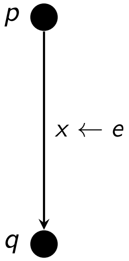
SSA Introduction

Sebastian Hack
hack@cs.uni-saarland.de

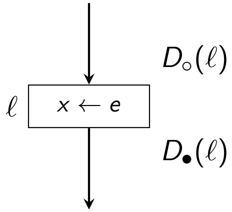
Compiler Construction 2013

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Another kind of CFGs



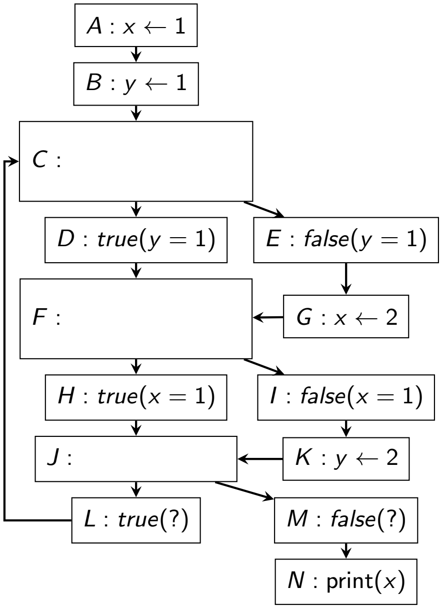
Effects on edges. Nodes called program points. One data flow fact per program point. Join of data flow facts done in fixpoint iteration (cf. data flow slides).



Nodes are basic blocks of instructions. Closer to the hardware. Edges denote flow of control. Every node has incoming ($\circ$) and outgoing ($\bullet$) data flow information:

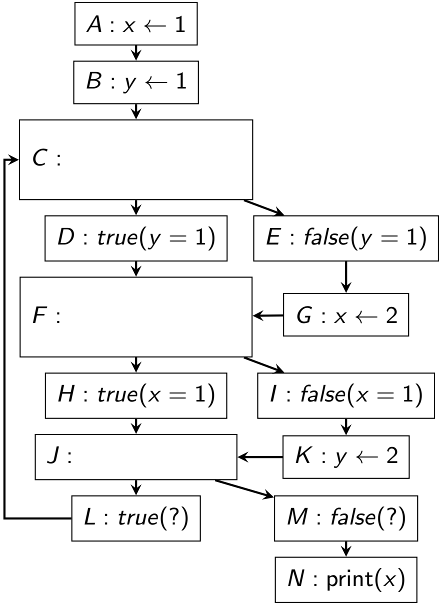
$$D_\circ(\ell) := \bigsqcup_{p \in pred(\ell)} D_\bullet(p)$$

Problem and Motivation



- Consider Constant Propagation
- Lattice: $\mathbb{D} := (Vars \to \mathbb{Z}^{\top})_{\perp}$
- Per CFG node we have to keep a mapping from $V := |Vars|$ variables to abstract values
- Space requirement $N \times V$
- Thus runtime $O(N \times V)$ rounds in the fixpoint iteration
- and $O(N \times V^2)$ in analysis updates per variable

Flow-Insensitive Constant Propagation

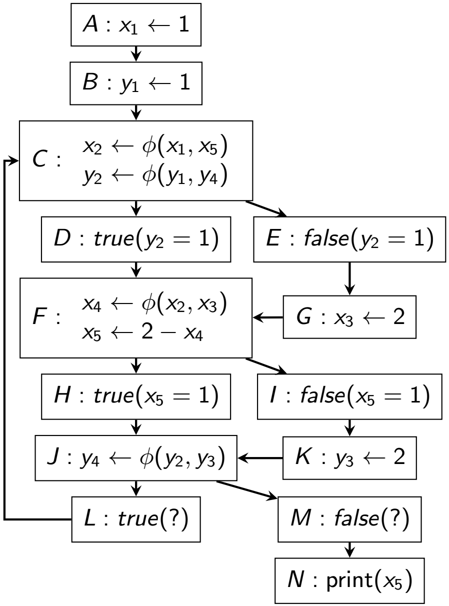


- Get around storing a map from vars to $\mathbb{Z}^\top$ at every program point
- Keep one element $x \in \mathbb{D}$ per CFG not per program point
- Solve the single equation

$$d \sqsupseteq \bigsqcup_i f_i(d)$$

- Loss of precision because abstract values of all definitions of a variable are joined

