Outline

- Introduction to SSA
- Construction, Destruction
- How to capture analysis results
- Optimizations
  - Classic analyses and optimizations on SSA representations
  - Heap analyses and optimizations

1

1

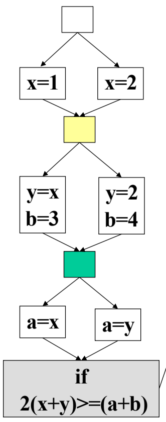
Context-sensitive analysis revisited

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

2

2

Context-insensitive Data flow analysis



x+y = {2,3,4}

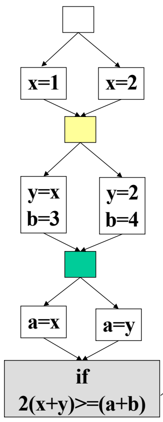
a+b = {4,5,6}

2(x+y) >= a+b ?

3

3

Context-sensitive Data flow analysis



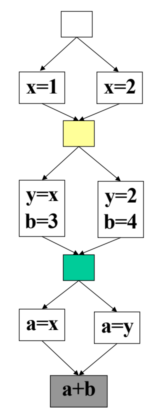
	000	001	010	011	100	101	110	111
x	1	1	1	1	2	2	2	2
y	1	1	2	2	2	2	2	2
a	1	1	1	1	2	2	2	2
b	3	3	4	4	3	3	4	4

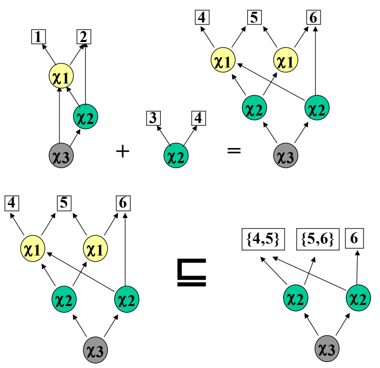
Representation of context in decision tables
Exponentially large! Too much redundancy!

4

4

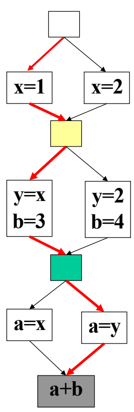
a + b

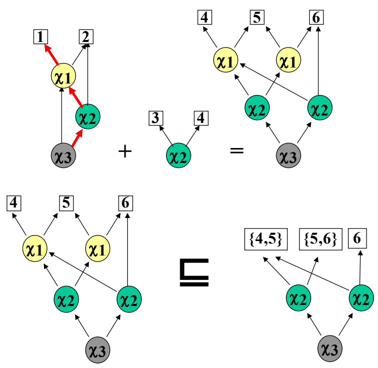




5

Concept of χ terms





6

Advantages of χ-Terms

- Compact representation of context sensitive information
- Delayed widening (recall 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
- Particularly interesting for address values in order to distinguish memory partitions as long as possible

7

7

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)
  - $[n]$ :: type(i_1) $\times \ldots \times$ type(i_k) $\rightarrow$ type(o_1) $\times \ldots \times$ 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 :: type($a(\, i_1)$) $\times \ldots \times$ type($a(\, i_k)$) $\rightarrow$ type($a(\, o_1)$) $\times \ldots \times$ type($a(\, o_l)$)
- For each abstract analysis semantics value representation U = type($a(\bullet)$) – analysis universe – there is a partial order relation $\subseteq$ and a meet operation $\cup$ (supremum),
  - $\subseteq : U \times U$
  - $\cup : U \times U \rightarrow U$ with $\cup = T_\phi$
  - $(U, \subseteq)$ defines a complete partial order

8

8

DFA on SSA (cont.)

- Provided that transfer functions are monotone: $x \leq y \Rightarrow T_n(x) \leq T_n(y)$ with $x,y \in U_1 \times \ldots \times U_k$ and $\leq$ defined element-wise with $\subseteq$ of the respective analysis values U_i
- The 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 with its abstract types
  - Attach each node with its corresponding transfer function
  - Compute abstract output values of the nodes
    - Any fair random selection of nodes terminates
    - Fewer updates use SCC and interval analysis to determine a traversal strategy, backward problems analyzed analogously)

9

9

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 $a \in U$ introduced
  - $a \in U \Rightarrow a \in \mathrm{X}_U$
  - $t_1, t_2 \in \mathrm{X} \Rightarrow \chi_i(t_1, t_2) \in \mathrm{X}_U$
  - Induces sensitive CPO for abstract values $(\mathrm{X}_U, \sqsubseteq)$ and for $a_1, a_2 \in U$ and $t_1, t_2, t_3, t_4 \in \mathrm{X}_U$:
    $a_1 \subseteq a_2 \Rightarrow a_1 \sqsubseteq a_2$
    $\chi_i(a_1, a_2) \sqsubseteq a_1 \cup a_2$
    $t_1 \sqsubseteq t_3,\; t_2 \sqsubseteq t_4 \Rightarrow \chi_i(t_1, t_2) \sqsubseteq \chi_i(t_3, t_4)$
- New transfer functions on X_U are induced

