The Layers of Larceny’s Foreign Function Interface

Felix S Klock II
Northeastern University
pnkfelix@ccs.neu.edu

Abstract

The Foreign Function Interface (FFI) in Larceny supports interacting with dynamically loaded C libraries via glue code written in Scheme. The programmer does not need to develop any C code to interface with C libraries. The FFI is divided into layers of Larceny Scheme code; the lower layers implement kernel functionality, and the higher layers support portable glue code development.

The lower level has two main features of interest. First, the FFI supports callbacks: objects that look like simple function pointers to foreign code but invoke Scheme closures when called. This requires generating specialized machine code, and is further complicated by the potential movement of the closures during garbage collections. Second, Larceny heaps can be dumped to disk and reloaded during a fresh runtime execution. The FFI automatically relinks the foreign procedures in such heaps.

The higher level layers provide macros and procedures for extracting information from header files and dictating how values translate between Scheme and foreign code. These tools ease development of portable glue code. The upper layers have simple implementations and do not require much Larceny-specific functionality; they may be useful for the FFIs of other Scheme systems.

1. Introduction

Scheme implementations cannot provide built-in access to all low-level libraries, and clients cannot be expected to reimplement them from scratch. Many Scheme implementations provide a Foreign Function Interface (FFI) to allow the connection of Scheme programs with foreign C libraries.

An FFI has many design axes. First, an FFI that only allows Scheme to hook into C functions that receive and produce values of a single scheme_value type (as in [Kelsey and Sperber(2003)]) forces the client to develop (write, compile, debug, etc) glue code written in C, rather than accessing the external library directly via Scheme expressions. [Barzilay and Orlovsky(2004)] motivates support for a more expressive FFI.

Second, transmitting complex objects requires bridging the gap between the semantics of Scheme and that of C. For example, making a Scheme closure appear to the C world as a C function pointer requires some semantic gymnastics, as the calling convention for invoking a closure may differ significantly from that of a C function pointer.

Finally, a low-level interface to a foreign library that requires hard coding offsets into native structures (see Figure 1), or transcribing full C structure definitions from header files can lead to glue code that works on one host but not others. Such code is fragile in the presence of C source-compatible changes to the library’s header files, such as the addition (or reordering) of fields in its structure definitions. Specifying an interface portably requires a more sophisticated approach.

In Larceny, we have developed a layered FFI. The lower layers constitute the kernel of Larceny’s FFI implementation; their description here is targeted at Scheme implementors. The upper layers aid development of portable glue code, and illustrate ideas worth incorporating into other Scheme systems. In particular, the interface provided by the define-c-info special form is a simple, structure-shy approach for portably interfacing with library frameworks written in C.

The next section shows example uses of the FFI. Section 3 describes how the lower layers of the FFI libraries work together with the Larceny runtime to handle value marshaling and procedure invocation. Section 4 describes the middle layer, which provides the most primitive interface we expect developers to use. Section 5 describes the higher layers that ease interfacing with foreign libraries. Section 6 describes related work and section 7 concludes.

2. Example FFI code

This section presents code using foreign functions, starting with low-level file-system examples and working up to GUI interactions. The tour starts with a misuse of the Larceny FFI: a low-level definition with a portability bug. The bug motivates our higher level tools, which we present in the remaining examples.

Figure 1 defines a directory listing procedure. Lines 5 through 10 link the UNIX procedures for opening, traversing, and closing a directory. It then defines dirent->name, a procedure that extracts filenames from dirent structures1 via the low-level (and unsafe) %peek-string procedure that constructs a Scheme string from a zero-terminated string of bytes at the given memory address.

There is significant machinery beneath the surface of Figure 1. For example, unix/opendir marshals its argument Scheme string to a zero-terminated byte array, matching the char* idiom for C strings.

On Mac OS X (Intel), Figure 1’s list-directory misbehaves:

```
> (begin (for-each system '("mkdir dtmp"
   "touch dtmp/abcdef"
   "touch dtmp/mnopqrst"))
   (list-directory "dtmp"))
("" "" "" "def" "pqrst")
```

1The author of Figure 1 presumably determined that the d_name field is located 11 bytes after the start of the dirent structure, perhaps by manual inspection of the header files on a Linux distribution, or perhaps by writing C code to reveal this information.
On Mac OS X, list-directory returns strict suffixes of the actual filenames in the directory. The hard-coded offsets to the names ties the code to one host and does not work on other systems.

Figure 2 shows a more portable definition of dirent->name. It uses the define-c-info special form of Larceny's foreign-ctools library, binding the identifier name-offs to the offset appropriate to the host. The developer did not have to provide the entire definition for the struct dirent type (a definition that may differ between operating systems, introducing a new portability issue). One only indicates a source file, via include<> "dirent.h"), and lists the fields of interest ("d_name") alongside identifiers to bind their offsets (name-offs).

