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 \( s \) changes observable behavior of the program
  - Computation of essential dependencies is not decidable
- Compact representation
  - No (few) redundant expression representation

Static Single Assignment - SSA

- Goal:
  - Increase efficiency of inter/intra-procedural analyses and optimizations
  - Speed up dataflow analysis
  - Represent def-use relations explicitly
- Idea:
  - Represent program as a directed graph of operations \( r \)
  - Represent triples as a (single) assignment \( t := t' \rightarrow t'' \) with \( t, t', t'' \) a variable/register/edge connecting operations
- SSA-Property: there is only one single (static) position in a program/procedure defining \( i \)
- Does not mean \( i \) computed only once (due to iterations the program point is in general executed more than once, possibly each time with different values)
  - But, there is no doubt which static variable definition is used in arguments of operations

Avoid redundant computations

- Assign each (partial) expression a unique number.
  - Good optimization in itself 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**, if they compute the same value - Not decidable
- Two expressions are **syntactically equivalent**, if the operator is the same and the operands are the same or syntactically equivalent
- Generalization towards semantic equivalence using algebraic identities, e.g., $a^2 = 2^1 a$
- In practice, provable equivalence (conservative approximation): two expressions are **congruent**, if they are syntactically equivalent or algebraically identical

Value Numbering

- **Type of value numbers:**
  - int for integer constants; bool for Boolean constants etc.
  - Use ids (labels): $t_1, \ldots, t_i$ otherwise
- **Data structure:**
  - Mapping (hash table) gets value number $vn$ of operations defining the values, i.e., for tuples $t := t' \cdot t''$, a lookup of $t \in (t') \cup (t'')$ gets the value number of $t$.
  - Mapping of operation labels $\tau$ to tuples $t \cdot t'$ (implicit),
  - $\tau$ is an operator symbol,
  - $vn(t') \cup vn(t'')$ are value numbers of tuples labeled $t', t''$, i.e., $t, t'$ have hash table entries or are constants.
- **For a first try:**
  - Computation basic block local
  - One hash table per basic block.

Example

<table>
<thead>
<tr>
<th>Original</th>
<th>Result</th>
</tr>
</thead>
</table>
| $t_1$: ST, $>$a$< 2  
$t_2$: LD, $<$a$>  
$t_3$: ADD, $t_4 1  
$t_5$: ST, $>$b$<  
$t_6$: MUL, $t_7  
$t_8$: LD, $<$c$>  
$t_9$: MUL, $t_2  
$t_{10}$: ADD, $t_{11} 1  
$t_{12}$: LD, $<$d$>  
$t_{13}$: ADD, $t_{12} 1  
$t_{14}$: ST, $>$e$< $t_{11}$ |
| $v_1$: ST, $>$a$< 2  
$v_2$: LD, $<$a$>  
$v_3$: MUL, $2  
$v_4$: ADD, $v_1 1  
$v_5$: ST, $>$b$<  
$v_6$: ST, $>$e$< $v_7$ |

Value Number Graph of Basic Block

Result

| $v_1$: ST, $>$a$< 2  
$v_2$: LD, $<$a$>  
$v_3$: MUL, $2  
$v_4$: ADD, $v_1 1  
$v_5$: ST, $>$b$<  
$v_6$: ST, $>$a$<  
$v_7$: ADD, $t_4  
$v_8$: ST, $>$e$< $v_7$ |
Value Numbering
with Global Variables without Alias Problem

- Case (a') as before case (a) for local variables
  \( \text{case (a')} \rightarrow ST > \text{global} \phi \)
  \( \text{if } \text{vn}(i) = \text{void} \text{ then } \text{vn}(\text{LD}) = \text{global} \phi \) \( \text{generate: } \text{vn}(i) = \text{new value number} \)

- Procedures:
  - Case (d) as before, but if global (potentially) redefined in \( \text{proc} \), set value number for tuple \( \text{ST} > \text{global} \phi \) and transitive depending tuples to \text{void}
  - Easy implementation: set all value numbers to \text{void}
  - Optimization for non-recursive leaf procedures:
    - New case (d): analyze procedure as if it was inlined
    - Not trivial if \( \text{proc} \) has more than one basic block