10

10

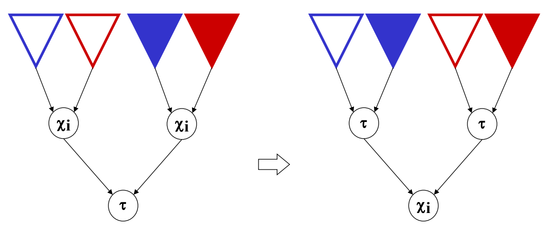
New transfer functions

- ϕ-node' s transfer functions:
  - Insensitive: $T_\phi = \cup = a(\, i_1) \cup \ldots \cup a(\, i_k)$
  - Sensitive: $S_\phi = \chi_b\,(a'(i_1),\; \ldots\; a'(i_k))$ with b block number of the ϕ-node
- Ordinary operation's (τ node's) transfer functions:
  - Insensitive (w.l.o.g. binary operation): $T_\tau : U_a \times U_b \rightarrow U_c$
  - Sensitive: $S_\tau : \mathrm{X}_a \times \mathrm{X}_b \rightarrow \mathrm{X}_c$ and for $a_1, a_2 \in U_a, U_b$ and $t_1, t_2 \in \mathrm{X}_a, \mathrm{X}_b$:
    $S_\tau\,(a_1\,,\, a_2\,) = T_\tau\,(a_1\,,\, a_2\,)$
    $S_\tau\,(\,\chi_x(t_1, t_2)\,,\, \chi_y(t_3, t_4)\,) = \chi_k(S_\tau(\mathrm{cof}(\chi_x(t_1,t_2),\chi_k,1),\mathrm{cof}(\chi_y(t_3,t_4),\chi_k,1)),$
    $S_\tau(\mathrm{cof}(\chi_x(t_1,t_2),\chi_k,2),\mathrm{cof}(\chi_y(t_3,t_4),\chi_k,2)))$
    with k larger of x,y and cof the co-factorization

11

11

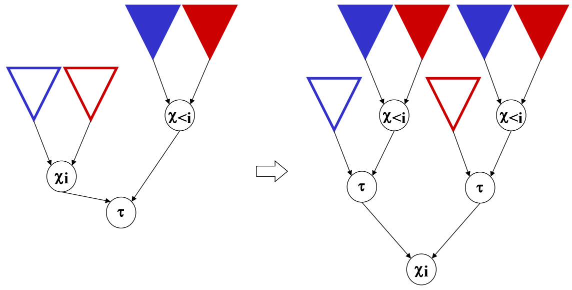
Sensitive Transfer Schema (case a)



12

12

Sensitive Transfer Schema (case b)



13

13

Co-factorization

- cof($\chi_x(t_1,t_2),\chi_k,i$) selects the i-th branch of a χ-term if χ_x=χ_k and returns the whole χ-term, otherwise

- cof($\chi_x(t_1,t_2),\chi_k,1$) $= t_1$ iff $k = x$ (case a)
- cof($\chi_x(t_1,t_2),\chi_k,2$) $= t_2$ iff $k = x$ (case a)
- cof($\chi_x(t_1,t_2),\chi_k,i$) $= \chi_x(t_1,t_2)$ iff $k > x$ (case b)

14

14

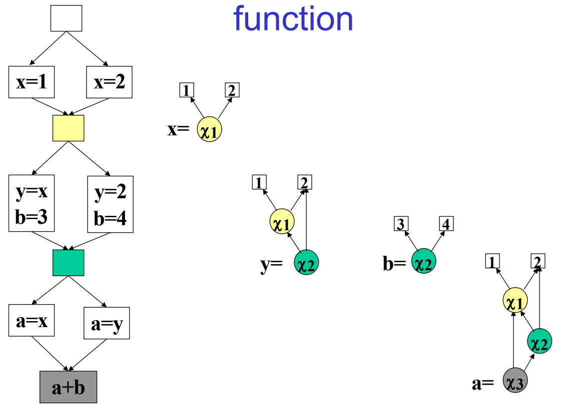
Example revisited: insensitive

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

15

15

Context-sensitive ϕ-node transfer function



16

16

Context-sensitive a $\oplus$ b

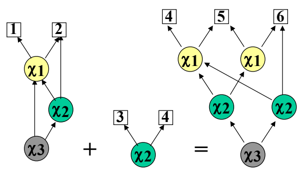
$S_\oplus$($\chi_3(\chi_1(1,2),\chi_2(\chi_1(1,2),2))$, $\chi_2(3,4)$)
$= \chi_3($ $S_\oplus$(cof($\chi_3(\chi_1(1,2),\chi_2(\chi_1(1,2),2))$, χ_3,1),cof($\chi_2(3,4)$, χ_3,1)),
$S_\oplus$(cof($\chi_3(\chi_1(1,2),\chi_2(\chi_1(1,2),2))$, χ_3,2),cof($\chi_2(3,4)$, χ_3,2)))
$= \chi_3($ $S_\oplus$($\chi_1(1,2)$,$\chi_2(3,4)$),
$S_\oplus$($\chi_2(\chi_1(1,2),2)$, $\chi_2(3,4)$))
$= \chi_3($ χ_2($S_\oplus$(cof($\chi_1(1,2)$, χ_2,1),cof($\chi_2(3,4)$, χ_2,1)) ,
$S_\oplus$(cof($\chi_1(1,2)$, χ_2,2),cof($\chi_2(3,4)$, χ_2,2))),
χ_2($S_\oplus$(cof($\chi_2(\chi_1(1,2),2)$, χ_2,1),cof($\chi_2(3,4)$, χ_2,1)) ,
$S_\oplus$(cof($\chi_2(\chi_1(1,2),2)$, χ_2,2),cof($\chi_2(3,4)$, χ_2,2))))
$= \chi_3($ χ_2($S_\oplus$($\chi_1(1,2)$, 3) ,
$S_\oplus$($\chi_1(1,2)$,4)),
χ_2($S_\oplus$($\chi_1(1,2)$,3),
$S_\oplus$(2,4)))

