Linnéuniversitetet Kalmar Växjö

Static Single Assignment (SSA) Form

Construction - Analyses - Optimizations

Welf Löwe

Welf.lowe@lnu.se

1

1

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

2

2

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

3

3

What makes 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 $s;s'$ of operations is essential *iff* execution $s';s$ changes observable behavior of the program
  - Computation of essential dependencies is not decidable
- Compact
  - Representation of dependencies
  - No (few) redundant expression

4

4

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 triples of assignment $t := \tau\, t'\, t''$ with t, t', t'' a variable/label/register
  - Represent program as a directed graph of operations τ with explicit def-use edges ($t\ t'$) connecting operations
- SSA-Property: there is only one single (static) position (label) in a program/procedure defining t
  - Does not mean t 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

5

5

Avoid redundant computations

- Assign each (partial) expression a unique number (label).
  - Good optimization in itself as values can be reused instead of recomputed
  - Basic idea for SSA
- Syntactic different computations that produce provably equivalent values get the same number
- How to statically find computations with provably equivalent values?
  - Can be computed by data flow analysis
  - It's a forward, must problem
  - Known as value numbering

6

6

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 or 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 algebraically identical (according to a number of algebraic rules implemented)

7

7

Idea of Value Numbering

- Congruent values get the same value number (in general a label)
- Values are defined by operations and used by other operations
- Values computed only once (by one operation) and then reused (referring to the value number of that operation)
- Algorithmic idea to prove equality of expression values at different program points (congruence of tuples) follows the congruence definition:
  - 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 (postponed):
  - Alias/Points-to problem: Addresses, hence address content, is not exactly computable. Where are values stored in and later loaded from in, e.g., an array with index expressions? Not decidable.
  - Meets in control flow: which definition holds? Simple trick.

8

8

Value Numbering

- Type of value numbers:
  - *INT* for integer constants; *BOOL* for Boolean constants etc.
  - Otherwise, ids (labels): $\{vn_1, . . . , vn_n\}$
- Data structures:
  - Process tuples $t := \tau\ t'\ t''$ with τ is a constant operator symbol,
  - We construct a mapping of the original label t to its value number vn
  - We construct an auxiliary mapping such that a lookup of $\tau\ vn(t')\ vn(t'')$ gives a unique value number vn or *void*, if not known yet
    - vn becomes the value number of t,
    - $vn(t')\ vn(t'')$ are value numbers of tuples labeled $t'\ t''$
    - We make sure $vn(t')\ vn(t'')$ always already computed or are constants.
- For a first try:
  - Computation basic block local
  - One such mapping t to vn per basic block.

9

9

Value Numbering
with Local Variables without Alias Problem

(1) Initially: value number $vn(constant)=constant$; $vn(t) = void$ for all tuples t.
(2) for all tuple t in program order:
case (a) $t = ST$ >*local*< t' -- *write to a local variable*
    $vn(t) := vn(ST$ >*local*< $vn(t'))$
    if $vn(t) = void$ then
        $vn(LD$ <*local* >$) := vn(t')$,
        $vn(t)$:= new value number,
        generate: $vn(t)$: ST >*local*< $vn(t')$
(b) $t = LD$ <*local*> -- *read from a local variable*
    $vn(t) := vn(LD$ <*local*>$)$.
    if $vn(t) = void$ then
        $vn(t)$:= new value number,
        generate: $vn(t)$: LD <*local*>
(c) $t = \tau\ t'\ t''$ -- *any operation* τ
    $vn(t) := vn(\tau\ vn(t')\ vn\ (t''))$
    if $vn(t) = void$ then
        $vn(t)$:= new value number,
        generate: $vn(t)$: $\tau\ vn(t')\ vn(t'')$.
(d) $t = call\ \ proc\ t'\ t''$ ... -- *analogously to* (c) *with* $\tau = call\ \ proc$

10

10

Example

Original	Result
t_1: *ST* >*a*< 2	v_1: *ST* >*a*< 2
t_2: *LD* <*a*>	
t_3: *LD* <*x*>	v_2: *LD* <*x*>
t_4: *MUL* t_2 t_3	v_3: *MUL* 2 v_2
t_5: *ADD* t_4 1	v_4: *ADD* v_3 1
t_6: *ST* >*b*< t_5	v_5: *ST* >*b*< v_4
t_7: *LD* <*x*>	
t_8: *MUL* 2 t_7	
t_9: *ST* >*a*< t_8	v_6: *ST* >*a*< v_3
t_{10}: *LD* <*a*>	
t_{11}: *ADD* t_{10} 1	
t_{12}: *LD* <*b*>	
t_{13}: *ADD* t_{11} t_{12}	v_7: *ADD* v_4 v_4
t_{14}: *ST* >*c*< t_{13}	v_8: *ST* >*c*< v_7

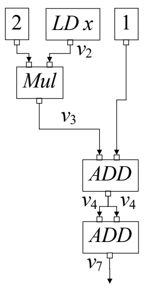
11

11

Value Number Graph of Basic Block

Result

v_1: *ST* >*a*< 2
v_2: *LD* <*x*>
v_3: *MUL* 2 v_2
v_4: *ADD* v_3 1
v_5: *ST* >*b*< v_4
v_6: *ST* >*a*< v_3
v_7: *ADD* v_4 v_4
v_8: *ST* >*c*< v_7



12

12

Value Numbering
with Global Variables without Alias Problem

- Case (a') as before case (a) for local variables
  case (a') $t = ST > global < t'$
  $vn(t) := vn(ST > global < vn(t'))$
  if $vn(t) = void$ then
  $vn(LD < global >) := vn(t')$,
  $vn(t)$:= new value number,
  generate: $vn(t)$: $ST > global < vn(t')$
- Procedures:
  - Case (d) as before
  - But as *global* (potentially) redefined in *proc*, set value number for tuple $ST > global < t'$, $LD < global >$ to *void*
- Improvement for non-recursive sequential leaf procedures:
  - New case (d): analyze procedure as if it was inlined
  - Too complex if *proc* has more than one basic block (interprocedural analysis)