Figure 3 defines a predicate distinguishing directories from other nodes in the file system. It illustrates some subtle policies in communicating with C procedures.

The definition of file-directory? uses the define-c-struct form to bind make-stat to a struct constructor and stat-mode to a field accessor. It also binds opendir-const to a preprocessor constant needed to compute with the mode. It then binds unix/stat to the foreign function:

```scheme
int stat(const char *path, struct stat *buf);
```

The foreign-procedure invocation linking unix/stat to stat uses *string to say that its first parameter is a Scheme string to be marshaled to a zero-terminated byte array. The linkages use *boxed to say that the second parameter is a Scheme heap-allocated object. The invocation of unit/stat maps the bytevector buf (produced by make-stat) to a pointer to the memory immediately after the bytevector's header and passes that pointer to the C stat function. stat initializes the bytevector's contents with information about the argument path. file-directory? finally determines whether the path is a directory by performing the Scheme equivalent of the C expression: !!((stat.st_mode & S_IFDIR). The FFI also supports callbacks: marshaling closures to C function pointers. Figure 4 presents an example with the C quicksort function, qsort. Callback invocation could cause garbage collections, which may relocate objects; therefore this code copies the unsorted bytevector into non-relocatable (but still managed) memory. The callback itself uses void*->word-ref to access memory via an address held in an opaque void*->rt record.

---

**Figure 1.** A [mis]use of the FFI

```scheme
(define file-directory?
  (let ()
    (define-c-info (include<> "sys/stat.h")
      (struct "stat" (st_mode "st_mode" (stat-mode)))
    (define unix/stat
      (foreign-procedure "stat" '(string boxed) 'int))
    (define file-directory? filename
      (let* ((buf (make-stat))
        (errcode (unix/stat filename buf))
        (cond ((zero? errcode)
         (let* ((buf (make-stat))
            (errcode (unix/stat filename buf)))
          (not (zero? (integer-logand mode s-ifdir))))
          (else (error 'file-directory? filename))))))
)
```

**Figure 2.** More portable dirent->name definition

```scheme
(define dirent->name : [Addressof dirent] -> String
define-c-info (include<> "dirent.h")
(struct "dirent" (name-offs "d_name"))
#peek-string (+ ent name-offs)))
```

**Figure 3.** Semi-portable file-directory? definition

```scheme
(define file-directory?
  (let ()
    (define-c-info (include<> "sys/stat.h")
      (const s-ifdir uint "S_IFDIR")
    (define-c-struct ("struct stat" make-stat
      (include<> "sys/stat.h"))
      ("st_mode" (stat-mode)))
    ;; unix/stat : String Bytevector -> Int
    (define unix/stat
      (foreign-procedure "stat" '(string boxed) 'int))
    ;; file-directory? : String -> Boolean
    (define file-directory? filename
      (let* ((buf (make-stat))
        (errcode (unix/stat filename buf)))
        (cond ((zero? errcode)
          (not (mode (stat-mode buf)))
        (else (error 'file-directory? filename))))))
)
```

**Figure 4.** Callback example (with result on little-endian systems)
Figure 5. Example of FFI callbacks in the Gimp Toolkit (GTK+)

Marshaling Scheme closures is handled by all of the layers working together; the lower layers provide the basic functionality for creating and invoking C callbacks, while the middle and upper layers ease interfacing to foreign functions with callbacks.

As a final example, figure 5 uses the Gimp toolkit to create a window that responds to key presses by printing their character value. This code builds upon the gtk library. Figures 6 and 7 present relevant snippets of the gtk library using high-level functionality further described in section 5.

Figure 5 marshals the Scheme symbol 'toplevel to the integer value of the C enum GTK_WINDOW_TOPLEVEL. Figure 6 uses the define-c-enum macro to introduce a gtkwindowtype symbolic enumeration, which marshals 'toplevel and 'popup to and from GTK_WINDOW_TOPLEVEL and GTK_WINDOW_POPUP. This marshaling happens only in contexts expecting gtkwindowtype, such as gtk-window-new invocations. The upper layers of the FFI implement enum support; the lower layers are oblivious to C enums.

Figure 6's invocation of establish-void*-subhierarchy! establishes classes of C pointers extending the void*-rt type. Foreign function invocations with arguments that do not satisfy the encoded subtyping relation signal an error. The special form (define-foreign (foo-bar-baz _____) _____) searches for a foreign export named foo_bar_baz (note the underscores) and then fooBarBaz, binding foo-bar-baz to the resulting foreign function if found.

Figure 7 links to the GTK+ function gtk_init. To satisfy the interface of gtk_init, it uses the combinators call-with-char** (marshaling a vector of strings to a char**) and call-with-boxed (taking values of C type T to 7*).

After tasting the FFI programming experience, we now delve into its implementation.