17

17

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

$S_\oplus$($\chi_3(\chi_1(1,2),\chi_2(\chi_1(1,2),2))$, $\chi_2(3,4)$) = …
$= \chi_3($ χ_2($S_\oplus$($\chi_1(1,2)$, 3), $S_\oplus$($\chi_1(1,2)$,4)), χ_2($S_\oplus$($\chi_1(1,2)$,3), $S_\oplus$(2,4)))
$= \chi_3($ χ_2($S_\oplus$($\chi_1(1,2)$, 3), $S_\oplus$($\chi_1(1,2)$,4)), χ_2($S_\oplus$($\chi_1(1,2)$,3), 6))
= …
$= \chi_3($ χ_2($\chi_1(4,5)$, $\chi_1(5,6)$), χ_2($\chi_1(4,5)$, 6))



18

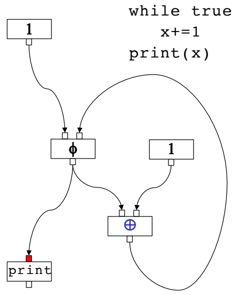
18

Example

Given the SSA fragment on the left

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

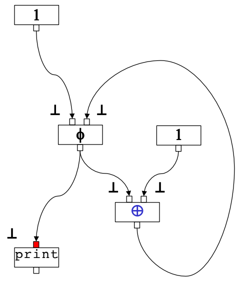
```
x=1
while true
    x+=1
print(x)
```



19

Example (cont'd)

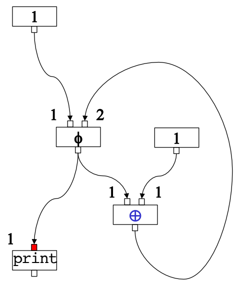
- Context-insensitive, initial situation:



20

Example (cont'd)

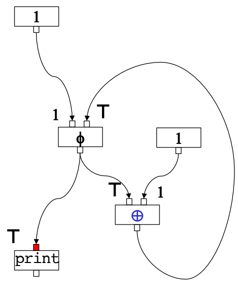
- Context-insensitive, after first iteration:



21

Example (cont'd)

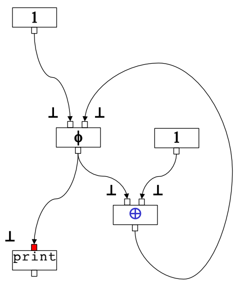
- Context-insensitive, after second iteration (stable):



22

Example (cont'd)

- Context-sensitive, initial situation:

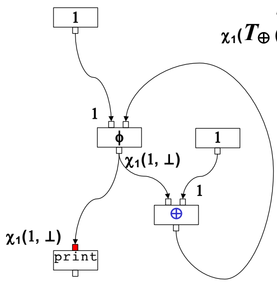


23

Example (cont'd)

- Context-sensitive, after first iteration:

$$S_{\oplus}(\chi_1(1, \perp), 1) = \chi_1(T_{\oplus}(1, 1), T_{\oplus}(\perp, 1)) = \chi_1(2, \perp)$$



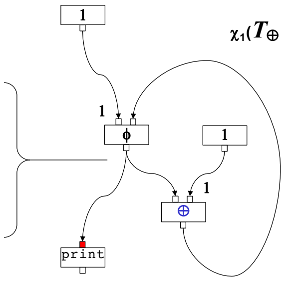
24

Example (cont'd)

- Context-sensitive, after second iteration:

$S_{\oplus}(\chi_1(1, 2), 1) = \chi_1(T_{\oplus}(1, 1), T_{\oplus}(2, 1)) = \chi_1(2, 3)$

$\chi_1(1, \chi_1(2, \perp)) = \chi_1(1, \cup(2, \perp)) = \chi_1(1, 2)$



25

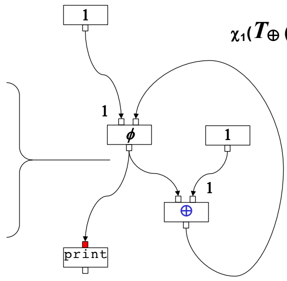
25

Example (cont'd)

- Context-sensitive, after third iteration (stable):

$S_{\oplus}(\chi_1(1, \top), 1) = \chi_1(T_{\oplus}(1, 1), T_{\oplus}(\top, 1)) = \chi_1(2, \top)$

$\chi_1(1, \chi_1(2, 3)) = \chi_1(1, \cup(2, 3)) = \chi_1(1, \top)$



26

26

Outline

- Introduction to SSA
- Construction, Destruction
- How to capture analysis results
- Optimizations
  - Classic analyses and optimizations on SSA representations
  - Heap analyses and optimizations

27

27

Analyses and Optimizations

- Analyses in Compiler Construction allow to safely perform optimizations
- Cost model: runtime of a program
  - Statically only conservative approximations
    - Loop iterations
    - Conditional code
  - Even for linear code not known in advance:
    - Instruction scheduling
    - Cache access is data dependent
    - Instruction pipelining: execution time is not the sum of individual operations costs
- Alternative cost models:
  - memory size, power consumptions
  - Same non-decidability problem as for execution time
- Caution: cost of a program ≠ sum of costs of its elements

