Optimizations

Identify inefficiencies in intermediate or target code
Replace with equivalent but better sequences

- equivalent = "has the same externally visible behavior"

Target-**independent** optimizations best done on IL code
Target-**dependent** optimizations best done on target code

"Optimize" overly optimistic

- "usually improve" better

An example

Source code:

```
x = a[i] + b[2];
c[i] = x - 5;
```

Intermediate code (if array indexing calculations explicit):

```
t1 = *(fp + ioffset);    // i
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);    // a[i]
t5 = 2;
t6 = t5 * 4;
t7 = fp + t6;
t8 = *(t7 + boffset);    // b[2]
t9 = t4 + t8;
*(fp + xoffset) = t9;    // x = ...
t10 = *(fp + xoffset);  // x
t11 = 5;
t12 = t10 - t11;
t13 = *(fp + ioffset);  // i
t14 = t13 * 4;
t15 = fp + t14;
*(t15 + coffset) = t15;  // c[i] := ...
```


Kinds of optimizations

Scope of study for optimizations:

- **peephole**:
  look at adjacent instructions
- **local**:
  look at straight-line sequence of statements
- **global** (**int*ra*procedural**):
  look at whole procedure
- **int*er*procedural**:
  look across procedures

Larger scope ⇒ better optimization, but more cost & complexity

Peephole optimization

After code generation, look at adjacent instructions
(a "peephole" on the code stream)

- try to replace adjacent instructions with something faster

Example:

```
sw $8, 12($fp)
lw $12, 12($fp)
   ⇒
sw $8, 12($fp)
mv $12, $8
```

More examples

On 68k:

```
sub sp, 4, sp
mov r1, 0(sp)
   ⇒
mov r1, -(sp)
```

```
mov 12(fp), r1
add r1, 1, r1
mov r1, 12(fp)
   ⇒
inc 12(fp)
```

Do complex instruction selection through peephole optimization

Peephole optimization of jumps

Eliminate jumps to jumps
Eliminate jumps after conditional branches

"Adjacent" instructions = "adjacent in control flow"

Source code:

```
if (a < b) {
  if (c < d) {
    // do nothing
  } else {
    stmt1;
  }
} else {
  stmt2;
}
```

IL code:

Algebraic simplifications

"constant folding", "strength reduction"

```
z = 3 + 4;

z = x + 0;
z = x * 1;

z = x * 2;
z = x * 8;


z = x / 8;

double x, y, z;
z = (x + y) - y;
```

Can be done by peephole optimizer, or by code generator

Local optimization

Analysis and optimizations within a **basic block**

Basic block: straight-line sequence of statements

- no control flow into or out of middle of sequence

Better than peephole
Not too hard to implement

Machine-independent, if done on intermediate code

Local constant propagation

If variable assigned a constant,
replace downstream uses of the variable with constant

Can enable more constant folding

Example:

```
final int count = 10;
...
x = count * 5;
y = x ^ 3;
```

Unoptimized intermediate code:

```
t1 = 10;
t2 = 5;
t3 = t1 * t2;
x = t3;

t4 = x;
t5 = 3;
t6 = exp(t4, t5);
y = t6;
```


Local dead assignment elimination

If l.h.s. of assignment never referenced again before being overwritten, then can delete assignment

E.g. clean-up after previous optimizations

Example:

```
final int count = 10;
...
x = count * 5;
y = x ^ 3;
x = 7;
```

Intermediate code after constant propagation:

```
t1 = 10;
t2 = 5;
t3 = 50;
x = 50;
t4 = 50;
t5 = 3;
t6 = 125000;
y = 125000;
x = 7;
```


Local common subexpression elimination

Avoid repeating the same calculation

- CSE of repeated loads: **redundant load elimination**

Keep track of **available expressions**

Source:

```
... a[i] + b[i] ...
```

Unoptimized intermediate code:

```
t1 = *(fp + ioffset);
t2 = t1 * 4;
t3 = fp + t2;
t4 = *(t3 + aoffset);

t5 = *(fp + ioffset);
t6 = t5 * 4;
t7 = fp + t6;
t8 = *(t7 + boffset);

t9 = t4 + t8;
```