Automatic FFI and API for Compiled High-Level Languages

Dakota Fisher
University of Colorado, Boulder

Miles Rufat-Latre
University of Colorado, Boulder

Abstract

To be able to write programs to perform certain classes of tasks, programming languages need to be able to refer to functions, variables and other symbols defined externally. FFI and API code permits programmers to import and export symbols, but this costs excessive programmer time, despite the type information needed to create this code being already available. This process can be automated extensively, reducing the time needed to port code from one language to another and freeing up programmers to work on other things.

1. Introduction

1.1 Terminology

C callback (a function in C called by Scheme)
Scheme [1] callback (a function in Scheme called by C)

2. Calling Convention

To fit all the constraints imposed by C, Scheme and System V ABI compliant x86-64 compiled code, calling convention requires special attention. The constraint imposed by C is that it should be able to write a function in C compilable in Scheme. The constraints imposed by the ABI are the calling convention. Scheme functions must fit proper tail recursion as described in Section 3.5 of R5RS [1]. An additional constraint is thus imposed by Scheme in that tail-call elimination must be possible.

2.1 Scheme Calling Convention

A new calling convention was required to fit the design constraints imposed by Scheme that are disallowed in compiled C code. In particular, the combination of both tail-call elimination and variable arguments using the calling convention described for x86-64 are impossible because the calling convention is a caller cleanup calling convention. For simplicity, the calling convention was reduced to storing an argument list in the rdi register of the processor. The choice of rdi is essential in permitting the straightforward production of wrapper functions in C.

2.2 Partial Application of Scheme Code

Given that a list of arguments is stored in the rdi register, partial application can be done by creating a new function which performs cons on the partially-applied parameter and the callers argument list, effectively adding an argument to the argument list, then jumping to a pre-determined function. The choice of rdi is to permit this partial application scheme to work on API and callback calls from C code with minimal effort (see next subsection, Partial Application on Compiled C Code). Effectively, partial application produces general-purpose closures.

Calling or dispatching to scm_cons (as the C function) provides some additional difficulties. In particular, a copy has to be made of every register that may be used as an argument in the callee of the partial application. For functions internal to Scheme, this is not a problem. However, for C callbacks, this means making copies of a large number of registers. To avoid this problem, the storage for a cons pair is provided in the closure, with the caveat that recursive calls of the function may change the second value stored in the pair. This adds an additional constraint to partially-applied functions. Any recursive calls utilize the same storage and thus overwrite the second value of the pair and thus the final access of the pair that obtains its second value must be made before any call is made that could eventually become a recursive call.

```
.align 16
jit_begin:

jit_func: /* places */
movq %rdi, jit_store(%rip)
```

```
leaq (jit_const+2)(%rip), %rdi
jmp *jit_const_func(%rip)

.align 16
jit_const:
.quad 0
jit_store:
.quad 0
jit_const_func:
.quad 0

jit_end:
```

3. Modifications to R5RS Scheme

3.1 New syntax

Three new special forms, import, export and exportfunc, are added to the subset of R5RS chosen, and are shown in BNF form below:

```
<import> -> (import <string>)
<export> -> (export <identifier>)
<exportfunc> -> (exportfunc <
    identifier> <type> (<type>*))

<type> -> (func type (<type>*))
   | (ptr <type>)
   | (union (<type>*))
   | (struct (<type>*))
   | <type_prim>
<type_prim> -> <anyctype> | SCM
```

The BNF rule ¡command or definition¿ specified in the formal syntax of R5RS Scheme is replaced with:

```
<command or definition> -> <original
     command or definition>
   | <import>
   | <export>
   | <exportfunc>
<original command or definition> ->
    <command>
   | <definition>
   | <syntax definition>
   | (begin <original command or
      definition>+)
```

3.2 New Semantics

All import statements are evaluated first and bind all procedures and variables specified in the C file to converted Scheme variables of the same name. Procedures are bound as new functions which converts all arguments and the return value to and from corresponding Scheme values appropriately, based on the C types. Variables are bound as their addresses in a foreign type and manipulated as such. The rationale for choosing this method is that variables in C do not correspond to values but addresses. An alternative semantics could automatically convert the type of the variable to and from its corresponding Scheme values, but has caveats in what it means to bind the variable (in particular, there are difficulties regarding what happens to the type of the contained data).

Export statements have no effect on internal Scheme code except that one cannot export an imported function or variable, as doing so creates two instances of a single global variable, and makes linking impossible. Running top-level code is necessary to initialize all Scheme variables, so exported procedures must first run all top-level code if not already run. Once that is done, the procedures return the converted result of applying the relevant Scheme function on converted arguments. Exported variables are exported as SCM types.