28

28

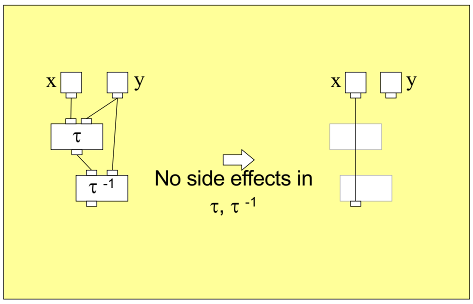
Optimization: Implementation

- Legal transformations in SSA-Graphs:
  - Simplifying transformations reduce the costs of a program
  - Preparative transformations allow the application of simplifying transformations
- Using
  - Algebraic Identities (e.g., Associative / Distributive law for certain operations)
  - Moving of operations
  - Reduction of dependencies
- Optimization is a sequence of goal directed, legal simplifying and legal preparative transformations
- Legibility proven
  - Locally by checking preconditions
  - Due to static data-flow analyses

29

29

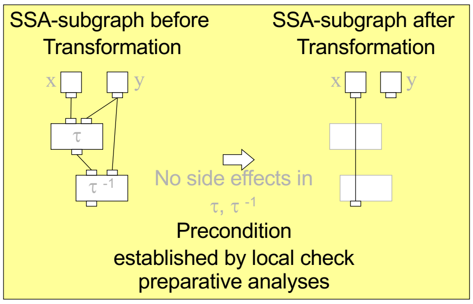
Algebraic Identity: Elimination of Operations and its Inverse



30

30

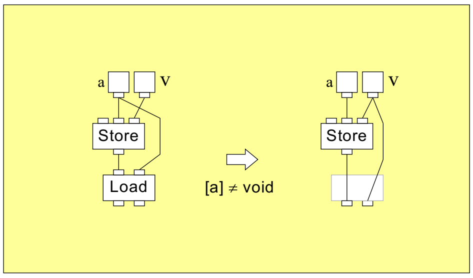
Graph Rewrite Schema



31

31

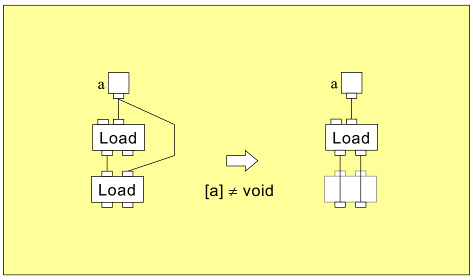
Elimination of Memory Operation and its Inverse



32

32

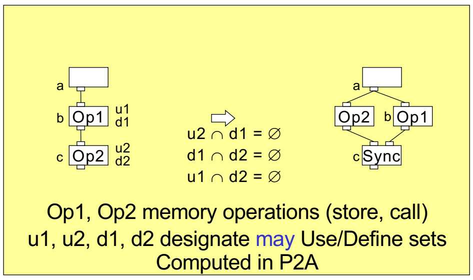
Elimination of Duplicated Memory Operations



33

33

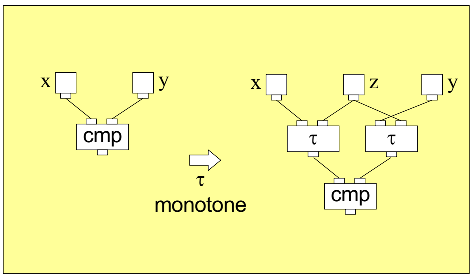
Elimination of non-essential dependencies



34

34

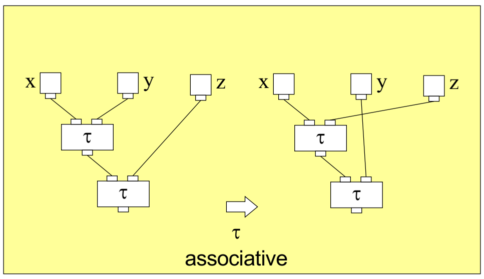
Algebraic Identity: Invariant Compares



35

35

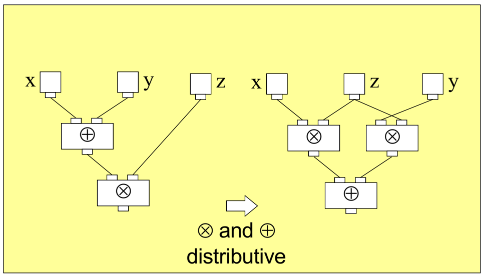
Associative Law



36

36

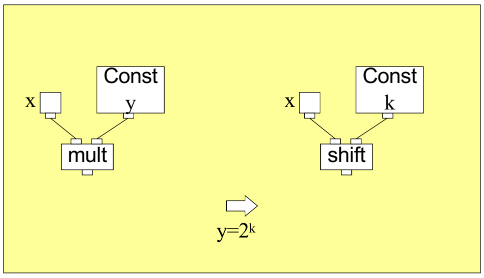
Distributive Law



37

37

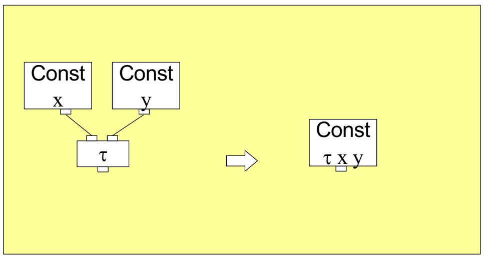
Operator Simplification



38

38

Constant Folding

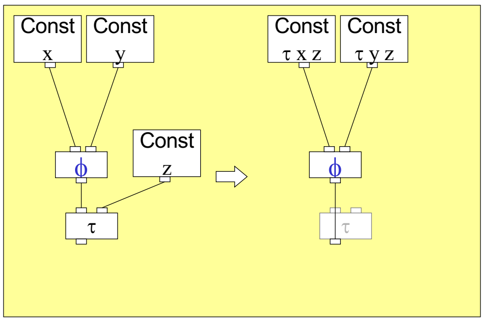


Evaluation using source algebra
or target algebra (if allowed by source language)