SSA



$A : x_1 \leftarrow 1$
$B : y_1 \leftarrow 1$
$C : \begin{array}{l} x_2 \leftarrow \phi(x_1, x_5) \\ y_2 \leftarrow \phi(y_1, y_4) \end{array}$
$D : \mathit{true}(y_2 = 1)$
$E : \mathit{false}(y_2 = 1)$
$F : \begin{array}{l} x_4 \leftarrow \phi(x_2, x_3) \\ x_5 \leftarrow 2 - x_4 \end{array}$
$G : x_3 \leftarrow 2$
$H : \mathit{true}(x_5 = 1)$
$I : \mathit{false}(x_5 = 1)$
$J : y_4 \leftarrow \phi(y_2, y_3)$
$K : y_3 \leftarrow 2$
$L : \mathit{true}(?)$
$M : \mathit{false}(?)$
$N : \text{print}(x_5)$

- Flow-Insensitive Analyses
- Each Variable has a static single assignment, i.e. one program point where it occurs on the left-hand side of an assignment
- Identify program points and variable names
- ϕ-functions select proper definitions at control-flow joins

(Un-Conditional) Constant Propagation in SSA

- Perform flow-insensitive analysis on SSA-program
- Domain: $\mathbb{D} := (\mathit{Vars} \to \mathbb{Z}_\bot^\top)$
- Transfer functions:

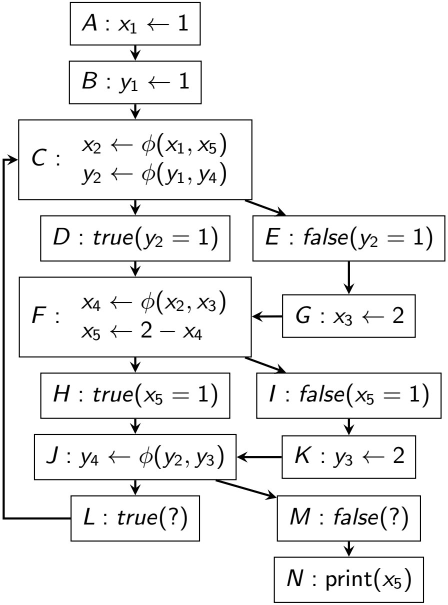
$$\begin{array}{rcl} [\![;]\!]^\sharp\, D & := & D \\ [\![x \leftarrow e;]\!]^\sharp\, D & := & D[x \mapsto [\![e]\!]^\sharp] \\ [\![x \leftarrow M[e];]\!]^\sharp\, D & := & D[x \mapsto \top] \\ [\![M[e_1] \leftarrow e_2]\!]^\sharp\, D & := & D \\ [\![x_0 \leftarrow \phi(x_1, \ldots, x_n)]\!]^\sharp\, D & := & D[x_0 \mapsto \bigsqcup_{1 \le i \le n} D(x_i)] \end{array}$$

- ϕ-functions make join over different reading definitions explicit
- Solve single inequality

$$D \sqsupseteq \bigsqcup_i f_i\, D$$

by fixpoint iteration

Example



	0	1	2	3
x_1	$\perp$	1	1	1
y_1	$\perp$	1	1	1
x_2	$\perp$	$\perp$	1	$\top$
y_2	$\perp$	$\perp$	1	$\top$
x_3	$\perp$	2	2	2
x_4	$\perp$	$\perp$	$\top$	$\top$
x_5	$\perp$	$\perp$	$\top$	$\top$
y_3	$\perp$	2	2	2
y_4	$\perp$	$\perp$	$\top$	$\top$

Round-robin iteration. Initialization with $\perp$. Fixed point reached after three rounds. Precision loss at ϕs because we could not exclude unreachable code.

