SSA Introduction

Sebastian Hack
hack@cs.uni-saarland.de

Compiler Construction 2013
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 (∘) and outgoing (●) data flow information:

\[ D_\circ(\ell) := \bigsqcup_{p \in \text{pred}(\ell)} D_\bullet(p) \]
Problem and Motivation

- Consider Constant Propagation
- Lattice: $\mathbb{D} := (\text{Vars} \rightarrow \mathbb{Z}^\top) \perp$
- Per CFG node we have to keep a mapping from $V := |\text{Vars}|$ 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 $x \in D$ per CFG not per program point
- Solve the single equation
  \[ d \supseteq \bigsqcup_i f_i(d) \]
- Loss of precision because abstract values of all definitions of a variable are joined
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

$\phi$-functions select proper definitions at control-flow joins

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

$\phi$-functions select proper definitions at control-flow joins
(Un-Conditional) Constant Propagation in SSA

- Perform flow-insensitive analysis on SSA-program

- Domain: $\mathbb{D} := (\text{Vars} \rightarrow \mathbb{Z}^\top)$

- Transfer functions:

  \[
  \begin{align*}
  [\cdot; \cdot]^\# D & := D \\
  [x \leftarrow e; \cdot]^\# D & := D[x \mapsto [e]^\#] \\
  [x \leftarrow M[e]; \cdot]^\# D & := D[x \mapsto \top] \\
  [M[e_1] \leftarrow e_2]^\# D & := D \\
  [x_0 \leftarrow \phi(x_1, \ldots, x_n)]^\# D & := D[x_0 \mapsto \bigcup_{1 \leq i \leq n} D(x_i)]
  \end{align*}
  \]

- $\phi$-functions make join over different reaching definitions explicit

- Solve single inequality

  \[
  D \supseteq \bigcup_i f_i D
  \]

  by fixpoint iteration
Round-robin iteration. Initialization with $\perp$. Fixed point reached after three rounds. Precision loss at $\phi$s 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:
  \[
  \mathbb{D} := \text{Vars} \rightarrow \mathbb{Z}^{\top} \quad \text{Blocks} \rightarrow \mathbb{C} := \{d, r\}
  \]

- \(d = \text{dead code}, \ r = \text{reachable code}\)

- Two transfer functions per program point \(i\):
  \[
  f_i : \mathbb{D} \times \mathbb{C} \rightarrow \mathbb{D} \text{ for constant propagation}
  \]

  \[
  g_i : \mathbb{D} \times \mathbb{C} \rightarrow \mathbb{C} \text{ for reachability}
  \]

- Solve system of equations

  \[
  x \sqsubseteq \bigsqcup f_i(x, y) \\
  y \sqsubseteq \bigsqcup g_i(x, y) \\
  x \in \mathbb{D}, y \in \mathbb{C}
  \]
Round-robin iteration. 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 $\bot$ 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 $f_i$)

  \[
  \begin{align*}
  \llbracket \ell : x \leftarrow e; \rrbracket^D, C & := D[x \leftarrow [e]^D] \\
  \llbracket \ell : x \leftarrow M[e]; \rrbracket^D, C & := D[x \leftarrow \top] \\
  \llbracket \ell : x_0 \leftarrow \phi(x_1, \ldots, x_n) \rrbracket^D, C & := D[x_0 \mapsto \bigcup X] \\
  X & := \{x_i \mid C(\text{pred}(\ell, i)) = r\} \\
  \llbracket \cdot \rrbracket^D, C & := D
  \end{align*}
  \]

- For reachability (functions $g_i$)

  \[
  \begin{align*}
  \llbracket \ell : \text{true}(e) \rrbracket^D, C & := C \\
  \ell \mapsto \begin{cases} 
  d & [e]^D \subseteq 0 \\
  r & \text{otherwise}
  \end{cases} \\
  \llbracket \ell : \text{false}(e) \rrbracket^D, C & := C \\
  \ell \mapsto \begin{cases} 
  r & 0 \subseteq [e]^D \\
  d & \text{otherwise}
  \end{cases} \\
  \llbracket \cdot \rrbracket^D, C & := C
  \end{align*}
  \]
\(\phi\)-functions have semantics

Commonly stated as

All \(\phi\)-functions are evaluated \textit{simultaneously} at the beginning of the block
Where to place $\phi$-functions?

- $\phi$-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 $\phi$-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. $X_{j'} = Y_{k'} \implies j = j' \lor k = k'$

- A program point $Z$ needs a $\phi$-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 $\phi$-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-$\phi$) 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: $X > Y \equiv X \geq Y \land X \neq Y$
- Immediate/direct dominator: $idom(Z) = X$ with $X > Z \land \nexists Y : X > Y > Z$
Dominance Frontiers

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

Definition (Dominance Frontier)

\[ DF(X) = \{ Y \mid X \not\succ Y \land (\exists Z \text{ predecessor of } Y : X \geq Z) \} \]

- \( DF^+(X) \) 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 \( p : X \rightarrow Z \) there is a node in \( \{X\} \cup DF^+(X) \) on \( p \) that dominates \( Z \)
  2. Show that the convergence point \( Z \) of two paths \( X \rightarrow Z, Y \rightarrow Z \) 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 entry \( \in S \)
  5. Using induction on \( DF^i \) show that \( DF^+(S) \subseteq J(S) \)
Dominance Frontiers

**Definition (Dominance Frontier)**

\[ DF(X) = \{ Y \mid X \not\succ Y \land (\exists Z \text{ predecessor of } Y : X \geq Z) \} \]

- 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 \( DF(X) \)

\[ DF(X) = \{ Y \text{ successor of } X \mid X \not\succ Y \} \]
\[ \cup \bigcup_{X=\text{idom}(Z)} \{ Y \in DF(Z) \mid X \not\succ Y \} \]
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 $\phi$-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 \( \exists v : \phi \text{.args} \subseteq \{\phi, v\} \)
        same \leftarrow v
    else:
        same \leftarrow \phi
        users \leftarrow \phi \text{.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