

Static Single Assignment (SSA) Form

Construction - Analyses - Optimizations

Welf Löwe
Welf.loewe@lnu.se

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

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

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 $s; s'$ 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 representation

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 program as a directed graph of operations τ
 - 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 **single** (static) position 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

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)

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

7

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 prove 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 to and loaded from? Not decidable.
 - Meets in control flow: which definition holds?

8

Value Numbering

- Type of value numbers:**
 - INT* for integer constants; *BOOL* for Boolean constants etc.
 - Use ids (labels): $\{t_1, \dots, t_n\}$ otherwise
- Data structure:**
 - Mapping (hash table) gets value number vn of operations defining the values, i.e., for tuples $t := \tau t' t''$, a lookup of $\tau vn(t') vn(t'')$ gets the value number of t ,
 - Mapping of operation labels t to tuples $\tau t' t''$ (implicit),
 - τ is an operator symbol,
 - $vn(t')$ $vn(t'')$ 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.

9

Value Numbering with Local Variables without Alias Problem

- Initially: value number $vn(\text{constant}) = \text{constant}$; $vn(t) = \text{void}$ for all tuples t .
- for all tuple t in program order:

```

case (a)  $t = ST \>local \leftarrow t'$ 
       $vn(t) := vn(ST \>local \leftarrow vn(t'))$ 
      if  $vn(t) = \text{void}$  then
         $vn(LD \<local \>) := vn(t')$ ,
         $vn(t) := \text{new value number}$ ,
        generate:  $vn(t): ST \>local \leftarrow vn(t')$ 
(b)  $t = LD \<local \>$ 
       $vn(t) := vn(LD \<local \>)$ ,
      if  $vn(t) = \text{void}$  then
         $vn(t) := \text{new value number}$ ,
        generate:  $vn(t): LD \<local \>$ 
(c)  $t = \tau t' t''$ 
       $vn(t) := vn(\tau vn(t') vn(t''))$ 
      if  $vn(t) = \text{void}$  then
         $vn(t) := \text{new value number}$ ,
        generate:  $vn(t): \tau vn(t') vn(t'')$ 
(d)  $t = \text{call proc } t' t'' \dots$  -- analog (c) with  $\tau = \text{call proc}$ 
    
```

10

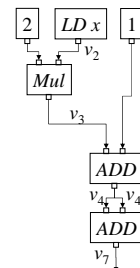
Example

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

11

Value Number Graph of Basic Block

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



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

13

General Value Numbering

- t'* is an address with unknown value (no compile time constant address, no name)
- 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 semantically equivalent to *tt*: -- Points-to analysis


```

      vn (ST vn (tt) ..) := void ,
      vn (LD vn (tt)) := void ,
      value number of transitively depending tuples := 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

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 new entry in the hash table generates a new value number. After *call proc*, entries depending on non-local 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
- Value number graph of a basic block
 - No (provable) unnecessary dependencies
 - No (provable) redundant computation

15

Value number graph → 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 than one position?
- E.g.


```

      if ... then i := 1 else i := 2 end; x := i
      i := 0; while ... loop ...; i := i + 1; ... end; x := i
      
```

16

φ-Functions

- Solution:
 - Each assignment to a variable *a* defines a new version *a_n*,
 - 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 defining a new version (value number) of that variable

$$a_3 := \phi(a_1, a_2)$$
- E.g.

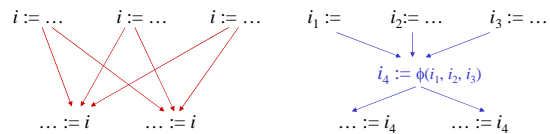

```

      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 all evaluated simultaneously for a block
 - guarantee that there is exactly one definition/assignment for each use of a variable
- Assignment $i_k := \phi(i_1, \dots, i_j)$ in a basic block indicates that the block has *k* direct predecessors in the control flow

17

Compact representation of dependencies

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



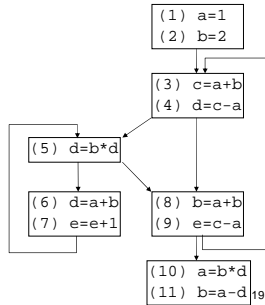
18

Example Program and Basic Block Graph

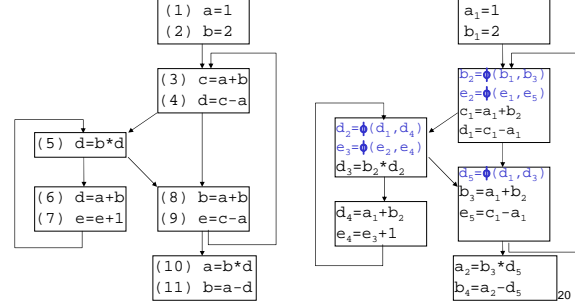
```

(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;

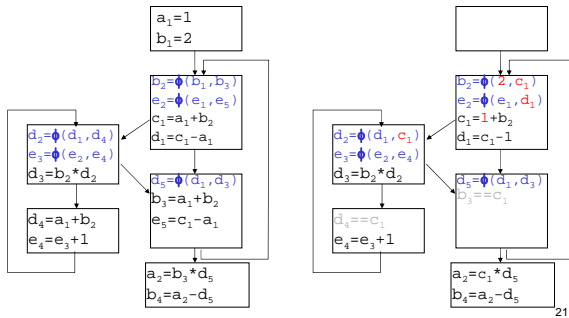
```



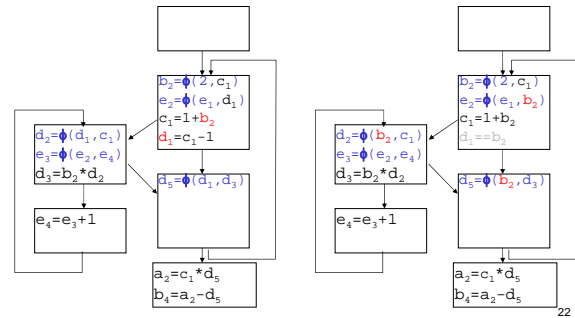
Basic Block and SSA Graph



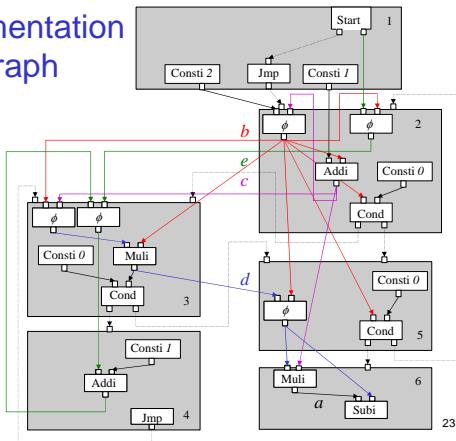
SSA-Graph before and after Constant and Copy Propagation



SSA-Graph before and after using Algebraic Identities



Implementation SSA graph

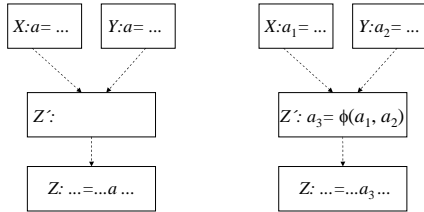


SSA properties

- P1: Typed in-/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 *Start*, *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 $B_x \rightarrow^* B_y$.
- P7a: (Special case of P7) operation y is a ϕ -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 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 ϕ -operation for a .

