Outline

- Introduction to SSA
- Construction, Destruction
- Representation of analysis values
- Representation of analysis values
- Optimizations
  - Classic analyses and optimizations on SSA representations
  - Heap analyses and optimizations

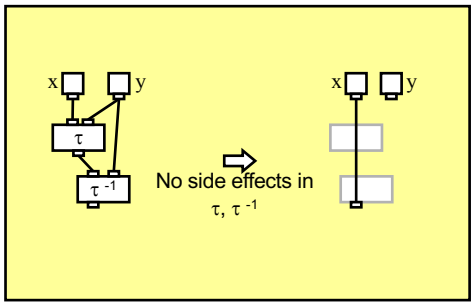
Analyses and Optimizations

- 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

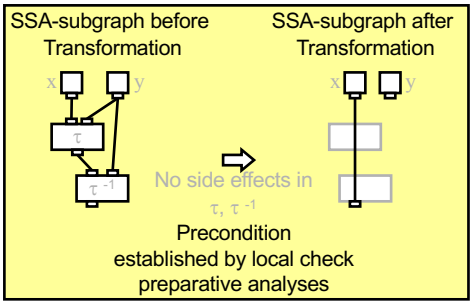
Optimization: Implementation

- 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

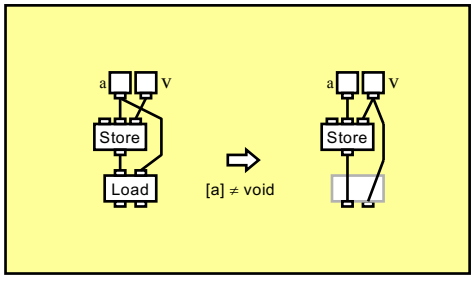
Algebraic Identity: Elimination of Operations and its Inverse



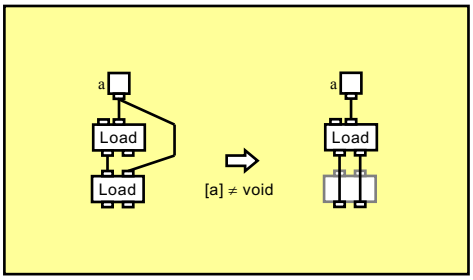
Graph Rewrite Schema



Elimination of Memory Operation and its Inverse

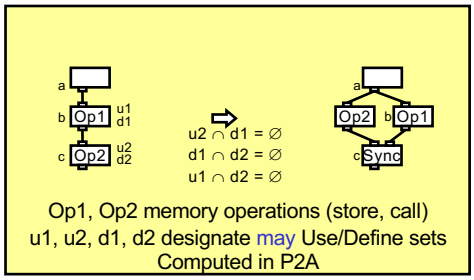


Elimination of Duplicated Memory Operations



33

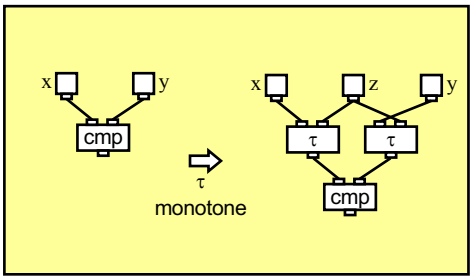
Elimination of non-essential dependencies



Op1, Op2 memory operations (store, call)
u1, u2, d1, d2 designate may Use/Define sets
Computed in P2A

