CS 131 Programming Languages

October 15, 2009

Types in the Homework 2

What is the type of *matcher* in the homework 2? In OCaml type inference is so smart that it will deduce a more general type that you'd want for your code. Here's a declaration of a type in C: `enum color { R, G, B };` We can then use the structure like this: `enum color x, y; int i, j; x = y; x = i;` This is a valid code because enum is of type int in C. In OCaml this can be also done: `type color = R | G | B;;` This type is more abstract than the int type in C. The designers of Java hated enumerated types so they left them out, but after users complained, enumerated types were finally included in the language.

Type Polymorphism

There's lots of different kinds of polymorphism. Let's talk about the different kinds and see how well they work.

Operator polymorphism: an operator or a method can be overloaded in C++ so that it has different implementations depending on the objects it is applied to. A function that accepts arguments of varying types is polymorphic. Example: `complex operator+ (complex, complex);` We've overloaded the operator + to work on *complex* values. When `a = b + c` is an operation with all complex values, instead of a regular operator, the overloaded one is called. Technical term for this is *name mangling* because a function name is derived for the function based on its arguments (something like `$plus$complex$complex`). One problem is that there is no standard on generating these names, so code compiled by different compilers may be impossible to link. Another problem is compatibility with C itself – how would one call the function whose name is automatically generated? Another problem is that the generated names might not always be unique, resulting in linking problems. What's happening at the abstract level is pretty simple. We're simply having short name at the syntactic lever stand for longer names at the implementation level.

Consider this C code: `double cos(double);` and the call is `cos(1)` – this will not result in an error because the *integer* 1 will be converted by the compiler into the appropriate type *double*. There is a run-time action that's needed to do that since one can't pass the integer data structure to a function that expects a double. What really happens is a call like `cos((double)(1))`. This is called *coercion* or *implicit type conversion*. There are some downsides to coercion.

Let's see polymorphism and coercion gone bad. `unsigned u = some_function(); int i = complicated_function(); if (i < u) { print("negative"); }` Let's say that `-1` is assigned to `i` and `0` is assigned to `u`. The code will not print "negative" as some would expect because in C, *a comparison of a signed and unsigned number is an unsigned one*, so `1` and `0` are compared. How would one fix that problem? One could reject such case at compile time, or compare it with a wider int, or do two comparisons to get the right answer.

Another example. `long long int i = …; double d = i; print(i); print(d);` (assume print functions output full, exact value). `if (i < d) { print("lt"); }` When i is greater than d, the output will nevertheless be "lt". The `long long integer i` is cast to `double` for the comparison – it is squashed from 64-bit value to a 52-bit value that's the floating part of the double. Therefore, sometimes an incorrect answer will be output when the value of integer `i` is so large that it cannot be correctly represented by 52 bits.

The moral is: coercions that lose information are dangerous. In C++ standard, rules on coercion and polymorphism take many pages to explain. Let's say we have functions `f(double,int)` and `f(int,double)` – which one would be called if we have a statement such as `f(3,5)`? We can either call the first one (3 will be converted to double) or the second one (5 will be converted to double). In such cases, C++ doesn't simply choose one over the other, but stops and complains about the ambiguity.

Duck typing: "If it quacks like a duck and it waddles like a duck, then it's a duck." In Python, there is no type checking other than runtime checking (?). If you want to know that something is a dictionary then you'll have to see if you can store key/value pairs, lookup key values, get next key, etc. If you can do these things in the object, then Python will let you use it as a dictionary. In Python, every function is polymorphic because a defined function will succeed for any type of its arguments as long as its operators work on those arguments. There's no coercion needed, therefore. It's a lot simpler than the complicated rules of C++ that go on for pages. The downside is that it's easier to make mistakes (since there's no compile-time checking) and it is slower than C++'s approach.

Parametric Polymorphism (not Ad-Hoc)

A function or a method has a *type* that's not exactly known at the time of writing, but has parameters of its own that are the *type parameters* that are filled in at the run time (or compile time) based on the parameters that are given to the function.

