Static Single Assignment (SSA) Form
Construction - Analyses - Optimizations

Welf Löwe
wlo@msi.vxu.se
http://www.msi.vxu.se/~wlo/teaching.htm

Outline

- Introduction to SSA
  - Motivation
  - Value Numbering
  - Definition, Observations
- Construction, Destruction
  - Theoretical, Pessimistic, Optimistic Construction
  - Destruction
  - Memory SSA
- Analyses and Optimizations

Intermediate Representations

- Intermediate representations (like BB, SSA graphs) separate compiler front-end (source code related representation) from back-end (target code related representation)
- Analyses and optimizations can be performed independently of the source and target languages
- Tailored for analyses and optimizations

What is an IR tailored for analyses and optimizations?

- Represents dependencies of operations in the program
  - Control flow dependencies
  - Data dependencies
- Only essential dependencies (approximation)
  - A dependency \( s \rightarrow s' \) of operations is essential if and only if execution of \( s \) changes observable behavior of the program
  - Computation of essential dependencies is not decidable
- Compact representation
  - … of dependencies
  - No redundant computations (approximation)
Static Single Assignment - SSA

- **Goal:**
  - Increase efficiency of inter-procedural analyses and optimizations
  - Speed up dataflow analysis
  - Represent def-use relations explicitly

- **Idea:**
  - Represent program as a directed graph of operations $\tau$
  - Represent triples as a single assignment $t := \tau \ t' \ t''$ with $t$, $t'$, $t''$ a variable/label/register/edge connecting operations

- **SSA-Property:** there is only one position in a program/procedure defining $t$

  - Does not mean is computed $t$ only once (due to iterations the program point is executed more than once with different values)

Avoid redundant computations

- Assign each (partial) expression a unique number.
  - Good for optimization, as values can be reused instead of recomputed
  - Known as value numbering
  - Basic idea for SSA

- Values that are provable equivalent get the same number

- How to find equal values?

  - Can be computed by data flow analysis (forward, must)

Equivalent Values

- Two expressions are semantically equivalent, \[iff\] they compute the same value

  - Not decidable

- Two expressions are syntactically equivalent, \[iff\] the operator is the same and the operands are the same of syntactically equivalent

- Generalization towards semantic equivalence using algebraic identities, e.g. $a + a = 2 * a$

  - In practice provable equivalence (conservative approximation): two expressions are congruent, \[iff\] they are syntactically equivalent or algebraic identical

Idea of Value Numbering

- Congruent values get the same value number

  - Values are defined by operations and used by other operations

  - Values computed only once and then reused (referring to their value number)

- Algorithmic idea to prove equality of expression values at different program points (congruence of tuples):
  - Basic case: constants are easy to proof equivalent

  - Induction: see definition of syntactic equivalence: if inputs of two operations equal and the operator is equal the computed values are also equal

  - Also apply algebraic identities to prove congruence

- Problems:
  - Alias/Points-to problem: Addresses not exactly computable. Where are values stored to and loaded from? Not decidable.

  - Meets in control flow: which value holds?
Value Numbering

- Type of value numbers:
  - INT for integer constants; BOOL for Boolean constants etc.
  - Use labels: \( \{ t_1, \ldots, t_n \} \) otherwise.

- Data structure:
  - Hash table for the assignment of value numbers to tuples \( t : \tau t' t'' \).
  - \( t', t'' \) are value numbers as well, i.e., hash table entries or constants.

- For a first try:
  - Computation basic block local.
  - One table per basic block.

Value Numbering with Local Variables without Alias Problem

(1) Initially: value number \( \text{vn}(\text{constant}) = \text{constant} \);
    \( \text{vn}(t) = \text{void} \) for all tuples \( t \).

(2) for all tuple \( t \) in program order:
    
    \( \text{case (a)} \ t \sim \text{ST} \sim \text{local} < t' \):
    
    \( \text{vn}(t) \sim \text{vn}(\text{ST} \text{local} < t') \)
    
    if \( \text{vn}(t) = \text{void} \) then
    
    \( \text{vn}(t) \sim \text{vn}(\text{LD} \text{local} < t') \)
    
    \( \text{vn}(t) \sim \text{new value number}, \)
    
    \( \text{generate: } \text{ST} \text{local} < \text{vn}(t') \)
    
    \( \text{case (b)} \ t \sim \text{LD} \text{local} > t' \):
    
    \( \text{vn}(t) \sim \text{vn}(\text{LD} \text{local} > t') \)
    
    if \( \text{vn}(t) = \text{void} \) then
    
    \( \text{vn}(t) \sim \text{new value number}, \)
    
    \( \text{generate: } \text{LD} \text{local} > \text{vn}(t') \)
    
    \( \text{case (c)} \ t \sim \tau t' t'' \):
    
    \( \text{vn}(t) \sim \text{vn}(\tau \text{vn}(t') \text{vn}(t'')) \)
    
    if \( \text{vn}(t) = \text{void} \) then
    
    \( \text{vn}(t) \sim \text{new value number}, \)
    
    \( \text{generate: } \tau \text{vn}(t') \text{vn}(t'') \).
    