Remarks

- Initially all value numbers are set to \text{void}. By knowing the values of predecessor basic blocks, this can be relaxed
- Each new entry in the hash table generates a new value number.
- A store operation \( ST > a' < i' \) sets \text{void} all \text{vn}(LD < a'>) and \text{vn}(ST < a' > i') if it is not clear, whether \( a = a' \) or \( a = a' \) (alias problem)
- Special case: arrays with index expressions

Value number graph of a basic block

- No (provable) unnecessary dependencies
- No (provable) redundant computation

\( \phi \)-Functions

- Solution:
  - Each assignment to a variable \( a \) defines a new version \( a_i \)
  - This version is actually the value number of the assigned expression
  - At meets in the control flow, we add a pseudo expression selecting a value number for tuple \( \text{ST} > \text{global} \phi \)

E.g.

\[ \text{if } i \neq 0 \text{ then } i : = i + 1 \] \( \text{else } i : = i + 2 \text{ end; } \text{while } i : = 1; \text{... end; } x : = i \]

- Functions always occur at the beginning of a block
- All are evaluated simultaneously for a block
- Guarantee that there is exactly one definition/assignment for each use of a variable

Assignment \( i : = \phi(i_1,...,i_k) \) in a basic block indicates that the block has \( i \) direct predecessors in the control flow

General Value Numbering

- \( i \) is an address with unknown value (no compile time constant address, no name)
- Computed in an operation with value number \( \text{vn}(i) \)

- Case (a') \( ST > i \)
  \( \text{vn}(i) = \text{void} \text{ then } \text{vn}(\text{LD}) = \text{void} \)
  \( \text{generate: } \text{vn}(i) = \text{new value number} \)

- Case (b) \( i \text{ is } \text{LD} < a' > \) \( i' \) \( \text{vn}(i) ) = \text{new value number} \)
  \( \text{generate: } \text{vn}(i) = \text{new value number} \)

Value number graph \( \rightarrow \) SSA

- SSA-Property: there is only one position in a program/procedure defining \( i \)
- 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 than one position?
  - E.g.
    \[ \text{if } i \neq 0 \text{ then } i : = i + 1 \text{ else } i : = i + 2 \text{ end; } x : = i \]
    \[ \text{while } i : = 1; \text{... end; } x : = i \]
- \( \phi \)-functions
  - \( \text{always occur at the beginning of a block} \)
  - \( \text{are all evaluated simultaneously for a block} \)
  - \( \text{guarantee that there is exactly one definition/assignment for each use of a variable} \)

Assignment \( i : = \phi(i_1,...,i_k) \) in a basic block indicates that the block has \( i \) direct predecessors in the control flow

Compact representation of dependencies

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

```
\text{i} := \ldots \quad \text{i} := \ldots \quad \text{i} := \ldots \quad \text{i} := \ldots
\text{i} := \ldots \quad \text{i} := \ldots \quad \text{i} := \ldots \quad \text{i} := \ldots
\text{...} \quad \text{...} \quad \text{...} \quad \text{...}
\text{i}_1 := \text{i}_1 \ldots \quad \text{i}_2 := \text{i}_2 \ldots \quad \text{i}_3 := \text{i}_3 \ldots \quad \text{i}_4 := \text{i}_4 \ldots
\text{...} \quad \text{...} \quad \text{...} \quad \text{...}
\text{i} := \phi(i_1,...,i_k)
```


Example Program and Basic Block Graph

(1) a=1;
(2) b=2;
while true {
(3) c=a+b;
(4) d=c-a;
if (d=b*d) {
(5) 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;
}

Basic Block and SSA Graph

SSA-Graph before and after Constant and Copy Propagation

SSA-Graph before and after using Algebraic Identities

Implementation SSA graph

SSA properties

P1: Typed in- and 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 are a DAG.
Note: correspondence to value number graphs and expression trees
P3: Input of \( \phi \)-operations have the 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 an operation \( x \) in a BB \( B_x \) defines a operand of operation \( y \) in a BB \( B_y \), 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 \( x \) be BBs each with a definition of \( u \) that may reach a use of \( u \) 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 \( u \).
Property P8 revisited

Let X, Y be BBs each with a definition of a that may reach 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 α-operation for a.

\[ X:a=\ldots, Y:a=\ldots, Z:a_{1}=\ldots, Z:a_{2}=\ldots \]

Example

(Iterated) Dominance Frontiers

- **Dominance Frontier \( DF(a) \)**
  - Set of nodes just not strictly dominated by a anymore
  - \( DF(a) = \{ m \mid \exists b < pre(m) : a \geq b \} \).

