Compiler Construction 2016/2017
SSA—Static Single Assignment Form

Peter Thiemann

January 19, 2017

Outline

Redundancy Elimination

Goal of optimization: redundancy elimination

- Value numbering
- Constant propagation
- Common subexpression elimination (CSE)

def-use chain

- important data structure for RE
- links definitions and uses to flow-graph nodes

Wanted: IR that makes RE more efficient

- simplifies data structures like DU chains
- speeds up analysis

Static Single-Assignment (SSA) Form

An answer: SSA

- Intermediate representation
- Statically, each variable has exactly one definition
- simplifies and speeds up analysis
- simplifies DU-chains (every U has exactly one D)

SSA Example

$$\begin{array}{lcl} a & \leftarrow & x + y \\ b & \leftarrow & a - 1 \\ a & \leftarrow & y + b \\ b & \leftarrow & x \cdot 4 \\ a & \leftarrow & a + b \end{array}$$

straight-line program

SSA Example

$$\begin{array}{lcl} a & \leftarrow & x + y \\ b & \leftarrow & a - 1 \\ a & \leftarrow & y + b \\ b & \leftarrow & x \cdot 4 \\ a & \leftarrow & a + b \end{array}$$

straight-line program

$$\begin{array}{lcl} a_1 & \leftarrow & x + y \\ b_1 & \leftarrow & a_1 - 1 \\ a_2 & \leftarrow & y + b_1 \\ b_2 & \leftarrow & x \cdot 4 \\ a_3 & \leftarrow & a_2 + b_2 \end{array}$$

program in SSA form

Why Bother?

Example: Value Numbering

Value Numbering

determines that $j == l$

```
i = read();
j = i + 1;
k = i;
l = k + 1;
```

Value Numbering

determines that $j == l$

```
i = read();
j = i + 1;
k = i;
l = k + 1;
```

Basic idea

- tag each computation
- same tag $\Rightarrow$ same value at run time

Congruence

$x \oplus y \sim a \otimes b$ iff

- $\oplus = \otimes$ and
- $x \sim a$ and $y \sim b$
- (commutativity if applicable)

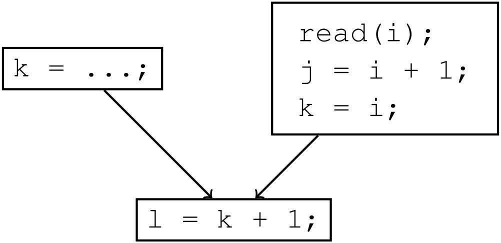
Value Numbering

Implementation

- Hash function H that respects congruence (i.e., $x \sim y$ implies $H(x) = H(y)$)
- Symbolic execution
- $V(t)$: tag of t's value
- Consider $t_1 = t_2 + 1$
- $h = H(V(t_2) + 1)$
- if temporary t_h holding tag h exists, then replace statement by $t_1 = t_h$
- otherwise, remember $V(t_1) = h$

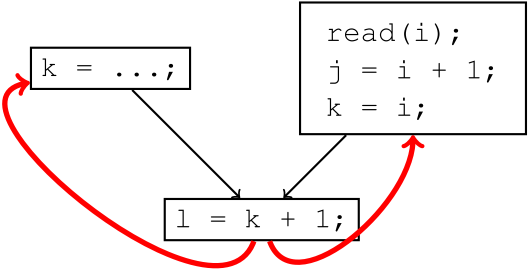
Global Value Numbering

- Local value numbering straightforward (inside basic block)
- Value numbering within a procedure requires more care



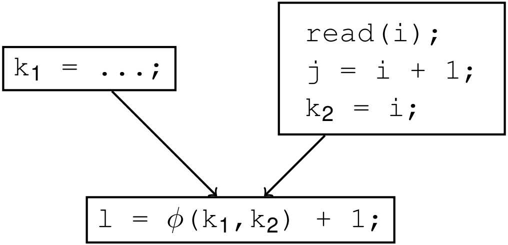
Def-use Information

- Problem: keeping track of relation between definitions and uses of a variable
- Dataflow analysis