13

13

General Value Numbering

-- *t' is an address with unknown value (no compile time constant address, no variable)*
-- *computed in an operation with value number vn(t')*
Case (e) $t = ST\ t'\ t''$
$vn(t) := vn(ST\ vn(t')\ vn(t''))$
if $vn(t) = void$ then
$vn(LD\ vn(t')) := vn(t'')$,
$vn(t)$:= new value number,
Generate: $vn(t)$: $ST\ vn(t')\ vn(t'')$
if t' may be an alias of another address tt: -- *requires points-to analysis*
$vn(ST\ vn(tt)\ \ldots) := void$,
$vn(LD\ vn(tt)) := void$,
Case (f) $t = LD\ t'$
$vn(t) := vn(LD\ vn(t'))$
if $vn(t) = void$ then
$vn(t)$:= new value number,
Generate: $vn(t)$: $LD\ vn(t')$

14

14

Remarks

- Values numbering gets complex when involving
  - Global variables
  - Procedure calls
  - Indirect address computations
- So-called strong updates of value numbers required additional
  - Dataflow analyses, especially, def-use and points-to analyses
  - In an interprocedural way
- On what IR? We are about to construct an IR that is suitable for these dataflow analyses.
- Recommendation: for constructing value numbers and SSA, take an easy conservative implementation: in case of doubt set the computed value numbers to *void* especially:
  - After *call proc*, entries of global variables get *void*
  - A store operation $ST > a < t'$ sets *void* all $vn(LD < a' >)$ and $vn(ST < a' > t')$ if it is not clear, whether $a = a'$ or $a \neq a'$ (alias-problem). Special case: arrays with index expressions

15

15

Observation

- Value number graph of a basic block
  - No (provable) unnecessary dependencies
  - No (provable) redundant computation
- Initially all value numbers are set to *void* (for each basic block)
- By knowing the values of predecessor basic blocks, this initialization can be improved
- Such an initializations over basic block leads to SSA form

16

16

Value number graph → SSA

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

17

17

Simple trick: ϕ-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 just add a pseudo expression selecting a value number from the control flow predecessor blocks
  - Defining itself a new version (value number) of that variable $a_3 := \phi(a_1, a_2)$
- E.g.

```
if … then i1:=1 else i2:=2 end; i3:=ϕ(i1,i2); x:=i3
i1:=0; while i3:=ϕ(i1,i2); … loop … ; i2:=i3+1; … end; x:=i3
```

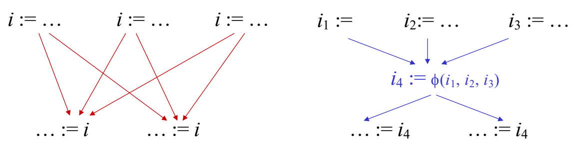
- ϕ–functions
  - always occur at the beginning of a block
  - are non-strict; switches selecting the either of the arguments
  - are all evaluated simultaneously for a block, with all having the same selection behavior
  - guarantee that there is exactly one static definition/assignment for each use of a variable
- Assignment $i_0 := \phi(i_1,\ldots,i_k)$ in a basic block indicates that the block has k direct predecessors in the control flow

18

18

Compact representation of dependencies

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

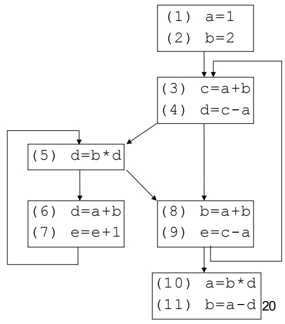


19

19

Example Program and Basic Block Graph

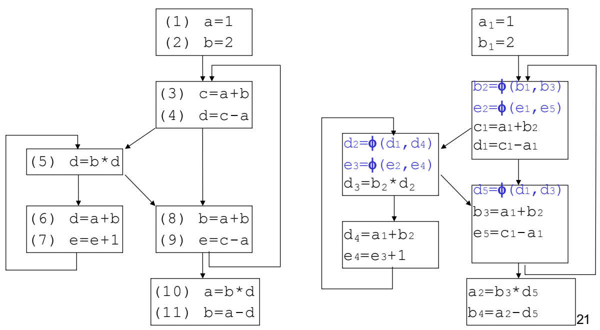
```
(1) a=1;
(2) b=2;
    while true{
(3)   c=a+b;
(4)   if (d=c-a)
(5)     while (d=b*d){
(6)        d=a+b;
(7)        e=e+1;
        }
(8)   b=a+b;
(9)   if (e=c-a) break;
    }
(10)a=b*d;
(11)b=a-d;
```



20

20

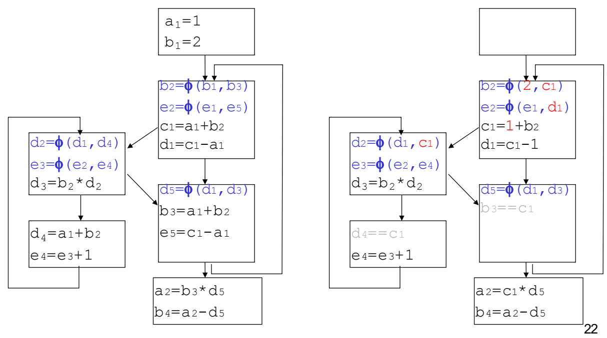
Basic Block and SSA Graph



21

21

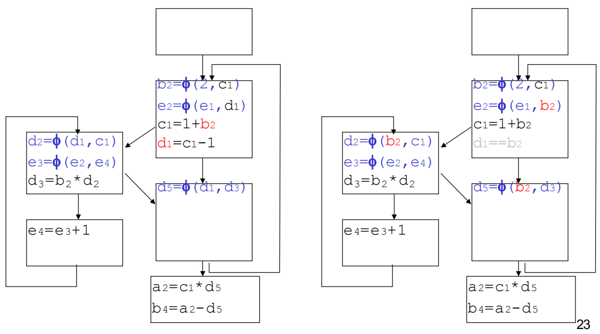
SSA-Graph before and after Constant and Copy Propagation