3. Lower layers of the FFI

This section describes the implementation of the FFI’s kernel functionality. During the invocation of callbacks and callbacks, control flows from the MacScheme machine through the Larceny runtime and into C code (and back again). Support for this is distributed amongst structures allocated on the Larceny heap.
3.1 Control flow of FFI invocations

Scheme code in Larceny is compiled and run in the environment of an abstract MacScheme machine, with its own stack and heap representations and conventions for using registers. The abstract MacScheme machine is supported by the Larceny runtime, implemented in C. System calls shift control from the MacScheme machine model to the runtime; during such shifts, MacScheme state is copied into C-accessible memory and the processor is reconfigured to follow the machine model expected by the runtime’s compiled C code.

Foreign libraries expect to be invoked using the C machine model. It would be nice for FFI invocations to reuse the shift of machine model implemented to support Larceny system calls. That is, we desire an FFI callout that jumps into the Larceny runtime and then directly to the target foreign function. We would also like a callback to be a pointer to a Larceny runtime function that shifts into the MacScheme machine when invoked.

Unfortunately, we cannot implement this approach directly.

3.1.1 Customized machine code is necessary

The foreign target of an FFI callout expects its parameters to be set up according to the calling convention of the application binary interface (ABI). We do not want to code a separate system call for each possible argument combination. Also, an FFI callback must appear to be a C function pointer that consumes some number of parameters that depends on what function type the callback is emulating and somehow knows which Scheme closure it is associated with; no fixed function implemented in the runtime would suffice for this purpose.

Instead of having the Larceny runtime directly interact with foreign functions, a fragment of dynamically generated machine code sits between the Larceny runtime and the world of foreign functions. We call each such fragment an FFI trampoline.

3.1.2 Control flow of a callout

The FFI trampoline generated for calling out to a foreign function \( f \) declared to have type \( T \) can be thought of as implementing a “scatter arguments for \( T \) and invoke \( f \)” operation, illustrated in figure 8.

The trampoline code has a fixed input interface where it receives a set of arguments (packaged as an array in memory). It is responsible for distributing the arguments from the ABI-specified format expected by the compiled C code for a function of type \( T \). The trampoline code must then invoke \( f \) according to the calling convention. The trampoline code copies the value returned from \( f \) into a receiving area established by the runtime, and then returns control to the runtime. The runtime marshals the returned value back to the MacScheme machine. Section 3.2.3 has more details on this structure.

3.1.3 Control flow of a callback

The trampoline code generated for a callback to a Scheme procedure \( p \) and emulating a foreign function of type \( T \) can be thought of as implementing a “gather arguments of \( T \) and invoke \( p \)” operation, illustrated in figure 9.

The machine code receives its arguments according to the callback’s type \( T \) and the ABI. The code packages pointers to its arguments (copying from positions specified by the calling convention into a C stack allocated array when necessary), and then directly invokes the \texttt{ffi_convert_and_call} Larceny runtime function to perform the remaining work: set up the MacScheme machine, unpack the arguments according to the MacScheme calling con-
3.2 Structures supporting the FFI

Larceny Scheme source code is responsible for constructing the machine code that lies between the runtime and the foreign functions. Larceny FFI’s lower layers are factored into three components: the Larceny runtime itself, ABI-dependent Scheme source providing a small interface for constructing FFI trampolines for each target architecture and operating system, and ABI-independent Scheme source implementing the remainder of the low-level FFI.

This section describes the different structures allocated from Scheme code to support the FFI. We illustrate them using heap diagrams in figures 10 and 11. In the diagrams, circles denote objects scanned by the garbage collector (e.g., closures, vectors), rectangles denote unscanned objects (e.g., bytevectors), solid arrows denote object references traced by the collector (tagged pointers), and dashed arrows are untraced memory references (integer addresses).

Here are three invariants that the diagrams must observe to reflect a sound heap structure:

1. Solid arrows originate at circles
2. Dashed arrows cannot point into relocatable memory
3. No solid arrows point into the unmanaged C runtime state

These invariants motivate constructions introduced in this section.

3.2.1 Anatomy of a trampoline

The core of each FFI trampoline object is a list of bytevectors (called an ilist for “instruction list”), where each bytevector holds ABI-dependent machine code to accomplish a task, such as copying a double word argument packaged by the runtime into the appropriate location according to the calling convention, or performing the actual foreign invocation invocation. New bytevectors can be added to this list via the mutation procedures tr-at-end and tr-at-beginning.

After the necessary bytevectors have been added to a trampoline, the tr-pasteup allocates a nonrelocatable bytevector and copies all of the machine code fragments to it. This bytevector is the code for the trampoline; it is the intermediary between the runtime and the foreign function. The trampoline also clears the processor instruction cache if necessary.

Each callout trampoline must support a change-fptr operation, which takes an integer address of a foreign function as an additional argument. This operation modifies the ilist so that the invocation code targets the new foreign function. After change-fptr is invoked, tr-pasteup regenerates the code for the trampoline. The change-fptr operation supports relinking foreign functions during heap loads; see section 3.3.3.

