Polymorphism vs. Overloading

In ML, the + operator is overloaded, and the `map` function is polymorphic. Both operators can work on different datatypes. What's the difference?

As we know, + can work on both ints and reals. However, it is not the same function at runtime. When the compiler sees a + sign, it determines whether it is acting on `ints` or on `reals`, and decides, before runtime, which addition function to call.

This is not the case for polymorphic functions. In this case, the compiler generates one piece of code, that will be used for any data type passed through it. However, different types have different sizes: a `real` often takes more bytes than an`int`, and custom types could be even larger. How does the compiler generate a single piece of machine code, if it doesn't know how many bytes to allocate for the function argument?

The solution to that is to exclusively pass by reference in a polymorphic function. All pointers are the same size, so the compiler only needs to allocate enough space for a pointer for the argument. This solution comes at a cost. Passing by reference is often less efficient than passing by value, which is a reason why many languages do not support polymorphic functions.

The code for `swap` illustrates the difference between polymorphism and overloading.

```
fun swap a b = let val temp = !a
                in (a:= !b; b := temp)

                end;

template <typename T>
void swap(T& x, T& y){
T tmp = x; .....   ;
}
```

The C++ version doesn't generate a single piece of code for each type. It generates unique code for each type. The C++ `swap` is overloaded because different code is executed at run time depending on the type. While the ML `swap` is polymorphic because the same code is generated independent of the types. This require a uniform representation, which in case of ML consists of a pointer. For example, the variable `temp` will correspond to a pointer to an `int`, in case we are swapping integers. Instead, the variable `tmp` in the C++ code will be allocated on the stack and it will occupy the space for an `int`.