Airs – Ian Lance Taylor

Go Interface Values

November 29, 2009 at 5:53 pm · Filed under Programming

While interface values in Go are flexible, they do have confusing aspects.

An interface value—e.g., a variable of an interface type— holds a value of some other type. The interface type is known as the static type, since that is the type that the compiler sees at compile type. The other type, which is only visible at runtime, is known as the dynamic type. The dynamic type is, by definition, never an interface type; it can be any other sort of type, though.

When you copy an interface value, via assignment or a function call, you are making a copy of the value of the dynamic type. This is how most types work in general. However, a very common case of using an interface is to have the dynamic type be a pointer type. In that case, when you copy the interface, you copy the pointer, but you of course do not copy the value to which the pointer points. In many cases, the methods of a type will take a receiver of pointer type; this is required if the methods are going to change the value itself. When that happens for a method which the interface requires, then the interface actually requires that the dynamic type be a pointer type.

What this means is that although interfaces are technically always copied as a value, in actual use they often behave as though they are copied by reference. That is, although there is no explicit marker, interface objects often act like pointers. This can be confusing until you understand what is really going on.

In my last post I mentioned that for gccgo, an interface value always holds a pointer to the value stored in the interface. Now I'm going to correct that: if a program stores a pointer (or a value of the type `unsafe.Pointer`) in an interface, then the value stored in the interface is the pointer itself. That is, gccgo does not store a pointer to the pointer (which would require allocating heap space to hold the pointer). This is a natural efficiency hack, since in practice most interface objects hold pointers.

This efficiency hack carries through to the implementation of methods. Methods are implemented to always accept a pointer as the receiver parameter. If the receiver type of the method is actually not a pointer, then the pointer is implicitly dereferenced, and the value copied, at the start of the method's code. This means that when calling a method on an interface, the value stored in the interface can be passed directly to the method, regardless of whether the dynamic type is a pointer or not. (As with the last post, the gc compiler does things somewhat differently.)