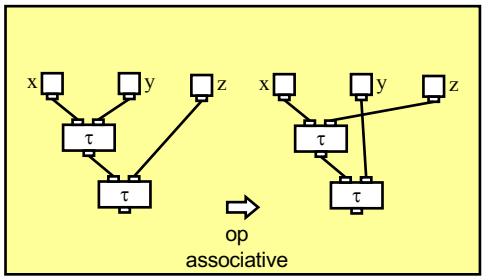
34

Algebraic Identity: Invariant Compares



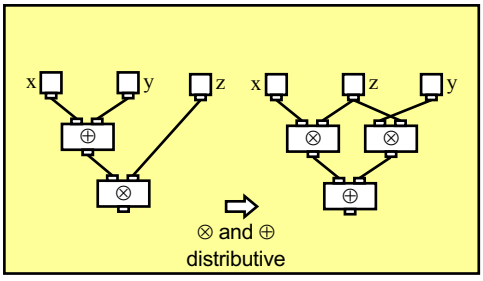
35

Associative Law



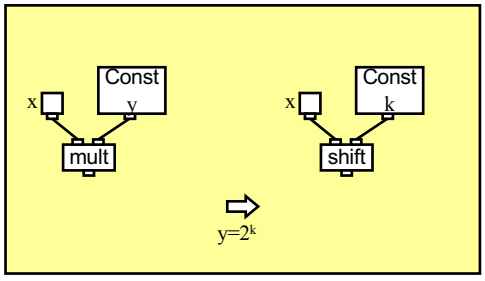
36

Distributive Law



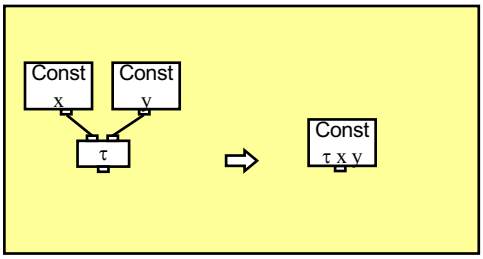
37

Operator Simplification



38

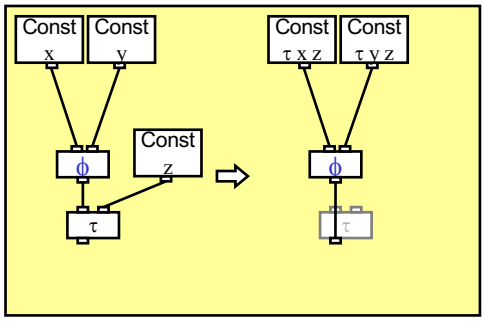
Constant Folding



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

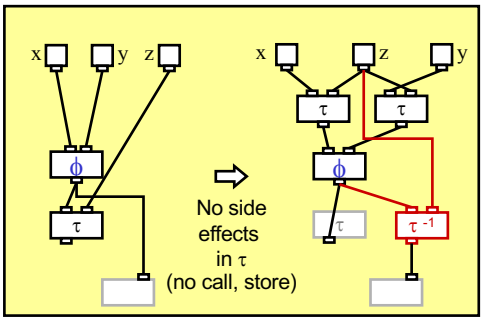
39

Constant folding over ϕ-functions



40

General: Moving arithmetic operations over ϕ-functions



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

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
```

43

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 computed in enclosing loop
- Example (cont'd), consider the inner loop:
  `for (j=0;j<n;j++){…}`
  - Direct induction variable: `j`, implicit `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` a loop constants for the inner loop