39

39

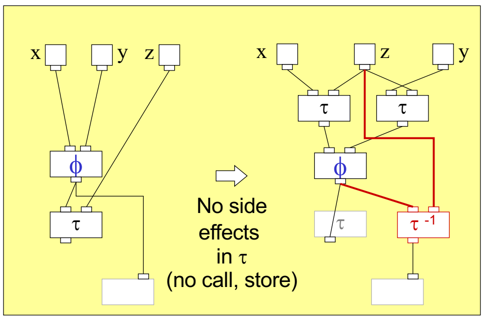
Constant folding over ϕ-functions



40

40

General: Moving arithmetic operations over ϕ-functions



41

41

Optimizations

- Strength reduction:
  - Bauer & Samelson 1959
  - Replace expensive by cheep operations
    - Loops contain multiplications with iteration variable,
    - These operations could be replaced by add operations (Induction analysis)
  - One of the oldest optimizations: already in Fortran I-compiler (1954/55) and Algol 58/60- compiler
- Partial redundancy elimination (PRE):
  - Morel & Renvoise 1978
  - Eliminate partially redundant computations
    - SSA eliminates all static but not dynamic redundancies
    - Problem on SSA: which is the best block to perform the computation
    - Move loop invariant computations out of loops, into conditionals
  - subsumes a number of simpler optimization

42

42

Example: Strength reduction

```
for (i=0;i<n;i++){
   for (j=0;j<n;j++){
      b[i,j]=a[i,j];
   }
}

//Original loop body:
ao    = i*n*d + j*d
aij   = >a[0,0]<+ao
bo    = i*n*d + j*d
bij   = >b[0,0]<+bo
<bij> = <aij>
```

```
         adda=>a[0,0]<
         addb=>b[0,0]<
         d=4
         addend=adda+n*n*d;
LOOP:    jump(addend==adda) END
         <addb>=<adda>
         adda=adda+d
         addb=addb+d
         jump LOOP
END:     exit
```


Induction Analysis Idea

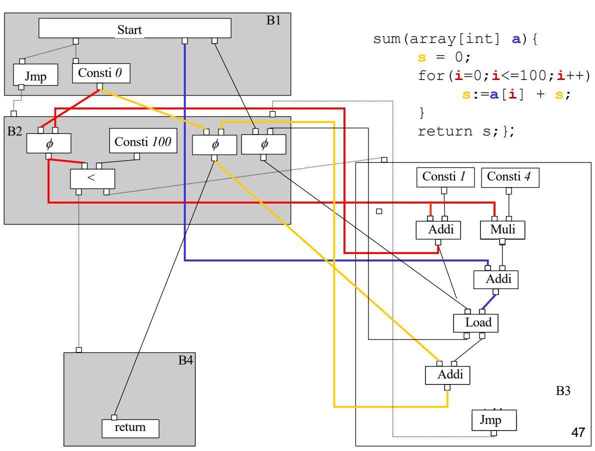
- Find Induction variable i for a loop using DFA
  - i is induction variable if in loop only assignments of form $i := i+c$ with loop constant c or, recursively, $i := c' * i' + c''$ with i' induction variable and loop constants c', c''
  - c is a loop constant iff c does not change value in loop, i.e.
    - c is static constant,
    - c defined in enclosing loop
- Example (cont'd), consider the inner loop: `for (j=0;j<n;j++){…}`
  - Direct induction variable: `j`, as `j=j+1` (c= `1`)
  - Indirect induction variable: `ao=i*n*d+j*d` (c' = `d`, c'' = `i*n*d`)
  - Note that `i*n*d` and `d` are loop constants for the inner loop

Induction Transformation Idea

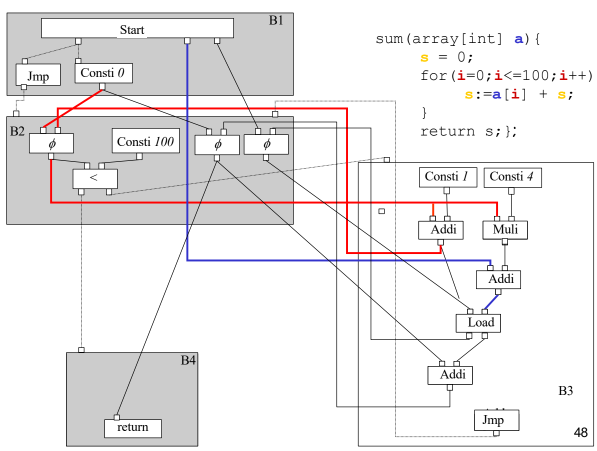
- Transformation goal: values of induction variables should grow linearly with iteration; add operations replace `mult` operations
- Transformation:
  - Let i_0 initialization of i and induction variables, $i := i+c$ and $i' := c' * i + c''$
  - New variable ia initialized $ia := c' * i_0 + c''$
  - At loop end insert $ia = ia + c' * c$
  - Replace consistently operands i' by ia
  - Remove all assignments to i, i' and i, i' themselves if i is not used elsewhere (DFA)
- Example:
  - Before: `loop ao = i*n*d+j*d … j++ end loop`
  - After: `aoa = i*n*d loop … aoa = aoa + d end loop`