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

- **Iterated Dominance Frontier \( DF^i(M) \)**
  - Minimum fix point of:
  - \( DF_{0} = DF(M) \)
  - \( DF_{i+1} = DF \left( \bigcup_{k \leq i} DF_{k} \right) \).

Discussion P8

- Defines, where to position α-functions
- Let X, Y be the only BBs containing a definition of a. Both may 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:
  - \( a_{1} := \phi(a_{1}, a_{2}) \) is assigned to Z'
  - Z' ≥ Z otherwise \( \forall Y \) are not the only BBs (a not initialized)
  - Z' dominates Z, actually Z' is the first common successor of X and Y dominating Z
  - Z' = DF(X,Y), i.e. Z' is in the dominance frontier of X and Y
- Since \( a_{2} := \phi(a_{1}, a_{2}) \) itself is a definition of a, the dominance frontiers \( DF(X,Y,Z) \) must contain α functions.
- Fixed point iteration required.
- Observation: α-functions in the iterated dominance frontiers \( DF^i(\ldots) \) of the BB defining a variable

Discussion P8

- **Dominance**: \( X \geq Y \)
  - On each path from the starting node S in the basic block graph to Y, X before Y.
  - ≥ is reflexive: \( X \geq X \).
- **Strict Dominance**: \( X > Y \)
  - \( X > Y \iff X \geq Y \land X \neq Y \)
- **Direct Dominance**: \( ddom(X) \)
  - \( X = ddom(Y) \Leftrightarrow X > Y \land \forall Z: X > Z > Y \).
- **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.

Example

Proof (idea) of this observation

1. \( a_{1} := \phi(a_{1}, a_{2}) \) 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. \( a_{2} := \phi(a_{1}, a_{2}) \) should not (but may) be inserted later in other dominators of Z dominated by \( Z' \) since on the path \( Z' \rightarrow Z \) there is no change (new definition) of a.
4. Iteration \( DF^i(\ldots) \) is required, as there is now a further definitions of a in a block \( Z' \) and \( DF(X,Y,Z) \in DF(X,Y,Z') \).
Example Property E8

Outline

- Introduction to SSA
  - Motivation
  - Value Numbering
  - Definition, Observations

- Construction, Destruction
  - Theoretical, Pessimistic, Optimistic Construction
  - Destruction
  - Memory SSA
  - Interprocedural analysis based on Memory SSA: example P2SSA
  - How to capture analysis results

- Optimizations

Construction Theory

- Program of size \( n \) may contain \( \Omega(n) \) variables
- In the worst case there are \( \Omega(n) \) \( \phi \)-function (for the variables) in \( \Omega(n) \) BBs, hence worst case complexity is \( \Omega(n^2) \)
- Previous discussion gives a straightforward 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

Extended Initialization

(1) Initialization for current block \( Z \):
   - (A) always: \( \text{vn}\text{(constant)} = \text{constant}; \)
   - (B) if \( Z \) = start block: \( \text{vn}(i) = \text{void} \) for all tuples \( i \);
   - (C) else: let \( \text{Pred}(Z) = \{X_1, X_2, \ldots\} \) be the predecessors of \( Z \) in basic block graph for all variables \( j \) used in current block \( Z \):
     - if \( \text{vn}(j) = \text{vn}(j)_i \) \( \text{vn}(j)_i \) = new value number
     - \( \text{vn}(j) = \phi(\text{vn}(j)_i, \text{vn}(j)_j) \)
     - if \( \text{vn}(j) = \text{vn}(j)_i \) \( \text{vn}(j)_i \) = new value number
     - \( \text{vn}(j) = \phi(\text{vn}(j)_i, \text{vn}(j)_j) \)