SSA = Embedding def-use in the IR

- SSA represents def-use information explicitly

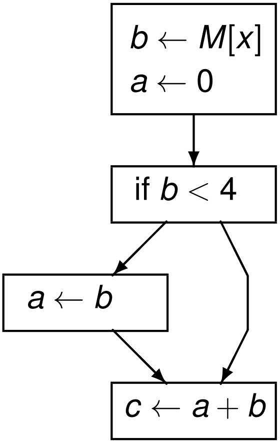


Usefulness of SSA Form

- Dataflow analysis becomes simpler
- Optimized space usage for def-use chains
  N uses and M definitions of var: $N \cdot M$ pointers required
- Uses and defs are related to dominator tree
- Unrelated uses of the same variable are made different

ϕ-Functions

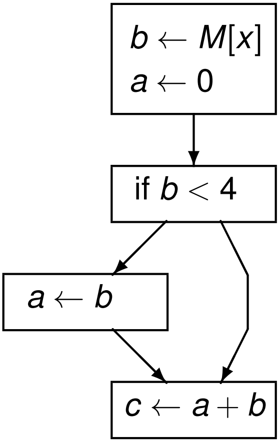
CFG with a control-flow join



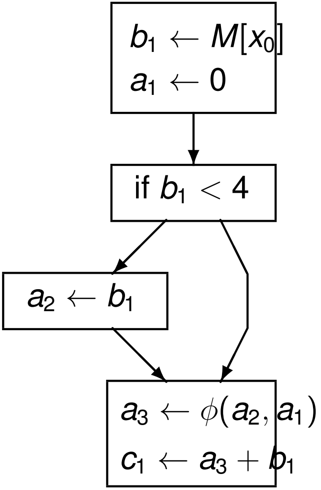
Original

ϕ-Functions

CFG with a control-flow join … transformed to SSA form



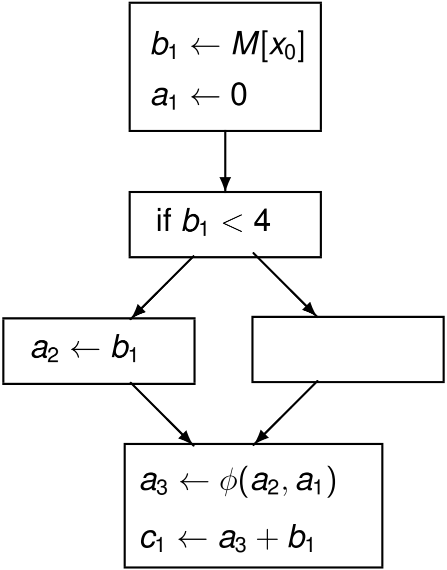
Original



SSA Form

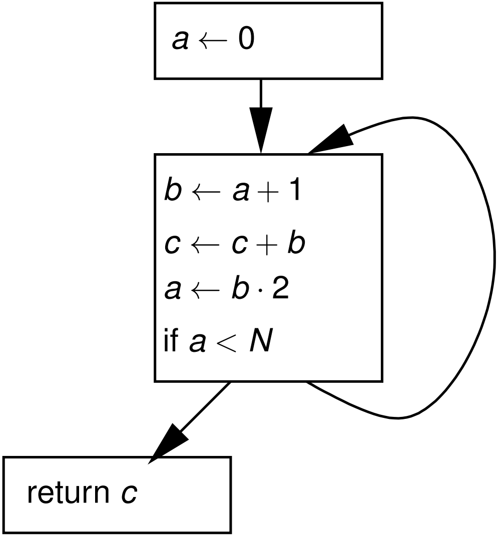
ϕ-Functions

. . . to edge-split SSA form



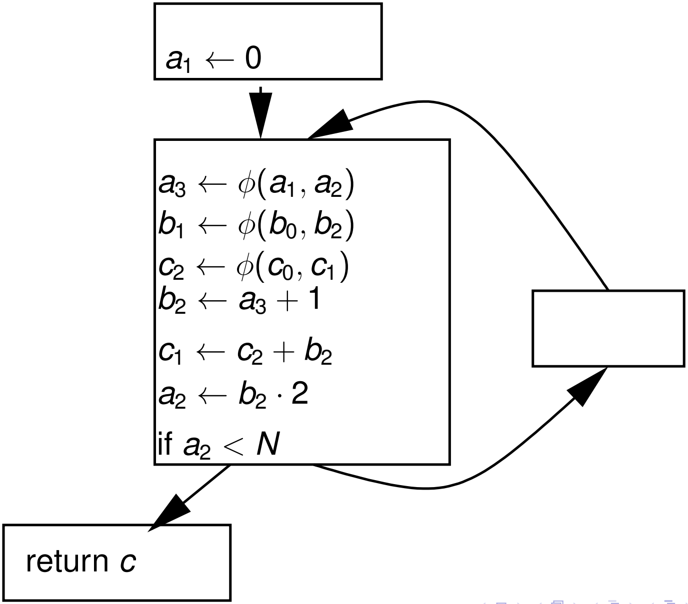
ϕ-Functions

Program with a loop



ϕ-Functions

… transformed to edge-split SSA form



Features of SSA Form

- SSA renames variables
- SSA introduces ϕ-functions
  - not “real” functions, just notation
  - implemented by move instruction on incoming edges
  - can often be ignored by optimization
- SSA with edge-splitting
  - a critical edge connects a node with multiple successors to a node with multiple predecessors
  - poses problems for SSA destruction (see below)
  - one remedy: split critical edges

Outline

SSA Construction

- Transform program $\rightarrow$ CFG
- Insert ϕ-functions
  naive: add a ϕ-function for each variable at each join point