Induction Analysis: Implementation

- Assume initially optimistically: all variables are induction variables
- Finding induction variable i for a loop follows definition
- Iteratively until fix point: i is not induction variable if not:
  - $i := i+c$ with loop constants c (direct induction variable)
  - $i := c' * i' + c''$ with i' induction variable and loop constants c', c'' (indirect induction variable)
- On SSA, simplifications of that analysis are possible
  - Any direct loop variable corresponds to a cyclic subgraph over $i := \phi(i_1 \ldots i_n)$
  - Find Strongly Connected Component (SCC) and check those subgraphs for the direct induction variable condition first
  - Then find the indirect induction variables

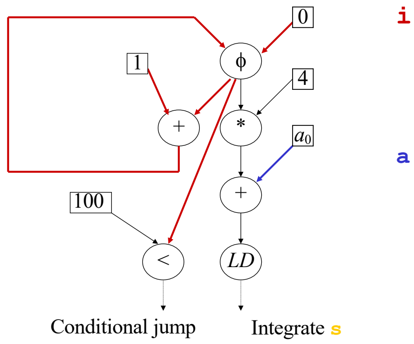



```
sum(array[int] a){
    s = 0;
    for(i=0;i<=100;i++){
        s:=a[i] + s;
    }
    return s;};
```


```
sum(array[int] a){
    s = 0;
    for(i=0;i<=100;i++){
        s:=a[i] + s;
    }
    return s;};
```

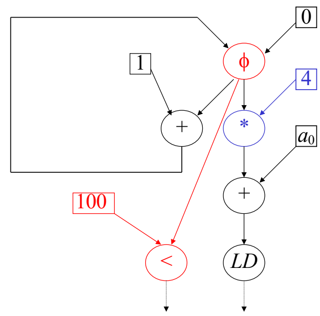
Induction Variables (Schematic)



49

49

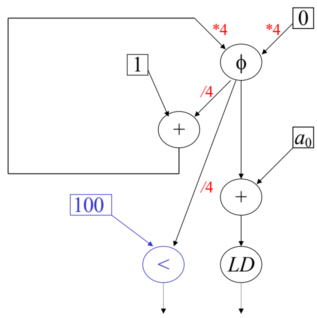
Move × over ϕ-function



50

50

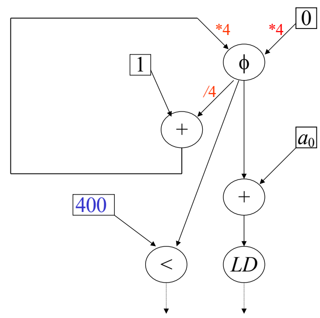
Invariant Comparison (×4) then constant folding and removal of operation and its invariant (/4 ×4)



51

51

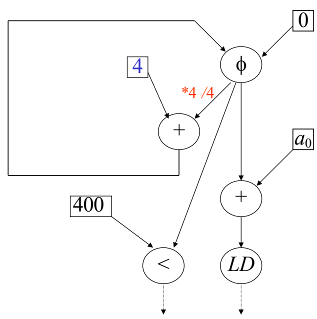
Distributive Law then constant folding



52

52

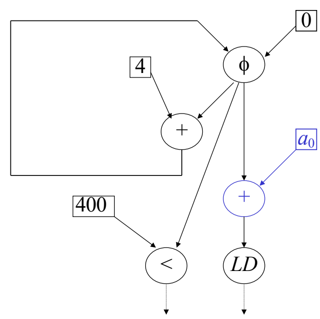
Removal of operation and its inverse



53

53

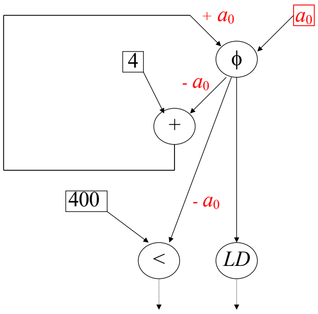
Move + over ϕ-function



54

54

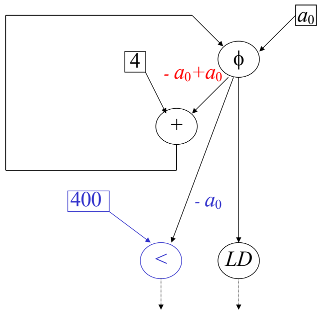
Associative Law



55

55

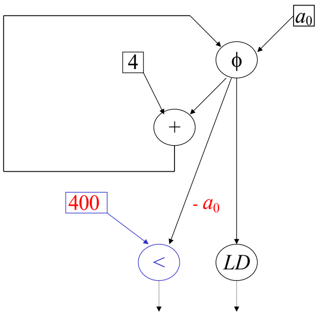
Removal of operation and its inverse



56

56

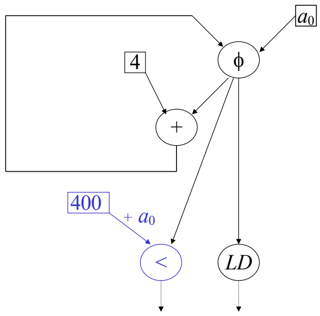
Invariant Comparison ($+a_0$) then removal of operation and its inverse