    \( \text{case (d)} \ t \sim \text{call proc } t' t'' \ldots \sim \text{call proc } \)

Example

<table>
<thead>
<tr>
<th>Original</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t_1: \text{ST} \sim \text{a} &lt; 2 )</td>
<td>( v_1: \text{ST} \sim \text{a} &lt; 2 )</td>
</tr>
<tr>
<td>( t_2: \text{LD} \sim \text{a} )</td>
<td>( v_2: \text{LD} \sim \text{a} )</td>
</tr>
<tr>
<td>( t_3: \text{LD} \sim \text{c} )</td>
<td>( v_4: \text{ADD} \sim v_3 \sim 1 )</td>
</tr>
<tr>
<td>( t_4: \text{MUL} \sim t_2 \sim t_3 )</td>
<td>( v_5: \text{ST} \sim \text{b} &lt; v_4 )</td>
</tr>
<tr>
<td>( t_6: \text{ST} \sim \text{b} &lt; t_5 )</td>
<td>( v_6: \text{ST} \sim \text{a} &lt; v_3 )</td>
</tr>
<tr>
<td>( t_7: \text{LD} \sim \text{c} )</td>
<td>( v_7: \text{ADD} \sim v_4 \sim v_4 )</td>
</tr>
<tr>
<td>( t_8: \text{ST} \sim \text{a} &lt; t_8 )</td>
<td>( v_8: \text{ST} \sim \text{c} &lt; v_7 )</td>
</tr>
<tr>
<td>( t_9: \text{LD} \sim \text{a} )</td>
<td>( v_9: \text{ADD} \sim v_4 \sim v_4 )</td>
</tr>
<tr>
<td>( t_{10}: \text{ADD} \sim t_{10} \sim 1 )</td>
<td>( v_{10}: \text{ST} \sim \text{b} &lt; v_4 )</td>
</tr>
<tr>
<td>( t_{11}: \text{ADD} \sim t_{11} \sim t_{12} )</td>
<td>( v_{12}: \text{ST} \sim \text{c} &lt; v_7 )</td>
</tr>
<tr>
<td>( t_{14}: \text{ST} \sim \text{c} &lt; t_{13} )</td>
<td>( v_{13}: \text{ADD} \sim v_4 \sim v_4 )</td>
</tr>
</tbody>
</table>
Value Numbering

with Global Variables without Alias Problem

- Case (a') $t = ST > \text{global} < i'$
  - $vn(t) := vn (ST > \text{global} < vn(i'))$
  - if $vn (t) = \text{void}$:
    - $vn (LD < \text{global} >) := vn (i')$
    - $vn (t) := \text{new value number}$
  - Generate always: $vn (t) := t$.

- Non recursive procedures:
  - New case (d) let $t = \text{call proc} t \cdots$
  - Analyze procedure as if it was inlined (caution: exponential but might be ok in practice)

- Recursive procedures:
  - Case (d) as before, but if global (potentially) redefined in proc, set value number for tuple $ST > \text{global} <, LD < \text{global} >$ and transitive depending tuples to void
  - Easy implementation: set all value numbers to void

Remarks

- Initially all value numbers are set to void. By knowing the values of predecessor basic blocks, this can be relaxed (initializations over basic block solved by SSA, next issue)
- Each entry in the hash table generates a new value number. After calls $t$ entries $i'$ depending on non-local variables get void
- A store operation $ST > a < i'$ sets void all $vn (LD < a' >)$, if it is not clear, whether $a = a'$ or $a \neq a'$ (alias-problem). Special case: arrays with index expressions

Value number graph $\rightarrow$ SSA

- SSA-Property: there is only one position in a program/procedure defining $t$
- Half way to SSA representation due to value numbering, i.e. value number graph is SSA graph of a basic block
- Problem: What to do with variables having assignments on more then one position?
- E.g.
  
  ```
  if ... then i:=1 else i:=2 end; x:=i
  i:=0; while ...loop ...; i:=i+1; ... end; x:=i
  ```
**φ-Functions**

- **Solution:**
  - Each assignment to a variable $a$ defines a new version $a_i$.
  - This version is actually its value number.
  - At meets in the control flow, just select a value which introduces a new version (value number) defined by a pseudo operation $a_i := \phi(a_1, a_2)$
  - E.g.
    