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

Extended Value Numbering

Remainder Value Numbering

(1) Initially: value number \( \text{vn}\text{(constant)} = \text{constant}; \)
(2) for all tuple \( i \) in program order:
case
   (a) \( i = \text{ST} \text{local} \):
      - \( \text{vn}(i) = \text{vn}(i) \) if \( (i) = \text{old} \)
      - \( \text{vn}(i) = \text{vn}(i) \) if \( (i) = \text{new value number} \)
      - \( \text{vn}(i) = \text{vn}(i) \) if \( (i) \) = new value number
   (b) \( i = \text{LD} \text{local} \):
      - \( \text{vn}(i) = \text{vn}(i) \) if \( (i) = \text{old} \)
      - \( \text{vn}(i) = \text{vn}(i) \) if \( (i) = \text{new value number} \)
      - \( \text{vn}(i) = \text{vn}(i) \) if \( (i) \) = new value number
   (c) \( i = \text{call proc} \):
      - \( \text{vn}(i) = \text{vn}(i) \) if \( (i) = \text{old} \)
      - \( \text{vn}(i) = \text{vn}(i) \) if \( (i) = \text{new value number} \)
   (d) \( i = \text{call proc} \):
      - analog \( (i) \) with \( (i) \) = call proc
Extended Initialization

(1) Initialization for current block \( Z \):

(A) always: \( vn(c) = \text{constant} \);

(B) if \( Z = \text{start block} \): \( vn(i) = \text{void} \) for all tuples \( i \);

(C) else:

let \( \text{Pred}(X, Y, \ldots) \) be the predecessors of \( Z \) in basic block graph

for all variables \( i \) used in current block \( Z \):

- if \( \text{vn}(i) = \text{undefined} \)
  - recursively, initialize block \( X \) \( \text{vn}(i) \) with (1)

- if \( \text{vn}(i) = \text{new value number} \)
  - generate: \( \text{vn}(i) := \phi(\text{vn}(i), \text{vn}(j)) \)

- if \( \text{vn}(i) = \text{new special value number} \)
  - with (1)

(2) for all tuple \( i \) in program order:

-- as before

Extended Value Numbering

Example I: Mature \( \phi^i \)-Functions

Example II: Mature \( \phi^i \)-Functions

Eliminate/Mature \( \phi^i \)-Functions

- After value numbering is finished for each block \( X \):
  - replace special value numbers in \( \phi^i \)-functions of \( X \) by last valid real value numbers in \( \text{pre}(X) \)
  - replace \( \phi^i \)-functions by mature \( \phi^i \)-functions using real value numbers
  - delete: \( \text{vn}(i) := \phi(\text{vn}(i), \text{vn}(j)) \)
    - if not changed in previously unvisited blocks, no \( \phi \) function required
    - replace then use of \( \text{vn} (i) \) by \( \text{vn}(i) \)
  - Insight:
    - deletion could prove some other \( \phi \)-functions unnecessary
    - iterative deletion till fix point
Example III: Mature $\phi^+$-Functions

Example Program and BB Graph

SSA Construction Block 1

SSA Construction Block 2

SSA Construction Block 3 - Initialization
SSA Construction Block 3

SSA Construction Block 4 - Initialization

SSA Construction Block 4

SSA Construction Block 5 - Initialization

SSA Construction Block 5

SSA Construction Block 6 - Initialization

SSA Construction Block 6
Final Simplifications

\[ a = 1 \]
\[ b = 2 \]

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 \( \phi \)-functions to mature
- Disadvantage:
  - Detects sometimes congruence that are not 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 \( \phi \)-functions

Optimistic SSA Construction

Construction Algorithm

- Generate BB graph and perform Definition-Use-Analysis (data flow analysis) for all variables. Notations:
  - \( v_i \) = \( v \) in statement \( i \) defined
  - \( u_j \) = \( v \) in statement \( j \) used with may reaching definitions in statements \( x, y, z, \ldots \)
  - Statements can be abstracted to blocks
- Set \( v_i = u_j \) for all \( v_i, u_j \) in the program
- Iterate until a fixed point over:
  - Set \( v_i = u_j \) for:
    - \( v_i = c \) and \( u_j \neq c \)
    - \( v_i = \text{op1}(\ldots) \) and \( u_j \neq \text{op1}(\ldots) \)
    - \( v_i = \text{op1}(x_1, y_1) \) and \( u_j = \text{op1}(x_2, y_2) \) and \( x_1 \equiv x_2 \) or \( y_1 \equiv y_2 \)
  - Find a unique value number for each equivalence class
  - Replace variables consistently by value number for each equivalence class
  - Insert, if necessary, \( \phi \)-functions (also possible during iteration)
Optimistic SSA Construction