22

22

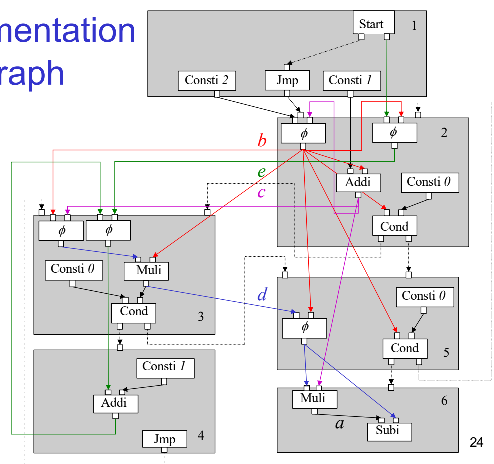
SSA-Graph before and after using Algebraic Identities



23

23

Implementation SSA graph



24

24

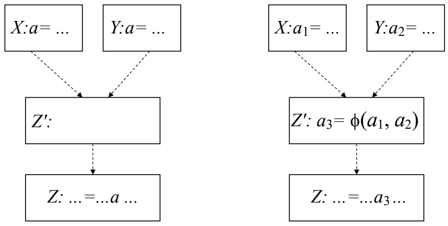
SSA properties

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 are a DAG.
Note: correspondence to value number graphs and expression trees
P3: Input of ϕ-operations have same type as their output.
P4: i-th operand of a ϕ-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 *End, Jump, Cond, Ret*
P7: If operation x in a BB B_x defines an operand of operation y in a BB B_y then there is a path $B_x \to^+ B_y$.
P7a: (Special case of P7) operation y is a ϕ-operation and x is defined in $B_x = B_y$ then there is a cyclic path $B_y \to^+ B_y$.

P8: 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 $X \to^+ Z$, $Y \to^+ Z$.
Then Z' contains a ϕ-operation for a.

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 $X \to^+ Z$, $Y \to^+ Z$.
Then Z' contains a ϕ-operation for a.



Remark: Z' is in the dominance frontier of X, Y. This is often used to explain the placement of ϕ nodes. Our lazy approach leads to the same result.

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

Remainder Value Numbering

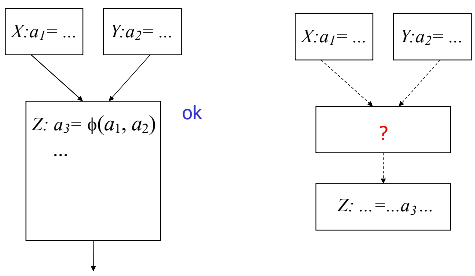
(1) Initially: value number $vn(constant)=constant$; $vn(t) = void$ for all tuples t.
(2) for all tuple t in program order:
case (a) $t = ST >local< t'$ -- *write to a local variable*
$vn(t) := vn(ST >local< vn(t'))$
if $vn(t) = void$ then
$vn(LD <local>) := vn(t')$,
$vn(t)$:= new value number,
generate: $vn(t)$: $ST >local< vn(t')$
(b) $t = LD <local>$ -- *read from a local variable*
$vn(t) := vn(LD <local>)$.
if $vn(t) = void$ then
$vn(t)$:= new value number,
generate: $vn(t)$: $LD <local>$
(c) $t = \tau\ t'\ t''$ -- *any operation* τ
$vn(t) := vn(\tau\ vn(t')\ vn(t''))$
if $vn(t) = void$ then
$vn(t)$:= new value number,
generate: $vn(t)$: $\tau\ vn(t')\ vn(t'')$.
(d) $t = call\ \ proc\ t'\ t''$ ... -- *analogously to* (c) *with* $\tau = call\ \ proc$

Extended Initialization

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

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

Extended Value Numbering



Extended Initialization

(1) Initialization of mapping for current block Z

(A) always: $vn(constant)_Z = constant$;

(B) if Z = start block: $vn(t) = void$ for all tuples t.

(C) else: let $Pred=\{X, Y, \ldots\}$ be the predecessors of Z in basic block graph

for all variables t used in current block Z:

if for any $B \in Pred$: $vn(t)_B = void$

recursively, initialize block B with (1) and get $vn(t)_B$

if $vn(t)_X \neq vn(t)_Y$

$vn(t)_Z$:= new value number

generate: $vn(t)_Z := \phi(vn(t)_X\ vn(t)_Y)$

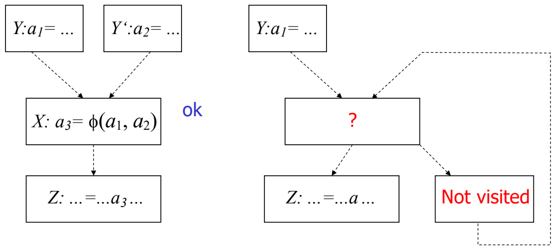
if $vn(t)_X = vn(t)_Y$

$vn(t)_Z := vn(t)_X$

(2) for all tuple t in program order:

-- *as before*

Extended Value Numbering



Extended Initialization

(1) Initialization of mapping for current block Z

(A) always: vn(constant)$_Z$ =constant;

(B) if Z = start block: $vn(t) = void$ for all tuples t.

(C) else: let $Pred=\{X, Y, \ldots\}$ be the predecessors of Z in basic block graph

for all variables t used in current block Z:

if for any $B \in Pred = unvisited$

$vn(t)_B$ = guess a new special value number

if for any $B \in Pred$: $vn(t)_B = void$

recursively, initialize block B with (1) and get $vn(t)_B$

if $vn(t)_X \neq vn(t)_Y$

$vn(t)_Z$:= new value number

generate: $vn(t)_Z := \phi(vn(t)_X\ vn(t)_Y)$

if $vn(t)_X$ or $vn(t)_Y$ is guessed

generate: $vn(t)_Z := \phi'(vn(t)_X\ vn(t)_Y)$

if $vn(t)_X = vn(t)_Y$

$vn(t)_Z := vn(t)_X$

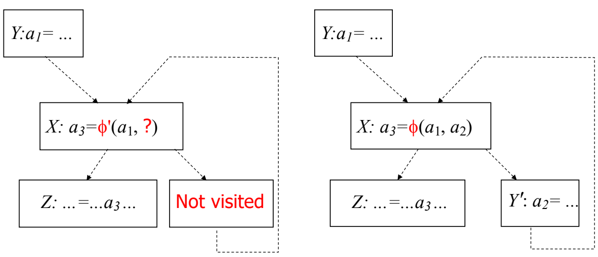
(2) for all tuple t in program order:

-- *as before*

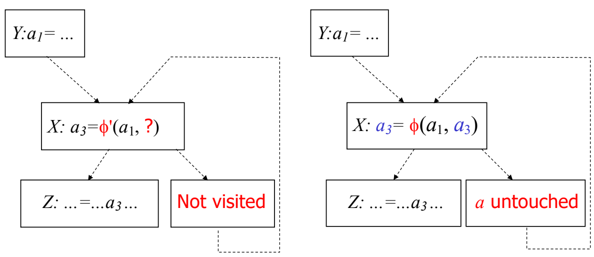
Eliminate/Mature ϕ'-Functions

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

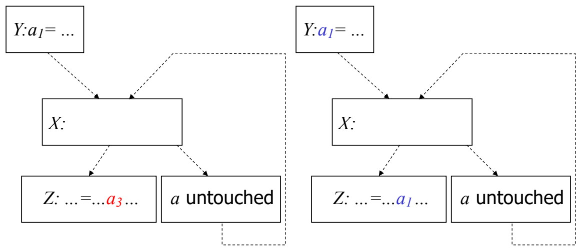
Example I: Mature ϕ'-Functions



Example II: Mature ϕ'-Functions



Example III: Mature ϕ′-Functions

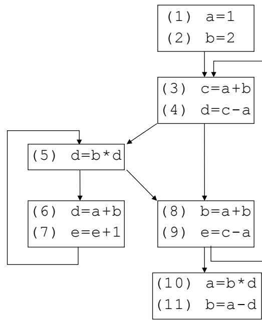


47

47

Example Program and BB Graph

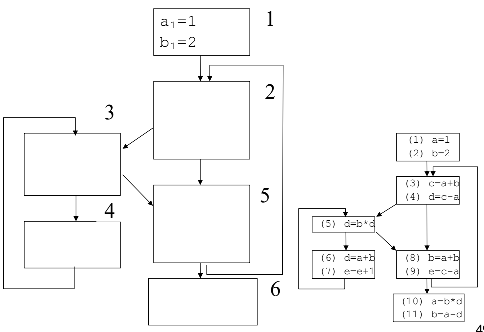
```
(1) a=1;
(2) b=2;
    while true{
(3)    c=a+b;
(4)    if (d=c-a)
(5)       while (d=b*d){
(6)          d=a+b;
(7)          e=e+1;
          }
(8)    b=a+b;
(9)    if (e=c-a) break;
    }
(10)a=b*d;
(11)b=a-d;
```



48

48

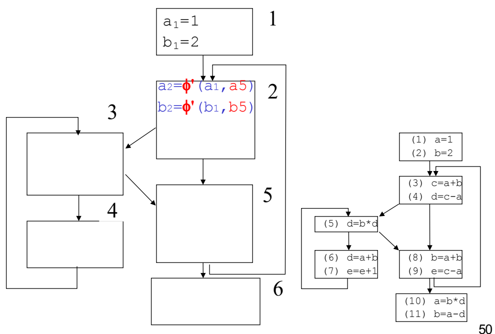
SSA Construction Block 1



49

49

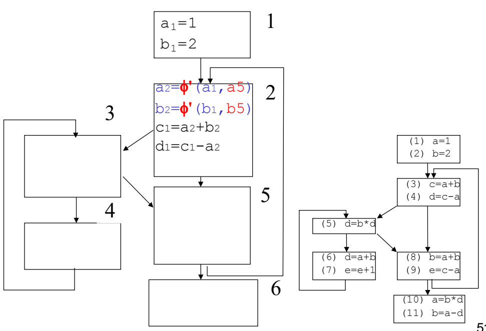
SSA Construction Block 2 - Initialization



50

50

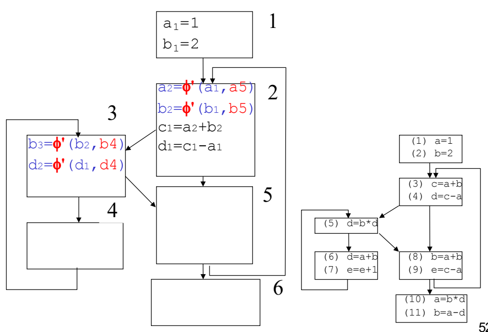
SSA Construction Block 2



51

51

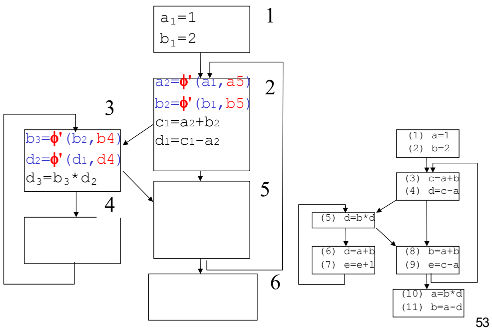
SSA Construction Block 3 - Initialization