57

57

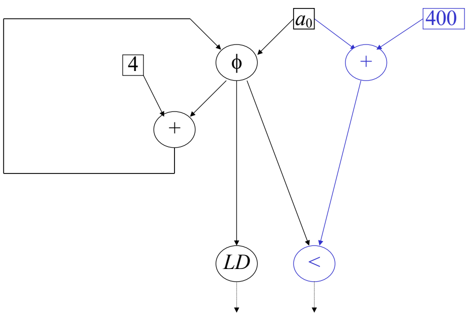
Rearranging the drawing



58

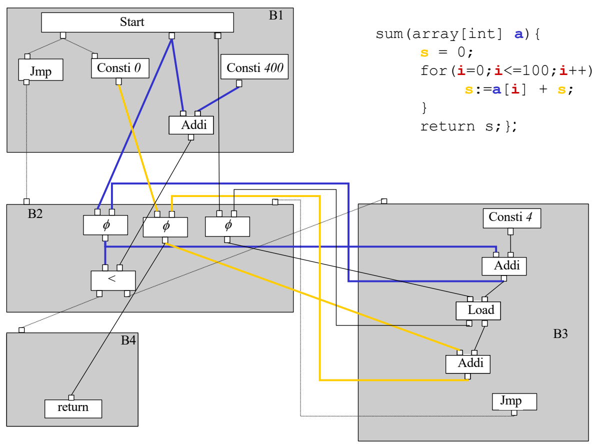
58

Result



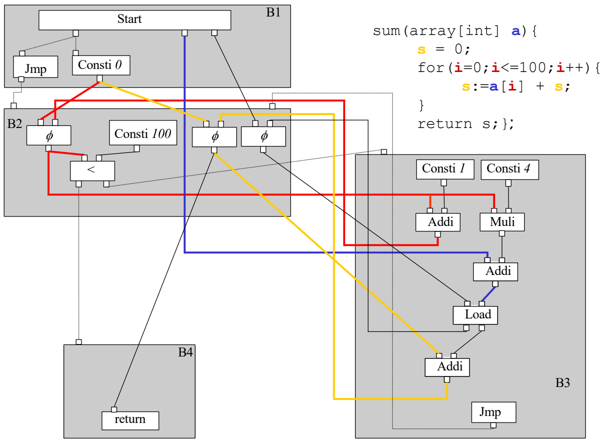
59

59




```
sum(array[int] a){
    s = 0;
    for(i=0;i<=100;i++){
        s:=a[i] + s;
    }
    return s;};
```

60



61

Further Optimizations

- Constant evaluation (simple transformation rule)
- Constant propagation (iterative application of that rule)
- Copy propagation (on SSA construction)
- Dead code elimination (on SSA construction)
- Common subexpression elimination (on SSA construction)

- Specialization of basic blocks, procedures, i.e.cloning
- Procedure inlining
- Control flow simplifications
- Loop transformations (Splitting/merging/unrolling)
- Bound check eliminations
- Cache optimizations (array access, object layout)
- Parallelization
- …

76

76

Observations

- Order of optimizations matters in theory:
  - Application of one optimization might destroy precondition of another
  - Optimization can ruin the effects of the previous once
- Optimal order unclear (in scientific papers usual statements like: "Assume my optimization is the last …"
- Simultaneous optimization too complex
- Usually, first optimization gives 15% sum of remaining 5%, independent of the chosen optimizations
- Might differ in certain application domains, e.g., in numerical applications operator simplification gives factor >2, cache optimization factor 2-5

77

77

Outline

- Introduction to SSA
- Construction, Destruction
- How to capture analysis results
- Optimizations
  - Classic analyses and optimizations on SSA representations
  - Heap analyses and optimizations

78

78

Optimizations on Memory

- Elimination of memory accesses.
- Elimination of object creations.
- Elimination nonessential dependencies.
- Those are normalizing transformations for further analyses

- Nothing new under the sun:
  - Define abstract values, addresses, memory
  - Define context-insensitive transfer functions for memory relevant SSA nodes (Load, Store, Call) (discussed already)
  - Generalization to context-sensitive analyses (discussed already)
  - Optimizations as graph transformations (discussed already)

79

79

Memory Values

- Differentiation by Name Schema
- Distinguish e.g.:
  - local arrays with different name
  - disjoint index sets in an array (odd/even etc.)
  - different types of heap objects
  - objects with same type but statically different creation program point
  - objects with same creation program point but with statically different path to that creation program point (execution context, context-sensitive)

80

80

Abstract values, addresses, memory

- References and values
  - allocation site lattice $\wp^O$ abstracts from objects O
  - arbitrary lattice X abstracts from values like `Integer` or `Boolean`
  - abstract heap memory M:
    - $O \times F \rightarrow \wp^O$ (F set of fields with reference object semantics)
    - $O \times V \rightarrow X$ (V set of fields with value semantics)
- Arrays
  - Treated as objects
  - Abstract heap memory M:
    - $O \times F_{[]} \times I \rightarrow O$ ($F_{[]}$ set of fields with type array of reference object)
    - $O \times V_{[]} \times I \rightarrow X$ ($V_{[]}$ set of fields with type array of value)
    - I an arbitrary integer value lattice (e.g., constant or interval lattice)
- Abstract address $Addr \subseteq \wp^O \times F \times I$ (F set of field names)
  - object-field-(index) triples where index might be ignored

81

81

Updates of Memory

- Given an abstract object-field of a store operation
- In general, this abstract object points to more than one real memory cell
- A store operation overwrites only one of these cells, all others contain the same value
- Hence, a store to an abstract object-field adds a new possible (abstract) value - weak update
- Only if guaranteed that abstract object-field-(index) matches one and only one concrete address, a new (abstract) value overwrites the old value - strong update
- Auxiliary function:

