAvail

- Initialized to *true* for all Φs
- Boundary Φs:
  - Not Down-safe, and
  - At least one argument is $\perp$
- Set *false* for boundary Φs
- Propagate *false* value along the chain of def-use to other Φs
  - exclude edges along which *HasRealUse* is *true*

Later

- Determines latest (final) insertion points

Later

- Determines latest (final) insertion points
- Initialize Later to *true* wherever CanBeAvail is *true*, otherwise false

Later

- Determines latest (final) insertion points
- Initialize Later to *true* wherever CanBeAvail is *true*, otherwise false
- Assign *false* for Φs with at least one operand with HasRealUse flag *true*

Later

- Determines latest (final) insertion points
- Initialize Later to *true* wherever CanBeAvail is *true*, otherwise false
- Assign *false* for Φs with at least one operand with HasRealUse flag *true*
- Propagate *false* value forward to other Φs

Later

- Determines latest (final) insertion points
- Initialize Later to *true* wherever CanBeAvail is *true*, otherwise false
- Assign *false* for Φs with at least one operand with HasRealUse flag *true*
- Propagate *false* value forward to other Φs
- Later $\Rightarrow$ Φs that are CanBeAvail, but do not reach any real occurrence of E

Insertion Points

- Insertions are done for Φ operands

Insertion Points

- Insertions are done for Φ operands
- Along the corresponding predecessor edges

Insertion Points

- Insertions are done for Φ operands
- Along the corresponding predecessor edges
- Insertion done along i^{th} predecessor of Φ if *Insert* is *true*, i.e.

Insertion Points

- Insertions are done for Φ operands
- Along the corresponding predecessor edges
- Insertion done along i^{th} predecessor of Φ if *Insert* is *true*, i.e.
  - *WillBeAvail*(Φ) == *true*; AND

Insertion Points

- Insertions are done for Φ operands
- Along the corresponding predecessor edges
- Insertion done along i^{th} predecessor of Φ if *Insert* is *true*, i.e.
  - *WillBeAvail*(Φ) == *true*; AND
  - Arg_i is $\perp$; OR

Insertion Points

- Insertions are done for Φ operands
- Along the corresponding predecessor edges
- Insertion done along i^{th} predecessor of Φ if *Insert* is *true*, i.e.
  - *WillBeAvail*(Φ) == *true*; AND
  - Arg_i is $\perp$; OR
    - (HasRealUse(Arg_i) == *false*), AND

Insertion Points

- Insertions are done for Φ operands
- Along the corresponding predecessor edges
- Insertion done along i^{th} predecessor of Φ if *Insert* is *true*, i.e.
  - $WillBeAvail(\Phi)$ == *true*; AND
  - Arg_i is $\perp$; OR
    - (HasRealUse(Arg_i) == *false*), AND
    - Arg_i is defined by Φ' with $WillBeAvail(\Phi')$ == *false*

Finalize

- Transforms the program with RCVs into a valid SSA form

Finalize

- Transforms the program with RCVs into a valid SSA form
- For every real occurrence of E, decide whether it is a *def* or a *use*

Finalize

- Transforms the program with RCVs into a valid SSA form
- For every real occurrence of E, decide whether it is a *def* or a *use*
- For every Φ with *WillBeAvail* being *true*, insert E along incoming edges with *Insert* being *true*

Finalize

- Transforms the program with RCVs into a valid SSA form
- For every real occurrence of E, decide whether it is a *def* or a *use*
- For every Φ with *WillBeAvail* being *true*, insert E along incoming edges with *Insert* being *true*
- For each Φ for E

Finalize

- Transforms the program with RCVs into a valid SSA form
- For every real occurrence of E, decide whether it is a *def* or a *use*
- For every Φ with *WillBeAvail* being *true*, insert E along incoming edges with *Insert* being *true*
- For each Φ for E
  - If *WillBeAvail* is *true*, it is replaced by SSA temporary with appropriate version (h_x)

Finalize

- Transforms the program with RCVs into a valid SSA form
- For every real occurrence of E, decide whether it is a *def* or a *use*
- For every Φ with *WillBeAvail* being *true*, insert E along incoming edges with *Insert* being *true*
- For each Φ for E
  - If *WillBeAvail* is *true*, it is replaced by SSA temporary with appropriate version (h_x)
  - If *WillBeAvail* is *false*, it is not part of SSA form, and is removed

Finalize: AvailDef

- AvailDef: Table to mark def of expression occurrences

Finalize: AvailDef

- AvailDef: Table to mark def of expression occurrences
- Computed for each class (say h_x) of E

Finalize: AvailDef

- AvailDef: Table to mark def of expression occurrences
- Computed for each class (say h_x) of E
- Preorder traversal of dominator tree

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:
    - If WillBeAvail is *false*, ignore.