- Rename variables
- Split critical edges

Add a ϕ-function for variable a at node z of the flow graph iff

1. There is a block x containing a definition of a.
2. There is a block $y \neq x$ containing a definition of a.
3. There is a non-empty path π_{xz} from x to z.
4. There is a non-empty path π_{yz} from y to z.
5. Paths π_{xz} and π_{yz} have only z in common.
6. Node z does not appear in <u>both</u> π_{xz} and π_{yz} prior to the end, but it may appear before in one of them.

Iterated Path-Convergence Criterion

Remarks

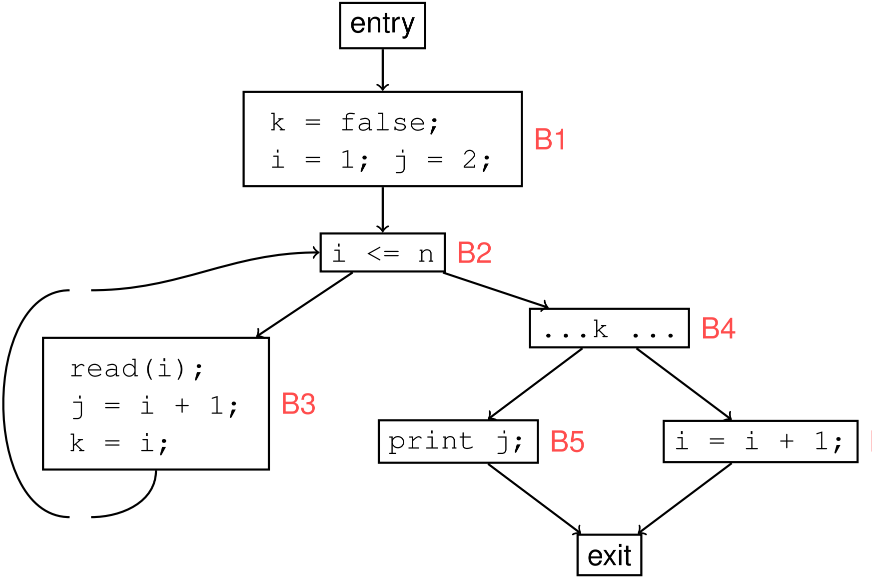
- Start node contains an implicit definition of each variable
- A ϕ-function counts as a definition
- Compute by fixpoint iteration

Algorithm

while there are nodes x, y, z satisfying conditions 1–5
and z does not contain a ϕ-function for a
do insert $a \leftarrow \phi(\underbrace{a, \ldots, a}_{p})$

where $p = \#$ predecessors of z

SSA Construction



Dominance Property of SSA Form

In SSA, each definition dominates all its uses

1. If x is the ith argument of a ϕ-function in block n, then the definition of x dominates the ith predecessor of node n.
2. If x is used in a non-ϕ statement in block n, then the definition of x dominates node n.

The Dominance Frontier

Towards a more efficient algorithm for placing ϕ-functions

Conventions

- Traversing the CFG:
  successor and predecessor for graph edges.
- Traversing the DT:
  parent and child for tree edges, ancestor for paths.