3.2.2 Descriptors for primitive type signatures

At this lower level of the FFI, the argument list for a callout is made up of only fixnums or objects allocated on the Scheme heap. This argument list does not indicate on its own whether a given argument should be marshaled as a pointer, a signed 32-bit integer, or an unsigned 64-bit integer, etc. Invocations of c-ffi-apply pass along an encoding of the argument signature for the target function; we use a bytevector based encoding, where the ith byte indicates a primitive type.

<table>
<thead>
<tr>
<th>byte</th>
<th>primitive type</th>
<th>scheme types accepted</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>signed32</td>
<td>exact integer in $[-2^{31}, 2^{31}]$</td>
</tr>
<tr>
<td>1</td>
<td>unsigned32</td>
<td>exact integer in $[0, 2^{32}]$</td>
</tr>
<tr>
<td>2</td>
<td>iee32 (“float”)</td>
<td>flonum</td>
</tr>
<tr>
<td>3</td>
<td>iee64 (“double”)</td>
<td>flonum</td>
</tr>
<tr>
<td>4</td>
<td>pointer</td>
<td>bytevector, vector, pair</td>
</tr>
<tr>
<td>5</td>
<td>signed64</td>
<td>exact integer in $[-2^{63}, 2^{63}]$</td>
</tr>
<tr>
<td>6</td>
<td>unsigned64</td>
<td>exact integer in $[0, 2^{64}]$</td>
</tr>
</tbody>
</table>

Likewise, for return types we encode the primitive types signed32, unsigned32, iee32, iee64, signed64, unsigned64, as well as void. pointer is not a primitive return type; the FFI design assumes that if a foreign function is returning a pointer, it is a pointer into the C heap, could not be sensibly treated as a pointer into the Scheme heap, and thus should be marshaled as an integer, not pointer.

The current FFI does not support direct struct parameters or return types; only pointers to structures.

3.2.3 Anatomy of a callout

Figure 10 shows the heap structure for a callout: a closure that invokes a foreign procedure. The callout’s lexical environment carries three key components: an FFI trampoline, a bytevector describing the argument signature for the foreign function, and an integer
describing the type of the return value it expects from the foreign function.

The invocation of a callout first extracts the code associated with its trampoline. It then invokes the \texttt{c-ffi-apply} runtime system call, passing the trampoline’s code object, the signature bytector, the return code integer, and the list of arguments for the invocation. The system call first marshals the arguments into an \texttt{args} array according to the signature bytector and sets up a location on the stack for the trampoline code to write the result value returned by the foreign invocation. The system call then invokes the trampoline code, passing the \texttt{args} array and the result location along as arguments. When that returns, the \texttt{c-ffi-apply} system call proceeds to convert the result held in the return location into a Scheme value and returns it to the MacScheme machine.

### 3.2.4 Anatomy of a callback

Each callback is associated with a Scheme closure targeted for invocation. The garbage collector may \textit{move} the target closure. The callback’s code is just a bytector of machine code; if the closure’s address were directly encoded in the bytector, the garbage collector would not update that address (because the garbage collector does not modify the contents of bytectors), and the callback’s encoded reference to the closure would become invalid.

We resolve the problem of closures moving during garbage collections by introducing a level of indirection. Instead of putting a direct reference to the targeted closure in the callback code bytector, we create a nonrelocatable handle for every callback. The handle points to the closure, and the callback’s machine code holds an untraced reference to the handle.

Also, the callback and its target closure need to live as long as the foreign library could invoke it. Since the garbage collector is not going to scan foreign memory, we keep extra references to the callback and its handles in a manually managed list.

Figure 11 illustrates the resulting structure of a callback. From the Scheme side, a callback is an FFI trampoline coupled with two nonrelocatable handles: one that points to the target Scheme closure and another that points to a bytector holding the argument descriptors. From the C side, a callback is the address of the trampoline’s code. When foreign code invokes the trampoline, it first ensures that the arguments are all stack-allocated, and then invokes the runtime function \texttt{ffi_convert_and_call}, passing along the handles for the closure and the bytector of argument descriptors, as well as an array of argument addresses, and a descriptor and receiving location for the result when the callback returns. The implementation of \texttt{ffi_convert_and_call} is careful not to dereference the handles until after it has finished allocating state on the heap, so that potential garbage collections will not invalidate the dereferenced values.

### 3.3 Source code factoring of the lower layers

#### 3.3.1 Runtime system calls supporting the FFI

The runtime provides a small set of system calls to support the FFI. We limit the runtime code supporting the FFI, moving functionality into Scheme when possible.