\[ a_1 = 1 \]
\[ b_1 = 2 \]
\[ b_2 = \phi(b_1, b_5) \]
\[ c(2) = a_1 + b_2 \]
\[ d(2) = c(2) - a_1 \]
\[ d_1 = \phi(d(4), d(2)) \]
\[ d(3) = b_2 \cdot d_1 \]
\[ b_5 = a_1 + b_2 \]
\[ e(5) = c(2) - a_1 \]
\[ a(6) = b_5 \cdot d(2, 3) \]
\[ b(6) = a(6) - d(2, 3) \]
\[ d(4) = a_1 + b_2 \]
\[ e(4) = e(0, 4, 5) + 1 \]
Optimistic vs. Pessimistic SSA

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
  - "Variable" means program- or auxiliary variable Solution of the (may) Reaching-Definitions-Problem
  - Problems with array elements and indirectly addressed variables retain (discussed and solved later)
- Minimal SSA-form: set $\phi$-function $a_0 := \phi(a_1, a_2, ...)$ in block $B$ iff value $a_0$ is live in $B$.
  - Use data flow analysis liveIn($B$) and check $a \in$ 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 syntactic expression its basic block number
  - compute precedence relation on basic blocks
  - generate expression triples into the BBs
- Right-Left Traversal (2. Round):
  - compute, for each live (beginning with the results of a procedure) expression, the 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 liveliness is 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) there is different set of actions when visiting the nodes of that type including:
  - 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

0

1

2

3

4

5

6

7

8

9

10

11

12

13
SSA from AST

- while actions
  - Finalize current block $B(\text{Parent})$
  - Create a new current block $B(\text{Expr})$
  - Add control flow $B(\text{Parent})$ to $B(\text{Expr})$
  - Recursively, generate code for $\text{Expr}$ computing value numbers locally
  - Finalize current block $B(\text{Expr})$
  - Create a new current block $B(\text{Parent})$
  - Add control flow $B(\text{Expr})$ to $B(\text{Parent})$
  - Recursively, generate code for $\text{Body}$
  - After return current block is $B(\text{Parent}),$ finalize it
  - Add control flow $B(\text{Parent})$ to $B(\text{Body})$

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

```plaintext
(1) a=1;
(2) b=2;

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

Introduce Variables for Edges

Remove $\phi$-functions
Remove $\phi$-functions

Copy Propagation

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

Why Load Defines Memory?
- Anti-depending memory operations: Read an address essentially before redefine the value

Memory state
- $M \rightarrow M'$
- $a \rightarrow v$
- $a \rightarrow v'$
- $a \rightarrow M'$
- $a \rightarrow v$
- $a \rightarrow M$
- $M \rightarrow M'$
- $M \rightarrow v$
- $M \rightarrow v'$
- $M \rightarrow M$
- $M \rightarrow v$

Why 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
    - Heap partitions for different object types
  - Distinction often only locally possible
  - Union necessary
  - Sync operation unifies disjoint memory fragments
  - Like $\phi$-functions but sync is strict

Properties of Memory SSA

P1-P8: as before

P9: New! Lifetime of memory states do not overlap if they define different values of the same memory slot

- 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 only have to analyze the program, P9 could be ignored

Reduced SSA Representations

- Assume goal is analysis not compilation as in many software tech.
  applications in IDEs
  - Rename ids
  - Go back in debugging
  - Move code
  - ...
- Not the whole program is directly relevant for an analysis
  - Certain data types are uninteresting, e.g., Int, Bool in call graph construction
  - Consequently, operation nodes consuming/defining values of these types and edges connecting them can be removed
- More compact program representation
  - Faster in analysis
  - Still SSA properties hold
- Example Points-To SSA capturing only reference information necessary for Points-To analysis (ignoring basic types and operations)

Example Points-to-SSA

