CSCI 742 - Compiler Construction

Lecture 32
Control Flow Graphs
Instructor: Hossein Hojjat

April 13, 2018

Recap: Optimizations

- **Optimizations:** code transformations that improve the program
  - Usually to improve execution time
  - Sometimes to reduce program size or power usage
- Can be done at high-level or low-level
  - e.g. constant folding
- Optimizations must preserve the original behavior of program
- Execution of transformed code must yield same result as the original code for every possible input

- **Example:** dead code elimination
- Variable is dead if value is never used after definition
- Eliminate assignments to dead variables

Optimization Correctness: Dead Code Elimination

- Which assignments are dead and can be removed?

```
x = y - 1;
y = z * 2;
x = y - z;
z = 10;
z = x;
```

Optimization Correctness: Dead Code Elimination

- Which assignments are dead and can be removed?

```
x = y - 1;
y = z * 2;
x = y - z;
z = 10;
z = x;
```

- Is `x` dead at first statement?
- Need to know if values assigned to `x` is never used later
- Obvious for this simple example (with no control flow)
- Not obvious for complex flow of control

Optimization Correctness: Dead Code Elimination

- Which assignments are dead and can be removed?

```
x = y - 1;
y = z * 2;
x = y - z;
z = 10;
z = x;
```

- Is `x` dead at first statement?
- Need to know if values assigned to `x` is never used later
- Obvious for this simple example (with no control flow)
- Not obvious for complex flow of control

Optimization Correctness: Dead Code Elimination

- Add control flow to example
- Is `x = y - 1` dead code? Is `z = 10` dead code?

```
x = y - 1;
y = z * 2;
if (c1) x = y - z;
z = 10;
z = x;
```

Optimization Correctness: Dead Code Elimination

- Add control flow to example
- Is `x = y - 1` dead code? Is `z = 10` dead code?

```
x = y - 1;
y = z * 2;
if (c1) x = y - z;
z = 10;
z = x;
```

- Statement `x = y - 1` is not dead code anymore
- On some executions, value is used later

Optimization Correctness: Dead Code Elimination

- Add more control flow to example
- Is `x = y - 1` dead code? Is `z = 10` dead code?

```
while(c2) {
   x = y - 1;
   y = z * 2;
   if (c1) x = y - z;
   z = 10; }
   z = x;
```

Optimization Correctness: Dead Code Elimination

- Add more control flow to example
- Is `x = y - 1` dead code? Is `z = 10` dead code?

```
while(c2) {
   x = y - 1;
   y = z * 2;
   if (c1) x = y - z;
   z = 10; }
   z = x;
```

- Statement `x = y - 1` not dead anymore
- Statement `z = 10` not dead either
- On **some** executions, value from `z = 10` is used later

Low-level Code

- Harder to eliminate dead code in low-level code

```
 0: iload_1
 1: ifeq          32
 4: iload_3
 5: iconst_1
 6: isub
 7: istore_2
 8: iload         4
10: iconst_2
11: imul
12: istore_3
13: iload_0
14: ifeq          22
17: iload_3
18: iload         4
20: isub
21: istore_2
22: bipush        10
24: istore        4
26: iload_2
27: istore        4
29: goto          0
```

Optimizations and Control Flow

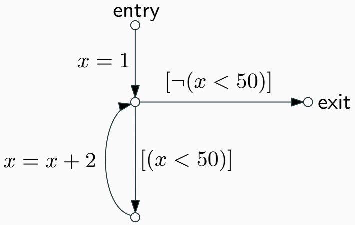
- Application of optimizations requires information
  - e.g. dead code elimination needs to know if variables are dead when assigned values
- Required information are not usually explicit in the program
- We must compute it statically (at compile-time)
- Must characterize all dynamic (run-time) executions
- Control flow makes it hard to extract information

- Branches and loops in the program
- Different executions =
  different branches taken,
  different number of loop iterations executed

Control Flow Graphs

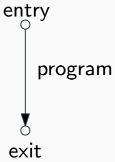
- **Control Flow Graph:** graph representation of computation and control flow in the program
- Specifies all possible execution paths

```
x = 1
while (x < 50) {
  x = x + 2
}
```



Generating Control-Flow Graphs

- Control-Flow graph is similar to AST
- Start with graph that has one **entry** and one **exit** node
- Draw an edge from entry to exit and label it with the entire program

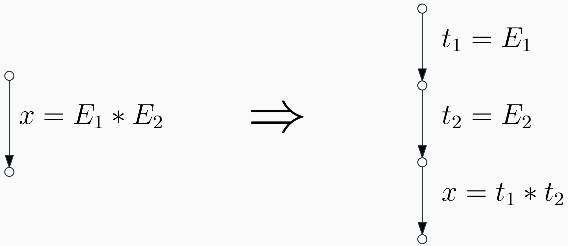


- Recursively decompose the program to have more edges with simpler labels
- When labels cannot be decomposed further, we are done

