Supplementary notes for lecture 4 – 1/30/09

Prof. Kamin

This note contains some further discussion on a few disparate topics that you identified as confusing in lecture 4, specifically: (1) What's the deal with "interpretation"? (2) How is JIT (just-in-time) compilation consistent with portability? (3) How are languages with run-time type-checking advantageous? (4) How do tagged values work?

Interpretation

As noted in class, programs can be executed either by translating them to machine language, or translating them to an abstract machine language (also called a "virtual machine language" or "bytecode") and then executing them in a simulator (like the SPIM simulator for MIPS machine code - http://pages.cs.wisc.edu/~larus/spim.html). It is also possible to execute the program directly by traversing the abstract syntax tree, without ever translating it to anything.

A short, but clear and accurate, discussion of these options is given in the Wikipedia entry: http://en.wikipedia.org/wiki/Interpreter_(computing).

It seems that the last of these options was the most confusing. To help explain it, here is a very, very simple example, based on the example given at the end of Wednesday's class. I gave the following abstract syntax (to which I've added one new constructor):

```
type stmt = Assign of string * expr
          | If of expr * stmt * stmt
          | Sequence of stmt list
and expr = Int of int | Var of string
          | Plus of expr*expr | Greater of expr*expr ;;
```

For example, this program:

```
x = 1;
y = 2;
if (x>y) {
   x = x+y;
   y = y+1;
}
else
   z = x+(y+10);
```

would be translated to this abstract syntax tree:

```
Sequence [Assign("x", Int 1);
          Assign("y", Int 2);
```

```
        If ( Greater(Var "x", Var "y"),
             Sequence [Assign("x", Plus(Var "x", Var "y"));
                       Assign("y", Plus(Var "y", Int 1))],
             Assign("z", Plus(Var "x", Plus(Var "y", Int 10))))];;
```

Here is an interpreter for this language –a program that executes programs in this language, without translating them at all:

```
let rec eval expr store = match expr with
     Int i -> i
   | Var x -> lookup x store
   | Plus(e1,e2) -> let i1 = eval e1 store
                    and i2 = eval e2 store
                    in i1+i2
   | Greater(e1,e2) -> let i1 = eval e1 store
                    and i2 = eval e2 store
                    in if i1>i2 then 1 else 0;;

let rec interpret stmt store = match stmt with
     Assign(s,e) -> let v = eval e store
                    in add s v store
   | If(e,s1,s2) -> let t = eval e store
                    in if t = 1
                       then interpret s1 store
                       else interpret s2 store
   | Sequence stmts -> match stmts with
                    [] -> store
                  | s::ss -> let store' = interpret s store
                             in interpret (Sequence ss) store';;
```

`interpret` executes a statement, while `eval` evaluates an expression. Each one uses a store, which gives values of variables. I haven't shown the definitions of add, which replaces a value in the store, and lookup, which gets the value of a variable from the store. There should also be a constant `emptystore` that represents a store at the start of a program. If prog is the program above, then calling

```
interpret prog emptystore;;
```

will return a store with x having value 1, y value 2, and z value 13.

As an exercise, try adding a constructor for while statements: `While expr stmt`, and adding this to interpret.

Why dynamic type-checking?

A student asked why people would use a language with dynamic type-checking? Static type-checking is very useful because it catches many errors before a program runs at all. With dynamic type-checking – where the compiler does not check for potential type errors – the problem is that type errors may escape detection until the program is already in production. (The solution for this problem is, of course, to test the program thoroughly; but it is still very helpful for the compiler to filter out programs with potential type errors.)

The answer is that people find dynamically-typed languages convenient. But why? One answer is that they don't like to declare types, but that's not a very good answer, because it is possible to design a language where you don't declare types but there is still static type-checking – Ocaml is an example.

I wish I had a really good answer for this, but here is one example of how dynamically-typed languages like Lisp or Python or Perl are convenient to use. If you want to create a collection of values – i.e. put several values of different types into a "package" that can be referred to by a single name – in a statically-typed language, you need to define a struct or class. You can't use an array or list, because those collections are homogeneous. (You may well ask why this has to be the case – why we can't have heterogeneous lists in a language with static type-checking – but that is a technical question I won't try to answer here.) So you have to define a structure in which you give the types. The great convenience you get in a dynamically-typed language is that you can just throw all these values into a list. Lists are so useful in these languages that you need to write many fewer class definitions. This makes writing small programs quick and convenient, because, unlike, say, Java, you don't need to start by defining a bunch of classes.