52

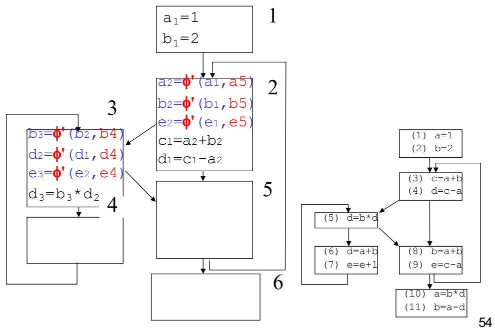
52

SSA Construction Block 3



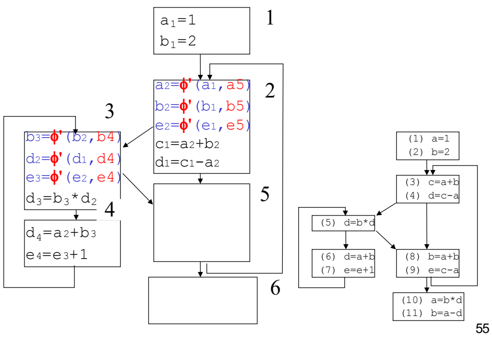
53

SSA Construction Block 4 - Initialization



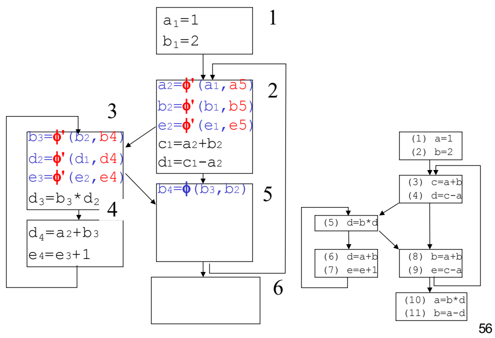
54

SSA Construction Block 4



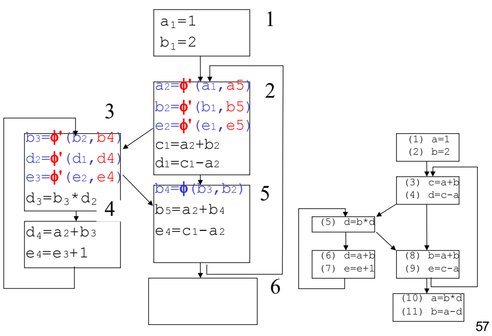
55

SSA Construction Block 5 - Initialization



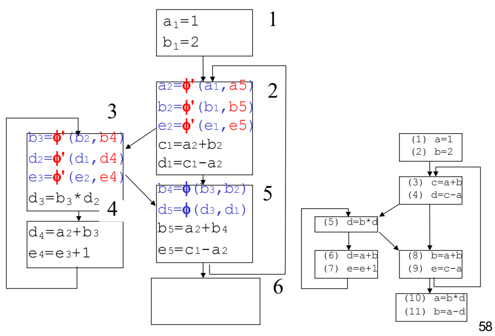
56

SSA Construction Block 5



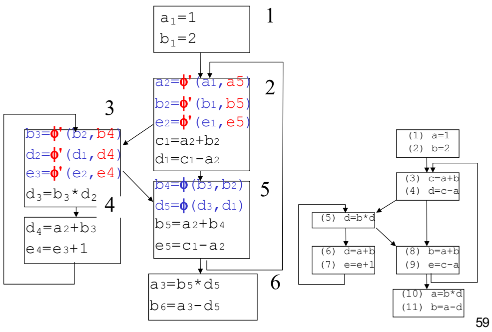
57

SSA Construction Block 6 - Initialization



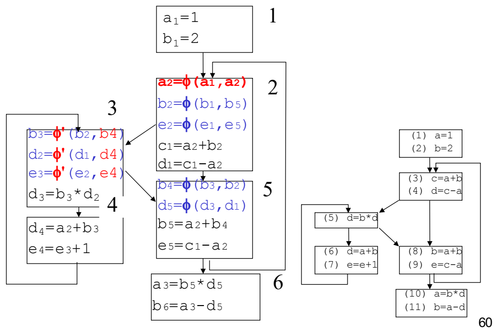
58

SSA Construction Block 6



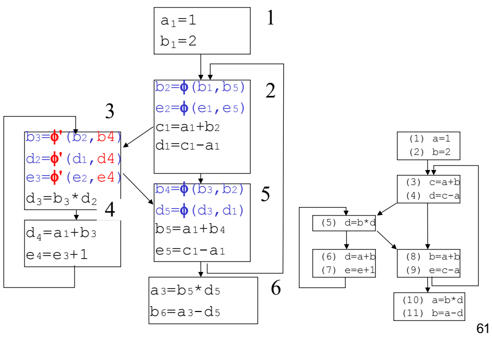
59

SSA Mature Block 2



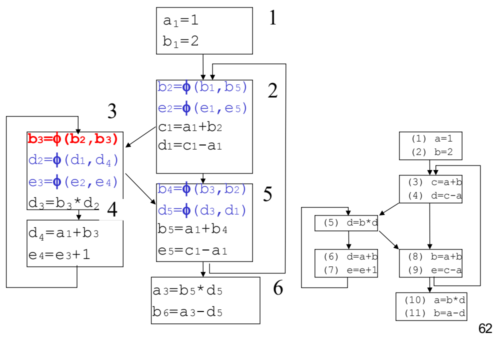
60

SSA Mature Block 2



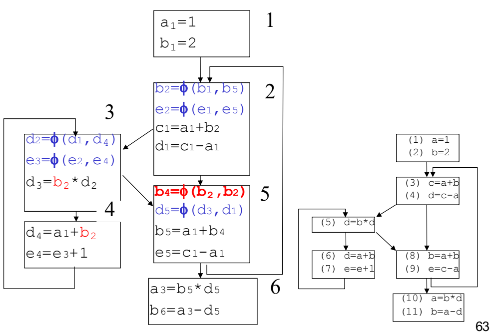
61

SSA Mature Block 3



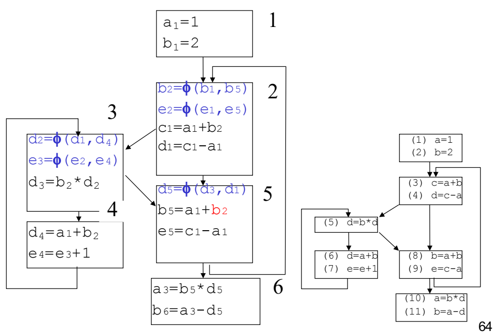
62

SSA Mature Block 3



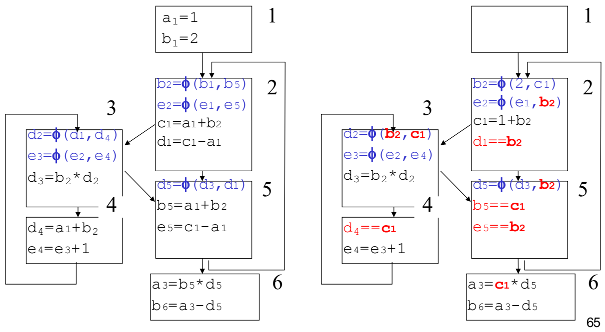
63

SSA Mature Block 3



64

Final Simplifications



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 form syntactical different operations
    - Values are generated form 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:
  - $v_{(i)}$= ... Variable v defined in statement (i)
  - $u_{(j)}= \tau(\ldots v_{(x,y,z,\ldots)} \ldots)$ Variable v used defined (may reaching definitions) in statements $(x,y,z,\ldots)$
- Set $v_{(i)} \equiv u_{(j)}$ for all $v_{(i)}$, $u_{(j)}$ in the program
- Iterate until a fixed point over:
  - Set $v_{(i)} \not\equiv u_{(j)}$ for:
    - $v_{(i)} = constant$ and $u_{(j)} \neq constant$
    - $v_{(i)} = \tau(\ldots)$ and $u_{(j)} \neq \tau(\ldots)$
    - $v_{(i)} = \tau(x_1,y_1)$ and $u_{(j)} = \tau(x_2,y_2)$ but $x_1 \not\equiv x_2$ or $y_1 \not\equiv y_2$
- Find a unique value number for each equivalence class
- Replace variables consistently by value number for each equivalence class
- Insert, if necessary, ϕ-functions eventually at the dominance frontiers or during the fixed-point iteration

Minimal SSA-Form

- Insight:
  - ϕ-functions guarantee that for each use of a variable there is exact one definition ("variable" means program- or auxiliary temp variable)
  - Encodes solution of the (may) Reaching-Definitions-Problem
  - Problems with array elements and indirectly addressed variables remain (to be discussed and solved later)
- Minimal *SSA*-form: set ϕ-function $a_0 := \phi(a_1, a_2, \ldots)$ in block B iff value a_0 is live in B.
  - Use data flow analysis $liveIn(B)$ and check $a \in liveIn(B)$.
- Faster but potentially larger:
  - generate value numbers only on demand
  - lazy initialization integrated in the construction algorithms