Conditional Constant Propagation on SSA

called *sparse conditional constant propagation* (SCCP) [Wegman et al. 1991]

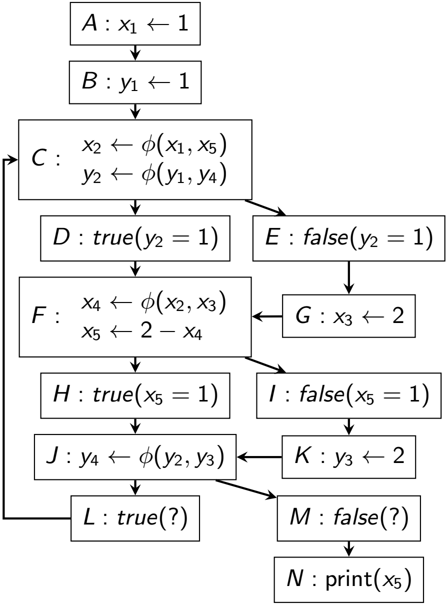
- Consider control flow as well. Perform two analysis in parallel
- Cooperation between two domains:

$$\mathbb{D} := \mathit{Vars} \to \mathbb{Z}_{\perp}^{\top} \qquad \text{Blocks} \to \mathbb{C} := \{\mathtt{d}, \mathtt{r}\}$$

- $\mathtt{d}$ = dead code, $\mathtt{r}$ = reachable code
- Two transfer functions per program point i:
  $f_i : \mathbb{D} \times \mathbb{C} \to \mathbb{D}$ for constant propagation
  $g_i : \mathbb{D} \times \mathbb{C} \to \mathbb{C}$ for reachability
- Solve system of equations

$$\begin{array}{rcl} x & \sqsupseteq & \bigsqcup f_i(x, y) \\ y & \sqsupseteq & \bigsqcup g_i(x, y) \end{array} \qquad x \in \mathbb{D}, y \in \mathbb{C}$$

Example



	0	1	2
x_1	$\perp$	1	1
y_1	$\perp$	1	1
x_2	$\perp$	1	1
y_2	$\perp$	1	1
x_3	$\perp$	2	2
x_4	$\perp$	1	1
x_5	$\perp$	1	1
y_3	$\perp$	2	2
A	r	r	r
B	d	r	r
C	d	r	r
D	d	r	r
E	d	d	d
F	d	r	r
G	d	d	d
H	d	r	r
I	d	d	d
J	d	r	r
K	d	d	d
L	d	r	r
M	d	r	r
N	d	r	r

Round-robin interation. Each column shows the value of $x \in \mathbb{D}$ (upper rows) and $y \in \mathbb{C}$ (lower rows) in a single iteration of the fixpoint algorithm. Initial values are $\perp$ and d. Root node A initialized with r. Fixed point reached after one round. Can prove code dead in cooperation with constant propagation information.

Transfer Functions

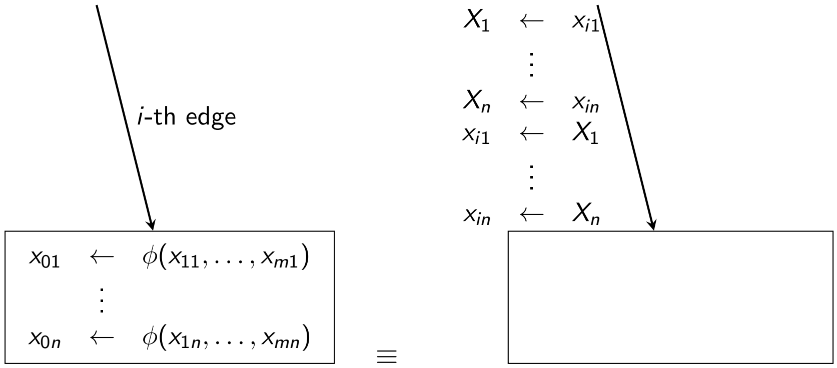
- For constant propagation (functions f_i)