```
public class list {
  Object value = null;
}

public List (object v) {
  value = v;
}

public void append (object v) {
  if (next == null) {
    next = new list(v);
    return;
  }
  next.append(v);
}

public void print (list l) {
  if (l == null)
    return;
  if (l.value != null)
    print(l.value);
  System.out.println(l.value);
}
```

Cliffhanger from the first day

- Inter-Procedural analysis
- Call graph construction (fast)
- Call graph construction (precise)
- Call graph construction (fast and precise)

Recall: Call graph construction

- Points-to Analysis (P2A) computes reference information:
  - Which abstract objects may a variable refer to.
  - Which are possible abstract target objects of a call.
- In general: for any expression in a program, which are the abstract objects that are possibly bound to it in an execution of that program.
- Call graph construction is a simple client analysis of P2A
Recall: Precise call graph construction

- Construction of a Points-to Graphs (P2G):
  - Node for objects and variables,
  - Edges for assignments and calls
  - Propagate objects along edges, i.e., data-flow analysis on that graph

- The baseline P2G approach is locally and globally flow-insensitive; it focuses on data-dependencies over variables and ignores the control flow
  - An analysis is flow-sensitive if it takes into account the order in which statements in a program are executed
  - Using global def-use analysis does not scale

- The baseline P2G approach is context-insensitive
  - An analysis is context-sensitive if distinguishes different contexts in which methods are called
  - Object-sensitivity distinguish methods by the abstract objects they are called on – can be understood as copies of the method’s graph
  - Scales but is quite slow

Fast and Precise P2A

1. Data values
   - Allocation site abstract from objects
   - Abstract heap memory: $O_{a} = O_{f}$ (set of fields) heap size: $I_{nt}$

2. Data-flow graph: Points-to-SSA graph for each method (constructor)
   - Nodes with ports represent operations relevant for P2A, ports correspond to operands, special $\phi$-nodes for merge points in control flow
   - Edges represent intra-procedural control- and data-flow

3. Transfer function:
   - update the heap according to the abstract semantics of the special $\phi$-nodes
     - $O$ on $0$ values and $\max$ on $I_{nt}$ values, resp.

4. Initialization: $\emptyset$ for $0$ ports and $0$ for $I_{nt}$ ports, resp.

5. Simulated execution

Recall: Simulated Execution

- Interleaving of process method and update call nodes’ transfer function
- Processes a method:
  - Starts with main
  - propagates data values analog the edges in P2-SSA graph
  - updates the heap and the data values in the nodes according to their transfer functions
  - If node type is a call then …
- Call nodes’ transfer function; if input changes:
  - Interrupts the processing of a caller method
  - Propagates arguments $P_{\phi}(v_{1}, ..., v_{n})$ to the all callees $P_{\phi}(a)$
  - Processes the callees (one by one) completely (iterate in case of recursive calls)
  - Propagates back and merges the results $P_{\phi}(r)$ of the callees
  - Updates heap size $size(x')$
  - Continue processing the caller method …

Example transfer functions

if input changes, update as below else skip:

- Alloc
- Load
- Store

Example: Global flow-sensitivity

Flow-sensitivity

- The Points-to-SSA approach has two features that contribute to flow-sensitivity:
  1. Locally flow-sensitive: We have SSA edges imposing the correct ordering among all operations (calls and field accesses) within a method.
  2. Restricted globally flow-sensitive: We have the simulated execution technique that follows the inter-procedural control-flow from one method to another.

- Effect:
  - An access $a_{1}.x$ will never be affected by another $a_{2}.x$ that we process after $a_{1}.x$.
  - Each return only contains contributions from previously processed calls, i.e. reduced mixing of values returned by calls targeting the same method.
  - But: information is accumulated in method arguments to guarantee scalability.
Context-sensitivity

- Context-insensitive analysis