  - generates code for transitively dead variables, hence, larger result

SSA – Construction from AST

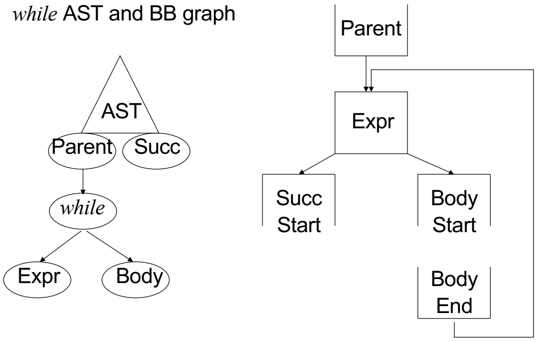
- Left-Right Traversal (1. Round):
  - compute for each syntactic expression its basic block number
  - compute precedence relation on basic blocks
  - generate expression triples into the BBs
- Right-Left Traversal (2. Round):
  - compute, for each live (beginning with the results of a procedure) expressions, the value numbers (contains ϕ') using the data structures known from value numbering
- Left-Right Traversal (3. Round):
  - Mature ϕ'-functions
  - generate SSA for nonempty blocks
- Further eliminations on SSA graph

SSA from AST

- One left-right tree traversal if we use lazy initialization instead of live analysis,
  - 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 BB nodes and control flow edges

SSA from AST

- *while* AST and BB graph

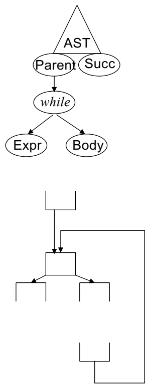


82

82

SSA from AST

- *while* actions
  - Finalize current block B(Parent)
  - Create a new current block B(Expr)
  - Add control flow B(Paret) to B(Expr)
  - Recursively, generate code for Expr computing value numbers locally
  - Finalize current block B(Expr)
  - Create a new current block B_{start}(Body)
  - Add control flow B(Expr) to B_{start}(Body)
  - Recursively, generate SSA code for Body
  - After return current block is B_{end}(Body), finalize it (B_{start} and B_{end} may be different)
  - Add control flow B_{end}(Body) to B(Expr)
  - Create a new current block B_{start}(Succ)
  - Add control flow B(Expr) to B_{start}(Succ)
  - Return with B_{start}(Succ) as current block



83

83

Deconstruction of SSA

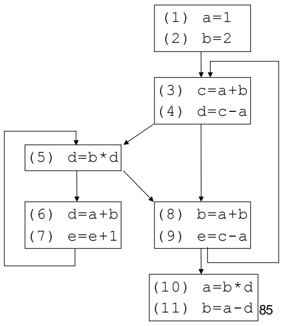
- Serialize the SSA graph
- Replace data dependency edges by variables
- Remove ϕ-functions $a_0 := \phi(a_1, a_2, ...)$:
  - Assume each variable a_0 designates a "register"
  - Copy values $a_1, a_2, ...$ at the end of the predecessor basic blocks into that register a_0
  - Requires possibly new blocks on some edges as $a_1, a_2, ...$ may be used in other successor blocks
  - Perform copy propagation to avoid unnecessary copy operations
- Allocate registers for the variables
  - Fixed number of registers
  - In general, more variables than registers
  - Idea: assign variables with non overlapping lifetimes to the same register
  - Later problem

84

84

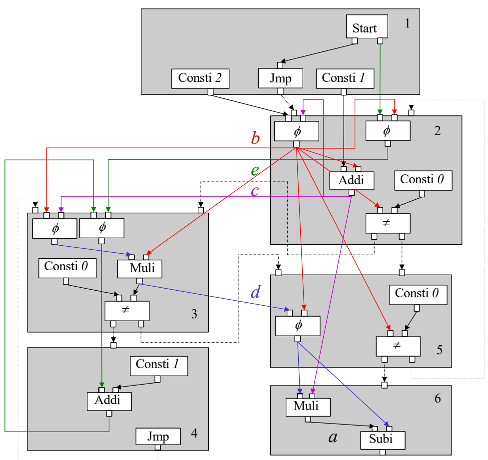
Example Program and BB Graph

```
(1) a=1;
(2) b=2;
    while true{
(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;
```



