CS 536 Announcements for Wednesday, April 24, 2024

Course evaluation – log into heliocampusac.wisc.edu using your NetID

Last Time

- optimization overview
- peephole optimization
- loop optimizations

Today

- wrap up optimization
- copy propagation

Optimization Review

Goal: Produce "better" code that does the "same thing" as the original code.

- better = faster code, fewer instructions
- same thing = determined by observable behavior of code

When?

- before code generation (ie, on intermediate representation)
- after code generation (ie, on generated machine code)

Important considerations

- **performance/profitability** – want to be sure optimization is "worth it"
- **safety** – orginal source code, non-optimized target code, and optimized target code all do the "same thing" / have the same "meaning"

Look at optimizations that

- are ***sound*** transformations sound = all results that are output are valid
- recognize a behavior in a program & replace it with a "better" version

Copy propagation

copy statement `x = y.`

definition of `x`

use of `y`

x is an L-value

y is an R-value ie y can be a variable or literal

Idea: Suppose we are at *use* U of *x* and a *definition* D of *x* (of the form *x* = *y*) reaches U

- If
  1) no other definition of *x* reaches U **and**
  2) *y* does not change between D and U
- then we can replace the use of *x* at U with *y*

Example

```
x = 3.

y = 5.          ← useless definition (can be removed - as part of dead code elimination)

p = ~~x~~3.

if w*~~x~~3 > 9 [

    x = 4.

    z = ~~x~~4 + w*~~y~~5.
]

else [
    z = 2*~~y~~5 + ~~x~~3.      → do constant folding to change this to 13 (another optimization)
]

q = 5*p.        can't change p to x
                Could change p to 3 on a 2nd pass of copy propagation

s = z + x.

t = s + ~~y~~5.
```

How is this an optimization?

- can create useless code (which can then be removed)

if <u>all</u> uses of x reach by D are replaced,
then definition D can be removed (eg, y=5)

- can create improved code

t=s+y. RHS requires (at a minimum) 2 loads & one add

t=s+5. RHS requires only 1 load & 1 add (can use immediate value in add instr)

- constant folding

z=2*5+3. → z=10+3 → z=13

↑ now can copy propagate this def of z

- if done before other optimizations, can improve results

```
x=2.
if x<7 [
    $stmts
]
```

copy prop →

```
x=2.
if 2<7 [
    $ stmts
]
```

other optimization(s) →

```
x=2.
$stmts
```

Copy propagation (cont.)

def

use

Recall: Suppose we are at *use* U of *x* and a *definition* D of *x* (of the form *x* = *y*) reaches U

constant or variable

- If
  1) no other definition of *x* reaches U **and**
  2) *y* does not change between D and U
- then we can replace the use of *x* at U with *y*

So, to do copy propagation, we must make sure two properties hold:

Property 1) No other definition of *x* reaches U

Property 2) *y* does not change between D and U

How?

