263-2810: Advanced Compiler Design

3.6 Variable renaming - example

Thomas R. Gross

Computer Science Department
ETH Zurich, Switzerland
Outline

- 3.1 Graphs
- 3.2 Approaches to insertion of $\phi$ nodes
- 3.3 Dominance frontier
- 3.4 Algorithm for insertion of $\phi$ nodes
- 3.5 Algorithm for variable renaming
- 3.6 Example
3.5 Algorithm for variable renaming

- Given a CFG, with $\phi$ functions inserted

- For each variable $V$: stack of versions
  
  stack[$V$]
  
  Initially empty

- For each variable: counter

  counter[$V$]
  
  Initialized to 0
Process (basic block X) {
    for all statements S in X {
        if (S is not a $\phi$ function) {
            for each variable V in RHS {
                replace V with $V_i = \text{Stack}[V].\text{top}$
            } // RHS
            for each variable V on the LHS {
                $c = \text{counter}[V]$
                $\text{Stack}[V].\text{push}(c)$
                replace V with $V_c$ on the LHS
                $\text{counter}[V] = c + 1$
            } // LHS
        } // for all statements
    } // for all statements
}
for all successors $Y$ of $X$ {
  for all $\phi$ functions $F$ in $Y$ {
    let $V$ be the LHS of $F$
    let $X$ be the $k$-th predecessor of $Y$
    set the $k$-th argument of $F$ to $V_i = \text{Stack}[V].\text{top}$
  } // for all $\phi$ functions
} // for all successors

for all children $C$ of $X$ { process ( $C$ ) }
for all assignments $A$ in $X$ {
  if $V$ is set by $A$ {
    Stack[$V$].pop
  }
} // for all assignments

} // for all blocks
3.6 Example

ENTRY

0

a = b =

1

b =

2

b = φ( , )

3

a = φ( , )
b = φ( , )

4

a =

5

b = φ( , )

6

b =

7

b = φ( , )

8

b = φ( , )

9

a = φ( , )
b = φ( , )

EXIT

EXIT
3.6 Example

**ENTRY**

\[
a_0 = \\
b_0 = \\
\]

**EXIT**

**a_4** = \( \phi(a_0, a_1) \)

**b_5** = \( \phi(b_0, b_4) \)

\[
a_0 = \\
b_0 = \\
\]

1

2

3

\[
b_1 = \\
a_1 = \phi(a_0, a_2) \\
b_2 = \phi(b_0, b_3) \\
\]

7

8

\[
b_4 = \phi(b_1, b_2) \\
\]

9

5

6

\[
\]

\[
\]

**EXIT**

**b_3** = \( \phi(b_0, b_3) \)

**a_1** = \( \phi(a_0, a_2) \)

**b_2** = \( \phi(b_0, b_3) \)

**a_4** = \( \phi(a_0, a_1) \)

**b_5** = \( \phi(b_0, b_4) \)
3.6 Example

Stack(a) = [ ] = [0]
Stack(b) = [ ] = [0]
C(a) = 0
C(b) = 0

process(0)
process(1)
process(2)
process(3)
process(4)
process(5)
process(6)
process(7)
process(8)
process(9)

ENTRY

a_0 =
b_0 =

0

1

b_1 =

2

a_2 = \phi(a_0, a_2)
b_2 = \phi(b_0, b_2)

3

a_3 = \phi(a_0, a_3)
b_3 = \phi(b_0, b_3)

4

b_4 = \phi(b_3, b_4)

5


EXIT

6

7

8

9
Stack = [ ]
Stack = [ 0 ]
Stack = [ 0, 1 ]
Stack = [ 0, 1, 2 ]
Stack = [ 0, 1, 2, 3 ]
Stack = [ 0, 1, 2, 3, 4 ]

C(a) = 0
C(b) = 0
C(c) = 0

process (0)
process (1)
process (2)
process (3)
process (4)
process (5)
process (6)
process (7)
process (8)
process (9)
4.0 Practical issues

- So far: method-local variables
  - Compiler-generated variables ("temporaries") for address arithmetic

- But real programs contain also other data types
  - Object (instances) & their fields
  - Arrays
4.0 Practical issues

- So far: method-local variables
  - Compiler-generated variables (“temporaries”) for address arithmetic

- But real programs contain also other data types
  - Object (instances) & their fields
  - Arrays

- Next: arrays
4.1 Arrays

- Why are arrays a problem for SSA format?
  - Arrays: start with arrays of int
4.1 Arrays

- Why are arrays a problem for SSA format?
  - Arrays: start with arrays of int

- Assignment to individual array elements

```c
a[1] = ... 
```

```c
int b[50], i, j, k
read(i, j, k)
```

```c
b[i] = ...
    = b[j]
```

```c
b[k] = ...
```
Dealing with arrays

- **Idea:** extend representation to include arrays
  - But limit optimization and analysis to scalars
  - Array representation does not cause damage

- **Two auxiliary functions**
  - `access()`: models the reading of an array element
  - `update()`: models the writing of an array element

- **Functions `access()` and `update()` must be executed in source-order**
  - No change of execution order ("no code movement")
update() and access()

read(i, j, k)

b[i]  = ...
      = b[j]
b[k] = ...

update() and access()

read(i, j, k)

\[ b[i] = \ldots = b[j] \]

\[ b[k] = \ldots \]

read\((i_6, j_2, k_{11})\)

update\((b, i_6, \ldots)\)

\[ \ldots = access(b, j_2) \]

update\((b, k_{11}, \ldots)\)
update() and access()

\[
\text{read}(i, j, k) \quad \text{read}(i_6, j_2, k_{11})
\]

\[
\begin{align*}
&b[i] = \ldots \\
&= b[j] \\
&b[k] = \ldots
\end{align*}
\]

\[
\begin{align*}
&\text{update}(b, i_6, \ldots) \\
&\ldots = \text{access}(b, j_2) \\
&\text{update}(b, k_{11}, \ldots)
\end{align*}
\]

These functions allow modification of arrays “in place”

- Efficient implementation is possible
- Can allow change of execution order for different access() functions
- Absence of aliasing allows changing order of access/update for different arrays
4.2 Objects

- Model instance of an object as a “special” array

```java
class X {
    int a;
    int b;
    int c;
}
X xref;
xref.a = ...
    = xref.b
xref.c = ...
```
4.2 Objects

- Model instance of an object as a “special” array

```java
class X {
    int a;
    int b;
    int c;
}
X xref;
xref.a = ...
  = xref.b
xref.c = ...
update(xref.data, 0, ...)
  ...
  = access(xref.data, 1)
  update(xref.data, 2, ...)
```
4.3 Properties of algorithm 3.4 (φ functions)

- The algorithm presented in Section 3.4 inserts the *minimal* number of φ functions.

- Is inserting the *minimal* number of φ functions *optimal*?
Example program (sketch)

```c
if (...) {
  if (...) {
    a = 0
  } else {
    a = 1
  }
}

b = a * f
```

} else {

...
Example program (sketch)

```plaintext
if (...) {
    if (...) {
        a = 0
    } else {
        a = 1
    }
    b = a * f
} else {
    ...
}
```