The Dominance Frontier Criterion

Definition

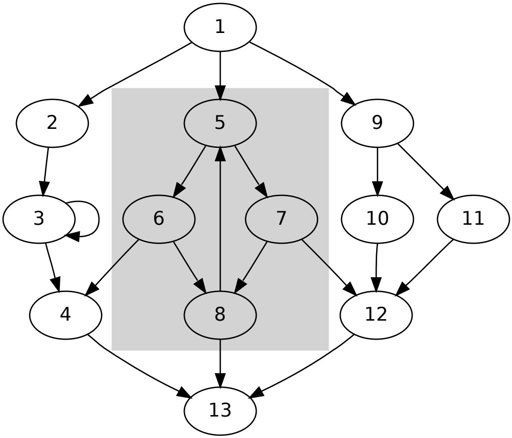
- x strictly dominates y if x dominates y and $x \neq y$.
- The dominance frontier of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w. (So w could be x.)

Dominance Frontier Criterion

If node x contains a definition of some variable a, then any node z in the dominance frontier of x needs a ϕ-function for a.

Dominance Frontier

Consider node 5



Iterated Dominance Frontier

- The dominance frontier criterion must be iterated: each inserted ϕ-function counts as a new definition

Theorem

The iterated dominance frontier criterion and the iterated path-convergence criterion specify the same set of nodes for placing ϕ-functions.

Computing the Dominance Frontier

$DF[n]$, the dominance frontier of node n, can be computed in one bottom-up pass through the dominator tree.

- $DF_{local}[n]$ successors of n not strictly dominated by n. $DF_{local}[n] = \{y \in succ[n] \mid idom(y) \neq n\}$
- $DF_{up}[n, c]$ nodes in the dominance frontier of n's child c (in the DT) that are not strictly dominated by n. $DF_{up}[n, c] = \{y \in DF[c] \mid idom(y) \neq n\}$

Computing the Dominance Frontier

$DF[n]$, the dominance frontier of node n, can be computed in one bottom-up pass through the dominator tree.

- $DF_{local}[n]$ successors of n not strictly dominated by n.
  $DF_{local}[n] = \{y \in succ[n] \mid idom(y) \neq n\}$
- $DF_{up}[n, c]$ nodes in the dominance frontier of n's child c (in the DT) that are not strictly dominated by n.
  $DF_{up}[n, c] = \{y \in DF[c] \mid idom(y) \neq n\}$

Theorem

$$DF[n] = DF_{local}[n] \cup \bigcup_{c \in children[n]} DF_{up}[n, c]$$

Computing the Dominance Frontier

Algorithm

```
computeDF(n) =
    S ← ∅
    {* compute DF_local(n) *}
    for each node y ∈ succ[n] do
        if idom(y) ≠ n then
            S ← S ∪ {y}
    {* compute DF_up(n, c) *}
    for each child c with idom(c) = n do
        computeDF(c)
        for each w ∈ DF[c] do
            if n = w or n does not dominate w then
                S ← S ∪ {w}
    DF[n] ← S
```

Inserting ϕ-Functions

```
Place-ϕ-Functions (A_def) =
  for each variable a do
    W ← {n | a ∈ A_def[n]}          {* A_def[n] = vars defined at n *}
    while W ≠ ∅ do
      remove some node n from W
      for each y ∈ DF[n] do
        if a ∉ A_ϕ[y] then
          insert statement a ← ϕ(a, ..., a) at top of block y,
          where the number of arguments is |pred[y]|
          A_ϕ[y] ← A_ϕ[y] ∪ {a}
          if a ∉ A_def[y] then
            W ← W ∪ {y}
  {* A_ϕ[n] = vars that have a ϕ at n *}
```

Renaming Variables

- Top-down traversal of the dominator tree
- Rename the different definitions (including ϕ-definitions) of variable *a* to $a_1, a_2, \ldots$
- Rename each use of *a* in a statement to the closest definition of an *a* that is above *a* in the dominator tree
- To modify the arguments of ϕ-functions, look ahead in the successor nodes.