  - Actual arguments of calls targeting the same method were mixed in the formal arguments.
  - Advantage: scalability ensures termination for recursive call sequences and reaches a fix point quickly.
  - Disadvantage: accuracy inaccurate due to mixed return values.

- Context-sensitive analysis
  - Divide calls targeting a given method into a finite number of different categories.
  - Analyze them separately as if they defined different copies of that method.
  - We can define contexts as an abstraction of call stack situations:
    \[ \text{context: } [m, \text{call id}, a, v_1, \ldots, v_n] \to [c_1, \ldots, c_p] \]
  - Often it is just one context per call stack situation.

Examples: Context abstractions

Object-sensitivity
- A context is given by a pair \((m, o)\) where \(o \in O\) is a unique abstract object in the points-to value analyzed for the call target variable this.
- Linear (in program size) many contexts, in practice slightly more precise than This-sensitivity.

This-sensitivity
- A context is given by a pair \((m, \text{this})\) where \(\text{this} \in 2^O\) is the unique points-to value (set of abstract objects) analyzed for the call target variable this.
- Exponentially many contexts (in practice ok), in practice an order of magnitude faster than Object-sensitivity.

Examples: Precision

In favor of Object-sensitivity
- Method definitions:
  - \(m() \{ \text{field = this; } \}
  - \(V n() \{ \text{return field; } \}
- Call 1:
  - \(a_1 = \{o_1, o_2\} \)
  - \(r_1 = a_1.m()\)
- Call 2:
  - \(a_2 = \{o_1\} \)
  - \(r_2 = a_2.m()\)

Object-sensitivity: \(\text{Pt}(r_2) = \{o_1\}\).
This-sensitivity: \(\text{Pt}(r_2) = \{o_1, o_2\}\).

In favor of This-sensitivity
- Method definition:
  - \(V m(V v) \{ \text{return v; } \}
- Call 1:
  - \(a_1 = \{o_1\}, v_1 = \{o_3\} \)
  - \(r_1 = a_1.m(v_1)\)
- Call 2:
  - \(a_2 = \{o_1, o_2\}, v_2 = \{o_4\} \)
  - \(r_2 = a_2.m(v_2)\)

Object-sensitivity: \(\text{Pt}(r_2) = \{o_3, o_4\}\).
This-sensitivity: \(\text{Pt}(r_2) = \{o_4\}\).

Results

- Fast and accurate P2A
- Points-to SSA -> locally flow sensitive PTA
- Simulated execution -> globally flow-sensitive PTA, fast
- Context-insensitive in the first presented baseline version
- More accurate (in theory and practice ca. 20%) and 2x as fast compared to classic flow- and context-insensitive P2A
- Fast: < 1 min on javac with > 300 classes.

Context-sensitive variant this sensitivity even more accurate
- As fast and up to 3x as precise compared to classic flow- and context-sensitive P2A
- As precise and 10x as fast compared to the best known context-sensitive variant (object sensitivity) of classic P2A
- Shows in clients analyses like synchronization removal and static garbage collection (escape and side effect analysis)

Assignment I

Compute the SSA graph for:

1. \(c := 0;\)
2. \(\text{while } a<>0 \text{ do} \)
3. \(\quad c := c+b;\)
4. \(\quad a := a-1;\)
5. \(\text{od};\)
6. \(\text{stdout} := c;\)

Define the program with immature \(\phi\) functions and the final version.

Draw the graph (replacing variables by edges). Note that \(\text{stdout} := c\) 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.

Context-sensitive analysis

- Distinguish different call context of a method
- In general: Distinguish different execution path to a program point
  - Exponentially (in program size) many path in a sequential program
  - Exponentially many analysis values
- (Let away the problem of analysis) How to capture the results efficiently?

Context-insensitive Data flow analysis

- \( x+y = \{2, 3, 4\} \)
- \( a+b = \{4, 5, 6\} \)
- \( 2(x+y) \geq a+b \) ?

Context-sensitive Data flow analysis

- \( x=1 \)
- \( x=2 \)
- \( y=x \)
- \( b=3 \)
- \( y=2 \)
- \( b=4 \)
- \( a=x \)
- \( a=y \)
- \( 2(x+y) = (a+b) \)

Representation of context in decision tables

Too large! Too much redundancy!

Concept of \( \chi \) terms

\[ a + b \]

\[ \begin{align*}
\chi_{1} & = 1 \\
\chi_{2} & = 2 \\
\chi_{3} & = 3 \\
\chi_{4} & = 4 \\
\chi_{5} & = 5 \\
\chi_{6} & = 6
\end{align*} \]
Advantages of $\chi$-Terms

- Compact representation of context sensitive information