    ```
    if ... then $i_1 := 1$ else $i_2 := 2$ end; $i_3 := \phi(i_1, i_2)$; $x := i_3$
    ```
  - $\phi$-functions always at the beginning of a block
  - Evaluated simultaneously
  - Assignment $i_0 := \phi(i_1, ..., i_k)$ in a basic block indicates that the block has $k$ direct predecessors in the control flow

**Compact representation of dependencies**

- Previous: #def $x$ #use dependency edges
- Now: #def + #use dependency edges

**Branches and Loops**

- $\phi$-function guarantees that there is exactly one definition/assignment for each use of a variable

**Example Program and BB Graph**

```plaintext
(1) a=1;
(2) b=2;
(3) c=a+b;
(4) if (d=c-a)
    (5) while (d=b*d)
        (6) d=a+b;
        (7) e=e+1;
    (8) b=a+b;
    (9) if (e=c-a) break;
(10) a=b*d;
(11) b=a-d;
```

![BB Graph](image)
SSA properties

P1: Typed in-/output of nodes, in- and output of operation node connected by edges have the same type.
P2: Operation nodes and edges of a basic block (BB) are a DAG. Note: correspondence to triples and expression trees
P3: Input of $\phi$-operations have same type as their output.
P4: $i$-th operand of a $\phi$-operation is available at the end of the $i$-th predecessor BB.
P5: A start node Start dominates all BBs of a procedure, an end node End post-dominates all nodes of a procedure.
P6: Every block has exactly one of nodes Start, End, Jump, Cond, Ret
P7: If operation $x$ of a BBs $B_x$ defines a operand of operation $y$ of a BBs $B_y$ then there is a path $B_x \rightarrow^* B_y$.
P7a: (Special case of P7) Operation $y$ is a $\phi$-operation and $x$ in $B_x = B_y$ then there is a cyclic path $B_y \rightarrow^* B_y$.
P8: Let $X, Y$ be BBs with a definition of $a$ reaching a use of $a$ in BB $Z$. Let $Z'$ be the first common BB of execution paths $X \rightarrow^* Z, Y \rightarrow^* Z$. Then $Z'$ contains a $\phi$-operation for $a$.

Property P8 revisited

Let $X, Y$ be BBs with a definition of $a$ reaching a use of $a$ in BB $Z$. Let $Z'$ be the first common BB of execution paths $X \rightarrow^* Z, Y \rightarrow^* Z$. Then $Z'$ contains a $\phi$-operation for $a$.

Dominance

Dominance: $X \geq Y$
On each path from the starting node $S$ in the basic block graph to $Y$, $X$ before $Y$.
$\geq$ is reflexive: $X \geq X$.
Strict Dominance: $X > Y$
$X > Y \iff X \geq Y \land X \not= Y$
Direct Dominance: $ddom(X)$
$X = ddom(Y) \iff X > Y \land \exists Z: X > Z > Y$.
Post-Dominance: $X \leq Y$
On each path from the end node $E$ in the basic block graph to $Y$, $X$ before $Y$.
Definitions of strict and direct post dominance analogously.

Dominance Analysis

- Can be formalized as a data flow problem (forward, must)
- Given basic block graph $G = (V, E)$ with $pre$ precedence relation in $G$:
  $D_{in}(n) = \bigcup_{p \in pre(n)} D_{out}(p)$ and $D_{out}(n) = D_{in}(n) \cup \{n\}$
- Initialization of nodes
  $D_{out}(n) = V$ (all nodes dominate each other)
- $pre(n)$ not defined for start node $n_0: D_{in}(n_0) = D_{out}(n_0) = \{n_0\}$.
- Simplification:
  - $D(n) = \{n\} \cup \bigcup_{p \in pre(n)} D(p)$
  - possible as in $n$ no information gets killed
  - initialization $\forall n \in V \setminus \{n_0\}$: $D(n) = V$ and $D(n_0) = \{n_0\}$
Example

Dominator Tree $G_D$

Dominator tree $G_D (G) = (V_D, E_D)$ of control flow graph $G = (V, E)$ with $V_D = V$ and $(v_D, v_D') \in E_D \iff e = ddom (e')$

(Iterated) Dominance Frontiers

- Dominance Frontier $DF (n)$
  - Set of nodes just not dominated by $n$ any more
  - $DF (n) = \{ m \mid n \not\sim m \land \exists b \in pre (m): n \geq b \}$.

- Dominance Frontiers of a Set $M$ of nodes $DF (M)$
  - $DF (M) = \bigcup_{n \in M} DF (n)$

- Iterated Dominance Frontier $DF^+ (M)$
  - minimum fix point of:
    - $DF_0 = DF (M)$,
    - $DF_{i+1} = DF (DF_i)$.