- \texttt{c-ffi-dlopen} takes a path to a file holding a foreign library; it delegates to \texttt{dlopen} on UNIX (\texttt{LoadLibrary} on Win32) and returns a library handle (or 0 for errors).
- \texttt{c-ffi-dlsym} takes a library handle and a symbol name; it delegates to \texttt{dlsym} on UNIX (\texttt{GetProcAddress} on Win32) and returns the associated address (or 0 for errors).
- \texttt{c-ffi-apply} is described in section 3.2.3.
- \texttt{ffi-getaddr} extracts functions within the runtime, its used to get the \texttt{ffi_convert_and_call} function (see section 3.2.4).
- \texttt{make-nonrelocatable} takes a size and a type tag; it allocates (and initializes) an object that the collector cannot move.

There are also system calls for low-level memory interactions: object->address produces the address for an object on the Scheme heap, while \texttt{peek-byte}, and \texttt{poke-byte} provide unchecked functionality to read and write C runtime memory.

#### 3.3.2 Construction of callouts and callbacks

Callouts and callbacks have ABI-independent interfaces. From the viewpoint of a client of the FFI, a callout can be specified by just the name of the function being invoked, the library exporting the function, and the primitive types of the function’s arguments and return type (see section 3.2.2). Likewise, a callback can be specified by just the target closure along with the primitive types of the function’s arguments and its return type (from the viewpoint of C code).

Every callout and callback is associated with a trampoline structure. The construction of the trampoline requires the injection of ABI-dependent machine code. The injected machine code is processor dependent as well as calling-convention dependent.

The Larceny code base separates the ABI-independent interface from the ABI-dependent implementation using an object-oriented style of implementation. Each target supported by the FFI provides a \texttt{callout-abi} object that implements methods for constructing callout trampolines, and likewise a \texttt{callback-abi} object for constructing callback trampolines. This object-oriented style eases code reuse of details (such as instruction encodings) between different hosts.

#### 3.3.3 Relink on load

The lower layer of the FFI provides the kernel interface for constructing callouts. The last part of this layer maintains a table of the foreign functions that it has linked. If the heap is dumped and subsequently reloaded, FFI attempts to \texttt{reload and relink} all of the libraries and foreign procedures that were linked at the time the heap was dumped.
Two operations act together to support this. First, the trampoline object provides a change-fptr operation, which allows one to change the function address associated with a trampoline. To support this, a foreign callout does not directly reference the trampoline’s machine code, but rather pulls the code out of the trampoline on demand (see figure 10).

Second, Larceny provides a primitive, add-init-procedure!, which registers a Scheme procedure as an initialization routine. When the heap is dumped and later loaded, all of the initialization routines for that heap are invoked. The FFI maintains a list of foreign objects and registers an initialization procedure that will relink them during a heap reload.

4. Middle layer of the FFI

The lower layer of the FFI offers all of the basic primitives required to dynamically load a foreign library and hook into symbols exported by the foreign library. However, the interface provided by the lower layer is baroque.

The remaining middle and upper layers of the FFI are built upon the lower layer. The middle layer provides procedures for loading libraries and linking foreign functions. Part of the linkage functionality is an extensible domain specific language (DSL) for defining the interface to foreign functions. The upper layers build upon this interface by adding common patterns and automating some of the work of extracting information from header files for C source code.

At its core, the middle layer provides the following procedures:

- (foreign-file lib) opens the dynamic library specified by lib and registers it on a list of of loaded libraries.
- (foreign-procedure name param-types ret-type) searches the loaded libraries for an export of name and generates a callout invoking the function at the exported address.
- (foreign-procedure-pointer addr param-types ret-type) generates a callout invoking the function at addr.

Above, param-types and ret-type are s-expressions of the middle layer’s interface DSL. These arguments guide the marshaling of parameters from Scheme to C and unmarshaling of values passed from C to Scheme. For the remainder of this section we focus on the interface DSL used for param-types and ret-type.

4.1 FFI attribute entries

The lowest layer of the FFI expresses all data in terms of a fixed set of primitive types like “unsigned 32-bit word” and “64-bit floating point number.” Foreign libraries are often written in terms of C types like char or int. Therefore, the interface DSL introduces symbolic names such as 'char or 'int with intuitive mappings to foreign values, richer names such as 'string or 'bool, and complex symbolic expressions like '(- (-> (string) int). The middle layer translates these specifications into the primitive types of the lower layer.

We call these symbolic type expressions FFI attributes, or just attributes.

Each attribute can be thought of as describing a domain of high-level Scheme values, a domain of low-level Scheme values (that trivially correspond to foreign values) and the functions necessary to map elements of the Scheme domain into and out of the foreign domain. The middle layer associates every attribute s-expression with three components: a low-level primitive type descriptor, a Scheme procedure that marshals values from the high-level domain to the low-level domain, and a Scheme procedure that unmarshals values from the low-level domain to the high-level domain.

There are two kinds of attributes: a core attribute is a Scheme symbol registered in a table maintained by the middle layer; this table stores the association between such symbols and their low level descriptor and mapping functions. A constructed attribute is a non-atomic s-expression which the middle layer maps to appropriate attribute components.