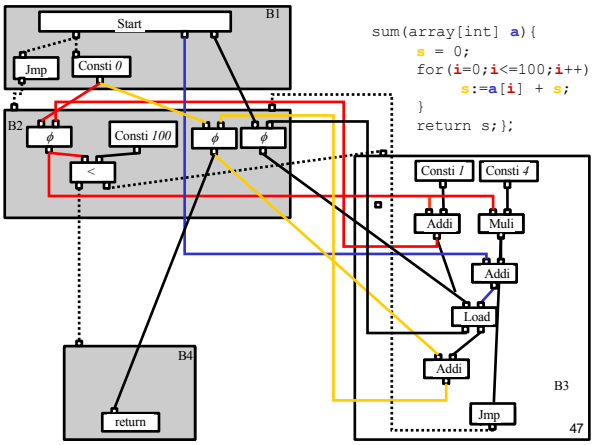
44

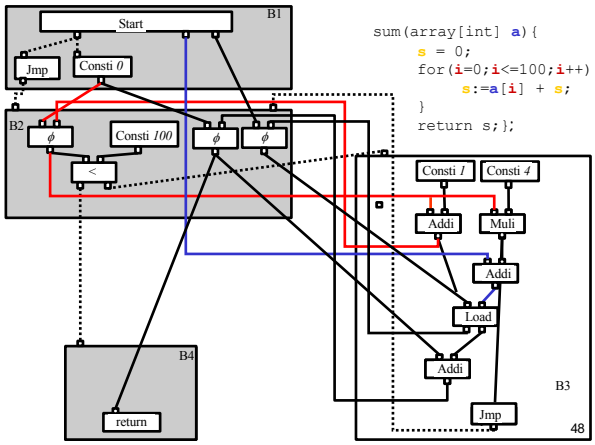
Induction Transformation Idea

- 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`

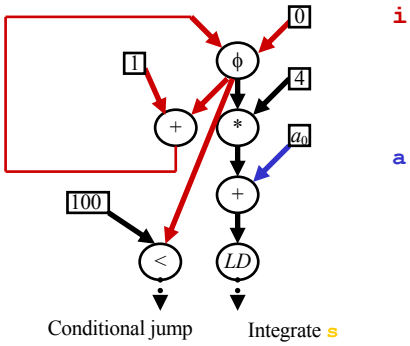
Induction Analysis: Implementation

- 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 variabes

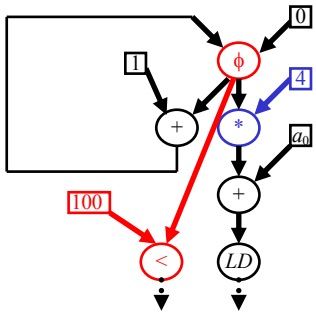




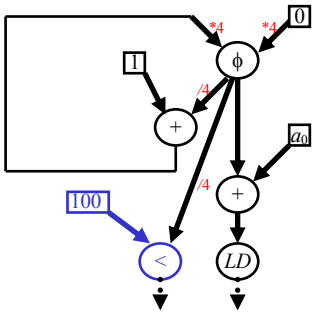
Induction Variables (Schematic)



Move * over ϕ-function

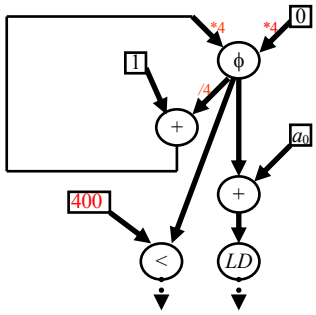


Move Multiplication



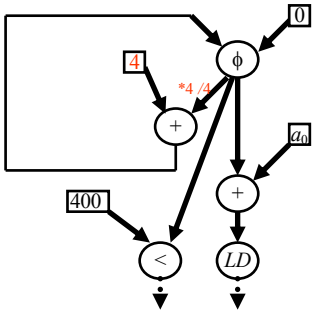
51

Invariant Compare



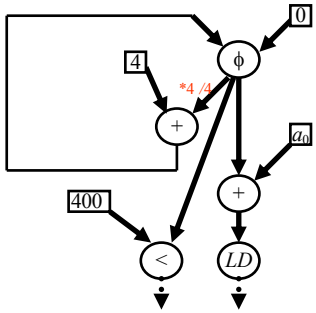
52

Distributive Law



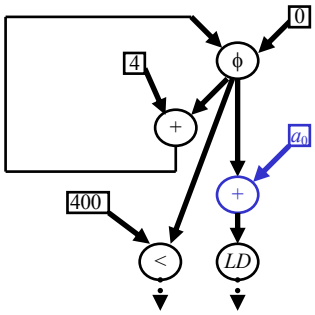
53

Operation and its Inverse



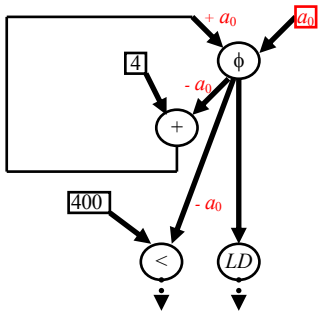
54

Move Addition



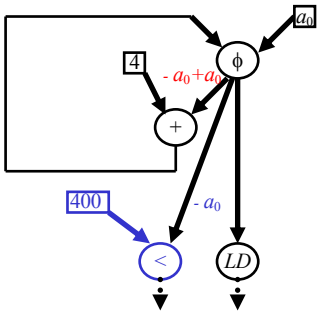
55

Move Addition



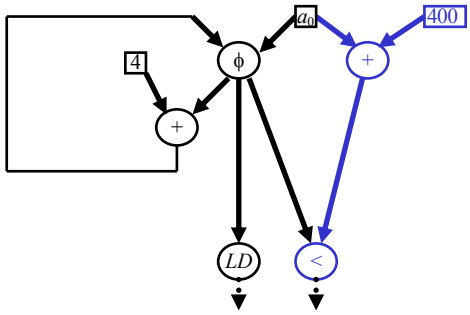
56

Associative Law



57

Change Compare



58