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:
    - If WillBeAvail is *false*, ignore.
    - Otherwise AvailDef[x] = this Φ (we must be visiting x for first time)

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:
    - If WillBeAvail is *false*, ignore.
    - Otherwise AvailDef[x] = this Φ (we must be visiting x for first time)

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:
    - If WillBeAvail is *false*, ignore.
    - Otherwise AvailDef[x] = this Φ (we must be visiting x for first time) – WHY?
  - Real occurrence:

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:
    - If WillBeAvail is *false*, ignore.
    - Otherwise AvailDef[x] = this Φ (we must be visiting x for first time) – WHY?
  - Real occurrence:
    - If AvailDef[x] is $\perp$, mark this occurrence as def

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:
    - If WillBeAvail is *false*, ignore.
    - Otherwise AvailDef[x] = this Φ (we must be visiting x for first time) – WHY?
  - Real occurrence:
    - If AvailDef[x] is $\perp$, mark this occurrence as def
    - Else, if AvailDef[x] does not dominate this occurrence, mark this occurrence as def

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:
    - If WillBeAvail is *false*, ignore.
    - Otherwise AvailDef[x] = this Φ (we must be visiting x for first time) – WHY?
  - Real occurrence:
    - If AvailDef[x] is $\perp$, mark this occurrence as def
    - Else, if AvailDef[x] does not dominate this occurrence, mark this occurrence as def
    - Else, mark this occurrence as use of AvailDef[x]

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:
    - If WillBeAvail is *false*, ignore.
    - Otherwise AvailDef[x] = this Φ (we must be visiting x for first time) – WHY?
  - Real occurrence:
    - If AvailDef[x] is $\perp$, mark this occurrence as def
    - Else, if AvailDef[x] does not dominate this occurrence, mark this occurrence as def
    - Else, mark this occurrence as use of AvailDef[x]
  - Φ operand (processed in predecessor block P)

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:
    - If WillBeAvail is *false*, ignore.
    - Otherwise AvailDef[x] = this Φ (we must be visiting x for first time) – WHY?
  - Real occurrence:
    - If AvailDef[x] is $\perp$, mark this occurrence as def
    - Else, if AvailDef[x] does not dominate this occurrence, mark this occurrence as def
    - Else, mark this occurrence as use of AvailDef[x]
  - Φ operand (processed in predecessor block P)
    - If WillBeAvail of Φ is false, ignore.

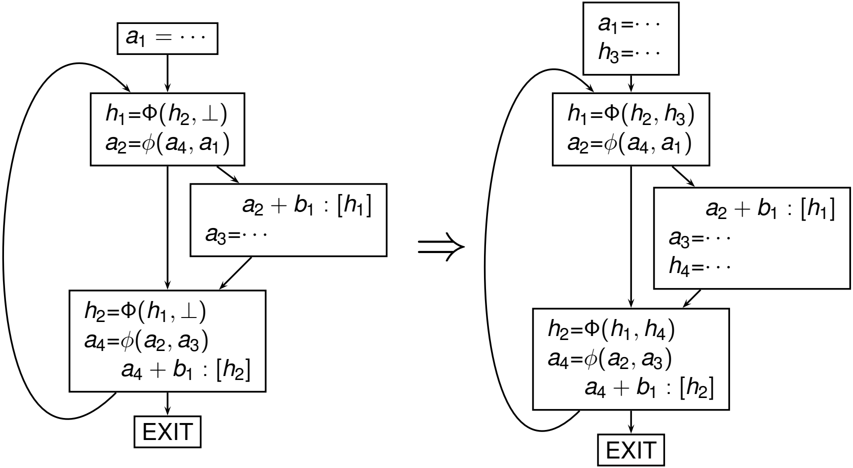
AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:
    - If WillBeAvail is *false*, ignore.
    - Otherwise AvailDef[x] = this Φ (we must be visiting x for first time) – WHY?
  - Real occurrence:
    - If AvailDef[x] is $\perp$, mark this occurrence as def
    - Else, if AvailDef[x] does not dominate this occurrence, mark this occurrence as def
    - Else, mark this occurrence as use of AvailDef[x]
  - Φ operand (processed in predecessor block P)
    - If WillBeAvail of Φ is false, ignore.
    - Else, if *Insert* is true for the operand, insert computation of E in block P, set it as a def, mark this occurrence as use of inserted.