Discussion P8

- Defines, where to position $\phi$-functions
- Let $X, Y$ be the (only) BBs containing a definition of $a$. Both reach a use of $a$ in block $Z$. Then there are paths $X \rightarrow^* Z$, $Y \rightarrow^* Z$ in the BB graph (P7).
  - Let $Z'$ be the first node common to both paths then:
    - $a_3 := \phi (a_1, a_2)$ is assigned to $Z'$
    - $Z' \geq Z$ ($Z'$ dominates $Z$) otherwise $a$ not initialized
    - $Z'$ is the first common successor of $X$ and $Y$, dominating $Z$
    - $Z' \in DF (X, Y)$: $Z'$ is in the dominance frontier of $X$ and $Y$
  - Since $a_3 := \phi (a_1, a_2)$ itself is a definition of $a$, this can trigger insertion of new $\phi$ functions.
  - Iteration required,
  - Observation: $\phi$-functions in the iterated dominance frontiers $DF^+ (...)$ of the BB defining a variable
  - Caution: don’t use this as construction idea!
Proof (idea) of this observation

1. $a_3 := \phi(a_1, a_2)$ cannot be defined before $Z'$
2. $X$ and $Y$, resp., must dominate all direct predecessors of $Z'$, otherwise there would be a use of $a$ without previous definition. Hence $Z' \in DF(X,Y)$
3. $a_3 := \phi(a_1, a_2)$ should not be inserted later since on the path $Z' \rightarrow Z$ no change of definition of $a$.
4. Iteration $DF^*(\ldots)$ is required, as there now a further definitions of $a$ in a block $Z' \notin DF(X,Y)$

Example Property E8

A($a$) = \{1, 2, 3\}
DF($A(a)$) = \{4\} = DF_{1}
DF_{2} = \{1, 4\} = DF^*

Outline

- Introduction to SSA
  - Motivation
  - Value Numbering
  - Definition, Observations
- Construction, Destruction
  - Theoretical, Pessimistic, Optimistic Construction
  - Destruction
  - Memory SSA
- Analyses and Optimizations

Construction Theory

- Program of size $n$ might contain $O(n)$ variables
- In the worst case there are $O(n)$ $\phi$–function (for the variables) in $O(n)$ BBs, hence worst case complexity is $\Omega(n^2)$
- Previous discussion gives a straight forward implementation:
  - Perform a value numbering and update BBs accordingly
  - For any used variable that is used but not locally (in BB) defined compute set of definition points (data flow analysis)
  - Compute iterated dominance frontiers of definition points
  - Insert $\phi$–function and rename variables accordingly
- In practice easier constructions possible
Remainder Value Numbering

(1) Initially: value number $\text{vn}(\text{constant}) = \text{constant}$; $\text{vn}(t) = \text{void}$ for all tuples $t$.

(2) for all tuple $t$ in program order:

\begin{enumerate}
  \item \text{case} $t = \text{ST}_{\text{local}} < t$
  \begin{itemize}
    \item $\text{vn}(t) := \text{vn}(\text{ST}_{\text{local}} < t)$
    \item if $\text{vn}(t) = \text{void}$:
      \begin{itemize}
        \item $\text{vn}(t) := \text{new value number}$,
        \item \text{generate}: $\text{ST}_{\text{local}} < \text{vn}(t)$
      \end{itemize}
  \end{itemize}

  \item $t = \text{LD}_{\text{local}}$
  \begin{itemize}
    \item $\text{vn}(t) := \text{vn}(\text{LD}_{\text{local}})$,
    \item if $\text{vn}(t) = \text{void}$:
      \begin{itemize}
        \item $\text{vn}(t) := \text{new value number}$,
        \item \text{generate}: $\text{vn}(t)$
      \end{itemize}
  \end{itemize}

  \item $t = \tau < t'$
  \begin{itemize}
    \item $\text{vn}(t) := \text{vn}(\tau < t')$
    \item if $\text{vn}(t) = \text{void}$:
      \begin{itemize}
        \item $\text{vn}(t) := \text{new value number}$,
        \item \text{generate}: $\text{vn}(t)$
      \end{itemize}
  \end{itemize}

  \item $t = \text{call proc } t' ...$ -- analog (c) with $\tau = \text{call proc}$
\end{enumerate}

Extended Initialization

(1) Initialization for current block $Z$

\begin{enumerate}
  \item always: $\text{vn}(\text{constant}) = \text{constant}$;
  \item if $Z = \text{start block}$: $\text{vn}(t) = \text{void}$ for all tuples $t$.
  \item else: let $\text{Pred} = \{X, Y, \ldots\}$ be the predecessors of $Z$ in basic block graph
    for all variables $t$ used in current block $Z$:
    \begin{itemize}
      \item if $\text{vn}(t) \neq \text{vn}(t)'$
        \begin{itemize}
          \item $\text{vn}(t)' := \text{new value number}$
          \item \text{generate}: $\text{vn}(t)' := \phi(\text{vn}(t)_{\text{X}}, \text{vn}(t)_{\text{Y}})$
        \end{itemize}
      \item if $\text{vn}(t)' = \text{vn}(t)'$
        \begin{itemize}
          \item $\text{vn}(t)' := \text{vn}(t)'$
        \end{itemize}
    \end{itemize}
\end{enumerate}