Edge Splitting

Algorithm: Edge Splitting

If there is a critical edge $a \to b$ in the CFG where $|succ[a]| > 1$ and $|pred[b]| > 1$, then create new, empty node z and replace edge $a \to b$ by $a \to z$ and $z \to b$.

- Some analyses and transformations (destruction!) are simpler if no control flow edge leads from a node with multiple successors to on with multiple predecessors.
- Edge splitting achieves the unique successor or predecessor property.

Efficient Computation of the Dominator Tree

- There are efficient, almost linear-time algorithms for computing the dominator tree [Lengauer, Tarjan 1979] [Harel 1985] [Buchsbaum 1998] [Alstrup 1999].
- But there are easy variations of the naive algorithm that perform good in practice. [Cooper, Harvey, Kennedy 2006]

Outline

Statement assignment, ϕ-function, fetch, store, branch. Fields: containing block, previous/next statement in block, variables defined, variables used

Variable definition site, list of use sites

Block list of statements, ordered list of predecessors, one or more successors

SSA: Useless Code Elimination

SSA Liveness

A variable definition is live iff its list of uses is non-empty.

Algorithm

$W \leftarrow$ set of all variables in SSA program
while $W \neq \emptyset$ **do**
  remove some variable v from W
  if v's list of uses is empty **then**
    let S be v's defining statement
    if S has no side effects other than the assignment to v
    then
      delete S from program
      for each variable x_i used by S **do**
        delete S from list of uses of x_i {in constant time}
        $W \leftarrow W \cup \{x_i\}$

SSA: Simple Constant Propagation

- If v is defined by $v \leftarrow c$ (a constant) then each use of v can be replaced by c.
- The ϕ-function $v \leftarrow \phi(c, \ldots, c)$ can be replaced by $v \leftarrow c$

Algorithm

$W \leftarrow$ set of all statements in SSA program
while $W \neq \emptyset$ **do**
  remove some statement S from W
  if S is $v \leftarrow \phi(c, \ldots, c)$ for constant c **then**
    replace S by $v \leftarrow c$
  if S is $v \leftarrow c$ for constant c **then**
    delete S
    for each statement T that uses v **do**
      substitute c for v in T
      $W \leftarrow W \cup \{T\}$

SSA: Further Linear-Time Transformations

Copy propagation

If some S is $x \leftarrow \phi(y)$ or $x \leftarrow y$,
then remove S and substitute y for every use of x.

Constant folding

If S is $v \leftarrow c \oplus d$ where c and d are constants, then

- compute $e = c \oplus d$ at compile time and
- replace S by $v \leftarrow e$.

SSA: Further Linear-Time Transformations

Constant conditions

Let **if** $a \sharp b$ **goto** L_1 **else** L_2 be at the end of block L with a and b constants and $\sharp$ a comparison operator.

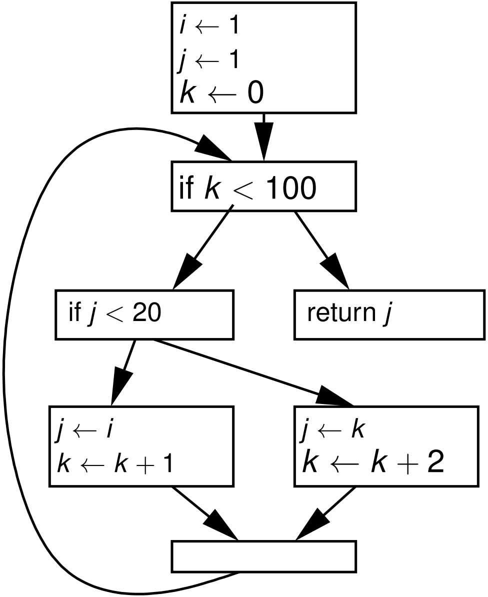
- Replace the conditional by **goto** L_1 or **goto** L_2 depending on the compile-time value of $a \sharp b$
- Delete the control flow edge $L \rightarrow L_2$ (L_1 respectively)
- Adjust the ϕ functions in L_2 (L_1) by removing the argument associated to predecessor L.

Unreachable code

Deleting an edge from a predecessor may cause block L_2 to become unreachable.

- Delete all statements of L_2, adjusting the use lists of the variables used in these statements.
- Delete block L_2 and the edges to its successors.

