Global Optimization

April 24, 2002

Administrative Issues

1. PA 5 is due Friday, May 3. The checkpoint is due today.

2. WA 10 will be up on Thursday.

3. Final is Friday, May 24.

4. There is only two more sections before the end of the semester.

Optimization

Given a program, we want to make it better. This usually means making the program faster, but it sometimes means making the code size of the program smaller.

However, optimizations may actually slow down the program due to issues outside of the compiler's control, such as hard-to-predict cache effects and pipelining issues.

Local Optimization

- Algebraic simplification
- Copy propagation
- Constant folding / propagation
- Dead code elimination
- Single-assignment form

Global Optimization

- Global dataflow analysis: Global constant propagation
- Liveness analysis: Dead code elimination
- Global escape analysis

Global Optimizations

1. Global Constant Propagation

We need to use global dataflow analysis to do this code optimization. Here, we save global constant information at each step.

We also use a *transfer function* which transfers information from one statement to another. For each statement *s*, we compute information about the value of *x* immediately before and after *s*.

$C_{in}(x,s)$ = value of *x* before *s*
$C_{out}(x,s)$ = value of *x* after *s*

In lecture, he defined C_{in} and C_{out} for constant propagation.

Notice that we begin at the top of the code and propagate the values down through the code. This is called *forward analysis*.

2. Dead Code Elimination

We need to use liveness analysis to do this code optimization. Here we determine if a variable is "live" at each step. By determining what code lines are still used through liveness, we can eliminate code no needed, i.e. *dead*.

A variable is *live* at statement *s* if the following is true:

1. There exists a statement s' that uses *x*
2. There exists a path from *s* to s'
3. That path has no intervening assignment to *x*

Again, we have transfer functions defined in the notes to help us determine liveness. Here, the transfer function is a *true* or *false* value indicating whether the code is live or dead.

To determine the liveness of a variable, we need to look at the code beneath the line using that variable. That means that we need to use *backward analysis*.

Examples:

1. In the following example, *x* is live at the line *x* = 10 since it is used by the print option.

$$x = 10$$
$$\vdots$$
$$print(x)$$

2.

$$x = 10$$
$$y = 5$$
$$z = x * 2$$
$$x = 3$$

print(*z*)
⋮ // x and y are not used anywhere below this code

In this example, z is live in the third line due to *print*(*z*). Similarly, $x = 10$ is alive because it is used by $z = x * 2$. However, $y = 5$ and $x = 3$ are both dead.

3. Global Escape Analysis

An object can *escape* if any of the flowing actions are preformed on it:

1. It is written into the field of an object.
2. It is passed as an argument to a method.
3. It has a method invoked on it.

We define the transfer function, E_{in} and E_{out}, as follows:

$E_{out} = true$ If the object referred to by x immediately after s can escape in the computation that follows s.

$E_{out} = false$ If no matter what path is taken after s, the object referred to by x can not escape.

This is *backward analysis* since you need to look at the code that follows to determine whether a variable escapes.

Examples:

1. Which of the new objects can escape?

$a \leftarrow new(A)$
$b \leftarrow new(B)$
$c \leftarrow new(C)$
$d \leftarrow new(D)$
$x \leftarrow a$
$y \leftarrow c$
$x.f \leftarrow d$
$x \leftarrow y.m(a)$

Answer:

new(A) escapes at *y.m(a)*
new(B) does not escape
new(C) escapes at *y.m(a)*
new(D) escapes at $x.f \leftarrow d$

2. What is the answer to the following transfer functions?

$$E_{in}(x, y \leftarrow x.m(z)) = true$$
$$E_{in}(x, y.f \leftarrow x) = true$$
$$E_{in}(x, x \leftarrow new(C)) = false$$