(2) for all tuple $t$ in program order:

-- as before

Extended Value Numbering

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
Z:_a_3 = \phi(_a_1, _a_2) \\
\end{array}
\]

ok

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

? ok

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]

\[
\begin{array}{c}
X:_a_1 = \ldots \\
Y:_a_2 = \ldots \\
\end{array}
\]
Extended Value Numbering

Extended Initialization

(1) Initialization for current block \( Z \)
   (A) always: \( vn(constant) = constant \);
   (B) if \( Z \) = start block: \( vn(t) = void \) for all tuples \( t \).
   (C) else: let \( Pred = \{ X, Y, ... \} \) be the predecessors of \( Z \) in basic block graph
        for all variables \( t \) used in current block \( Z \):
        if \( X \in Pred \) is unvisited
           \( vn(t)_X = new \) special value number (guessed number)
        if \( vn(t)_X = void \) recursively, initialize block \( X \) \( vn(t)_X \) with (1)
        if \( vn(t)_Y \neq vn(t)_Z \)
           \( vn(t)_Z = new \) value number
           generate if \( vn(t)_Y \) or \( vn(t)_Z \) is guessed: \( vn(t)_Z := \phi(vn(t)_X, vn(t)_Y) \)
           generate: \( vn(t)_Z := \phi(vn(t)_X, vn(t)_Y) \) otherwise
        if \( vn(t)_Y = vn(t)_Z \) \( vn(t)_Z := vn(t)_X \)

(2) for all tuple \( t \) in program order:
   -- as before

Eliminate/Mature \( \phi' \)-Functions

- After value numbering is finished for each block \( X \):
  - replace special value numbers in \( X \) by last valid in \( X \)
  - replace \( \phi' \)-functions by mature \( \phi \)-functions using replaced value numbers
  - delete: \( vn(t)_X := \phi(vn(t)_Y, vn(t)_Z) \)
    if \( t \) not changed in previously unvisited blocks, no \( \phi \) function required
  - replace then use of \( vn(t)_Z \) by \( vn(t)_Y \)
- Insight:
  - deletion could prove some other \( \phi \)-functions unnecessary
  - iterative deletion till fix point

Example I: Mature \( \phi' \)-Functions
Example II: Mature $\phi'$-Functions

Example III: Mature $\phi'$-Functions

Example Program and BB Graph

SSA Construction Block 1
SSA Construction Block 2 - Initialization

1. \( a_1 = 1 \)
   \( b_1 = 2 \)

2. \( a_2 = \phi \) \( b_2 = \phi \) \( (a_1, a_5) \)

3. \( b_3 = \phi \) \( (b_1, b_5) \)

4. \( c_1 = a_2 + b_2 \)
   \( d_1 = c_1 - a_2 \)

5. \( d = b_4 \)

6. \( e = e + 1 \)

SSA Construction Block 3 - Initialization

1. \( a_1 = 1 \)
   \( b_1 = 2 \)

2. \( a_2 = \phi \) \( b_2 = \phi \) \( (a_1, a_5) \)

3. \( b_3 = \phi \) \( (b_1, b_4) \)

4. \( d_3 = b_3 \cdot d_2 \)

5. \( e = e + 1 \)
SSA Construction Block 4 - Initialization

1. \( a_1 = 1 \)
2. \( b_1 = 2 \)
3. \( a = a_1 \)
4. \( b = b_1 \)
5. \( e = e_1 \)
6. \( d = d_1 \)
7. \( c = a + b \)
8. \( d = c - a \)

SSA Construction Block 4

1. \( a_1 = 1 \)
2. \( b_1 = 2 \)
3. \( a = a_1 \)
4. \( b = b_1 \)
5. \( e = e_1 \)
6. \( d = d_1 \)
7. \( c = a + b \)
8. \( d = c - a \)

SSA Construction Block 5 - Initialization

1. \( a_1 = 1 \)
2. \( b_1 = 2 \)
3. \( a = a_1 \)
4. \( b = b_1 \)
5. \( e = e_1 \)
6. \( d = d_1 \)
7. \( c = a + b \)
8. \( d = c - a \)

SSA Construction Block 5

1. \( a_1 = 1 \)
2. \( b_1 = 2 \)
3. \( a = a_1 \)
4. \( b = b_1 \)
5. \( e = e_1 \)
6. \( d = d_1 \)
7. \( c = a + b \)
8. \( d = c - a \)
SSA Construction Block 6 - Initialization