Wrappers: Due to using different calling conventions and due to the different type systems in play, FFI calls and API calls need compatibility wrappers. An FFI call wrapper takes a list of arguments, converts each argument into the appropriate C type, calls the function and then converts the result into a Scheme value. API call wrappers do the reverse, taking some arguments from C, converting them into Scheme values, applying the list of arguments to the function and then converting the result from a Scheme value to a C value. This is the bulk of the work that permits x86-64 ABI compatibility.

Suppose the imported function:

```
extern int add(int args);
```

The FFI call generated is:

```
SCM scm_import_add(SCM args) {
  int arg0 = scm_scm2int(scm_car(
     args));
  args = scm_cdr(args);
  int arg1 = scm_scm2int(scm_car(
     args));
  args = scm_cdr(args);
  return scm_int2scm(add(arg0,arg1))
     ;
```

```
}
```

At the beginning of evaluation of the Scheme program, a closure is created wrapping this function, with this assembly:

```
/* other variable declarations */
var_add:
.quad 0
/* other variable declarations */
scm_init:
.global scm_init
/* other variable initializations */
movq $scm_import_add, %rdi
call scm_make_closure
movq %rax, var_add
/* rest of code */
```

Suppose that instead, the add function is exported as the same type:

```
(exportfunc add int (int int))
```

The assembly produced makes var_add a global symbol. The API wrapper then becomes this C code:

```
int add(int arg0, int arg1) {
  if (!scm_initialized) scm_init();
  SCM f = var_add;
  SCM args = scm_null;
  args = scm_cons(scm_int2scm(arg0),
     args);
  args = scm_cons(scm_int2scm(arg1),
     args);
  return scm_scm2int(scm_apply(
     var_add,args));
}
```

Callbacks: If an FFI call returns a function pointer or an API call takes a function pointer as an argument, a new C callback (that is, a callback to C code, that Scheme calls) must be generated to convert the C function pointer to the appropriate Scheme function. To do this, a C callback wrapper is made. For example, suppose the imported function:

```
int force(int(*)());
```

Ideally, to generate this callback, one would be able to write a function:

```
SCM scm_import_force(SCM args) {
  int (*arg0)() = ???(scm_car(args))
     ;
  SCM args = scm_null;
  return scm_int2scm(force(arg0));
}
```

wherein ??? is a function converting scm values to appropriate C functions. The essential problem is that a new C function must be generated at runtime, which is impossible to do in pure C. While C has no facilities for functions which create functions, it is possible to import a facility which does do such a task. In particular, the partial application function for Scheme functions can also be utilized to generate new functions from C wrappers. Consider the following function:

```
int scm_icb_force_0(SCM sarg) {
  SCM f = scm_car(sarg);
  SCM args = scm_null;
  return scm_scm2int(scm_apply(f,
     args));
}
```

As compiled x64 code, the sarg argument is stored in %rdi. Recall that the Scheme method of partial application places the value of %rdi in a pair and then places the pair (as a SCM cons) in %rdi. The same partial application, on this wrapper, creates a function wherein the partially-applied function is evaluated on an empty list and its result converted to int. That is, given this function, a correct import of the force function is:

```
int scm_import_force(int (*arg0)())
   {
  int (*arg0)() = scm_get_func_ptr(
    scm_env_apply(scm_icb_force_0,
    scm_car(args)));
  SCM args = scm_null;
  return scm_int2scm(force(arg0));
```

Suppose instead that force is exported rather than imported. In Scheme:

The C callback wrapper for the first argument is:

```
SCM scm_ecb_force_0(SCM args) {
  int (*f)() = scm_car(args);
  return scm_int2scm(f());
}
```

```
extern SCM var_force;
int force(int (*arg0)()) {
```

```
  if (!scm_initialized) scm_init();
  SCM f = var_force;
  SCM args = scm_null;
  args = scm_cons(scm_get_func_ptr(
      scm_env_apply(scm_ecb_force_0,
      arg0)),args);
  return scm_scm2int(scm_apply(f,
      args));
}
```

Finally, functions can be returned from either by simply inverting what the callback wrappers do.

4. Conclusion

It has been demonstrated that it is feasible to considerably reduce the effort necessary to write FFI and API code for high-level languages by utilizing the type information available in header files. In addition, there exists a fairly general and relatively portable technique permitting this method to utilize callbacks, given availability of the instruction pointer.

References

[1] R. Kelsey, W. Clinger, J. Rees (eds.), Revised^5 Report on the Algorithmic Language Scheme, *Higher-Order and Symbolic Computation*, Vol. 11, No. 1, August, 1998.