ECE251: Programming Languages & Translators Fall 2010

Lecture 20 — October 21, 2010

Derek Rayside *version 1*

1 Optimization

Optimization is usually easiest to perform mechanically on three address code (which we saw last time). In these examples, since we are optimizing manually, we will work with the regular syntax.

There are many different kinds of optimizations that compilers perform. Some optimizations are inverses of other optimizations.

1.1 Loop invariant code motion

```
for (i = 0; i < n; ++i) {
    x = y + z;
    a[i] = 6 * i + x * x;
}
```

Loop invariant code motion:

```
x = y + z;
t1 = x * x;
for (int i = 0; i < n; ++i) {
    a[i] = 6 * i + t1;
}
```

1.2 Constant Propagation + Folding Example

```
int a = 30;
int b = 9 - a / 5;
int c;
c = b * 4;
if (c > 10) { c = c - 10; }
return c * (60 / a);
```

Constant propagation + Folding:

```
int a = 30;
int b = 3;
int c;
c = 3 * 4;
if (c > 10) { c = c - 10; }
return c * 2;
```

Dead code elimination on a and b:

```
int c;
c = 12;
if (12 > 10) { c = 2; }
return c * 2;
```

The conditional is always true:

```
return 4;
```

1.3 Thought questions

1. What order to apply optimizations in?
2. How many times to apply them? (Until we reach a fixpoint.)
3. Do we know that this process will converge?
4. Will it really make the program faster/smaller/etc?
5. Will we reach an optimum?

2 Register Allocation

The software illusion: unbounded space (local variables, dynamically-allocated memory). The hardware reality: a finite machine.

Storage hierarchy: disk, RAM, L2, L1, registers

OS implementor controls swapping RAM to disk. (swapping/paging/virtual memory)

Chip designer controls what's in the caches.

Compiler writer controls which variables are stored in the registers and which are *spilled* into main memory. *Register allocation* is the process of deciding which variables will be stored in the registers and which will be spilled. This is an NP-complete problem. A common technique is graph colouring.

The problem with register allocation is that there are usually more local variables than registers, especially on an architecture like the x86, which has 4 general-purpose registers. We allocate room for local variables both on the stack and in registers.

3 Runtime support: Object Header

In the memory of a running program, each object (in an object-oriented language) comprises two parts: the *object header* and the field values. The object header is meta-data that the runtime system needs to know about the object. What exactly is included in the object header varies by

language. Memory-safe object-oriented languages commonly include the following in the object header:

Pointer to the Type Information Block: Each object has a type, and each object header (usually) has a pointer to a record ('information block') about that type. The TIB may contain the virtual method dispatch table.

Identity HashCode: In Java each object has an *identity hashcode:* a unique integer that is associated with it. (These are unique for all of the live objects at any given point in time; however, it is possible for a number to be reused for a different object that is created later in the execution.)

Lock: Locks are used to ensure disciplined mutation of data in a multi-threaded program.

GC bits: The garbage collector may want to associate some information with each object. A GC based on reference counting will include an integer counting the number of incoming pointers to the object. A mark-sweep GC will include some bits (usually two) to indicate that the object is still live during the mark phase (the sweep phase then collects unmarked objects).

Array length: If the object is an array, then its header will also include an integer indicating how long the array is. This is used to ensure that the array is accessed only within bounds.

By contrast, arrays in C (a memory unsafe language) are just chunks of memory and do not have an object header. How does one know how long the array is? By convention there is a null in the last position. The *buffer overflow* problem you may have heard of is about running off the end of the array and clobbering whatever happens to be in the next memory location. This can happen because of incompetence or because of malfeasance. There is no way for C to prevent this from happening because arrays do not have an object header, and so one never really knows how long the array is supposed to be.

Native Code Generation

After the compiler finishes optimizing the three-address code (for instance by evaluating constant expressions like 2+2), it can create native code, usually in assembly language. Each target architecture needs its own native code generation backend, which is annoying, so C works well as a compiler target as well.

The main issue in creating native code is allocating resources. In particular, the compiler needs to allocate registers and space for local variables (since there is no symbol table at runtime). You also need to generate appropriate instructions corresponding to the three-address code instructions, in particular for calling procedures or methods.