Conditional Constant Propagation

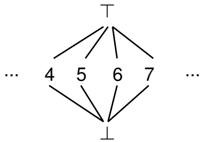


Conditional Constant Propagation

- does not assume that a block can be executed until there is evidence for it
- does not assume a variable is non-constant until there is evidence for it

Constant Propagation Lattice

- $V[v] = \perp$ no assignment to v has been seen (initially)
- $V[v] = c$ an assignment $v \leftarrow c$ (constant) has been seen
- $V[v] = \top$ conflicting assignments have been seen



Block Reachability

- $E[B] =$ *false* no control transfer to B has been seen (initially)
- $E[B] =$ *true* a control transfer to B has been seen

Least upper bound operation

$$
\begin{aligned}
\bot \sqcup \alpha &= \alpha \sqcup \bot = \alpha \\
\top \sqcup \alpha &= \alpha \sqcup \top = \top \\
a \sqcup b &= \begin{cases} a & a = b \\ \top & a \neq b \end{cases}
\end{aligned}
$$

Primitive operation

$$
\begin{aligned}
\bot \hat{\oplus} \alpha &= \alpha \hat{\oplus} \bot = \bot \\
\top \hat{\oplus} \alpha &= \alpha \hat{\oplus} \top = \top \\
a \hat{\oplus} b &= (a \oplus b)
\end{aligned}
$$

Conditional Constant Propagation

Algorithm Initialization

1. Initialize $V[v] = \perp$ for all variables v and $E[B] = \mathit{false}$ for all blocks B
2. If v has no definition, then set $V[v] \leftarrow \top$ (must be input or uninitialized)
3. The entry block is reachable: $E[B_0] \leftarrow \mathit{true}$

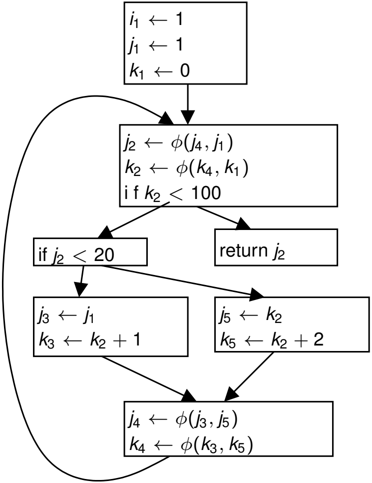
Conditional Constant Propagation

Algorithm

1. For each B with $E[B]$ and B has only one successor C, then set $E[C] = \mathit{true}$.
2. For each reachable assignment $v \leftarrow x \oplus y$ set $V[v] \leftarrow V[x] \hat{\oplus} V[y]$.
3. For each reachable assignment $v \leftarrow \phi(x_1, \ldots, x_p)$ set $V[v] \leftarrow \bigsqcup \{ V[x_j] \mid j\text{th predecessor is reachable}\}$
4. For each reachable assignment $v \leftarrow M[\ldots]$ or $v \leftarrow \mathit{CALL}(\ldots)$ set $V[v] \leftarrow \top$.
5. For each reachable branch **if** $x \sharp y$ **goto** L_1 **else** L_2 consider $\beta = V[x] \hat{\sharp} V[y]$.
   - If $\beta = \mathit{true}$, then set $E[L_1] \leftarrow \mathit{true}$.
   - If $\beta = \mathit{false}$, then set $E[L_2] \leftarrow \mathit{true}$.
   - If $\beta = \top$, then set $E[L_1], E[L_2] \leftarrow \mathit{true}$.