4.2 Core (symbolic) attribute entries

There are a number of predefined core attributes. The simplest, 'byte, 'short, 'int, 'long, 'unsigned, 'uint, 'ushort, and 'ulong, all map to one of the descriptors for primitive integers, with marshaling that performs a range check but is otherwise the identity. Likewise 'float maps to the primitive ieee32 and 'double maps to the primitive ieee64.

The 'char and 'uchar attributes map to 32-bit integers, with marshaling that identifies characters with corresponding ASCII values. Both attributes do not handle characters that fall outside the expected range of ASCII characters gracefully. The 'bool attribute maps to the signed32 domain, marshals non-False Scheme values to 1 (#t to #f) and unmarshals 0 or #f (other integers to #t).

The more interesting built-in core attributes are those that represent objects with more state than fixed-width integers. There are three of these: 'boxed, 'string, and 'void*.

The 'boxed attribute maps to the 'pointer low-level descriptor, and marshals heap-allocated objects (pairs, vector-likes, bytevector-likes, and procedures) to themselves and #f to the foreign null pointer. There is no unmarshaling function; it is an error for a callout to indicate that it returns a 'boxed. The main values used with 'boxed are bytevectors; other heap allocated objects hold Scheme formatted words that foreign libraries do not generally process.

The 'string attribute maps to the 'pointer low-level descriptor. Marshaling and unmarshaling of 'string allocates a fresh object on the Scheme heap and copies character data into it.

Finally, the 'void* attribute is used to encode pointers to memory unmanaged by the Scheme runtime system.

4.2.1 The 'void* FFI attribute and void*-rt

Using the 'void* attribute wraps addresses up in a Larceny record, so that standard numeric operations cannot be directly applied by accident. Larceny’s record system is similar to that proposed for ERR5RS [Clinger(2008)]. The FFI uses two properties of the record system: the record type descriptor is a first class value with an inspectable name, and record types are extensible via single-inheritance.

The FFI provides void*-rt, a record type descriptor with a single field (a wrapped address). The FFI provides a family of functions for dereferencing the pointer within a void*-rt.

The 'void* attribute maps to the unsigned32 low-level descriptor. Marshaling checks that its input is an instance of the void*-rt record type and then extracts its wrapped address. Its unmarshaling function constructs an instance of void*-rt.

4.2.2 Extending the set of core FFI attributes

The public interface to many foreign libraries is written in terms of types defined within that foreign library. One can introduce new types to the Larceny FFI by extending the core attribute entry table. The ffi-add-attribute-core-entry! procedure consumes four parameters: a symbol (the high-level attribute), a low-
level type descriptor symbol, a marshaling function, and an unmarshaling function; it extends the internal table with the new entry. This extensibility is crucial; one can add new domains that correspond to the abstractions provided by particular foreign library. The upper layers of the FFI assist with common extensions.

### 4.3 Constructed FFI attribute entries

Core attributes suffice for linking to simple functions. Constructed FFI attributes express more complex marshaling protocols.

A structured FFI attribute of the form \( \rightarrow (s_1 \ldots s_n) \) allows passing functions from Scheme to C and back again. The low-level descriptor for such a form is a pointer to non-relocatable (and possibly unmanaged) memory; an \( \text{unsigned32} \) on 32-bit architectures.

To marshal a closure \( p \) of arity \( n \), the \( \rightarrow (s_1 \ldots s_n) \) attribute:

1. wraps \( p \) in another closure \( p' \) that unmarshals the foreign arguments of \( p' \) according to \( \{s_1\ldots s_n\} \), feeds the results to \( p \), and then marshals the value returned by invoking \( f \) according to \( s_r \). Note that \( p' \) is itself \text{not} acceptable by the lower layers.
2. Next the marshaling procedure for \( \rightarrow \) constructs a callback trampoline, \( p'' \), from \( p' \), using the callback construction procedure provided by the FFI’s lower layer.
3. Finally the marshaling extracts the code bytevector from \( p'' \), passing the address of the trampoline machine code as the \( \text{unsigned32} \) received by the foreign code.

The unmarshaling of a \( \rightarrow (s_1 \ldots s_n) \) FFI attribute accepts an address (the function pointer to be invoked), and constructs a callout to that machine code, using \( \{s_1\ldots s_n\} \) as the callout’s parameter attributes and \( s_r \) as its return type, as one would expect.

These two mappings naturally generalize to arbitrary nesting of \( \rightarrow \) FFI attributes, so one can create callbacks that consume callouts, return callouts that consume callbacks, and so on.

Other structured attribute entries encode common marshaling patterns. The structured attribute \( \text{maybe \ t} \) captures the pattern of passing \text{NULL} in C and \#f in Scheme to represent the absence of information. The low-level descriptor of \( \text{maybe \ t} \) is the same as that of \( t \); it marshals \#f to the foreign null pointer, and otherwise applies the marshaling of \( t \). Likewise, it unmarshals the foreign null pointer to \#f and otherwise applies the unmarshaling of \( t \).