AvailDef Computation

- Initialize: AvailDef[x] = $\perp$ $\forall x$ (all classes of all expressions)
- During course of traversal, process occurrence x of E
  - Φ occurrence:
    - If WillBeAvail is *false*, ignore.
    - Otherwise AvailDef[x] = this Φ (we must be visiting x for first time) – WHY?
  - Real occurrence:
    - If AvailDef[x] is $\perp$, mark this occurrence as def
    - Else, if AvailDef[x] does not dominate this occurrence, mark this occurrence as def
    - Else, mark this occurrence as use of AvailDef[x]
  - Φ operand (processed in predecessor block P)
    - If WillBeAvail of Φ is false, ignore.
    - Else, if *Insert* is true for the operand, insert computation of E in block P, set it as a def, mark this occurrence as use of inserted.
    - Else (*Insert* is false), mark this occurrence as use of AvailDef[x]

Finalize



$a_1 = \cdots$
$h_1 = \Phi(h_2, \perp)$
$a_2 = \phi(a_4, a_1)$
$a_2 + b_1 : [h_1]$
$a_3 = \cdots$
$h_2 = \Phi(h_1, \perp)$
$a_4 = \phi(a_2, a_3)$
$a_4 + b_1 : [h_2]$
EXIT
$\Longrightarrow$
$a_1 = \cdots$
$h_3 = \cdots$
$h_1 = \Phi(h_2, h_3)$
$a_2 = \phi(a_4, a_1)$
$a_2 + b_1 : [h_1]$
$a_3 = \cdots$
$h_4 = \cdots$
$h_2 = \Phi(h_1, h_4)$
$a_4 = \phi(a_2, a_3)$
$a_4 + b_1 : [h_2]$
EXIT

Code Motion

- For real *def* occurrence of E, compute E in a new version of temporary t

Code Motion

- For real *def* occurrence of E, compute E in a new version of temporary t
- For real *use* occurrence of E, replace E by current version of t

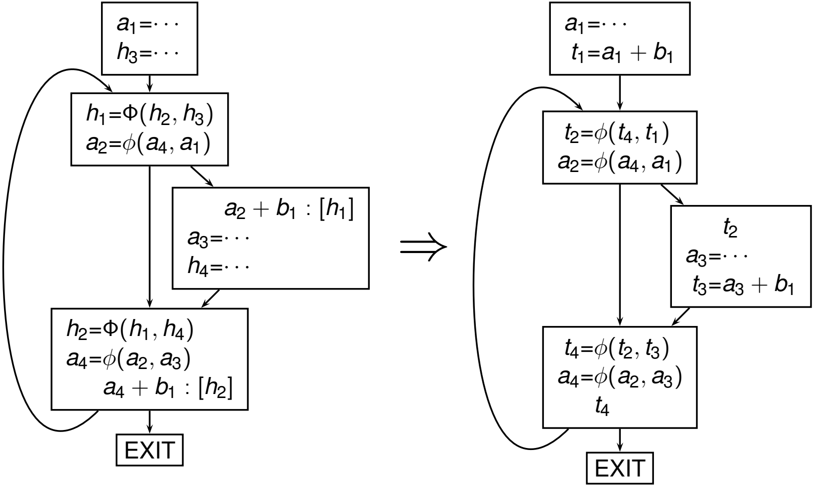
Code Motion

- For real *def* occurrence of E, compute E in a new version of temporary t
- For real *use* occurrence of E, replace E by current version of t
- For inserted occurrence of E, compute E in a new version of temporary t

Code Motion

- For real *def* occurrence of E, compute E in a new version of temporary t
- For real *use* occurrence of E, replace E by current version of t
- For inserted occurrence of E, compute E in a new version of temporary t
- For a Φ occurrence, insert appropriate ϕ for t

Code Motion



$a_1 = \cdots$
$h_3 = \cdots$
$h_1 = \Phi(h_2, h_3)$
$a_2 = \phi(a_4, a_1)$
$a_2 + b_1 : [h_1]$
$a_3 = \cdots$
$h_4 = \cdots$
$h_2 = \Phi(h_1, h_4)$
$a_4 = \phi(a_2, a_3)$
$a_4 + b_1 : [h_2]$
EXIT
$\Rightarrow$
$a_1 = \cdots$
$t_1 = a_1 + b_1$
$t_2 = \phi(t_4, t_1)$
$a_2 = \phi(a_4, a_1)$
t_2
$a_3 = \cdots$
$t_3 = a_3 + b_1$
$t_4 = \phi(t_2, t_3)$
$a_4 = \phi(a_2, a_3)$
t_4
EXIT