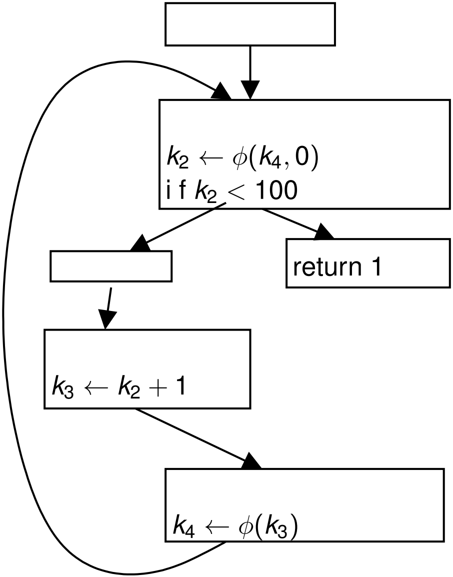
Conditional Constant Propagation

Example



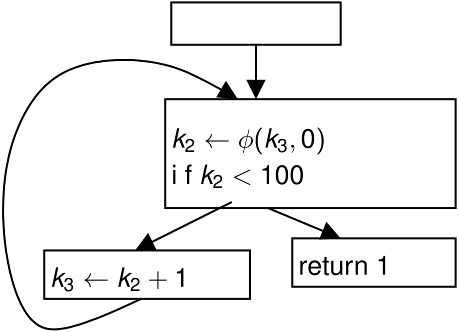
Conditional Constant Propagation

Example after propagation



Conditional Constant Propagation

Example after cleanup



Outline

Dependencies Between Statements

B depends on A

Read-after-write A defines variable v and B uses v

Write-after-write A defines variable v and B defines v

Write-after-read A uses v and then B defines v

Control A controls whether B executes

In SSA form

- all dependencies are Read-after-write or Control
- Read-after-write is evident from SSA graph
- Control needs to be analyzed

Memory Dependence

- Memory does not enjoy the single assignment property
- Consider

$$\begin{array}{lll} 1 & M[i] & \leftarrow & 4 \\ 2 & x & \leftarrow & M[j] \\ 3 & M[k] & \leftarrow & j \end{array}$$

Depending on the values of i, j, and k

  - 2 may have a read-after-write dependency with 1 (if $i = j$)
  - 3 may have a write-after-write dependency with 1 (if $i = k$)
  - 3 may have a write-after-read dependency with 2 (if $j = k$)
  so 2 and 3 may not be exchanged

Approach

- No tracking of memory dependencies
- Store instructions always live
- No reordering of memory instructions

Control Dependence Graph

Control Dependence

- Node y is control dependent on x if
  1. x has successors u and v
  2. there exists a path from u to *exit* that avoids y
  3. every path from v to *exit* goes through y
- The control-dependence graph (CDG) has an edge from x to y if y is control dependent on x.
- y postdominates v if y is on every path from v to *exit*, i.e., if y dominates v in the reverse CFG.

Construction of the CDG

Let G be a CFG

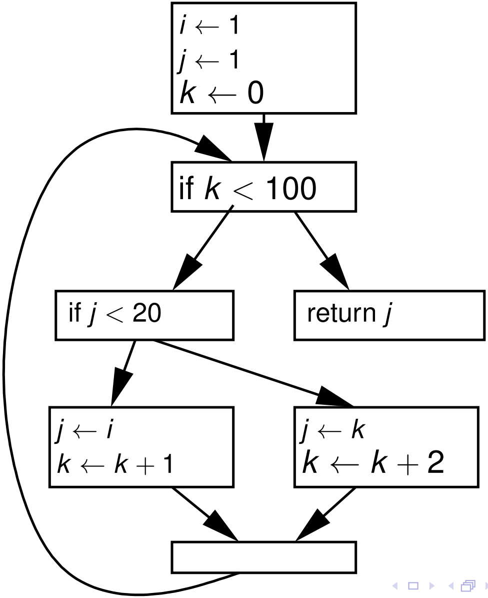
1. Add new entry node r to G with edge $r \rightarrow s$ (the original start node) and an edge $r \rightarrow exit$.
2. Let G' be the reverse control-flow graph with the same nodes as G, all edges reversed, and with start node *exit*.
3. Construct the dominator tree of G' with root *exit*.
4. Calculate the dominance frontiers $DF_{G'}$ of G'.
5. The CDG has edge $x \rightarrow y$ if $x \in DF_{G'}[y]$.

Use of the CDG

A must be executed before B
if
there is a path $A \to B$ using SSA use-def edges and CDG edges.
I.e., there are data- and control dependencies that require A to be executed before B.