1. \( a_1 = 1 \)
2. \( b_1 = 2 \)
3. \( a_2 = \phi(d_1, d_5) \)
4. \( b_2 = \phi(b_1, b_5) \)
5. \( e_2 = \phi(e_1, e_5) \)
6. \( d_1 = b_2 * d_2 \)
7. \( d_4 = a_1 + b_1 \)
8. \( e_4 = e_1 + 1 \)
9. \( c_1 = a_1 + b_2 \)
10. \( d_1 = c_1 - a_2 \)

SSA Construction Block 6

1. \( a_1 = 1 \)
2. \( b_1 = 2 \)
3. \( a_2 = \phi(a_1, a_5) \)
4. \( b_2 = \phi(b_1, b_5) \)
5. \( e_2 = \phi(e_1, e_5) \)
6. \( d_1 = b_2 * d_2 \)
7. \( b_4 = \phi(b_2, b_4) \)
8. \( d_4 = a_1 + b_3 \)
9. \( e_4 = e_3 + 1 \)
10. \( a_3 = b_5 * d_5 \)
11. \( b_6 = a_3 - d_5 \)

SSA Mature Block 2

1. \( a_1 = 1 \)
2. \( b_1 = 2 \)
3. \( a_2 = \phi(a_1, a_4) \)
4. \( b_2 = \phi(b_1, b_4) \)
5. \( e_2 = \phi(e_1, e_4) \)
6. \( d_1 = b_2 * d_2 \)
7. \( d_4 = a_1 + b_3 \)
8. \( e_4 = e_3 + 1 \)
9. \( a_3 = b_5 * d_5 \)
10. \( b_6 = a_3 - d_5 \)

SSA Mature Block 2

1. \( a_1 = 1 \)
2. \( b_1 = 2 \)
3. \( a_2 = \phi(a_1, a_4) \)
4. \( b_2 = \phi(b_1, b_4) \)
5. \( e_2 = \phi(e_1, e_4) \)
6. \( d_1 = b_2 * d_2 \)
7. \( d_4 = a_1 + b_3 \)
8. \( e_4 = e_3 + 1 \)
9. \( a_3 = b_5 * d_5 \)
10. \( b_6 = a_3 - d_5 \)
Optimistic SSA Construction

- **Idea:**
  - all values (value numbers) are equal until the opposite is proven
  - opposite is proven by:
    - Values are different constants
    - Values are generated from syntactical different operations
    - Values are generated from syntactical equivalent operations with proven different values as operands

- **Advantage:**
  - Detects sometimes congruence that are not detected by pessimistic construction
  - No φ−functions to mature

- **Disadvantage:**
  - Detects sometimes congruence not that are detected by pessimistic construction (e.g. algebraic identities)
  - Requires Definition-Use-Analyses on BB graph on construction
  - Requires computation of iterated dominance frontiers to position φ−functions

Construction Algorithm

- Generate BB graph and perform Definition-Use-Analysis (data flow analysis) for all variables. Notations:
  - \( v_{(i)} = ... \) Variable \( v \) in statement \( (i) \) defined
  - \( u_{(i)} = ... v_{(x,y,z,...)} \) Variable \( v \) in statement \( (i) \) used with possible definitions in statements \( (x,y,z,...) \)
  - statements can be abstracted to blocks
- Set \( v_{(i)} = u_{(i)} \) for all \( v_{(i)}, u_{(i)} \) in the program
- Iterate until fix point:
  - Set \( v_{(i)} = u_{(i)} \) for:
    - \( v_{(i)} = c \) and \( u_{(i)} \neq c \)
    - \( v_{(i)} = \text{op}(...) \) and \( u_{(i)} \neq \text{op}(...) \)
    - \( v_{(i)} = \text{op}(x_1, y_1) \) and \( u_{(i)} = \text{op}(x_2, y_2) \) and \( x_1 \neq x_2 \) or \( y_1 \neq y_2 \)
- Find a unique value number for each equivalence class
- Replace variables consistently by value number for each equivalence class
- Insert, if necessary φ−functions (also possible during iteration)
Optimistic SSA Construction

\[ a_1 = 1 \]
\[ b_1 = 2 \]

\[ a_6 = b_6 - d_{2,3} \]
\[ b_6 = a_6 - d_{2,3} \]

\[ a_1 = 1 \]
\[ b_1 = 2 \]

\[ a_6 = b_6 - d_{2,3} \]
\[ b_6 = a_6 - d_{2,3} \]

\[ a_1 = 1 \]
\[ b_1 = 2 \]

\[ a_6 = b_6 - d_{2,3} \]
\[ b_6 = a_6 - d_{2,3} \]
Optimistic SSA Construction

In examples pessimistic SSA better.