You pay for the convenience in the end. If you use lists in this way, then you need to remember where each value is stored in each list. That is, you may have decided that you will represent a value by a list in which the first element is a string, the second and third ints, and so on. But since the language does not enforce these types, you have to be responsible for using the list correctly everywhere. If you change your mind and decide to add a new "field" in the list, then you have to manually check that you are still using all parts of the list correctly wherever it is used. Again, this is fine for small programs, but when the program gets up to a few thousand lines, this can become an insupportable burden. (A good alternative would be a language where you could write dynamically-typed programs and then introduce static types gradually, but it has proven very difficult to define such a language.)

JIT compilation and portability

Java programs are built for portability – or "mobility" – in this way: The server compiles your program to a virtual machine program, which it puts in a ".class" file. The format of the class file is standard and published. You can write an emulator (a "Java virtual machine") in C, for example, and if you want to run it on a new architecture, you can just recompile it, as you would for any program. So mobility is achieved: as the writer of the program, I just need to compile it once and then I can make the program available to anyone and they can run it as long as they have a virtual machine. If

instead Java were implemented by translating to native machine code, I would have the burden of compiling it to native code for every different type of processor (and different OS's as well).

However, execution of the JVM program in a simulator will be inefficient. As a partial solution, if I'm writing a virtual machine for a new architecture, I can do this: Instead of emulating the JVM instructions, I can translate them to native machine code. This will improve efficiency. For the writer of the Java program, nothing has changed – there is just one .class file for each class. Every major Java Virtual Machine implementation for a popular architecture, like x86, includes such "just-in-time" compilation.

I said this is only a partial solution to the efficiency problem because the compiled programs will still not be as efficient as a compiled C program. There are two reasons for this: First, the language itself makes it hard to produce efficient code; for example, dynamic type checks still need to be done (the language requires it), which take time. Second, the just-in-time compiler is constrained to compile code quickly, because the user is waiting for an answer; the virtual machine has no way of knowing whether spending time on compiling to native machine code will pay off. As I mentioned in class, compiling to highly optimized native machine code is complicated and time-consuming. A C compiler can take a lot of time generating optimized code (especially if you use the –O4 option), but a just-in-time compiler can't. So it is likely to produce less efficient code.

Tagged values

Any language with dynamic type-checking, or a language like Java, where the run-time system has to do type-checking for enhanced security, has to "tag" values. If the Java virtual machine is executing an "ADDI" instruction, it needs to check whether the arguments it's adding are, in fact, integers; it can only do this if all values somehow contain information saying what types of values these are.

It is difficult to explain this, because there are many different ways to implement tagging. But let's just try something simple. Consider the Java virtual machine. In Java, values are either simple – integers, floats, bools, etc. – or objects. The stack contains these two kinds of values. We can do the following: All values on the stack will be represented with an additional word beyond what is required for the value itself; 32-bit integers will be represented by two words; doubles will be represented by three words; and objects, which are pointers into the heap, will be represented by two words. The first word will contain a tag: 0 for objects, 1 for integers, 2 for floats, etc. Objects are represented by chunks of memory in the heap, which give the values of the fields of the object. In addition to these fields, the heap item will contain an additional word that points to a description of the class of the object. That description will be a chunk of memory that gives the name of the class and the types of fields; those types will be represented in a similar way: a value of 1 for integers, 2 for floats, and, for objects, a pointer to the description of that class.

To execute an ADDI instruction, you're given two pointers, p1 and p2. Look at the word pointed to by each of these pointers (*[1 and *p2) and make sure it is a 1. If so, then add the second words (*(p1+4) and *(p2+4)), and create a new value consisting of one word containing 1 and one word with the sum.

Clearly, this will be a lot less efficient than doing an addition in a compiled language like C, which consists of a single native machine code add instruction, and will also use a lot more memory.

Again, there are many ways to implement tagging that are more efficient than this. But the basic idea is always the same: add some additional bits to characterize the type of the values, check those values when doing any operation, and add the tag bits whenever a new value is produced.