CS 131 Programming Languages

December 1, 2009

Storage Management

Two incompatible things were mentioned in the last lecture (they diametrically disagree with each other). First, if you're trying to build a program that creates lots of objects of same size, it could be efficient to do your own memory management on top of the built-in memory manager. For example, a *free list of pairs* is managed by the custom memory manager and the built-in memory manager is called only when the list needs to be grown. Such thing could be done to improve performance (versus calling `new()` every time). The second thing is that if you're using a modern garbage collector based on generations and copying, it will invoke `new()` really fast (~10 instructions). The problem is that if you have the first and the second, your program will run slower. The garbage collector will copy the free list even thought it might be mostly composed of free space (garbage).

Performance, Obsolete Performance Tricks

Here's another trick that is sometimes obsolete now. Suppose you're allocating a big object (such as a tree: `Tree t = new Tree(…)`). Suppose then you do a lot of computation based on `t`, after which you no longer need `t`, and then you do more computations. A common trick is to dereference `t` (set `t = Null`) so that garbage collector can reclaim it. In Python, this will reclaim the storage right away since it uses reference counts. However, modern compilers do *analysis of variables definitions and uses (defuse analysis)*, and after a last use of variable the compiler will arrange to garbage collect the variable, therefore rendering the trick obsolete. Also, a computation routine could create its own pointers to the target of `t` and therefore prevent garbage collector from claiming the storage. *Escape analysis* figures out whether the variable pointing to the object ever "escapes" (gets away into other parts of the code). If the variable stays in the local code (e.g., *no* functions are called with the variable as a parameter) then the compiler will not have to invoke heap manager to allocate the memory for the object. If variable is local to the code, its object is put on the local stack so a call to `new()` results in 0 instructions since the stack can be added to.

Let's look at another trick: in Java you can invoke a standard method `System.GC()`. Let's assume a traditional garbage collecting algorithm. One would put a call to the garbage collector before the code that was meant to be executed fast and without interruptions (e.g., real-time code that will fail if it doesn't execute within strict time bounds). There's an area of development that attempts to implement *real-time garbage collector* that is constantly freeing unused variables.

Let's look at another trick: there's a keyword `register` in C and C++. `register char *p, *q` declares two pointers and puts them into registers. `while(*p++ = *q++) continue;` is a string copy operations which put in registers is much faster (two instructions?). However, this relies on programmer intuition about performance (where the hot spots in the code are). For less experienced programmers, such intuition is usually wrong. Compiler will have to take other variables out of the registers (since it's a scarce resource) which might result in worse performance.

Here's another trick: `#define PUSH(p, v) (*--(p) = (v))` and `#define POP(p) (*(p)++)`. Let's say we have code as follows, which also does type checking and is in general nowadays preferred to macros:

```
static inline void push(T *p, T v) { *--p = v; }
static inline T pop(T *p) { return *p++; }
```

Let's look at a cost model for Prolog. Recall that in Prolog, goals are done left to right, rules are done left to right, and there's backtracking and failure. So, as you succeed, you push things on the stack (e.g., new structure) and as you fail, you pop from the stack and unbind older variables. For this to work, the compiler must keep track of which variables got bound (*the trail*). But there is also another way Prolog code could run: a recursive function that succeeds each time it calls itself, and therefore fills up the stack. When the end of the stack is reached, garbage collection can be done on the stack to free up the space to continue recursion.

Tail calls and inline functions are two ways to optimize the *call overhead* of the language.

Parameter passing conventions: *call by value* is the most common convention where caller evaluates the arguments and passes copies to the callee. The biggest problem with call by value is performance when passing big objects. The second most popular approach is to use *call by reference* where caller does not evaluate the arguments but rather addresses of the arguments and passes these addresses to the callee. The problem with call by reference is *aliasing*, which results in compilers generating slower, more confusing code. In Ada, there's another passing convention called *call by result* where caller passes a variable without evaluating it,

callee computes, and resulting variable is copied back to caller. *Call by value-result* fixes the aliasing issue in theory, but in practice aliasing in Ada results in an undefined behavior. *Call by name* does not evaluate the argument or its address; instead, it creates a *thunk* (parameter-less procedure) which will evaluate the argument when called. In that way, we're putting off argument evaluation for as long as possible.