Caution: can not be generalized!
Optimal (good) Policy

- Generate pessimistic SSA
  - Program size reduced
  - Definition-Use-Information computed
  - No immature $\phi$-functions
- Set all value(-number)s congruent
- Compute optimistic SSA
- Iteration (pessimistic-optimistic-pessimistic- ...)
  - until fix point possible
  - in practice only pessimistic SSA or pessimistic SSA with only one subsequent optimistic SSA computation (no iteration)

Minimal SSA-Form

- Insight:
  - $\phi$-functions guarantee that for each use of a variable there is exact one definition
  - Solution of the Reaching-Definitions-Problems
  - “Variable” means program- or auxiliary variable
  - Problems with array elements and indirectly addressed variables retain (discussed and solved later)
- **minimum SSA-form:** set $\phi$-function $a_0 := \phi(a_1, a_2, ...)$ in block $B$ iff value live in $B$.
  - Use data flow analysis $\text{liveIn}(B)$ and check $a \in \text{liveIn}(B)$.
  - Better: generate value numbers only on demand (Integration in construction algorithms)

SSA – Construction from AST

- Left-Right Traversal (1. Round):
  - compute for each expression basic block number
  - compute precedence relation on basic blocks
  - generate triples to the BBs
- Right-Left Traversal (2. Round):
  - compute for each live (beginning with the results of a procedure) expression value numbers (contains $\phi'$) using the data structures known from value numbering
- Left-Right Traversal (3. Round):
  - Mature $\phi'$-functions
  - generate SSA for non empty blocks
- Further eliminations on SSA graph

SSA from AST

- Rounds 1+2 on one left-right tree traversal if lifelines ignored,
  - Construct BBs
  - Construct SSA code for the basic blocks (value number graphs)
  - Construct control flow between BBs
- For each statement type (AST node type) different set of actions
  - Assignment to local variables and expressions: like local value numbering in a left-to-right traversal
  - Procedure calls like any other operation expressions
  - While, If, Exception, … on the fly introduce new BBs and control flow
SSA from AST

- *while* AST and BB graph

```
AST
  |   Parent
  |   Succ
  |   \_

  Expr
  \_/   \_
  Succ Start   Body Start

  Body End
```

Deconstruction of SSA

- Serialize the SSA graph
- Replace data dependency edges by variables
- Remove \(\phi\)-functions:
  - Define a new variable
  - Copy from value in predecessor basic blocks (requires possibly new blocks on some edges)
  - Perform copy propagation to avoid unnecessary copy operations
- Allocate registers for variables
  - Fixed number of registers
  - More variables than registers
  - Idea: assign variables with non overlapping lifetimes to the same register

Example Program and BB Graph

```
(1) a=1;
(2) b=2;
(3) c=a+b;
(4) if (d=c-a)
(5)   while (d=b*d)
(6)     d=a+b;
(7)     e=e+1;
(8) b=a+b;
(9) if (e=c-a) break;
(10) a=b*d;
(11) b=a-d;
```

```
(1) a=1
(2) b=2
(3) c=a+b
(4) d=c-a
(5) d=b*d
(6) d=a+b
(7) e=e+1
(8) b=a+b
(9) e=c-a
(10) a=b*d
(11) b=a-d
```
Remove $\phi$-functions

1. $b_2 = 2$
   $e_2 = e_1$

2. $c_1 = 1 + b_2$
   $d_2 = c_1$
   $e_2 = e_1$

3. $d_1 = b_2 * d_2$
   $e_1 = e_2$

4. $d_3 = d_1$
   $e_3 = e_2$

5. $a_1 = c_1 * d_3$
   $b_6 = a_3 - d_5$

6. $b_2 = 2$
    $e_1 = e_2$

7. $c_1 = 1 + b_2$
   $d_2 = c_1$
   $e_2 = e_1$

8. $d_3 = d_1$
   $e_3 = e_2$

9. $a_1 = c_1 * d_3$
   $b_6 = a_3 - d_5$

Copy Propagation

1. $b_2 = 2$
   $e_2 = e_1$

2. $c_1 = 1 + b_2$
   $d_2 = c_1$
   $e_2 = e_1$

3. $d_1 = b_2 * d_2$
   $e_1 = e_2$

4. $d_3 = d_1$
   $e_3 = e_2$

5. $a_1 = c_1 * d_3$
   $b_6 = a_3 - d_5$

6. $b_2 = 2$
   $e_2 = e_1$

7. $c_1 = 1 + b_2$
   $d_2 = c_1$
   $e_2 = e_1$

8. $d_3 = d_1$
   $e_3 = e_2$

9. $a_1 = c_1 * d_3$
   $b_6 = a_3 - d_5$
Copy Propagation

1. \( b_2 = 2 \)
   \( e_1 = e_1 \)
2. \( d_2 = 1 + b_2 \)
   \( d_4 = b_2 \)
3. \( a_5 = b_2 \times d_2 \)
4. \( e_4 = e_1 + 1 \)
   \( d_2 = b_2 \)
   \( e_1 = e_4 \)
5. \( a_4 = c_1 \times d_5 \)
6. \( b_6 = a_4 - d_5 \)
7. \( b_2 = 2 \)
   \( e_3 = e_1 \)

