Interfacing C Routines to Poly/ML

David C.J. Matthews

July 5, 1995

1 Abstract

ML is a powerful programming language and you can do almost anything in it you could do in any other language. The exceptions are where you want to use some feature of the operating system that has not been provided in the standard libraries, perhaps because it is specific to one operating system. In that case you might want to add a C routine to the run time system to call that function. Alternatively you may have a piece of code already written in another language which you want to be able to use from ML.

This report describes how to interface C routines to ML, and the problems you may encounter.

2 The Simple Method

Suppose you have a routine written in C, or perhaps some other language, which you want to be able to call from ML. The simplest way of doing this is to make it into a complete program, and then run it using `ExtendedIO.execute`. This starts the program and returns streams connected to its standard input and standard output. You can send arguments to it down one stream and get results from the other one. You can provide it with a suitable functional interface so that it appears to the ML as an ordinary ML function.

3 Adding a Function to the Run-Time System

That is by far the easiest way of interfacing a C routine because it avoids any need to know about the internals of the Poly/ML run-time system. However, if you really want to incorporate your C routine directly into the driver program for Poly/ML this is how to do it.

The first step is to add your C routine into the driver. You can edit it into `run_time.c` which is just a collection of assorted run time functions, or put it as a separate module and link it in. In the latter case you will have to change

Makefile to compile and link it. Each function you want to call from ML has to have a name which ends in the letter "c". So we will assume that the function you want to call is my_fun_c.

The next stage is to find a *function number* for your function. This will enable the ML code to find your C function. You do this by looking at the entries in globals.h which begin SYS_. In the standard system the numbers up to 255 are used with some gaps for future enhancements. To avoid problems with future releases use 256 or above. You must increase SYS_vec_size to be at least one more than the number you have used. You should add an entry for your function.

```
#define SYS_my_fun 260

#define SYS_vec_size 261
```

You now need to edit the appropriate machine dependent files. The Vax and Sun versions, and others as they are produced, each have a pair of files in the run-time system that deal with all the machine dependent parts. The rest of the run-time system is common to all versions. The two machine dependent files are a C source and an assembly code source. If you are running on the Sun, for example, the files are sun_dep.c and sun_assembly.s, and on the Vax they are vax_dep.c and vax_assembly.s. You will have to make changes in both of these files.

In the C source you must add an entry to the interface vector. You do this by editting the routine MD_init_interface_vector and the declarations immediately above it. You must add a declaration of your C function but replacing the final "c" by an "a".

```
extern word my_fun_a;
```

Inside the function you add a line which looks like

```
    add_word_to_io_area(SYS_my_fun, &my_fun_a);
```

You can put it anywhere among the other calls, they are in no particular order.

Next you must edit the assembly code file. There are macros to help you here. You will find a set of lines which look like

```
CALL_IO1(convlong, REF, IND)
CALL_IO0(processtime, IND)
CALL_IO2(openstream, REF, VAL, IND)
```

You must add a similar line for your function. Each line is a macro which expands into a piece of assembly code which will take the parameters supplied by the ML call and pass them to the C function, and then take the result of the C function and pass it back to the ML.

The number immediately after CALL_IO is the number of parameters, 0 if the function has no parameters. The first of the arguments to the macro is the name of your C function, but without the final "c". After this come entries which describe the way that the parameters should be passed to your C function, either as the values themselves (`VAL`), or by reference (`REF`). There must be one of these for each of the parameters to your function, and they are given in the order in which the parameters are passed from ML which is the reverse order to the order of the parameters of your C procedure. Finally there is an argument to the macro which describes whether the result of the C function should be returned directly to the ML (`NOIND`) or whether the result of the C function is the address of the actual result (`IND`). So if your function has two parameters, the first of which is passed by reference and the second by value, the result being returned directly, you need an entry which looks like

```
CALL_IO2(my_fun_, VAL, REF, NOIND)
```

When do you use VAL, REF, IND and NOIND? VAL and NOIND can be used when the argument or result is a boolean or an integer that you know will be between −32768 and 32767. If it is a structured value, a string or an integer outside this range you will almost certainly have to use REF or IND. If you use REF your C routine will be given the address of the argument so you may have to change your C routine to accept this. Similarly if the result is IND you may have to change the way you return the result.