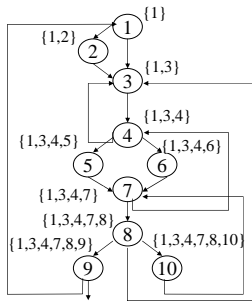
Property P8 revisited

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



25

Example



27

Discussion P8

- Defines, where to position ϕ -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 path $X \rightarrow^+ Z, Y \rightarrow^+ Z$ in the BB graph (P7). Let Z' be the first node common to both path then:
 - $a_3 := \phi(a_1, a_2)$ is assigned to Z'
 - $Z' \geq Z$ otherwise X, 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
 - $Z' \in DF(X, Y)$ i.e. Z' is in the dominance frontier of X and Y
- Since $a_3 := \phi(a_1, a_2)$ itself is a definition of a , the dominance frontiers $DF(X, Y, Z')$ must contain ϕ functions.
- Fixed point iteration required,
- Observation: ϕ -functions in the iterated dominance frontiers $DF^+(\dots)$ of the BB defining a variable

32

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: $X > Y$

$X > Y \Leftrightarrow X \geq Y \wedge X \neq Y$

Direct Dominance: $ddom(X)$

$X = ddom(Y) \Leftrightarrow X > Y \wedge \neg \exists Z: X > Z > Y$.

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.

(Iterated) Dominance Frontiers

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

$$DF_0 = DF(M),$$

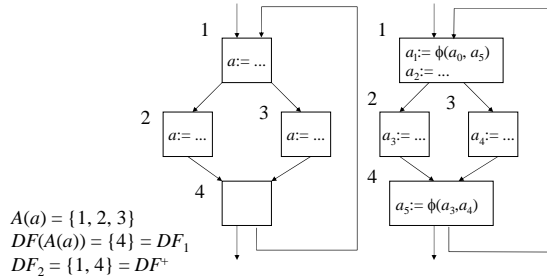
$$DF_{i+1} = DF\left(\bigcup_{k \in 0 \dots i} DF_k\right).$$

Proof (idea) of this observation

- $a_3 := \phi(a_1, a_2)$ cannot be defined before Z'
- X and Y , resp., must dominate all direct predecessors of Z' , otherwise there would be a use of a without previous definition. Hence $Z' \in DF(X, Y)$
- $a_3 := \phi(a_1, a_2)$ 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 .
- Iteration $DF^+(\dots)$ is required, as there is now a further definitions of a in a block Z' and $DF(X, Y) \subseteq DF(X, Y, Z')$

33

Example Property E8



34

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

35

Construction Theory

- Program of size n may contain $O(n)$ variables
- In the worst case there are $O(n)$ ϕ -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 ϕ -function and rename variables accordingly
- In practice easier constructions possible

36

Remainder Value Numbering

- (1) Initially: value number $vn(\text{constant}) = \text{constant}$; $vn(t) = \text{void}$ for all tuples t .
- (2) for all tuple t in program order:
 - case
 - (a) $t = ST \rightarrow local \leftarrow t'$
 - $vn(t) := vn(ST \rightarrow local \leftarrow vn(t'))$
 - if $vn(t) = \text{void}$:
 - $vn(LD \leftarrow local) := vn(t')$,
 - $vn(t) := \text{new value number}$,
 - generate: $ST \rightarrow local \leftarrow vn(t')$
 - (b) $t = LD \leftarrow local$
 - $vn(t) := vn(LD \leftarrow local)$.
 - if $vn(t) = \text{void}$:
 - $vn(t) := \text{new value number}$,
 - generate: $vn(t); t$
 - (c) $t = \tau t'$
 - $vn(t) := vn(\tau vn(t') vn(t'))$
 - if $vn(t) = \text{void}$:
 - $vn(t) := \text{new value number}$,
 - generate: $vn(t); \tau vn(t') vn(t')$.
 - (d) $t = \text{call proc } t' \dots$ -- analog (c) with $\tau = \text{call proc}$

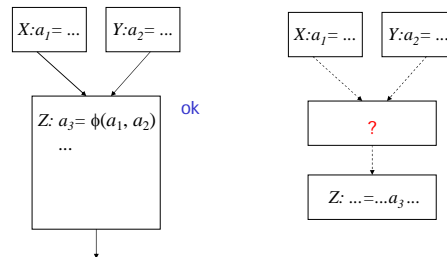
37

Extended Initialization

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

38

Extended Value Numbering



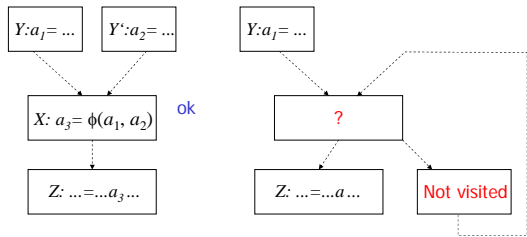
39

Extended Initialization

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

40

Extended Value Numbering



41

Extended Initialization

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

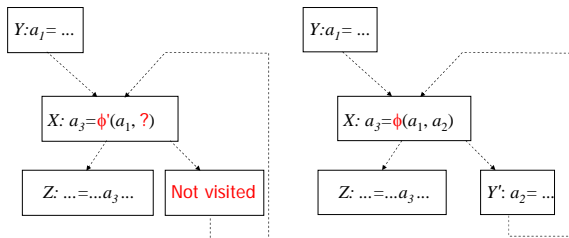
42

Eliminate/Mature ϕ' -Functions

- After value numbering is finished for each block X:
 - replace special 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

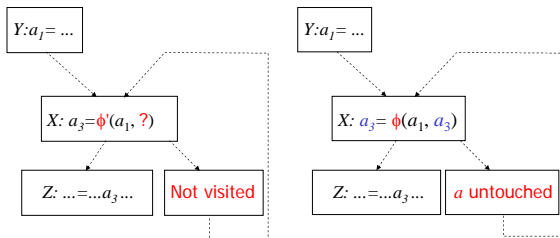
43

Example I: Mature ϕ' -Functions



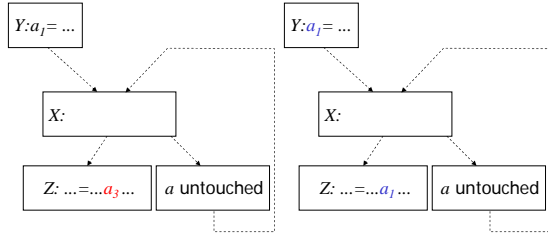
44

Example II: Mature ϕ' -Functions



45

Example III: Mature ϕ' -Functions



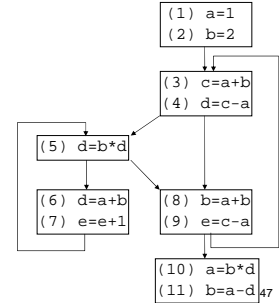
46

Example Program and BB Graph

```

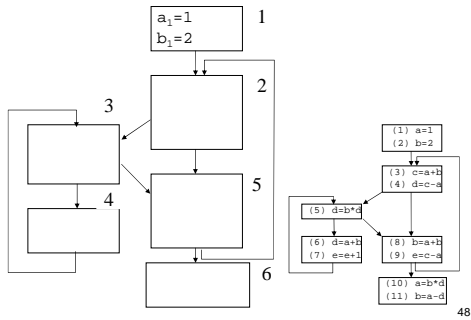
(1) a=1;
(2) b=2;
while true{
(3) c=a+b;
(4) if (d=c-a)
(5)   while (d=b*d){
(6)     d=a+b;
(7)     e=e+1;
(8)   }
(9)   b=a+b;
(10)  if (e=c-a) break;
(11) a=b*d;
}