### 4.4 Accessing foreign memory

If all foreign libraries provided a complete set of procedures for every kind of operation provided by the library, then the FFI might not need more than the \text{foreign-procedure} function. However, most C libraries are designed with the assumption that they will be used from C code that directly accesses and modifies the fields of structures in memory.

To support operations like extracting an integer field from a C structure, the middle layer provides a family of functions for reading and writing arbitrary addresses in memory. Such functions introduce a measure of unsafety to Larceny, since uncontrolled invocations could corrupt the internal state of the MacScheme machine.

On top of the two system calls \text{peek-byte} and \text{poke-byte}, the middle layer provides two large families of functions for observing and modifying low-level memory. One family is organized around exact bitwidths (e.g. \%peek8, \%peek16u, \%poke32); the other family is organized around primitive C types (e.g. \%peek-short, \%peek-ulong, \%poke-pointer).

### 5. Upper layer of the FFI

The upper layer of the FFI consists of various libraries that add syntactic sugar, capture common programming patterns, and aid in making code more abstract and portable.

#### 5.1 foreign-ctools

The \text{foreign-ctools} library provides a special form, \text{define-c-info}, that binds Scheme identifiers to values computed from the contents of C header files.

The interesting thing about \text{define-c-info} is its implementation (section 5.1.1); here we describe its specification.

Figure 12 presents the grammar of the \text{define-c-info} special form. The \( \langle c-decl \rangle \) clauses of \text{define-c-info} control how header files are processed. The compiler clause selects between \( cc \) (the default UNIX system compiler) and \( cl \) (the compiler included with Microsoft’s Windows SDK). The \text{path} clause adds a directory to search when looking for header files. The \text{include} and \text{include<>} clauses indicate header files to include when executing the \( \langle c-defn \rangle \) clauses; the two variants correspond to the quoted and bracketed forms of the C preprocessor’s \#include directive.

The \( \langle c-defn \rangle \) clauses bind identifiers. A \( \langle \text{const \ t \ "ae"} \rangle \) clause binds \( t \) to the integer value of \text{ae} according to the C language; \text{ae} can be any C arithmetic expression that evaluates to a value of type \( t \). (The expected usage is for \( t \) to be an expression that the C preprocessor expands to an arithmetic expression.)

The remaining clauses provide similar functionality:

- \( \langle \text{sizeof \ t \ "ae"} \rangle \) binds \( t \) to the size occupied by values of type \( t \), where \( t \) is any C type expression.
- \( \langle \text{struct \ "cn" \ · · ·} \langle x \ "cf" \ y \ · · ·\rangle \rangle \) binds \( t \) to the offset from the start of a structure of type \text{struct \ cn} to its \text{cf} field, and binds \( y \), if present, to the field’s size. A \text{fields} clause is similar, but it applies to structures of type \text{cn} rather than \text{struct \ cn}.
- \( \langle \text{ifdefconst \ t \ "cn"} \rangle \) clause binds \( t \) to the value of \text{cn} if \text{cn} is defined; \( x \) is otherwise bound to Larceny’s unspecified value.

#### 5.1.1 The implementation of \text{define-c-info}

Header files are usually written with the assumption that they will first be passed through a C preprocessor and then a C parser. Even after preprocessing, C is a tricky language to parse, due in part to its context-sensitivity. Furthermore, the contents of included system header files are sometimes written in a non-standard dialect of C, further complicating direct attempts to parse header files.

The \text{foreign-ctools} library resolves these problems by using a (perhaps surprising) “standard library”: the system’s C compiler itself.

The philosophy behind the \text{foreign-ctools} library is: “A C program generator is easier to write than a C parser.” That claim, combined with the common Scheme system procedure, procedural Scheme macros, and C pointer arithmetic, leads to the \text{define-c-info} design.

The \text{define-c-info} form is a procedural macro that:

1. generates a C program (in a temporary file).
2. compiles the program,
3. executes the program, printing results to another temporary file,
4. reads the output of the execution (usually numeric data), and,
5. expands to a Scheme expression binding the read values.

\footnote{This is binding in the sense of the \text{define} special form; at the top-level \text{define-c-info} introduces global definitions, and in internal definition contexts it introduces local definitions.}
of the expanded code does not define-c-info

entire set of fields or their ordering. One

be a useful way to develop code abstracted from system-specific

types, and the equivalences established by enum

or struct type, and the equivalences established by typedef.

Despite such shortcomings, the foreign-c Macros, the names of all of fields of an enum

or struct type, and the equivalences established by typedef.

5.2 foreign-structs

The foreign-structs library provides a more direct interface to C structures. Figure 14 presents the grammar of its define-c-struct

form. This form is layered on top of define-c-info; the latter provides the structure field offsets and sizes used to generate constructors and field accessors. The define-c-struct form combines them with the marshaling and unmarshaling procedures from the middle layer's DSL to provide high-level access to a structure.