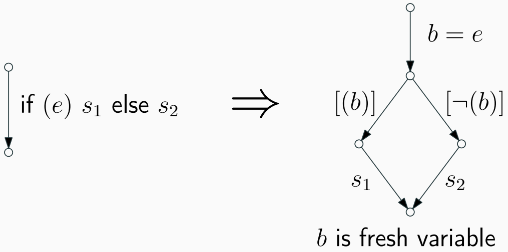
Flattening Expressions

- Label flattening: simplify a label, make an order on the side effects

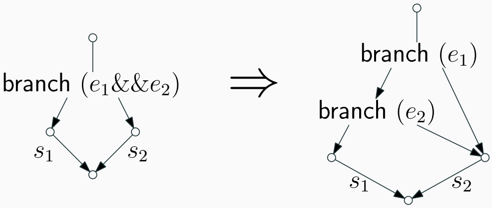
$$E_1, E_2 : \quad \text{complex expressions}$$
$$t_1, t_2 : \quad \text{fresh variables}$$



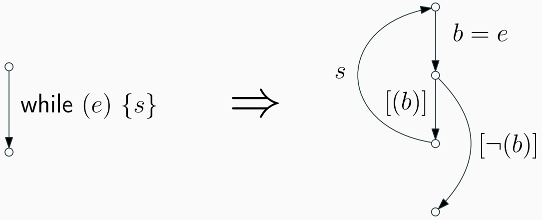
Conditional Statement



- Translation using `branch` instruction with two destinations

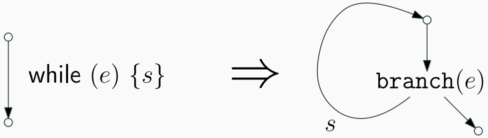


`while` Statement



b is fresh variable

- Translation using the `branch` instruction

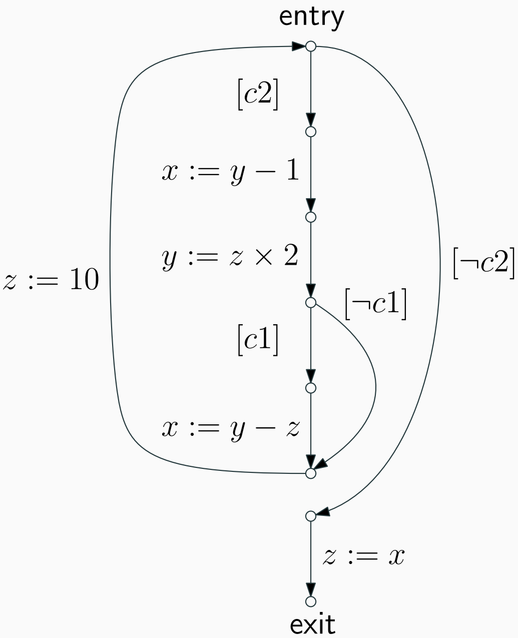


Exercise 1: Convert to CFG

```
while(c2) {
  x = y - 1;
  y = z * 2;
  if (c1) x = y - z;
  z = 10;
}
z = x;
```

Exercise 1: Convert to CFG

```
while(c2) {
  x = y - 1;
  y = z * 2;
  if (c1) x = y - z;
  z = 10;
}
z = x;
```



Exercise 2: Convert to CFG

```
int i = n;
while (i > 1) {
  println(i);
  if (i % 2 == 0) {
    i = i / 2;
  } else {
    i = 3*i + 1;
  }
}
```

Control Flow Graph Construction

$[s_1; s_2]\ v_{\textit{source}}\ v_{\textit{target}} =$

$[s_1]\ v_{\textit{source}}\ v_{\textit{fresh}}$

$[s_2]\ v_{\textit{fresh}}\ v_{\textit{target}}$

$\textbf{insert}(v_s, \text{stmt}, v_t) = \textit{cfg} + (v_s, \text{stmt}, v_t)$

$[x = y + z]\ v_s\ v_t = \textbf{insert}(v_s, x = y + z, v_t)$

where y, y are constants or variables

$[\texttt{branch}(x < y)]\ v_{\textit{source}}\ v_{\textit{true}}\ v_{\textit{false}} =$

$\textbf{insert}(v_{\textit{source}}, [x < y], v_{\textit{true}});$

$\textbf{insert}(v_{\textit{source}}, [!(x < y)], v_{\textit{false}})$