```



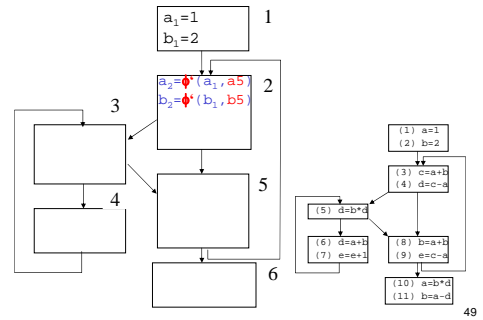
47

SSA Construction Block 1



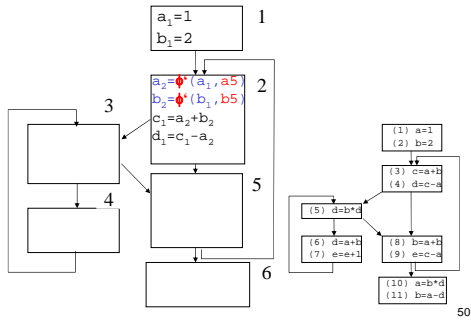
48

SSA Construction Block 2 - Initialization



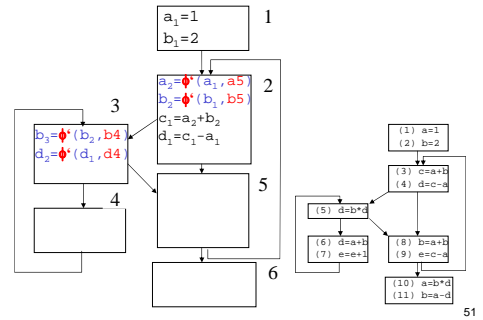
49

SSA Construction Block 2



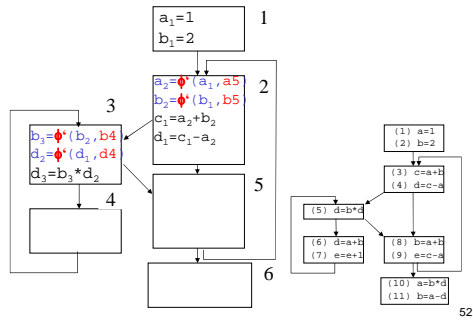
50

SSA Construction Block 3 - Initialization

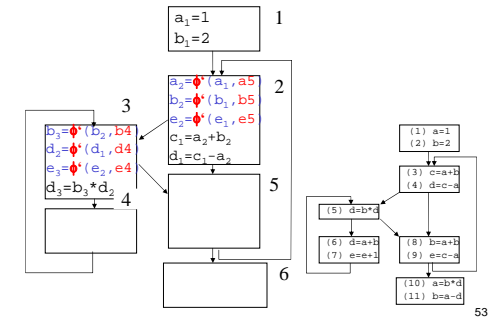


51

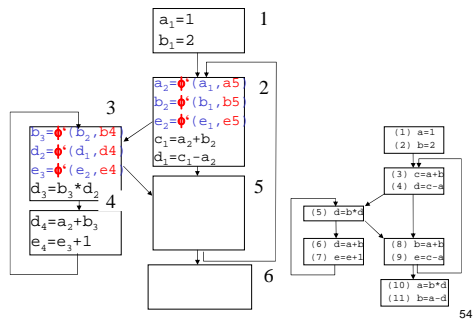
SSA Construction Block 3



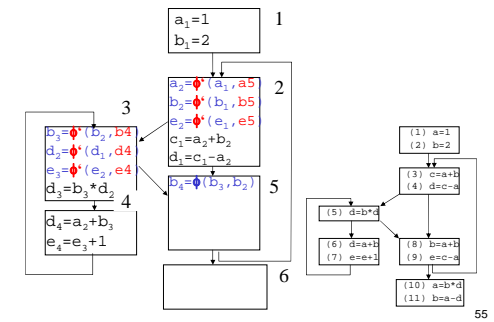
SSA Construction Block 4 - Initialization



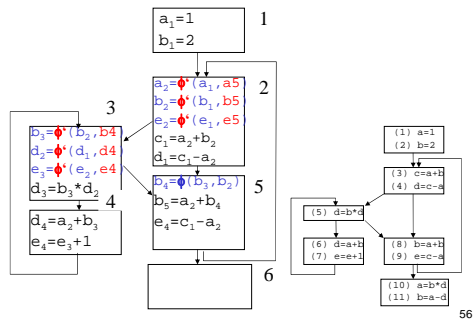
SSA Construction Block 4



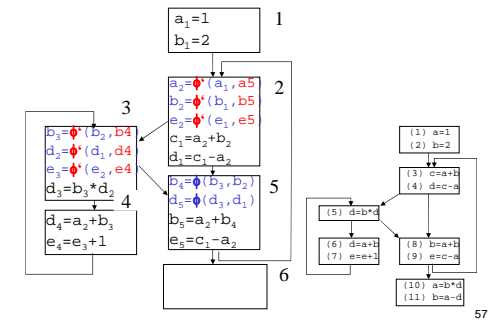
SSA Construction Block 5 - Initialization



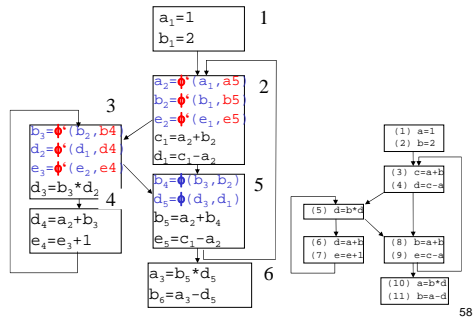
SSA Construction Block 5



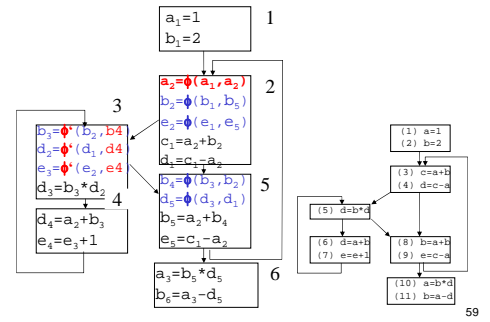
SSA Construction Block 6 - Initialization