$$\begin{aligned}
[\![\ell : x \leftarrow e;]\!]^\sharp\, D, C &:= D[x \leftarrow [\![e]\!]^\sharp\, D] \\
[\![\ell : x \leftarrow M[e];]\!]^\sharp\, D, C &:= D[x \leftarrow \top] \\
[\![\ell : x_0 \leftarrow \phi(x_1, \ldots, x_n)]\!]^\sharp\, D, C &:= D[x_0 \mapsto \bigsqcup X] \\
X &:= \{x_i \mid C(\mathit{pred}(\ell, i)) = \mathtt{r}\} \\
[\![\cdot]\!]^\sharp\, D, C &:= D
\end{aligned}$$

- For reachability (functions g_i)

$$\begin{aligned}
[\![\ell : \mathit{true}(e)]\!]^\sharp\, D, C &:= C\left[\ell \mapsto \begin{cases} \mathtt{d} & [\![e]\!]^\sharp\, D \sqsubseteq 0 \\ \mathtt{r} & \text{otherwise} \end{cases}\right] \\
[\![\ell : \mathit{false}(e)]\!]^\sharp\, D, C &:= C\left[\ell \mapsto \begin{cases} \mathtt{r} & 0 \sqsubseteq [\![e]\!]^\sharp\, D \\ \mathtt{d} & \text{otherwise} \end{cases}\right] \\
[\![\cdot]\!]^\sharp\, D, C &:= C
\end{aligned}$$

ϕ-functions have semantics



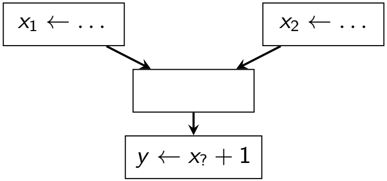
$$\begin{array}{lcl} x_{01} & \leftarrow & \phi(x_{11}, \ldots, x_{m1}) \\ & \vdots & \\ x_{0n} & \leftarrow & \phi(x_{1n}, \ldots, x_{mn}) \end{array} \equiv \begin{array}{lcl} X_1 & \leftarrow & x_{i1} \\ & \vdots & \\ X_n & \leftarrow & x_{in} \\ x_{i1} & \leftarrow & X_1 \\ & \vdots & \\ x_{in} & \leftarrow & X_n \end{array}$$

Commonly stated as

All ϕ-functions are evaluated simultaneously at the beginning of the block

Where to place ϕ-functions?

- ϕ-functions have to be placed such that
  1. SSA program P' has the same semantics as original program P
  2. Every variable has exactly one program point where it is defined



- Observation:
  - First point reached by two different definitions of (non-SSA) variable has to contain a ϕ-function
  - In the SSA-form program, every use is reached by a single unique definition

Join Points

Definition

Two paths $p : X_0 \xrightarrow{*} X_j$ and $q : Y_0 \xrightarrow{*} Y_k$ converge at a program point Z if

1. $X_0 \neq Y_0$
2. $Z = X_j = Y_k$
3. $X_{j'} = Y_{k'} \implies j = j' \vee k = k'$

- A program point Z needs a ϕ-function for variable a, if it is the convergence point of two program points X_0 and Y_0 where each is a definition of a
- Formally: $J(S) := \{Z \mid X, Y \in S \text{ converge at } Z\}$.
- $J(defs(a))$ is the set of program points where ϕ-functions have to be placed for a
- How to compute join points efficiently?

Dominance

- Every SSA variable has a unique program point where it is defined
- The definition of a SSA variable dominates all its (non-ϕ) uses

Definition (Dominance)

A node X in the CFG dominates a node Y if every path from *entry* to Y contains X. Write $X \geq Y$.