Property 1) No other definition of *x* reaches U

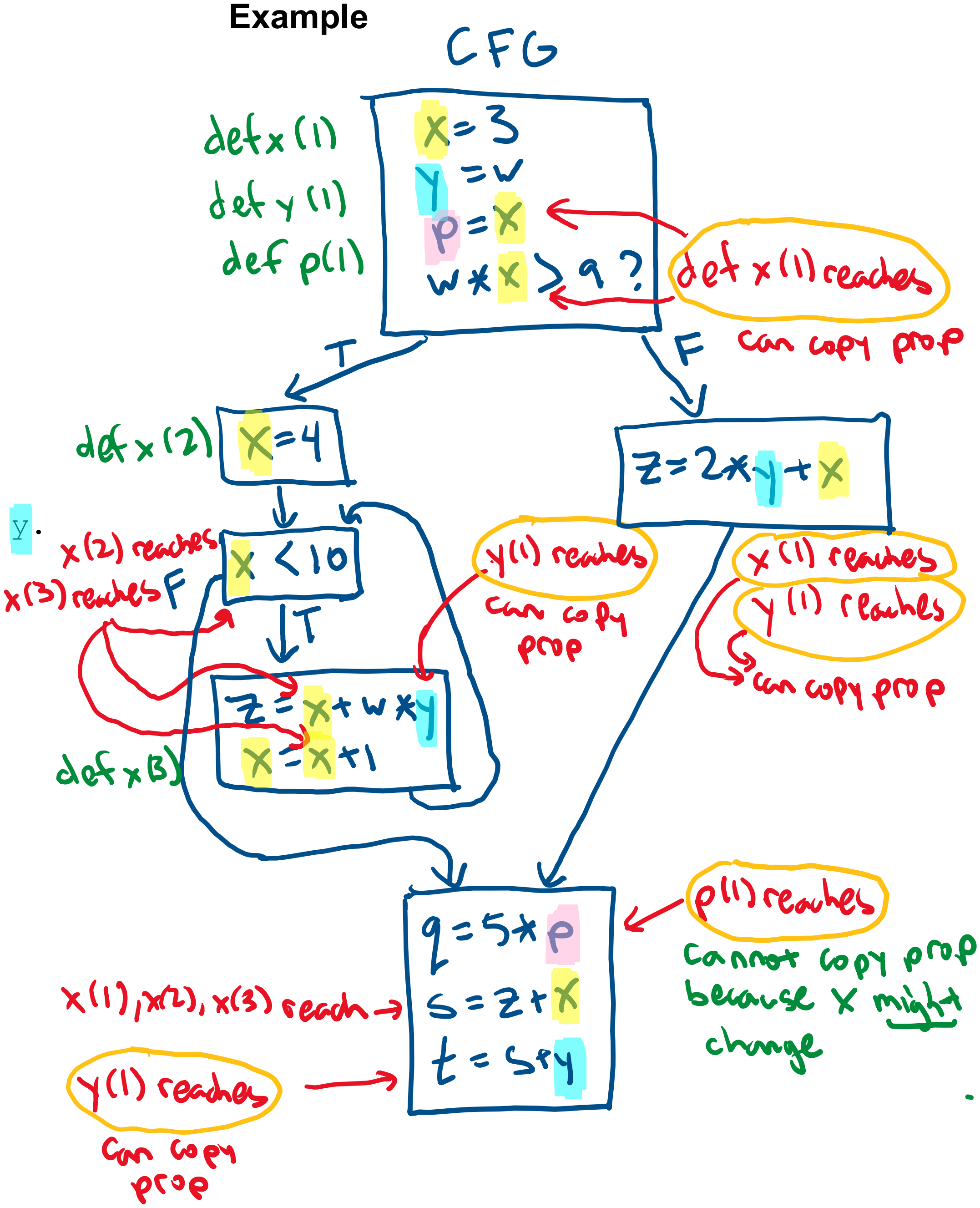
- How? Do a *reaching-definitions* analysis
  - one way: data flow analysis
  - another way: create control flow graph (CFG)

- do "backwards" search starting at U
- stop exploring a branch of a search when we find a def of x (but continue overall search)

Example

```
x = 3.
y = w.
p = x.
if w * x > 9 [
  x = 4.
  while x < 10 [
     z = x + w * y.
     x = x + 1.
  ]
]
else [
  z = 2 * y + x.
]
q = 5*p.
s = z + x.
t = s + y.
```

uses where only 1 def reaches



Copy Propagation (cont.)

Property 2) *y* does not change between D and U (of x)

- If *y* is a constant, then this is trivially true.
- If on any path through the CFG from D to U there is a definition of *y*, then

  y might change

- If *y* and *z* are aliases and there is a definition of *z* between D and U, then

  refer to same spot in memory

  y might change

```
x=y;
//code to make y&z aliases
z=5;
w=x+y;
```

can't replace x with y

In C/C++

```
x=y;
int *z=&y;
*z=5;
w=x+y;
```

*z & y are same place in memory

Example (cont.)

```
x = 3.

y = w.

p = ~~x~~ 3.

if w * ~~x~~ 3 > 9 [

   x = 4.

   while x < 10 [

      z = x + w * ~~y~~ w.

      x = x + 1.
   ]
]

else [
   z = 2 * ~~y~~ w + ~~x~~ 3.
]

q = 5 * p.

s = z + x.

t = s + ~~y~~ w.
```

← on a 2nd pass p(1) reaches & can copy prop (the constant 3) to get: q=5*3.

Optimization Wrap-up