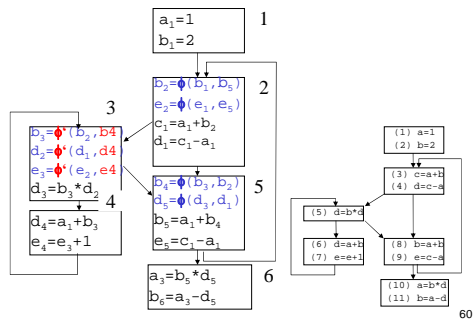
SSA Construction Block 6



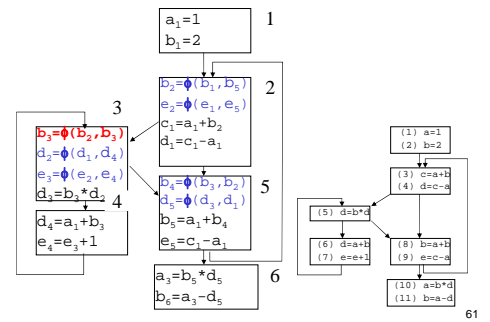
SSA Mature Block 2



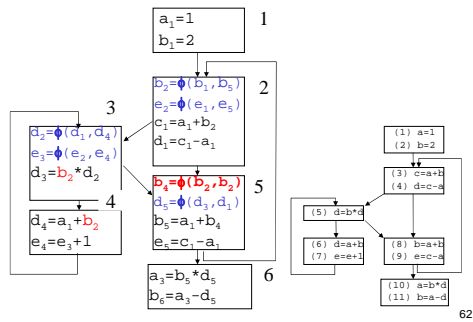
SSA Mature Block 2



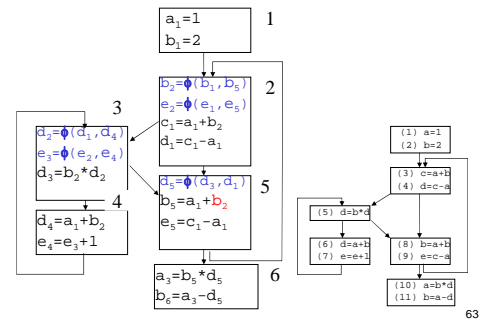
SSA Mature Block 3



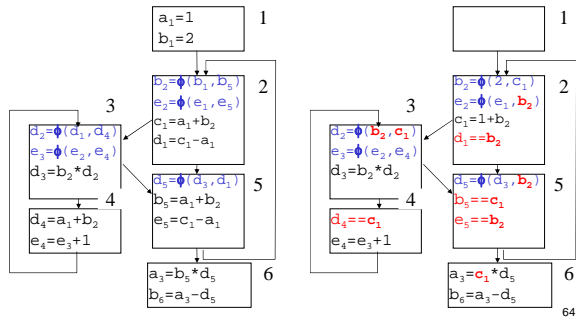
SSA Mature Block 3



SSA Mature Block 3



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 from syntactically different operations
 - Values are generated from syntactically 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

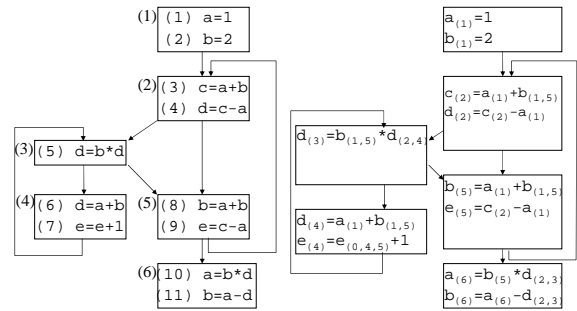
65

Construction Algorithm

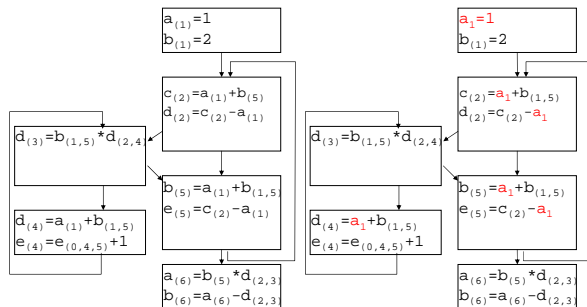
- Generate BB graph and perform Definition-Use-Analysis (data flow analysis) for all variables. Notations:
 - $v_{(i)} = \dots$ Variable v in statement (i) defined
 - $u_{(i)} = \dots, v_{(x,y,z), \dots}$ Variable v in statement (i) used with may reaching definitions in statements (x,y,z, \dots)
- statements can be abstracted to blocks
- Set $v_{(i)} = u_{(i)}$ for all $v_{(i)}, u_{(i)}$ in the program
- Iterate until a fixed point over:
 - Set $v_{(i)} \neq u_{(i)}$ for:
 - $v_{(i)} = c$ and $u_{(i)} \neq c$
 - $v_{(i)} = op_f(\dots)$ and $u_{(i)} \neq op_f(\dots)$
 - $v_{(i)} = op_f(x_1, y_1)$ and $u_{(i)} = op_f(x_2, y_2)$ 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)