85

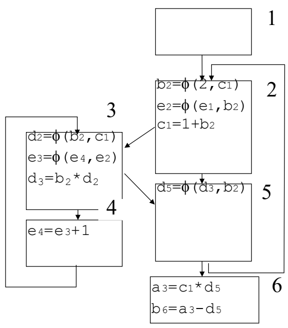
85

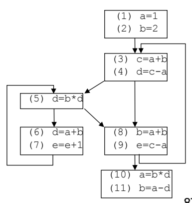


86

86

Introduce Variables for Edges

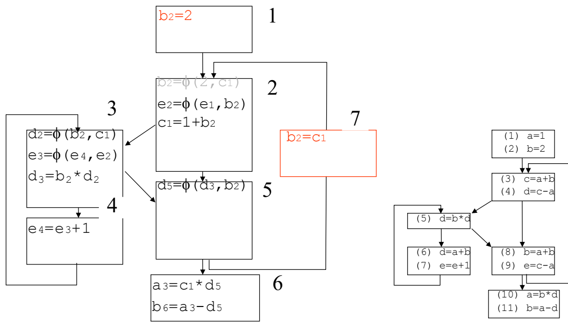




87

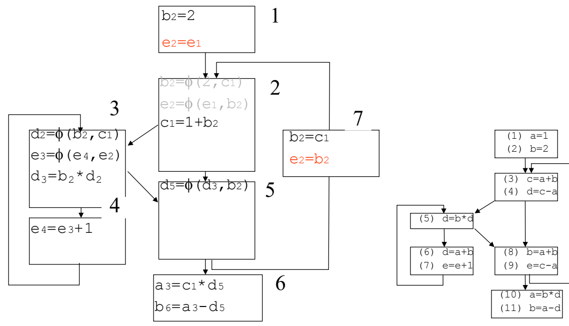
87

Remove ϕ-functions



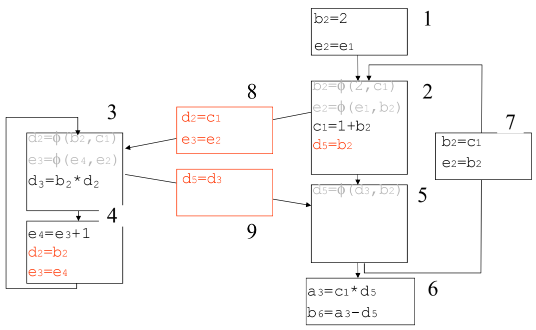
88

Remove ϕ-functions



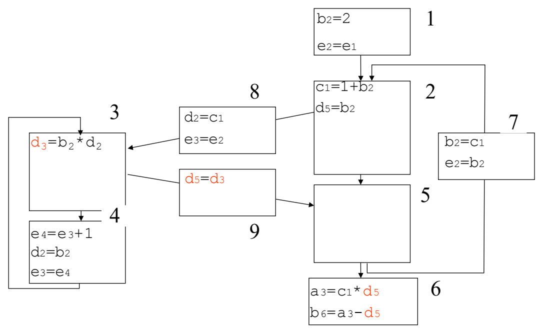
89

Remove ϕ-functions



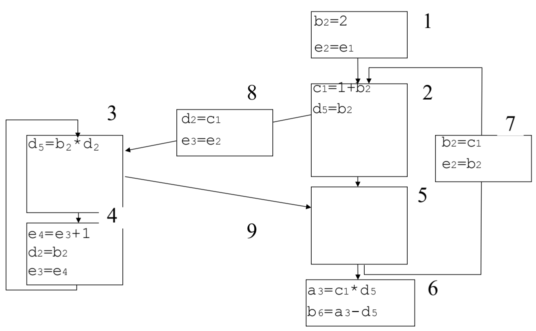
90

Copy Propagation



91

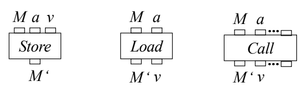
Remove Empty Block



92

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

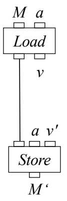


M, M' Memory state
a Address
v Value

93

93

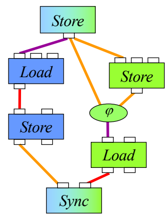
Why *Load* Defines Memory?



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 are easier, e.g.
    - Heap vs. Stack
    - Different arrays on the stack
    - Heap partitions for different object types
- Distinction often only locally possible
  - Union necessary
  - Sync operation unifies disjoint memory fragments
  - Like ϕ- 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 make the programs non-implementable

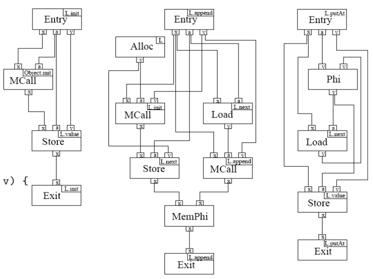
- Note: if we only have to analyze the program and not to generate code, P9 could be ignored

Reduced SSA Representations

- Not the whole program is directly relevant for all analyses
  - Certain data types are uninteresting, e.g., value types such as Int, Bool in Points-to analysis
- 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;
  List next = null;

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

  public void append(Object v) {
    if (next == null)
      next = new List(v);
    else
      next.append(v);
  }