Register Allocation

- Assign variables to registers
- Reuse the registers if the variables are not alive at the same time
  - Interference graph encodes colliding life times
  - Build upon define-use-relation of variables
- Optimal register allocation is an \( NP \)-hard problem
- Suboptimal solution uses graph coloring

Live Variables

- Which variables are used in assignment \( A \) or later?
- Data flow problem
  - Backward, May
  - General equations
    \[ \text{Live}_{\text{out}}(A) = \bigcup_{i \in \text{succ}(i)} \text{Live}_{\text{in}}(A) \]
    \[ \text{Live}_{\text{out}}(A) = \text{Live}_{\text{in}}(A) - \text{kill}(A) \cup \text{gen}(A) \]
- With:
  \( \text{kill}(A) = D(A) \) (defined in \( A \))
  \( \text{gen}(A) = U(A) \) (used in \( A \))

Live Variables

- \( b_1 = 2 \)
- \( e_1 = e_1 \)
- \( d_4 = 1 + b_2 \)
- \( d_6 = b_2 \)
- \( a_5 = b_2 \times d_4 \)
- \( e_4 = e_1 + 1 \)
- \( d_2 = b_2 \)
- \( e_1 = e_4 \)
- \( b_2 = 2 \)
- \( e_3 = e_1 \)
- \( c_1 = c_1 \times d_5 \)
- \( a_4 = c_1 \times d_5 \)
- \( b_6 = a_4 - d_5 \)
Live Variables

\[ d_5 = b_2 \]

\[ e_4 = e_3 + 1 \]

\[ d_2 = b_2 \]

\[ e_3 = e_1 \]

\[ b_2 = 2 \]

\[ a_3 = c_1 \cdot d_5 \]

\[ b_6 = a_3 - d_5 \]

\[ d_2 \cdot d_5 \cdot b_2 \cdot e_3 \]

\[ b_2 \cdot d_2 \cdot e_3 \]

\[ b_2 \cdot e_3 \]

\[ d_2 \cdot d_5 \]

\[ b_2 \cdot e_3 \]

\[ d_2 \cdot d_5 \]

\[ b_2 \cdot d_2 \cdot e_3 \]

\[ b_2 \cdot e_3 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]

\[ e_1 \]

\[ b_2, e_1 \]

\[ b_2, d_2, e_3 \]

\[ b_2, d_2, e_3, d_5 \]

\[ b_2, d_2, e_3 \]

\[ b_2, e_3 \]

\[ b_2 \]

\[ d_2 \]
Memory SSA

- By now we can only handle simple variables
- Extension:
  - Node: memory changing operations
  - Edges:
    - Data- and control flow.
    - Anti- / out dependencies between memory changing operations
- Functional modeling of memory changing operations

Why \textit{Load} Defines Memory?

- Anti-depending memory operations:
  - Read an address essentially before
  - Redefine the value

Memory SSA

- To capture only essential dependencies distinguish disjoint memory fragments
  - In general not decidable
  - Approximated by analyses
  - Initial distinctions e.g.
    - Heap vs. Stack
    - Different arrays on the stack
    - Heap partitions for different object types
- Distinction often only locally possible
  - Union necessary
  - \textit{Sync} operation unifies disjoint memory fragments
  - Like \( \phi \)- functions but \textit{sync} is strict

Properties of Memory SSA

- P1-P8: as before
- P9: \textbf{New!} Lifetime of memory states do not overlap if they define different values of the same variable
  - Otherwise we would need to keep two versions of the memory alive
  - Memory does not fit into a register (usually)
  - Would be make the programs non-implementable
- Note: if we were only to analyze the program analyses, P9 could be ignored
Assignment I

Compute the SSA graph for:

1: c := 0;
2: while a/=0 do
3:    c := c+b;
4:    a := a-1;
5:   od;
6: stdout := c

- Define the program with immature φ functions and the final version.
- Draw the graph (replacing variables by edges). Note that stdout := c is actually a function call.

Assignment II

- Leave the SSA graph for the program from Assignment I
- Remove φ functions
- Perform copy propagation
- Compare the produced program to the original one
- Perform register allocation for 3 registers
  - Do it mechanically using live analysis and graph coloring
  - Why is it immediately clear that 3 registers are sufficient but 2 are not.