17 Code generation 2 - Low level optimisation

What is peephole optimisation?

The final stages of optimisation are performed directly on the generated machine code. This is usually called peephole optimisation, since it works by moving a sliding window of fixed width (in terms of numbers of instructions) across the code and looking for patterns which can be replaced with more efficient ones. The scope for ingenuity is considerable and here we look at a few of the commoner cases. It is sensible to consider the code as being held in a list of chunks which correspond to the basic blocks used in high level optimisation, since many of the optimisations affect jumps and labels.

Redundant loading and storing

A sequence which derives from naïve code generation of the statements:

```
A := B + C;
A := A + 2 * B;
```

might be:

```
MOV B,R2
ADD C,R2
MOV R2,A
MOV A,R2
MOV B,R3
MUL 2,R3
ADD R2,R3
MOV R3,A
```

A two instruction window would spot the storing and immediate reloading of `A`, giving instead:

```
MOV B,R2
ADD C,R2
MOV R2,A
MOV B,R3
MUL 2,R3
ADD R2,R3
MOV R3,A
```

A larger window might spot that `A` is stored twice without being loaded in between, giving:

```
MOV B,R2
ADD C,R2
MOV B,R3
MUL 2,R3
ADD R2,R3
MOV R3,A
```

and also spot that B is loaded twice without being stored. This cannot be simply removed, but might exploit the lower cost of register to register copying to give:

```
MOV B,R2
MOV R2,R3
ADD C,R2
MUL 2,R3
ADD R2,R3
MOV R3,A
```

Strength reduction again

Strength reduction can also be applied in peephole optimisation. In fact the last optimisation above could be thought of as an example of this. More conventionally the multiplication by 2 can be replaced by a shift.

```
MOV B,R2
MOV R2,R3
ADD C,R2
LSH 1,R3
ADD R2,R3
MOV R3,A
```

Multiple operand substitution

For a three operand machine the above code can be reduced to:

```
MOV B,R2
ADD C,R2,R3
LSH 1,R3
ADD R2,R3,A
```

It is important not to make these changes without considering further uses of the registers, but if the code generation has been truly naïve it is safe in most cases.

It may also be important not to replace a load into a register of a value which is then used more than once with direct use of the value stored in memory in each case, since this implies repeated fetching of that value. This will actually depend on the caching strategy of the machine.

Unreachable code

If any code without a label is left following immediately after an unconditional jump, this may be removed. This can have the effect of providing conditional compilation even in languages without it in their definition. Thus source code containing the form:

```
if TRACE=0 then begin
      ...
      ...
end else if TRACE=1 begin
      ...
      ...
end
```

might have either sub-sequence removed. This on its own would not remove the jumps and labels.

Redundant jumps

The dead code removal above, as well as poor code generation earlier, might leave something like:

```
      TST TRACE
      JNZ L1
      GOTO L2
L1:      Tracing statements
L2:
```

This can be spotted as a jump over a jump and improved in various ways, such as:

```
      TST TRACE
      JZ L2
      Tracing statements
L2:
```

Constant folding

In general we can be very ingenious about exploiting constant values. If we have propagated constants through the code in high level analysis, they may now be visible. For example, in the example above the `TST TRACE` may be known to always be 1 and so both tests in the original code may be removed, leaving the unconditional execution of the code defined by the value of TRACE.

Other constants can be removed where the arithmetic/algebraic properties of expressions mean that they have no effect. This includes multiplication by one, addition of zero etc. Such effects may result from simplistic code generation for array accesses, particularly where memory allocation is postponed to take advantage of range information on types. Thus it may be possible on a particular machine to use single bytes to store values of an integer sub-range and so array accessing may result in scaling by an item size of one.

Other jump/label optimisations

Sometimes code may contain a number of jumps to a label which is attached to an unconditional jump. These can be redirected to the destination of this jump instead. If this leaves no further jumps to the original label, the jump it labels can be removed.

It may be useful to shorten all jump distances, to allow shorter forms of the jump instruction to be used. This can be done by chaining jumps which end at the same label. This results in the opposite of the effect in the previous paragraph and requires some careful weighing of the costs and benefits on a particular machine.

Macine specific instructions

Many of these optimisations depend on machine specific characteristics, but two not mentioned are use of auto-increment/decrement for variables and use of branch and link at the top of a loop to store a return address into a register.