

Tour of common optimizations

Simple example

```

    print
foo(z) {
    x := 3 + 6;
    y := x - 5;
    return z * 4;
}

```

Handwritten annotations:
 - A red arrow points from the word "print" to the function call "foo(z)".
 - A red circle is drawn around the expression "3 + 6" in the first line.
 - A red arrow points from the "9" in "3 + 6" to the "x" in the second line.
 - A red circle is drawn around the expression "x - 5" in the second line.
 - A red arrow points from the "4" in "x - 5" to the "4" in "z * 4" in the third line.
 - A red arrow points from the "4" in "z * 4" to the "z << 2" annotation.

Simple example

```

foo(z) {
    x := 3 + 6;
    y := x - 5;
    return z * 4;
}

```

Handwritten annotations:
 - "Constant folding (CF)" is written in red next to "3 + 6".
 - "Const prop (CP)" is written in red next to "x".
 - "4 (CF)" is written in red next to "x - 5".
 - "z << 2" is written in red next to "z * 4".
 - "Strength reduction" is written in red below "z << 2".
 - A large red bracket on the right side groups the annotations and is labeled "With simpl".

Another example

```

x := a + b;
...
y := a + b; x

```

Another example

```

x := a + b;
...
y := a + b; x

```

Handwritten annotation:
 - A red bracket on the right side groups the three lines of code and is labeled "only if x, a, b not modified!".

Another example

```

if (...) {
    a := read();
    x' := a + b;
    print(x);
} else { x' := a + b }

...
print(y)
y := a + b; x'

```

Another example

```
if (...) {  
  a := read(); t := a + b  
  x := a + b; t  
  print(x);  
} else { t := a + b; }  
  
...  
y := a + b; t
```

Partial Redundancy Elimination PRE

Another example

```
x := y  
...  
z := z + x
```

Another example

```
x := y  
...  
z := z + x * y
```

*x, y not modified
Copy prop*

Another example

```
( x := y  
  ...  
  z := z + x * x
```

What if we run CSE now?

x := E

*... E
x*

Another example

```
x := y  
...  
z := z + x * x
```

What if we run CSE now?

Another example

```
x := y**z  
...  
x := ...
```

Another example

```
x := y**z
...
x := ...
```

*if x is not used
dead assignment elim
(unused assignment elim)*

- Often used as a clean-up pass

```
x := y      Copy prop   x := y      DAE   x := y
z := z + x  →          z := z + y →   z := z + y
```

Another example

```
if (false) {
    ...
}
```

Another example

```
if (false) {
    ...
}
```

*dead code elim
(unreachable code elim)
Another common clean up opt*

Another example

- In Java:

```
a = new int [10];
for (index = 0; index < 10; index++) {
    a[index] = 100;
}
```

Another example

- In "lowered" Java:

```
[a = new int [10];] a.length = 10
for (index = 0; index < 10; index++) {
    if (index < 0 || index >= a.length()) {
        throw OutOfBoundsException;
    }
    a[index] = 10;
}
```

*note
[0..9]*

Another example

- In "lowered" Java:

```
a = new int [10];
for (index = 0; index < 10; index++) {
    if (index < 0 || index >= a.length()) {
        throw OutOfBoundsException;
    }
    a[index] = 0;
}
```

*Branch folding
+ unreachable
code elim
index ∈ [0..9] ← Range analysis
10 ← Kinda like CP
if we assume
that 0 acts
like a.length := 10*

Another example

```
p := &x;  
*p := 5  
y := x5 + 1;
```

Another example

```
p := &x;  
*p := 5  
y := x5 + 1; 6
```

pointer/alias analysis

```
x := 5;  
*p := 3  
y := x5 + 1; → ???
```

Another example

```
for j := 1 to N  
  for i := 1 to M  
    a[i] := a[i] + b[j]
```

*t₁ = a + i * 4*
*t₂ = *t₁*
~~*t₃ = a + i * 4*~~
**t₁ = t₂ + t*

Another example

```
for j := 1 to N  
  for i := 1 to M  
    a[i] := a[i] + b[j]
```

t := b[j]
Loop invariant
Code motion

Another example

```
area(h,w) { return h * w }  
  
h := ...;  
w := 4;  
a := area(h,w)
```

*h * w*
*h * 4*
h << 2

Another example

```
area(h,w) { return h * w }  
  
h := ...;  
w := 4;  
a := area(h,w)
```

*h * w*
~~*h * 4*~~
h << 2

Many "silly" opts become important after inlining

Optimization themes

- Don't compute if you don't have to
 - unused assignment elimination
- Compute at compile-time if possible
 - constant folding, loop unrolling, inlining
- Compute it as few times as possible
 - CSE, PRE, PDE, loop invariant code motion
- Compute it as cheaply as possible
 - strength reduction
- Enable other optimizations
 - constant and copy prop, pointer analysis
- Compute it with as little code space as possible
 - unreachable code elimination