66

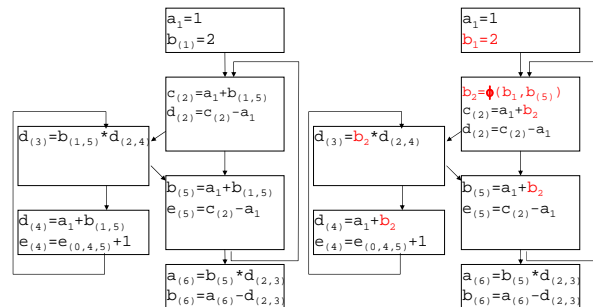
Optimistic SSA Construction



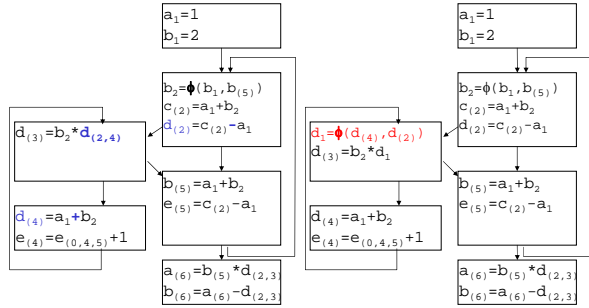
Optimistic SSA Construction



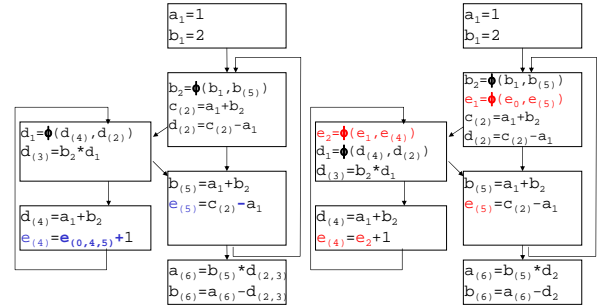
Optimistic SSA Construction



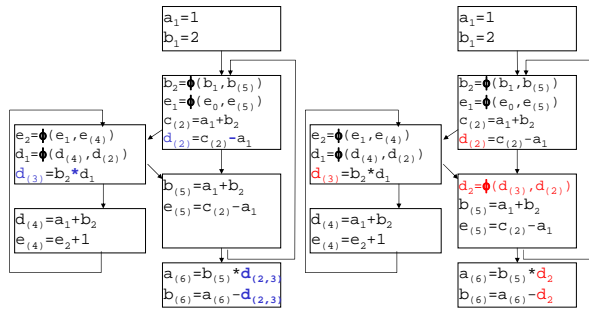
Optimistic SSA Construction



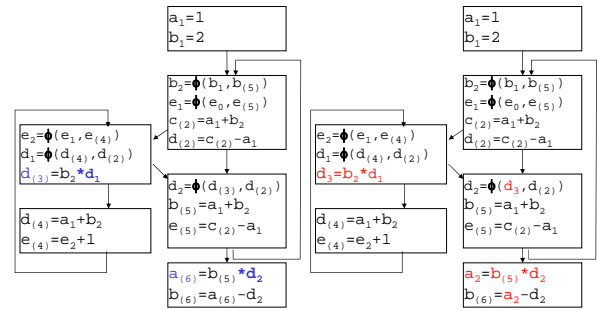
Optimistic SSA Construction



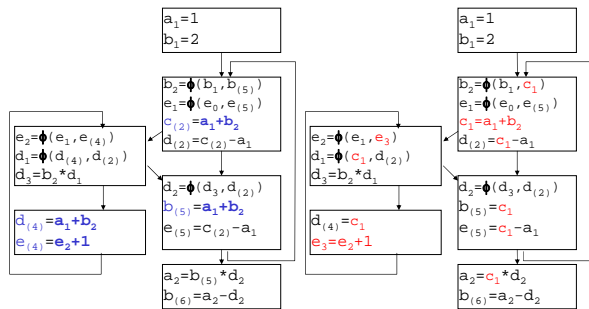
Optimistic SSA Construction



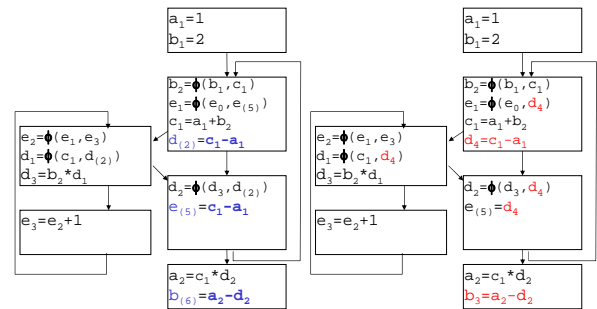
Optimistic SSA Construction



Optimistic SSA Construction

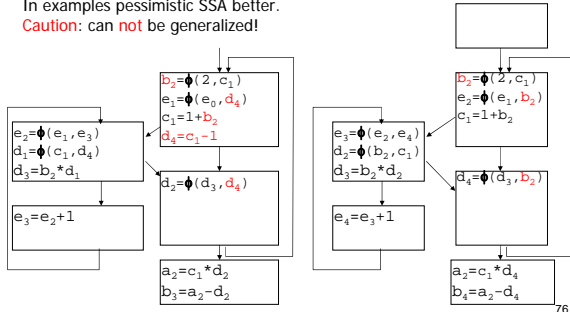


Optimistic SSA Construction



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 ϕ -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)

77

Minimal SSA-Form

- Insight:
 - ϕ -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 ϕ -function $a_0 := \phi(a_1, a_2, \dots)$ 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).

78

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 ϕ) using the data structures known from value numbering
- Left-Right Traversal (3. Round):
 - Mature ϕ -functions
 - generate SSA for non empty blocks
- Further eliminations on SSA graph

79

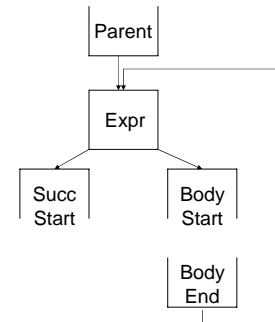
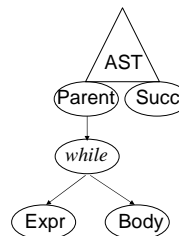
SSA from AST

- Rounds 1+2 on one left-right tree traversal if liveness 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

80

SSA from AST

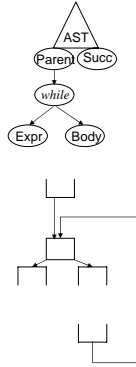
- while AST and BB graph



81

SSA from AST

- while actions
 - Finalize current block B(Parent)
 - Create a new current block B(Expr)
 - Add control flow B(Parent) 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 code for Body
 - After return current block is $B_{end}(Body)$, finalize it
 - 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



82

Deconstruction of SSA

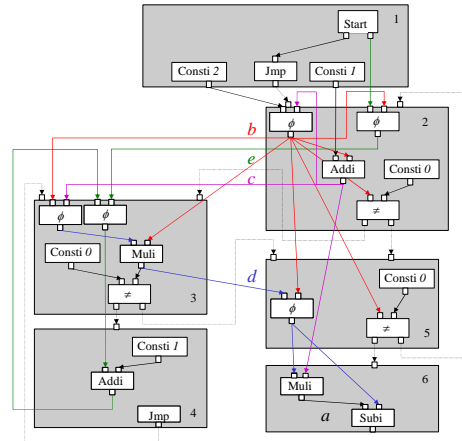
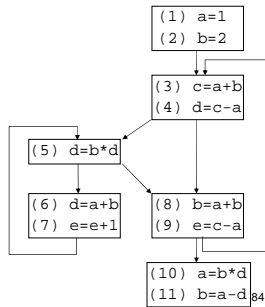
- Serialize the SSA graph
- Replace data dependency edges by variables
- Remove ϕ -functions:
 - Define a new variable
 - Copy from value in predecessor basic blocks (requires possibly new blocks on some edges)
 - Perform copy propagation to avoid unnecessary copy operations
- Allocate registers for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

83

Example Program and BB Graph

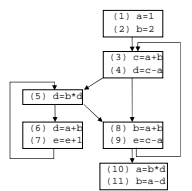
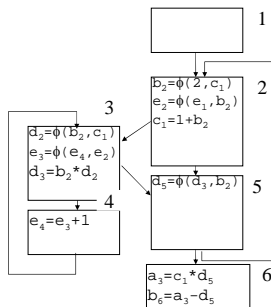
```

(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)    }
(9)  b=a+b;
(10) if (e=c-a) break;
(11) a=b*d;
(12) b=a-d;
}
    
```



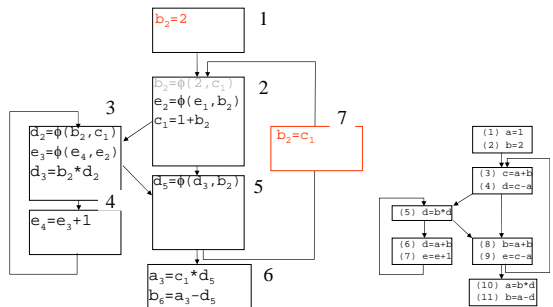
85

Introduce Variables for Edges



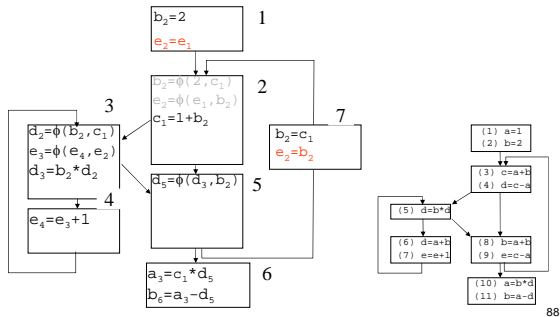
86

Remove ϕ -functions

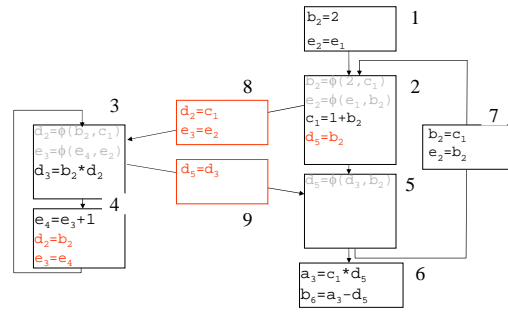


87

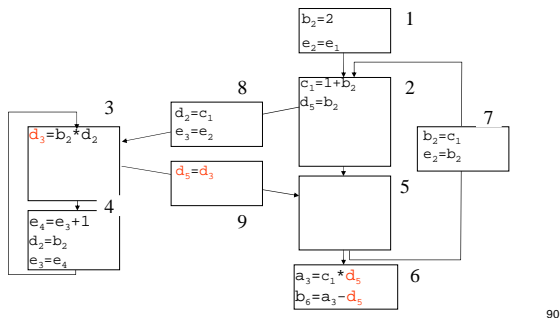
Remove ϕ -functions



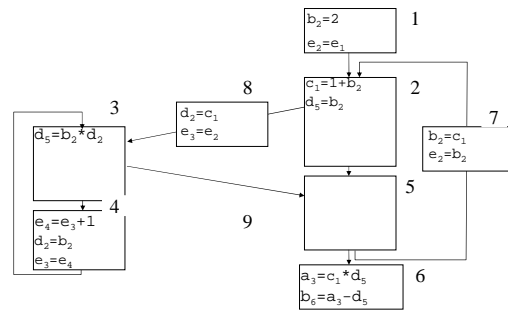
Remove ϕ -functions



Copy Propagation

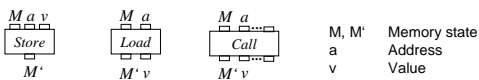


Remove Empty Block

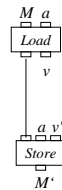


Memory SSA

- 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



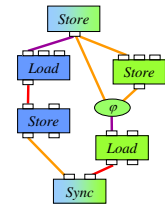
Why Load Defines Memory?



