Optimizing Compilers
6th Lecture

Bernhard Scholz
Institut f. Computersprachen
Argentinierstr. 8
scholz@complang.tuwien.ac.at

Outline

Machine-independent optimizations based on DFA

- Dead Computations based on Live Variable Analysis
- Constant Propagation

Dead Computations

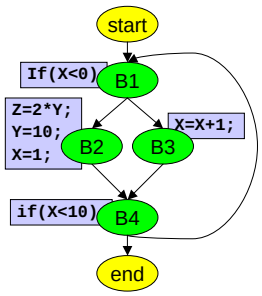
Goal:

Eliminate useless computations in basic blocks

Approach:

1. Global Analysis: DFA to determine useless computations
   - Compute Live Variable of all variables in a CFG
   - Dual set of live variables => dead variables!
2. Transformation: Remove useless computations

Example (cont'd)



- three variables: X, Y, Z
- computation of Z in B2 is not used
- analysis required to determine whether variable X and Y are alive
- if variable is not alive after definition => remove it.
- What about Y?

Liveness Analysis

- Definitions of *v* can be eliminated if variable *v* is not *alive* on the path between definition and exit node.
- Alive-Information is computed by Data Flow Analysis
- Alive-Problem is a Gen/Kill Problem
- Conservative Assumptions
  - functions can access globals and can make them alive
  - OIL does not have global variables!

Liveness Analysis(2)

- How to compute Alive information?
  1. Find type for Live Variable Problem
     - gen/kill problem
     - backward problem
     - union problem
  2. Get gen/kill sets for basic blocks
     - use(n) set is the set of variables that are used in n, i.e. gen
     - def(n) set is the set of variables that are killed in n, i.e. kill
  3. Compute solution with DFA Solver
     - a variable is alive at the entry of n if it is alive at the exit and not in def(n) or in use(n).
  4. Propagate solution for all statement (DFA solution only at an entry or exit of a basic block).

Liveness Analysis(3)

- DFA information
  subset of variables V={v1,v2,..}
- Equations of Live Variables
  $LvOut(end) = \{\}$
  $LvIn(n) = \cup_{p \in succs(n)} LvOut(p)$
  $LvOut(n) = [LvIn(n) - def(n)] \cup use(n)$

  where *def(n)* is the set of variables which are defined in basic block *n* and *use(n)* are set of variables which are used in *n*.
- Meaning of the Solution
  $LvIn(n)$: alive variables at exit of *n*
  $LvOut(n)$: alive variables at entry of *n*
  $[V - LvOut(n)]$: set of dead variables at entry
  $[V - LvIn(n)]$: set of dead variables at exit

Computation of Use/Def

For all statements in basic block *n*:

- Insert variable to *use(n)*,
  if variable occurs in an expression of n (i.e. read-access).
- Insert variable to *def(n)*,
  if variable is assigned a value of n (i.e. write-access).

Problem

- def-use situations in basic block
- For example X=1; Y=X+1;
- without precedence variable X would be alive before X=1

Algorithm for Use/Def

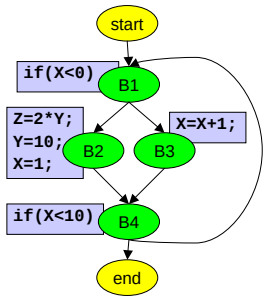
- Initialization
  $use(n)=def(n)=\varnothing$
- Process statements in reverse order

Algorithm

```
for all n∈N-{start,end}
 for all statements s in n
  if s = <v=expr;> then
   def(n) = def(n)∪{v}
   use(n) = use(n)-{v}
  endif
  for all v used in s
   use(n) = use(n) ∪ {v}
   def(n) = def(n)-{v}
  endfor
 endfor
endfor
```

Example

Live Variable Problem



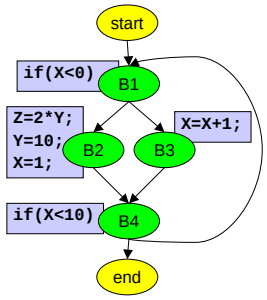
Use/Def Sets

Node	use	def
Start	{}	{}
B1	{X}	{}
B2	{Y}	{X,Z}
B3	{X}	{}
B4	{X}	{}
end	{}	{}

- Variable X is defined and used in B2!

Example(2)

Live Variable Problem



DFA Solution

Node	LvIn	LvOut
Start	{X,Y}	{X,Y}
B1	{X,Y}	{X,Y}
B2	{X,Y}	{X,Y}
B3	{X,Y}	{X,Y}
B4	{X,Y}	{X,Y}
end	{}	{}

- Variable Z is not alive
- What about Y?