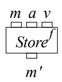
$update(mem, o, f, v)$ $= v$ (if strong update possible)
$= mem(o, f) \cup v$ (otherwise, weak update)

82

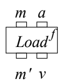
82

Transfer functions (insensitive, no arrays)

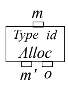
- $T_{store,f}\ (mem, addr, v)$:
  $(update(mem, o_1, f, v)\ldots update(mem, o_k, f, v))$
  $addr = \{o_1 \ldots o_k\}$



- $T_{load,f}\ (mem, addr)$:
  $(mem, mem(o_1) \cup \ldots \cup mem(o_k))$
  $addr = \{o_1 \ldots o_k\}$



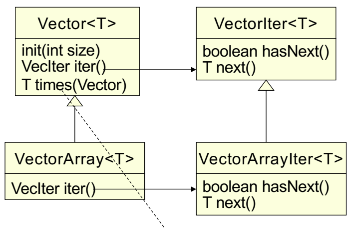
- $T_{alloc(type,id)}\ (mem)$:
  $(mem(o_{id}, f_1) \mapsto \bot]\ldots mem(o_{id}, f_k) \mapsto \bot],\ \{o_{id}\})$
  $\{f_1 \ldots f_k\}$ fields of $Type$



83

83

Example: Main Loop Inner Product Algorithm

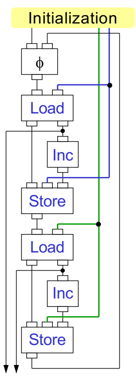


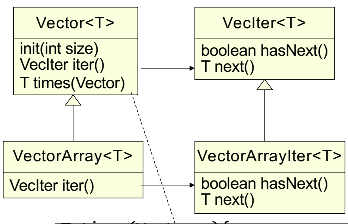

```
T times(Vector v){
  i1 = iter();
  i2 = v.iter();
  for (s = 0; i1.hasNext(); )
    s = s+i1.next()*i2.next();
}
```

84

84

Example: SSA



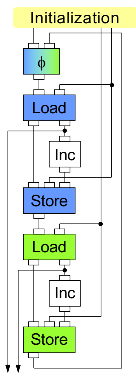


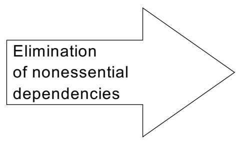

```
T times(Vector v){
  i1 = iter();
  i2 = v.iter();

  for (s = 0; i1.hasNext(); )
    s = s+i1.next()*i2.next();
}
```

85

85

Actually, two Iterators

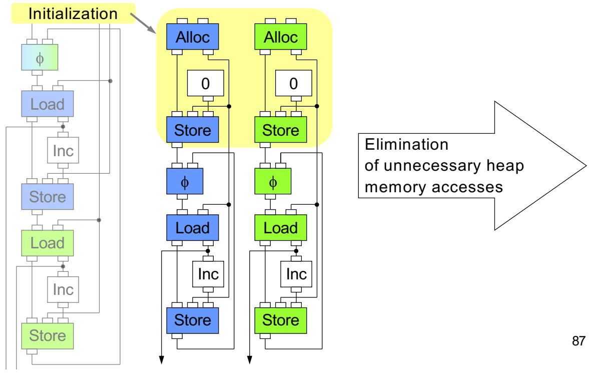




86

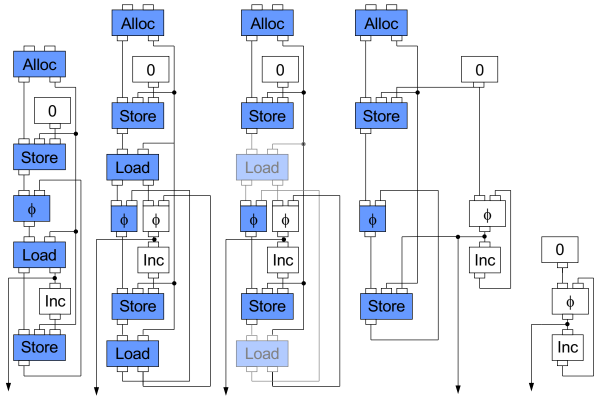
86

Initialization: disjoint memory guarantied



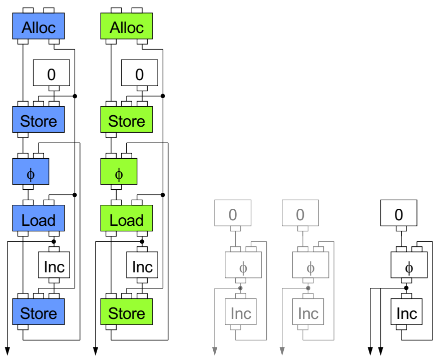
87

Memory objects replaced by values



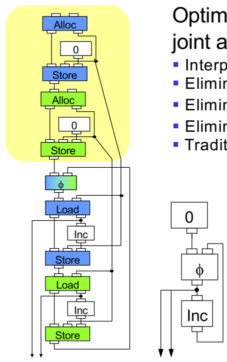
88

Value numbering proofs equivalence



89

Example revisited



Optimization only possible due to joint application of single techniques:

- Interprocedural analysis
- Elimination of polymorphism
- Elimination of nonessential dependencies
- Elimination von memory operations
- Traditional optimizations

90

90

Outline

- Introduction to SSA
- Construction, Destruction
- How to capture analysis results
- Optimizations
  - Classic analyses and optimizations on SSA representations
  - Heap analyses and optimizations

91

91