  public void putAt(int n,Object v) {
    int count = 0;
    List l = this;
    while (count < n) {
      l = l.next;
      count++;
    }
    l.value = v;
  }
}
```



Cliffhanger from the earlier today

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

Recall 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.
- “Static” or “dynamic” dispatch:
  - Call graph construction is required for P2A (static dispatch)
  - Can be integrated in P2A (dynamic dispatch)

Recall Points-to Analysis (P2A)

- 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 flow-insensitive; it focuses on data-dependencies over variables and ignores the intraprocedural control flow
  - An analysis is flow-sensitive if it takes into account the order in which statements in a program are executed
  - In principle, additional def-use analysis avoids this problem at the expenses of higher memory and analysis costs
- The baseline P2G approach is context-insensitive
  - An analysis is context-sensitive if distinguishes different contexts in which procedures/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 Accurate P2A

1. Data values
   - allocation site abstract from objects O
   - abstract heap memory: O × F→O (F set of fields) heap size: `Int`
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 ϕ-nodes for merge points in control flow
   - Edges represent intra-procedural control- and data-flow
   - Reduced general SSA graph
3. Transfer function:
   - update the heap according to the abstract semantics of the node kinds
   - Special for ϕ-nodes: ∪ on O values and max on `Int` values, resp.
4. Initialization: ∅ for O ports and 0 for `Int` ports, resp.
5. Simulated execution

1: Representation of Memory in P2A

- Assume there is only one abstract heap memory value (*mem*) valid at all program points during analysis, we can use global memory data structure
  - Mapping (abstract objects, attributes) → stored values
  - Note, stored values are references, i.e., a (set of) abstract object(s)
- Update heap memory value data structure as side effect of
  - *Store* and *Alloc* operations
  - Weak updates, i.e., *generate/add* but never *kill/delete* information
- Distinguish between abstract heap memory (*mem*) and abstract heap memory size values (*size*), we can use memory *size* as type of memory edge values in SSA
  - Changed size indicates changed memory speeds up the fixed-point iteration
  - Phi-functions over memory size

1: Types *addr*, *size* and *mem*

addr

- Each static allocation site i corresponds to an abstract object $o_i \in O$
- An address in the analysis is a subset of the finite set of abstract objects: $\mathcal{P}^O$
- Alloc produces *addr* values and Load, Store, Call uses them

size

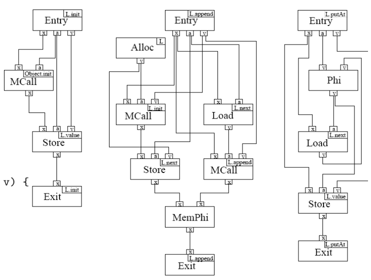
- The memory *size* is an indicator of the current size of the heap abstraction and guarantees the order of memory related operations
- Implemented as a special Integer
- Used instead of a heap memory value in the SSA graph edges

mem

- Is the global data structure modeling the heap memory (singleton)
- Is not a value in the SSA graph edges but updated as side effect of node interpretation
- A memory slot is a pair of abstract object and *field* $[o_i, field] \in O \times F$
- A state of a memory slot is a pair of memory slot and an *addr* value: $([o_i, f] \mapsto addr) \in O \times F \times \mathcal{P}^O$
- A state of the memory is the state of all memory slots
- Memory $mem \subseteq O \times F \times \mathcal{P}^O$
- Functions *set* and *get* to access the *addr* value of a slot

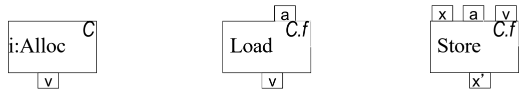
2: Points-to-SSA Graphs

```
public class List {
  Object value = null;
  List next = null;

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

  public void append(Object v) {
    if (next == null)
      next = new List(v);
    else
      next.append(v);
  }

  public void putAt(int n,Object v) {
    int count = 0;
    List l = this;
    while (count < n) {
      l = l.next;
      count++;
    }
    l.value = v;
  }
}
```



3: Transfer functions

if input changes, update as below else skip:



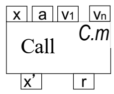
$Pt(v)=\{o_i\}$ (unique i) $Pt(v)=\cup_{oi \in Pt(a)}$ get(o_i,f) $\forall o_i \in Pt(a)$:
set (o_i,f, get(o_i,f) $\cup Pt(v)$)
size(x') = heap.size

106

106

5: 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 $Pt(v_1 \ldots v_n)$ to the all callees $Pt(a)$
  - Processes the callees (one by one) completely (iterate in case of recursive calls)
  - Propagates back and merges the results $Pt(r)$ of the callees
  - Updates heap size size(x')
  - Continue processing the caller method …



107

107

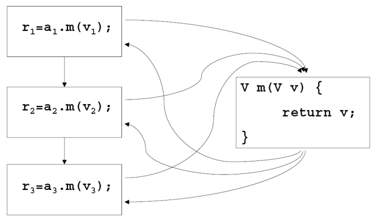
Flow-sensitivity

- 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: simulated execution 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 (method summaries) to avoid exponential explosion and guarantee scalability.

108

108

Example: Global flow-sensitivity



Flow-insensitive Result

$$Pt(r_1) = Pt(r_2) = Pt(r_3) = Pt(v_1) \cup Pt(v_2) \cup Pt(v_3)$$

Flow-sensitive Result

$$\begin{aligned} Pt(r_1) &= Pt(v_1), \\ Pt(r_2) &= Pt(v_1) \cup Pt(v_2), \\ Pt(r_3) &= Pt(v_1) \cup Pt(v_2) \cup Pt(v_3) \end{aligned}$$

109

109

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 `a.m(v ...)` 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: *context*: `[m, call id, a, v1,… , vn]` → C

110

110

Context-sensitive call handling

Call(m, call, x_{in}, a, v_1, … , v_n] → [x_{out}, r]
  [x_{out}, r] = [0, ⊥]
  for all c ∈ *context* `[m, call id, a, v1,… , vn]` **do**
    if [x_{in}, a, v_1, … , v_n] ⊑ previous arguments (m, c) **then**
      r = previous return (m, c)
      x_{out} = x_{in}
    else
      args = previous args (m, c) ⊔ [x_{in}, a, v_1, … , v_n]
      args previous (m, c) = args
      [x_{out}, r] = [x_{out}, r] ⊔ processMethod(m, args)
      previous return (m, c) = r
    end if
  end for
  return [x_{out}, r]

111

111

Examples: Context abstractions

Object-sensitivity
- A context is given by a pair `(m,o)`
- where o ∈ 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,this)`
- where `this` $\in \mathcal{P}^O$ 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.

112

112

Examples: Precision

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

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

113

113

Results

- Fast and accurate P2A
  - Points-to SSA ⇒ locally flow sensitive PTA
  - Simulated execution ⇒ globally flow-sensitive PTA, fast
  - Context-insensitive in the 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-insensitive P2A
  - As precise and 10x as fast compared to the best-known context-sensitive variant (object sensitivity) P2A
- Shows in clients analyses like synchronization removal and static garbage collection (escape and side effect analysis)

114

114

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 context-sensitive analysis results
- Optimizations

115

115