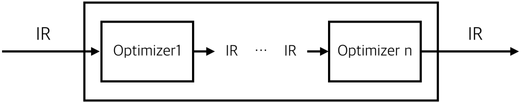
COSE312: Compilers

Lecture 18 — Optimization (1)

Hakjoo Oh
2017 Spring

Middle End: Optimizer

Converts the source program into a more efficient yet semantically equivalent program.



ex)

```
t1 = 10
t2 = rate * t1
t3 = init + t2
pos = t3
```

original IR

```
t1 = 10
t2 = rate * 10
t3 = init + t2
pos = t3
```

```
t2 = rate * 10
t3 = init + t2
pos = t3
```

```
t2 = rate * 10
pos = init + t2
```

final IR

Common Optimization Passes

- Common subexpressions elimination
- Copy propagation
- Deadcode elimination
- Constant folding

Common Subexpression Elimination

- An occurrence of an expression $\boldsymbol{E}$ is called a *common subexpression* if $\boldsymbol{E}$ was previously computed and the values of the variables in $\boldsymbol{E}$ have not changed since the previous computation.

  ```
  x = 2 * k + 1
  ...       // no defs to k
  y = 2 * k + 1
  ```

- We can avoid recomputing $\boldsymbol{E}$ by replacing $\boldsymbol{E}$ by the variable that holds the previous value of $\boldsymbol{E}$.

  ```
  x = 2 * k + 1
  ...       // no defs to k
  y = x
  ```

Copy Propagation

After the copy statement $u = v$, use v for u unless u is re-defined.

```
u = v                  u = v
x = u + 1              x = v + 1
u = x          =>      u = x
y = u + 2              y = u + 2
```

Deadcode Elimination

- A variable is *live* at a point in a program if its value is used eventually; otherwise it is *dead* at that point.
- A statement is said to be *deadcode* if it computes values that never get used.

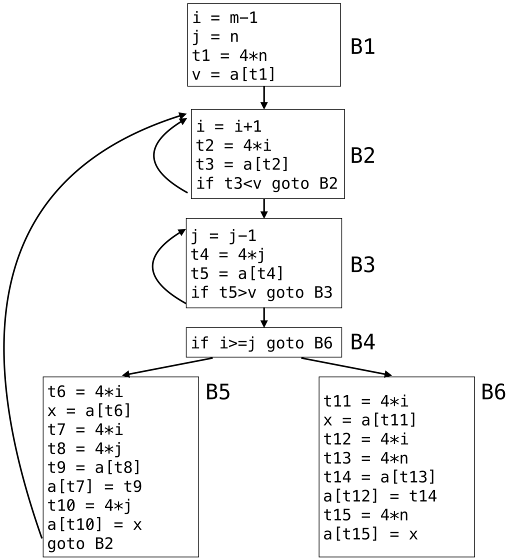
```
u = v     // deadcode
x = v + 1
u = x
y = u + 2
```

Constant Folding

Decide that the value of an expression is a constant and use the constant instead.

```
c = 1                         c = 1
x = c + c        =>           x = 2
y = x + x                     y = 4
```

Example: Original Program



B1

```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

B2

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

B3

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

B4

```
if i>=j goto B6
```

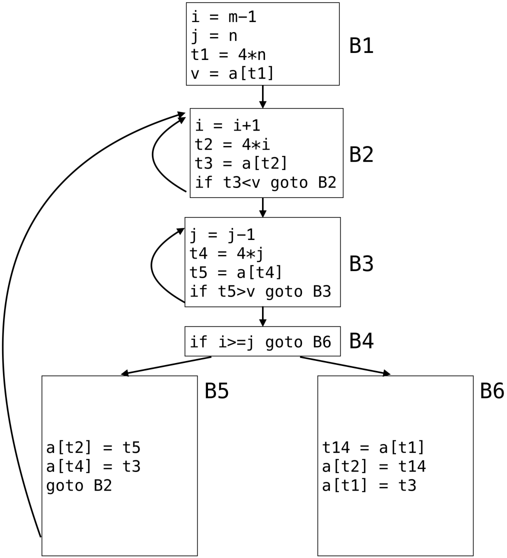
B5

```
t6 = 4*i
x = a[t6]
t7 = 4*i
t8 = 4*j
t9 = a[t8]
a[t7] = t9
t10 = 4*j
a[t10] = x
goto B2
```

B6

```
t11 = 4*i
x = a[t11]
t12 = 4*i
t13 = 4*n
t14 = a[t13]
a[t12] = t14
t15 = 4*n
a[t15] = x
```

Example: Optimized Program



B1

```
i = m-1
j = n
t1 = 4*n
v = a[t1]
```

B2

```
i = i+1
t2 = 4*i
t3 = a[t2]
if t3<v goto B2
```

B3

```
j = j-1
t4 = 4*j
t5 = a[t4]
if t5>v goto B3
```

B4

```
if i>=j goto B6
```

B5

```
a[t2] = t5
a[t4] = t3
goto B2
```

B6

```
t14 = a[t1]
a[t2] = t14
a[t1] = t3
```

Static analysis is needed

To optimize a program, we need static analysis that derives information about the flow of data along program execution paths. Examples:

- Do the two textually identical expressions evaluate to the same value along any possible execution path of the program? (If so, we can apply common subexpression elimination)
- Is the result of an assignment not used along any subsequent execution path? (If so, we can apply deadcode elimination).

Summary

Code Optimization:

- Code transformation to have better performance
- Execution of transformed code must produce same results as the original code for all possible executions
- Static analysis is needed (called data-flow analysis)