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
 - Sync** operation unifies disjoint memory fragments
 - Like ϕ -functions but sync is strict



94

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

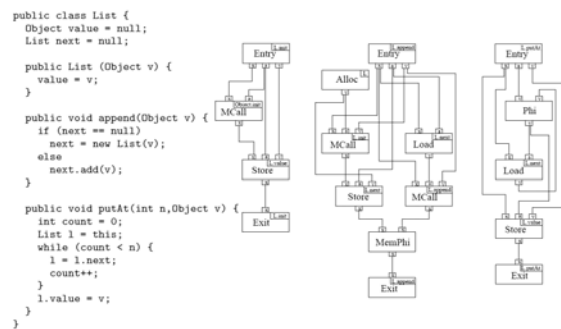
95

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)

96

Example Points-to-SSA



Cliffhanger from the first day

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

98

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

99

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
 - In principle, additional def-use analysis avoids this problem
 - 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

100

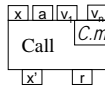
Fast and Precise P2A

- Data values
 - allocation site abstract from objects 0
 - abstract heap memory: $0 \times F \rightarrow 0$ (F set of fields) heap size: $l \ n \ t$
- 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
- Transfer function:
 - update the heap according to the abstract semantics of the
 - special ϕ -nodes: \cup on 0 values and \max on $l \ n \ t$ values, resp.
- Initialization: ϕ for 0 ports and 0 for $l \ n \ t$ ports, resp.
- Simulated execution

101

Recall: Simulated Execution

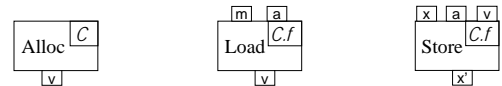
- Interleaving of **process method** and update **call nodes' transfer function**
- Processes a method:
 - Starts with main,
 - propagates data values analog the edges in P2-SSA graph
 - updates the heap and the data values in the nodes according to their transfer functions
 - If node type is a call then ...
- Call nodes' transfer function; if input changes:
 - Interrupts the processing of a caller method
 - Propagates arguments $Pt(v_1 \dots v_n)$ 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 ...



102

Example transfer functions

if input changes, update as below else skip:



$$Pt(v) = \{a_i\} \text{ (unique } i) \quad Pt(v) = \bigcup_{a \in Pt(a)} \text{heap}(a, f) \quad \forall a \in Pt(a): \text{heap}(a, f) \cup = Pt(v) \\ \text{size}(x') = \text{heap.size}$$

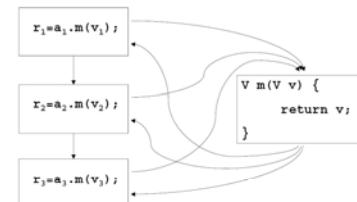
103

Flow-sensitivity

- The **Points-to-SSA** approach has two features that contribute to flow-sensitivity:
 - Locally flow-sensitive: We have SSA edges imposing the correct ordering among all operations (calls and field accesses) within a method.
 - 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.

104

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

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

105

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 $a.m(v_1, \dots)$ 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:
 - $context: [m, call\ id, a, v_1, \dots, v_n] \mapsto [c_1, \dots, c_p]$
 - Often it is just one context per call stack situations
 - $context: [m, call\ id, a, v_1, \dots, v_n] \mapsto c_i$

106

Context-sensitive call handling

```

Call(m, call, xin, a, v1, ..., vn) → [xout, r]
[xout, r] = [0, ⊥]
for all c ∈ context [m, call id, a, v1, ..., vn] do
  if [xin, a, v1, ..., vn] ⊆ previous arguments (m, c) then
    r = previous return (m, c)
    xout = xin
  else
    args = previous args (m, c) ⊔ [xin, a, v1, ..., vn]
    args previous (m, c) = args
    [xout, r] = [xout, r] ⊔ processMethod(m, args)
    previous return (m, c) = r
  end if
end for
return [xout, r]
    
```

107

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 $(m, this)$
- where $this \in 2^0$ 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.

108

Examples: Precision

In favor of Object-sensitivity

- Method definitions:
 - $m()$ {field = this;}
 - $v.n()$ {return field;}
- Call 1:
 - $a_1 = \{o_1, o_2\}$
 - $a_1.m()$
- Call 2:
 - $a_2 = \{o_1\}$
 - $r_2 = a_2.n()$

In favor of This-sensitivity

- Method definition:
 - $v.m(v)$ {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: $Pt(r_2) = \{o_1\}$.
- This-sensitivity: $Pt(r_2) = \{o_1, o_2\}$.
- Object-sensitivity: $Pt(r_2) = \{o_3, o_4\}$.
- This-sensitivity: $Pt(r_2) = \{o_4\}$.

109

Results

- Fast and accurate P2A
 - Points-to SSA \Rightarrow locally flow sensitive PTA
 - Simulated execution \Rightarrow 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 j avac 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) of classic P2A
- Shows in clients analyses like synchronization removal and static garbage collection (escape and side effect analysis)