- Delayed widening (abstract interpretation) of terms until no more memory; no unnecessary loss of information
- $\chi$-Terms are implementation of decision diagrams.
- Adaptation of OBDD implementation techniques.
- Symbolic computation with $\chi$-terms (simplification of terms).
- Transition of the idea of SSA-$\phi$-function to data-flow values.
- Especially interesting for address values to distinguish memory partitions as long as possible.

Data-Flow Analyses (DFA) on SSA

- Each SSA node type $n$ has a concrete semantics $\sigma(n)$.
- On execution of $n$, map inputs $i$ to outputs $o$ and $\gamma(n(i))$.
- Inputs $i$ and outputs $o$ are records of typed values (P1).
- $[\sigma] := \text{type}(\gamma(i)) \times \ldots \times \text{type}(\gamma(o)) \rightarrow \text{type}(\gamma(i)) \times \ldots \times \text{type}(\gamma(o))$.
- Each static data-flow analysis abstracts from concrete semantics and values.
- Abstract semantics $\Gamma_n$ is called transfer function of node type $n$.
- On analysis of $n$, map abstract analysis inputs $\alpha(i)$ to abstract analysis outputs $\alpha(o)$.
- $\alpha := \text{type}(\alpha(i)) \rightarrow \text{type}(\alpha(o))$.
- For each abstract semantics type $\cdot \alpha := \text{type}(\alpha(i))$ -- analysis universe -- there is a partial order relation $\leq$ and a meet operation $\sqcap$ (supremum),
- $\gamma(a) \sqsubseteq \alpha \rightarrow (\alpha \sqcap \gamma(a))$.
- $\sqcap$ defines a complete partial order.

DFA on SSA (cont.)

- Provided that transfer functions are monotone: $x \leq y \Rightarrow \Gamma_n(x) \leq \Gamma_n(y)$ with $x, y \in A_1 \times \ldots \times A_l$ and $\leq$ defined element-wise with $\leq$ of the respective abstract semantics type.

Generalization to $\chi$-terms

- Given such a context-insensitive analysis (lattices for abstract values, set of transfer functions, initialization of start node) we can systematically construct a context-sensitive analysis.

$\chi$-term algebras $\times$ over abstract semantic values $\alpha \in A$ introduced.

- $\times \in A \Rightarrow \times \in \times_1$.
- $\times_1, \times_2 \in \times \Rightarrow \times_1 \sqcup \times_2 \in \times_1$.
- $\Gamma_n$ induces sensitive $\times$ for abstract values $(\times_1, \leq)$.

New transfer functions

- Insensitive: $\Gamma_n = \alpha(i) \cup \ldots \cup \alpha(o)$.

Sensitive: $\Gamma_n = \alpha(i) \cup \ldots \cup \alpha(o)$ with $\alpha$ block number of $\phi$-node and for $\alpha, \beta \in A$ and $\times_1, \times_2, \times_3 \in \times_1$.

- $\times_1 \sqcup \times_2 \sqcup \alpha = \alpha(i) \cup \ldots \cup \alpha(o)$.

Ordinary operation's (i node's) transfer functions:

- Insensitive (w.r.t. binary operation): $\times_1 \times A \rightarrow \times_1$.

Sensitive: $\times_1 \times A \rightarrow \times_1$, and for $\alpha, \beta \in A$ and $\times_1, \times_2 \in \times_1$.

$\gamma_1(i \times_1 \times_2) = \gamma_1(i \times_1 \times_2)_{\text{cof}(\gamma_1(i \times_1 \times_2))}$.

with $k$ larger of $\times_1, \times_2$, and cof is the cofactorization.

Sensitive Transfer Schema (case a)
Context-sensitive $\phi$-node transfer function

Context-sensitive $a \oplus b$ (cont.)

$S_g(x_4(1,2), x_4(1,2), ..., x_4(3,4)) = ...$
$= z_d \cdot z_d \cdot z_d \cdot z_d(1,2), 3, z_4(1,2), 3, S_d(4,5,6) \cdot \chi_d(5,6) \cdot \chi_d(4,5,6, 6))$

Example revisited: insensitive

Co-factorization

- $\text{cof}(x_i(t_1, t_2), x_i, i)$ selects the $i$-th branch of a $x_i$-term if $x_i = x_i$
- $\text{cof}(x_i(t_1, t_2), x_i, i)$ = $x_i(t_1, t_2)$ iff $k > x$
- $\text{cof}(x_i(t_1, t_2), x_i, i)$ = $t_1$ iff $k = x$
- $\text{cof}(x_i(t_1, t_2), x_i, i)$ = $t_2$ iff $k = x$

Sensitive Transfer Schema (case b)
Assignment III

Given the SSA fragment on the left

- Perform context-insensitive data-flow analysis (using the definitions on the previous slides). What is the the value at the entry of node $x$?
- Perform context-sensitive data-flow analysis (using the definitions on the previous slides). What is the the value at the entry of node $x$?
- Why is the former less precise than the latter?
- Construct a scenario where you could take advantage of that precision in an optimization!