- Dominance is a partial order
- Dominance is a tree order: For every X, Y, Z with $X \geq Z$ and $Y \geq Z$ holds $X \geq Y$ or $Y \geq X$
- Strict dominance: $X > Y := X \geq Y \wedge X \neq Y$
- Immediate/direct dominator: $\mathit{idom}(Z) = X$ with $X > Z \wedge \nexists Y : X > Y > Z$

Dominance Frontiers

Efficiently computing SSA. . . [Cytron et al. 1991]

Definition (Dominance Frontier)

$$DF(X) = \{Y \mid X \not> Y \wedge (\exists Z \text{ predecessor of } Y : X \geq Z\}$$

- $DF^+(X)$ is the least fixed point S of $S = DF(X \cup S)$
- Theorem:
$$DF^+(X) = J(X)$$
- Proof Sketch:
  1. Show that for every path $p : X \xrightarrow{*} Z$ there is a node in $\{X\} \cup DF^+(X)$ on p that dominates Z
  2. Show that the convergence point Z of two paths $X \xrightarrow{*} Z, Y \xrightarrow{*} Z$ is contained in $DF^+(X) \cup DF^+(Y)$
  3. Using this, we can show that $J(S) \subseteq DF^+(S)$
  4. Show $DF(S) \subseteq J(S)$ for $\mathit{entry} \in S$
  5. Using induction on DF^i show that $DF^+(S) \subseteq J(S)$

Dominance Frontiers

Definition (Dominance Frontier)

$$DF(X) = \{Y \mid X \not> Y \wedge (\exists Z \text{ predecessor of } Y : X \geq Z\}$$

- Can be efficiently computed by a bottom up traversal over the dominance tree:
    1. Each CF-successor Z of X is either dominated by X or not
    2. if not, it is in the dominance frontier of X
    3. if yes, look at the dominance frontier of Z: All $Y \in DF(Z)$ not dominated by X are also in $DF(X)$

$$\begin{aligned} DF(X) = & \{Y \text{ successor of } X \mid X \not> Y\} \\ & \cup \bigcup_{X = idom(Z)} \{Y \in DF(Z) \mid X \not\geq Y\} \end{aligned}$$

SSA Construction

Cytron et al.

1. Compute dominance tree
2. Compute iterated dominance frontiers $DF^+(X)$ for all definitions of each variable
3. Rename variables
   - Every use takes lowest definition in the dominance tree
   - Note that ϕ-function uses happen at the end of the predecessors
   - First lemma of proof sketch guarantees that this definition is available

SSA Construction

Buchwald et al.

```
writeVariable(variable, block, value):
    currentDef[variable][block] ← value

readVariable(variable, block):
    if currentDef[variable] contains block:
        return currentDef[variable][block]
    return readVariableRecursive(variable, block)

readVariableRecursive(variable, block):
    if |block.preds| = 1:
        # Optimize the common case of one predecessor: No phi needed
        val ← readVariable(variable, block.preds[0])
    else:
        # Break potential cycles with operandless phi
        val ← new Phi(block)
        writeVariable(variable, block, val)
        val ← addPhiOperands(variable, val)
    writeVariable(variable, block, val)
    return val
```

SSA Construction (cont'd)

Buchwald et al.

```
addPhiOperands(variable, phi):
    # Determine operands from predecessors
    for pred in phi.block.preds:
        phi.appendOperand(readVariable(variable, pred))
    return tryRemoveTrivialPhi(phi)

tryRemoveTrivialPhi(phi):
  if $\exists v : \texttt{phi.args} \subseteq \{\texttt{phi}, v\}$
        same ← v
    else:
        same ← phi
    users ← phi.users.remove(phi) # Remember all users except the phi itself
    phi.replaceBy(same) # Reroute all uses of phi to same and remove phi

    # Try to recursively remove all phi users, which might have become trivial
    for use in users:
        if use is a Phi:
            tryRemoveTrivialPhi(use)
    return same
```