110

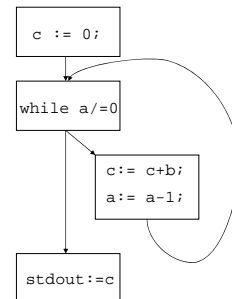
Assignment I

Compute the SSA graph for:

```

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



111

Assignment II

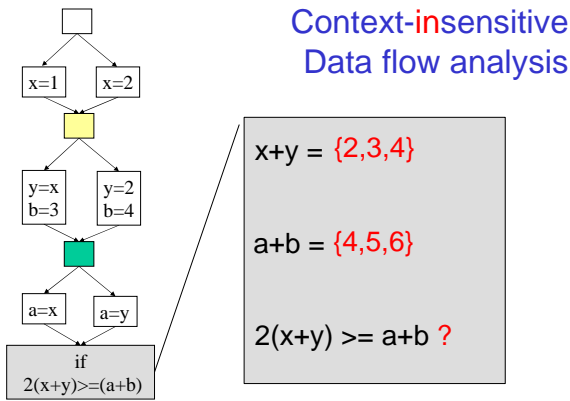
- Leave the SSA graph for the program from [Assignment I](#)
 - Remove ϕ functions
 - Perform copy propagation
 - Compare the produced program to the original one
- Perform register allocation for 3 registers
 - Do it mechanically using live analysis and graph coloring
 - Why is it immediately clear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

Context-sensitive analysis

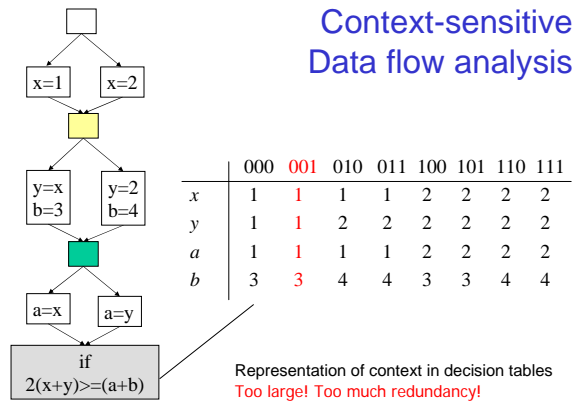
- Distinguish different call context of a method
- In general: Distinguish different execution path to a program point
 - Exponentially (in program size) many path in a sequential program
 - Exponentially many analysis values
- (Let away the problem of analysis) How to capture the results efficiently?

112

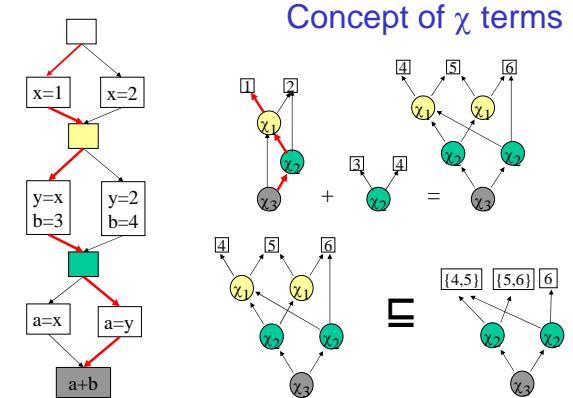
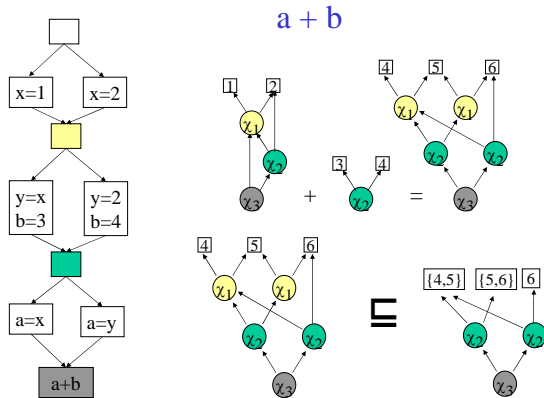
113



114



115



Advantages of χ -Terms

- Compact representation of context sensitive information
 - Delayed widening (abstract interpretation) of terms until no more memory: no unnecessary loss of information
- $$\chi_3(\chi_2(\chi_1(4,5),\chi_1(5,6)),\chi_2(\chi_1(4,5),6)) \sqsubseteq \chi_3(\chi_2(\{4,5\},\{5,6\}),\chi_2(\{4,5\},6)) \sqsubseteq \chi_3(\{4,5,6\},\{4,5,6\}) = \{4,5,6\} \sqsubseteq [4,6] \sqsubseteq \top$$
- χ -Terms are implementation of decision diagrams.
 - Adaptation of OBDD implementation techniques.
 - Symbolic computation with χ -terms (simplification of terms).
 - Transition of the idea of SSA ϕ -function to data-flow values
 - Especially interesting for [address values](#) to distinguish [memory partitions](#) as long as possible

118

Data-Flow Analyses (DFA) on SSA

- Each SSA node type n has a concrete semantics $[n]$,
 - On execution of n , map inputs i to outputs o and $o = [n](i)$
 - inputs i and outputs o are records of typed values (P1)
 - $[n] :: \text{type}(i_1) \times \dots \times \text{type}(i_k) \rightarrow \text{type}(o_1) \times \dots \times \text{type}(o_l)$
- Each static data-flow analysis abstracts from concrete semantics and values
 - Abstract semantics T_n is called transfer function of node type n
 - On analysis of n , map abstract analysis inputs $a(i)$ to abstract analysis outputs $a(o)$ and $a(o) = T_n(a(i))$
 - $T_n :: \text{type}(a(i_1)) \times \dots \times \text{type}(a(i_k)) \rightarrow \text{type}(a(o_1)) \times \dots \times \text{type}(a(o_l))$
- For each abstract semantics type $A = \text{type}(a(\bullet))$ – analysis universe – there is a partial order relation \sqsubseteq and a meet operation \sqcup (supremum),
 - $\sqsubseteq :: A \times A$
 - $\sqcup :: A \times A \rightarrow A$ with $\sqcup = T_n$
 - (A, \sqsubseteq) defines a complete partial order

119

DFA on SSA (cont.)

- Provided that transfer functions are monotone: $x \leq y \Rightarrow T_n(x) \leq T_n(y)$ with $x, y \in A_1 \times \dots \times A_k$ and \leq defined element-wise with \sqsubseteq of the respective abstract semantics type
- Following iteration terminates in a unique fixed point
 - Initialize the input of each node the SSA graph with the smallest (bottom) element of the Lattice/CPO corresponding its abstract semantics type
 - Initialize the *start* nodes of the SSA graph with proper abstract values of the corresponding its abstract types
 - Attach each node with its corresponding transfer function
 - Uniform randomly compute abstract output values
 - (Fewer updates use SCC and interval analysis to determine an traversal strategy, backward problems analyzed analogously)

120

Generalization to χ -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](#)
- χ -term algebras X over abstract semantic values $a \in A$ introduced
 - $a \in A \Rightarrow a \in X_A$
 - $t_1, t_2 \in X \Rightarrow \chi(t_1, t_2) \in X_A$
 - Induces sensitive CPO for abstract values (X_A, \sqsubseteq) and for $a_1, a_2 \in A$ and $t_1, t_2, t_3, t_4 \in X$:
 $a_1 \sqsubseteq a_2 \Rightarrow a_1 \sqsubseteq a_2$
 $\chi(a_1, a_2) \sqsubseteq a_1 \sqcup a_2$
 $t_1 \sqsubseteq t_3, t_2 \sqsubseteq t_4 \Rightarrow \chi(t_1, t_2) \sqsubseteq \chi(t_3, t_4)$
- New transfer functions induced

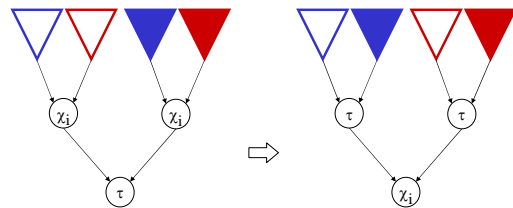
121

New transfer functions