Let's see an example in Java. We have a collection of strings *c* and we'd like to remove all strings of length 1 from it. In Java we can write code that does not use parametric polymorphism:

```
for (Iterator i = c.iterator(); i.hasNext(); )
        if (((String)i.next()).length == 1)   // the cast (String) will result in run time overhead
                i.remove();
```

Here's the code with parametric polymorphism:

```
for (Iterator<String> i = c.iterator(); i.hasNext(); )  // this is checked at compile time, no run time overhead
        if (i.next().length == 1)
                i.remove();
```

Implementing Parametric Polymorphism

There are two basic ways of doing it: *templates* and *generics*. With templates, if you define a class with a type parameter (`class Foo<T>` where `T` is the type parameter), instantiating this class (such as `Foo<String>`) compiles the template all over again after doing all the checking. After this instantiation, one ends up with a regular, non-template class which can be used ordinarily. This method used in languages like Ada and C++.

With generics (in OCaml, Java) the definition and use looks the same, but the type checking and code generation is done once – no recompilation is required. The system knows that it's a generic type and does all type checking in a generic way. So, the type checking is at compile time in both templates and generics approaches, but in templates case it's at the instantiation of the class while in generics it's at the definition of the function.

Advantage of the templates approach is that the generated code can be optimized just for the type of arguments that it was instantiated with (at compile time). Another advantage is that in the template approach the `T` values can be of any size (doesn't need to be a pointer/reference/etc.) while with generic approach the T is assumed to be the same size (a pointer to an actual object). However, with templates approach, changing the definition of the function that uses templates results in recompilation of any code that uses it.

Advantage of the generics approach include less code bloat, less recompilation. Disadvantages include too much forced abstraction since the type checking rules are too complicated.

Let's look at some more examples. Suppose we have Java code like this:

```
List<String> ls = …;
List<Object> lo = ls;  // this is OK because every String is an Object
                       // lo and ls are two pointers to the same object
lo.add(new Thread());  // this is OK because Thread is an Object
String s = ls.get();   // get the most recent thing out of ls.
                       // but ls points to an object that contains a Thread!
                       // therefore lo = ls is a TYPE ERROR in Java.
```

What we can do in Java to have a list of anything:

```
List<?> la = …;               // a list of anything (generic type) – we're using a wildcard '?'
void printall(List<?> la)     // function that prints a list of anything
        for (Object i:la)     // create an Iterator i from la
                System.out.println(i);
```

Suppose instead of *printall* we want to do *displayShapes*.

```
void displayShapes(List<Shape> lsh)
        for (Shape s:lsh)
                display(s);    // display() can only display objects of type Shape
```

This will compile. But suppose we have a sub-class called `BeautifulShape` that extends `Shape`. Would `displayShapes()` be able to display `BeautifulShape`? No! Let's fix the code:

```
void displayShapes(List<? extends Shape> lsh)        // a list of anything that extends Shape – bounded wildcard
        for (Shape s:lsh)
                display(s);                          // display() can only display objects of type Shape
```

Suppose we want to copy values from one collection to another – out of array and into a collection.

```
static void convert(Object [] a, Collection<Object> c)
        for (i = 0; i < a.length; i++)
                c.add(a[i]);
```

The above code will not work on an array of Strings. Let's fix that:

```
static void convert(Object [] a, Collection<?> c)
        for (i = 0; i < a.length; i++)
                c.add(a[i]);
```

This isn't right either – we can't copy an array of Strings to the collection of Strings. What we need is something that will let us convert any type to any other type. We need an "array of anything" and "a collection of anything" but the two must be the same type of anything.

```
static <T> void convert(T [] a, Collection<T> c)
        for (i = 0; i < a.length; i++)
                c.add(a[i]);
```

This almost works. We'd like to allow the types and super-types to be allowed to passed as arguments. Final version:

```
static <T> void convert(T [] a, Collection<? super T> c)
        for (i = 0; i < a.length; i++)
                c.add(a[i]);
```

Reading assignment: all the chapters about Java.