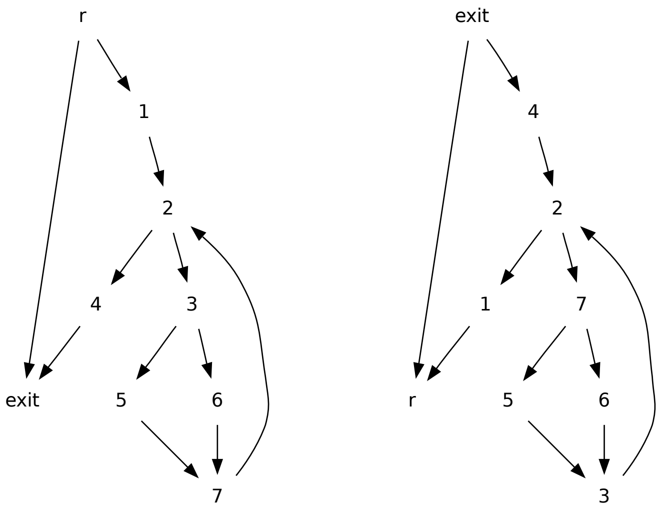
Construction of the CDG

Example



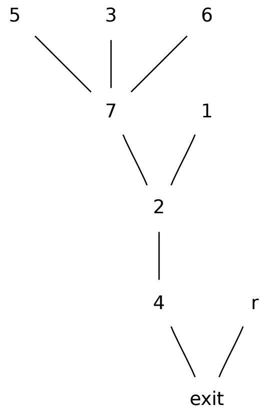
Construction of the CDG

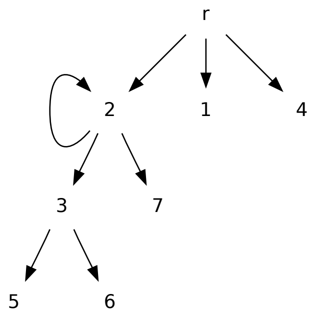
CFG and reverse CFG



Construction of the CDG

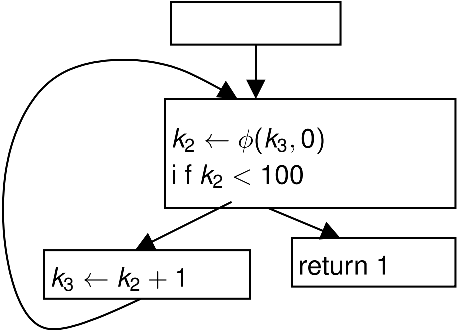
Postdominators and CDG





Aggressive Dead-Code Elimination

- Application of the CDG
- Consider



- k_2 is live because it is used in defining k_3
- k_3 is live because it is used in defining k_2

Aggressive Dead-Code Elimination

Algorithm

Exhaustively mark a live any statement that

1. Performs I/O, stores to memory, returns from the function, calls another function that may have side effects.
2. Defines some variable v that is used by another live statement.
3. Is a conditional branch, on which some other live statement is control dependent.

Then delete all unmarked statements.

- Result on example: return 1; loop is deleted

Outline

SSA Destruction

- ϕ-functions are not executable and must be replaced with real instructions to generate code
- $y \leftarrow \phi(x_1, x_2, x_3)$ is interpreted as
  - move x_1 to y if arriving from predecessor $\sharp 1$
  - move x_2 to y if arriving from predecessor $\sharp 2$
  - move x_3 to y if arriving from predecessor $\sharp 3$
- Insert these instructions at the end of the respective predecessor (possible due to edge-split assumption)
- Next step: register allocation

Liveness Analysis for SSA

```
LivenessAnalysis() =
  for each variable v do
    M ← ∅
    for  each statement s using v do
      if  s is a ϕ-function with ith argument v then
        let p be the ith predecessor of s's block
        LiveOutAtBlock(p, v)
      else
        LiveInAtStatement(s, v)

LiveOutAtBlock(n, v) =
  {v is live-out at n}
  if  n ∉ M then
    M ← M ∪ {n}
    let s be the last statement in n
    LiveOutAtStatement(s, v)
```

Liveness Analysis for SSA

LiveInAtStatement(s, v) $=$

{v is live-in at s}

if s is first statement of block n **then**

{v is live-in at n}

for each $p \in pred[n]$ **do**

LiveOutAtBlock(p, v)

else

let s' be the statement preceding s

LiveOutAtStatement(s', v)

LiveOutAtStatement(s, v) $=$

{v is live-out at s}

let W be the set of variables defined in s

for each variable $w \in W \setminus \{v\}$ **do**

add (v, w) to interference graph {needed if v defined?}

if $v \notin W$ **then**

LiveInAtStatement(s, v)