- ϕ -node's transfer functions:
 - Insensitive: $T_\phi = \sqcup = a(i_1) \sqcup \dots \sqcup a(i_k)$
 - Sensitive: $S_\phi = \sqcup_b = a'(i_1) \sqcup_b \dots \sqcup_b a'(i_k)$ with b block number of ϕ -node and for $a_1, a_2 \in A$ and $t_1, t_2, t_3, t_4 \in X$:
 $a_1 \sqcup_b a_2 = \chi_b(a_1, a_2)$
 $\chi_x(t_1, t_2) \sqcup_b \chi_y(t_3, t_4) = \chi_b(t_1 \sqcup t_2, t_3 \sqcup_b t_4)$ iff $x=b$ (cases $y = b$ analog)
 $\chi_x(t_1, t_2) \sqcup_b \chi_y(t_3, t_4) = \chi_b(t_1 \sqcup_b t_2, t_3 \sqcup_b t_4)$ otherwise
- Ordinary operation's (τ node's) transfer functions:
 - Insensitive (w.l.o.g. binary operation): $T_\tau :: A_a \times A_b \rightarrow A_c$
 - Sensitive: $S_\tau :: X_{A_a} \times X_{A_b} \rightarrow X_{A_c}$ and for $a_1, a_2 \in A_a, A_b$ and $t_1, t_2 \in X_a, X_b$:
 $S_\tau(a_1, a_2) = T_\tau(a_1, a_2)$
 $S_\tau(\chi_x(t_1, t_2), \chi_y(t_3, t_4)) = \chi_c(S_\tau(\text{cof}(\chi_x(t_1, t_2), \chi_{kx}, 1), \text{cof}(\chi_y(t_3, t_4), \chi_{ky}, 1))), S_\tau(\text{cof}(\chi_x(t_1, t_2), \chi_{kx}, 2), \text{cof}(\chi_y(t_3, t_4), \chi_{ky}, 2)))$
 with k larger of x, y and cof is the co-factorization

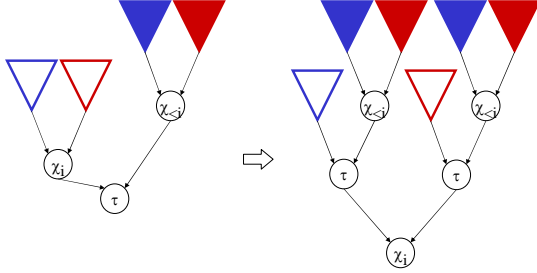
122

Sensitive Transfer Schema (case a)



123

Sensitive Transfer Schema (case b)



124

Co-factorization

- $\text{cof}(\chi_x(t_1, t_2), \chi_k, i)$ selects the i -th branch of a χ -term if $\chi_x = \chi_k$ and returns the whole χ -term, otherwise
- $\text{cof}(\chi_x(t_1, t_2), \chi_k, i) = \chi_x(t_1, t_2)$ iff $k > x$
- $\text{cof}(\chi_x(t_1, t_2), \chi_k, 1) = t_1$ iff $k = x$
- $\text{cof}(\chi_x(t_1, t_2), \chi_k, 2) = t_2$ iff $k = x$

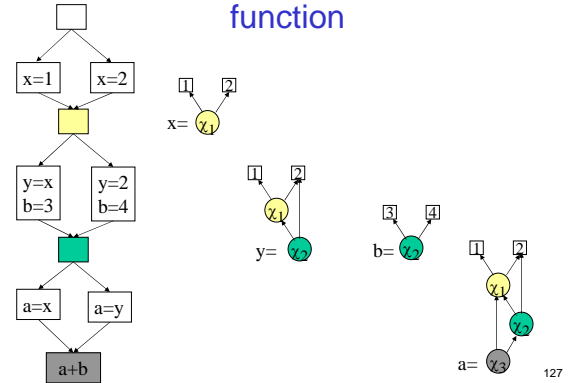
125

Example revisited: insensitive

- SSA node \oplus
- Semantic
 - $[\oplus] :: \text{Int} \times \text{Int} \rightarrow \text{Int}$
 - $[\oplus](a, b) = a + b$
- Abstract Int values $\{\perp, 1, 2, \dots, \text{maxint}, \top\}$
- Context-insensitive transfer function:
 - $T_{\oplus}(\perp, x) = T_{\oplus}(x, \perp) = \perp$
 - $T_{\oplus}(\top, x) = T_{\oplus}(x, \top) = \top$
 - $T_{\oplus}(a, b) = [\oplus](a, b) = a + b$ for $a, b \in \text{Int}$
- Context-insensitive meet function
 - $T_{\oplus}(\perp, x) = T_{\oplus}(x, \perp) = x$
 - $T_{\oplus}(\top, x) = T_{\oplus}(x, \top) = \top$
 - $T_{\oplus}(x, x) = x$
 - $T_{\oplus}(x, y) = \top$

126

Context-sensitive ϕ -node transfer function



127

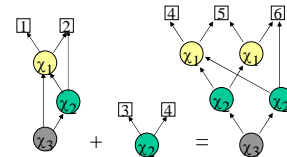
Context-sensitive $a \oplus b$

$$\begin{aligned}
 & S_{\oplus}(\chi_3(\chi_1(1,2), \chi_2(\chi_1(1,2), 2)), \chi_2(3,4)) \\
 &= \chi_3(S_{\oplus}(\text{cof}(\chi_3(\chi_1(1,2), \chi_2(\chi_1(1,2), 2)), \chi_3, 1), \text{cof}(\chi_2(3,4), \chi_3, 1)), \\
 & \quad S_{\oplus}(\text{cof}(\chi_3(\chi_1(1,2), \chi_2(\chi_1(1,2), 2)), \chi_3, 2), \text{cof}(\chi_2(3,4), \chi_3, 2))) \\
 &= \chi_3(S_{\oplus}(\chi_1(1,2), \chi_2(3,4)), \\
 & \quad S_{\oplus}(\chi_2(\chi_1(1,2), 2), \chi_2(3,4))) \\
 &= \chi_3(S_{\oplus}(S_{\oplus}(\text{cof}(\chi_1(1,2), \chi_2, 1), \text{cof}(\chi_2(3,4), \chi_2, 1)), \\
 & \quad S_{\oplus}(\text{cof}(\chi_1(1,2), \chi_2, 2), \text{cof}(\chi_2(3,4), \chi_2, 2))), \\
 & \quad \chi_2(S_{\oplus}(\text{cof}(\chi_2(\chi_1(1,2), 2), \chi_2, 1), \text{cof}(\chi_2(3,4), \chi_2, 1)), \\
 & \quad S_{\oplus}(\text{cof}(\chi_2(\chi_1(1,2), 2), \chi_2, 2), \text{cof}(\chi_2(3,4), \chi_2, 2)))) \\
 &= \chi_3(S_{\oplus}(S_{\oplus}(\chi_1(1,2), 3), \\
 & \quad S_{\oplus}(\chi_1(1,2), 4)), \\
 & \quad \chi_2(S_{\oplus}(\chi_1(1,2), 3), \\
 & \quad S_{\oplus}(2,4)))
 \end{aligned}$$

128

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

$$\begin{aligned}
 & S_{\oplus}(\chi_3(\chi_1(1,2), \chi_2(\chi_1(1,2), 2)), \chi_2(3,4)) = \dots \\
 &= \chi_3(\chi_2(S_{\oplus}(\chi_1(1,2), 3), S_{\oplus}(\chi_1(1,2), 4)), \chi_2(S_{\oplus}(\chi_1(1,2), 3), S_{\oplus}(2,4))) \\
 &= \chi_3(\chi_2(S_{\oplus}(\chi_1(1,2), 3), S_{\oplus}(\chi_1(1,2), 4)), \chi_2(S_{\oplus}(\chi_1(1,2), 3), 6)) \\
 &= \dots \\
 &= \chi_3(\chi_2(\chi_1(4,5), \chi_1(5,6)), \chi_2(\chi_1(4,5), 6))
 \end{aligned}$$

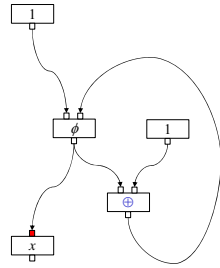


129

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 value at the entry of node x ?
- Perform **context-sensitive** data-flow analysis (using the definitions on the previous slides). What is 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!



130