Removing Dead Computations

Algorithm

- Process statements in reverse order!
- Uses DFA Solution
- Take care of local dead computations
- Cannot remove transitive dead computations => several iterations required.

```
for all n∈N-{start,end}
 alive = LvIn(n)
 for all statements s in n
  if s = <v=expr;> then
   if v ∉ alive then
    remove statement s
   endif
   alive = alive – {v}
  endif
  if statement not removed
   for all v used in s
    alive = alive ∪ {v}
   endfor
 endfor
endfor
```

Statement Removal

- Statement may contain function calls
- Without knowing side-effects of function, call must be preserved
- Example (conservative approach):

```
t = f(A)+f(B);
```

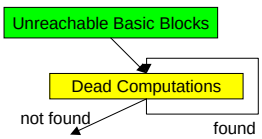
⇩

```
f(A); f(B);
```

- For complete removal: functions must be side-effect free (for OIL: no write-access to memory (*(x)=...;)

Dead Code Elimination

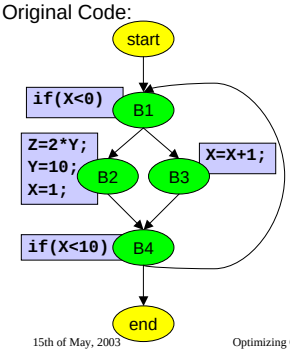
- Approach



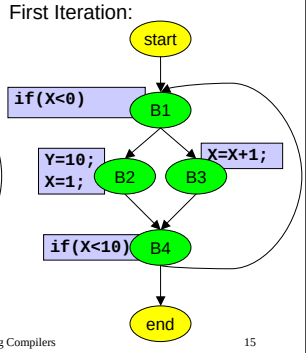
- Eliminate unreachable basic blocks beforehand
- Captures transitive dead computations in several iterations

Example (cont'd)

Original Code:

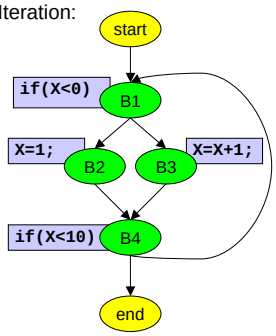


First Iteration:



Example (cont'd)

Last Iteration:



Constant Propagation(CP)

Goal:

Propagate constant values of variables for evaluating expressions at compile time

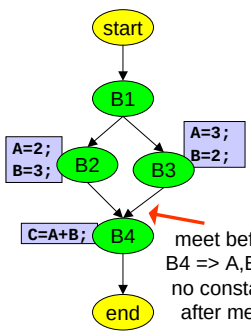
Approach:

1. Determine which variables are constant at a point
2. Replace variables by their constant value
3. Evaluate and simplify expressions at compile-time

Analysis-Details:

- constant propagation problem is not distributive
- cannot be simply mapped to gen/kill-problems

Example



meet before B4 => A,B are no constants after meet!

- problem is not distributive
- Solution of (B2 B4) $\wedge$ (B3 B4) $\neq$ (B2 $\wedge$ B3) B4
- two paths needs to be propagated to B4 for folding C (=5)
- with loops infinite different paths => undecideable in general
- not an easy problem
- find sufficient solution (maximum fixpoint)

Types of Constant Variables

- Simple Constants

```
X=10; ........; Z=A*X;
X=10; ........; Z= A*10;
```

- Copy Constants

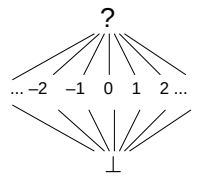
```
Y=5;X=Y; ........; Z=A*X;
Y=5;X=5;........; Z= A*5;
```

- Complex Constants

```
Y=5;X=2*Y; ........; Z=A*X;
Y=5;X=10;........; Z= A*10;
```


DFA for CP

- Number lattice



  - Bottom: not a constant
  - Top: still unknown (either constant or not).

Dataflow Information:

- variable environment
  $env=\{(v_1, c_1),(v_2, c_2),.....\}$
- variable is associated with an element of number lattice.
- not a simple bit-problem anymore!
- more complex operations are required

Number Lattice

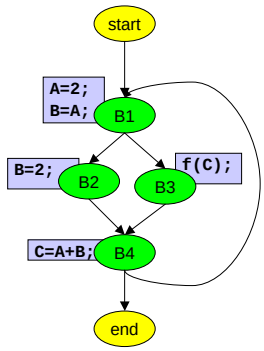
- Meet operator of number lattice
  - Numbers c_1, c_2 and $c_1 \neq c_2$
  - $c_1 \wedge c_2 = \perp$
  - $? \wedge c = c$
  - $\perp \wedge c = \perp$

$\wedge$	c_1	c_2	$\perp$	?
c_1	c_1	$\perp$	$\perp$	c_1
c_2	$\perp$	c_2	$\perp$	c_2
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$
?	c_1	c_2	$\perp$	?

- Meet operator of variable environment
  - join values of same variables

$env_a \wedge env_b = \{(v_1, {c_1}^a \wedge {c_1}^b),(v_2, {c_2}^a \wedge {c_2}^b),.....\}$

Example



Initialization
start: {(A, ⊥),(B, ⊥), (C, ⊥)}
others: {(A, ?),(B, ?), (C, ?)}

Solve iteratively

In/Out solutions for basic blocks
out(start): {(A, ⊥),(B, ⊥), (C, ⊥)}
in(B1): {(A, ⊥),(B, ⊥), (C, ⊥)}
out(B1): {(A, 2),(B, 2), (C, ⊥)}
in(B2): {(A, 2),(B, 2), (C, ⊥)}
out(B2): {(A, 2),(B, 2), (C, ⊥)}
in(B3): {(A, 2),(B, 2), (C, ⊥)}
out(B3): {(A, 2),(B, 2), (C, ⊥)}
in(B4): {(A, 2),(B, 2), (C, ⊥)}
out(B4): {(A, 2),(B, 2), (C, ⊥)}
in(end): {(A, 2),(B, 2), (C, 4)}

Stop

- Next lecture: 22.5.2003, 13:45 – 14:45