Now you have made all the changes to the run-time system you can remake the driver program. Assuming that works successfully you can use this new driver with your existing database and put in the ML function which will call your C routine. You need to put in an ML function with the appropriate type. You use a function from the PolyML structure which provides the interface and make a declaration of the form

```
val my_fun : string * int -> bool = PolyML.run_call2 260
```

The number 260 was the number used for `SYS_my_fun` in globals.h, and run_call2 is used because the function has two arguments. A type coercion must be given to give your function the correct type. The type you give it is up to you but remember that the arguments are in the reverse order to the C routine.

You cannot actually type in the ML as shown because the PolyML structure that is normally available does not have the run_call functions. They provide a loophole in the type system and the run-time system so they are deliberately made difficult to find. What you must do is put the declaration in a file, say `my_fun.ML`. You now have to enter the `poly` system and compile the file by using the following sequence of commands.

```
PolyML.poly();
runcompiler(ml_environ, ml_compiler, new(root_environ), ".ML")
    "my_fun";
^D
```

You can only do this in the top-level (parent) database and not in a database constructed with `make_database`. You will now be back in ML and you can try out your function.

4 Complications

That was the easy part. Unfortunately it is probably not going to be as easy as that. If your C routine is very simple, for example just calling an operating system function, you may be able get away with using it as it stands, otherwise you are probably going to have to make some changes to it to make it work with the rest of the run-time system.

The problem is that addresses of ML objects may be persistent addresses, and therefore cause a bus error if you use them as C pointers, or the objects they reference may move as a result of a garbage collection. This is the reason why `REF` is used to pass objects that may be addresses. A value passed as `REF` is put into the `save vector` and the value passed to the C routine is the address of the entry. The garbage collector knows about values in the save vector and will ensure that the objects they refer to are not thrown away, and that the addresses are updated if the objects move. Provided you always indirect through the vector you will be safe.

If the values passed in to your C routine can be addresses they may well be in the persistent form. To ensure that they are in the local form and to load the object if necessary there is a C routine called `load_persistent_object` which takes an address as an argument and returns it in the local form.

If you want to create an object on the heap you can call `alloc_and_save`. This takes as argument the number of words to allocate and allocates a piece of store, possibly after a garbage collection. The address of the store is put into the save vector and the address of the entry is returned. In this way multiple calls can be made without the results of previous calls being garbage collected. If you want to return such a value to the ML code you can return the address of the save vector entry and use `IND` in the `CALL_IO` macro.

Because strings are often passed into and out of the run time system there are two routines, `Poly_string_to_C` and `C_string_to_Poly` which take care of the conversion of formats, and any persistent address translation.

Putting addresses in the save vector is satisfactory if you only want them during a single call of your function. But sometimes you may want to be able to keep some information from one call to the next in a static variable. You can do this, but you must be careful if the values include addresses of objects on the heap. You will have to register a procedure with the garbage collector so that it will not throw your objects away. You do this by calling `register_gc_proc` with your procedure as the argument.

```
register_gc_proc(gc_proc)
```

```
gc_proc(update_op)
update_op(ptr, strength)
word **ptr;
int  strength;
```

The garbage collector will call `gc_proc` (your procedure) with a procedure to apply to the address of each of your addresses, `update_op`. The procedure you are given will update the address if the garbage collector moves the object. As well as the address it takes a `strength` value which is either 0, for a *strong* reference, or 1 for a *weak* reference. Strong references are the normal case and weak references are only used in exceptional circumstances. If you pass an address with the strength set to 1 the garbage collector may set the address to zero if there are no other references to the object. This can be used, for example, to close streams which are no longer referenced.

C programs often build up structures using calls to `malloc` and get rid of them with calls to `free`. The Poly/ML run time system has redefined these so that the store allocation will work with the rest of the system. The objects allocated by `malloc` are not moved by the garbage collector, but calling `malloc` may cause a garbage collection in order to find the space. Addresses allocated by `malloc` calls must not be passed back to the ML system directly since they do not follow the conventions of ML addresses. Instead you will have to put them into a static vector and return an index into it. In that way the ML program is given a small integer as the result of the call and this can be used as an argument to other calls. If you do this kind of thing you will have to think about what will happen if the user saves their database and then restarts in another session. Clearly the structures created with `malloc` will not be there in a new session.