5.3 foreign-enum

The foreign-enum library provides forms to associate the identifiers of a C enum type declaration with the integer values they denote. The foreign-enum library is layered above the foreign-c tools library.

The two forms introduced by the library are define-c-enum and define-c-enum-set (Figure 15). The define-c-enum form describes enums encoding a discriminated sum; define-c-enum-set describes bitmasks, mapping them to R\textsuperscript{0}RS enum-sets in Scheme.

Both forms expand into uses of the define-c-info form to extract integer values associated with the (c-name)'s. Both also invoke ffi-add-attribute-core-entry!, extending the attribute table with bindings for (enum-id).

The (define-c-enum en (---) (x\textsubscript{i} "cn\textsubscript{i}" ...)) form adds the 'en FFI attribute. The attribute marshals each symbol 'x\textsubscript{i} to the integer value that cn\textsubscript{i} denotes in C; unmarshaling does the inverse translation.

The (define-c-enum-set ens (---) (x\textsubscript{i} "cn\textsubscript{i}" ...)) form binds ens to an R\textsuperscript{0}RS enum-set constructor with universe resulting from (make-enumeration 'x (x ...)); it also adds the 'ens FFI attribute. The attribute marshals an enum-set s constructed by ens to the corresponding bitmask in C (that is, the integer one would get by logically or ing all cn\textsubscript{i} such that x\textsubscript{i} is in s). Unmarshaling attempts to do the inverse translation.

Unlike constructs derived from unguided automated processing of C header files, define-c-enum works on any set of integer

5 The constructors produce appropriately sized bytevectors, not record instances.

6 The inverse uniquely exists when the high-to-low mapping is a bijection, which depends on the denotations of \{cn\textsubscript{i} . . . \} assigned by the header files.
where capturing some common naming conventions found in C libraries. The define-foreign procedure is the heart of the foreign-stdlib library. It extends the FFI attribute entry table with a new primitive entry for (rtd-name sub-rtl), where sub-rtl must extend void*-rt. The resulting record represents a tagged wrapped C pointer, allowing one to encode type hierarchies.

This procedure is then used to establish the FFI attributes 'char', 'int', 'double', 'float', and 'char**'. For each such attribute 'T, there is a record type T-rt and a combinator function call-with-T that allocates (deallocates) an appropriately marshaled array on entry (exit) to a procedure parameterized over an instance of the corresponding record type.

For example, (call-with-char** strings proc) consumes a vector of strings and a procedure that consumes a char**-rt. It first allocates a array on the C heap, marshaling each argument string to a C string in the newly allocated array. Then it invokes proc on char**-rt wrapped address of the array. When proc returns, it deallocates the array. The call-with-boxed procedure uses a similar pattern to allocate a memory cell to hold any instance of void*-rt.

Finally, (establish-void*->subhierarchy! symbol-tree) is a convenience function for constructing large object hierarchies, such as that found in GTK+. It descends the symbol-tree, creates a record type descriptor for each symbol (where the root of the tree has the parent void*-rt), and invokes ffi-install-void*-subtype on all of the introduced types.

5.5 foreign-sugar

The (define-foreign (name arg-type ...) ret-type) form is the heart of the foreign-sugar library. This form is simple when name directly corresponds to a foreign function; then its expansion is:

(define name
  (foreign-procedure (symbol->string 'name)
    '(arg-type ...) 'ret-type))

The interesting case is when name is not a foreign export. Then the define-foreign form performs a search, applying a sequence of name generators to name until it finds an export from some foreign library. Each name generator maps a string to another string (or false when inapplicable). The library itself provides the sample name generators foo-bar-baz->foo_bar_baz and foo-bar-baz->fooBarBaz, which perform transformations capturing some common naming conventions found in C libraries.

The library also provides procedures to extend the set of name generators, changing the search strategy to deal with other naming conventions. One can devise “natural” mappings of foreign function names to Scheme procedure names. (However, there are phase issues when extending the set of name generators; one must ensure that the appropriate name generators are installed before performing the expansion of define-foreign.)

When this library was developed, Larceny’s reader case-folded by default, and many C identifiers did not directly correspond to Scheme identifiers. Automatically mapping Scheme-compatible names to their C counterparts was preferable to linking them by hand. With Larceny’s new case-sensitive reader, such name mapping is unnecessary and this library is less relevant.

6. Related Work

Almost every Lisp and Scheme implementation has some sort of foreign function interface; we cannot address all of them. Here we review some published treatments of interfacing to foreign libraries.

6.1 Interfacing high-level languages with foreign libraries

[Fisher et al.(2000)] present an FFI between SML/NJ and C based on data-level interoperability. It encodes much of the C type system directly into complex ML types. Their system supports preservation of foreign functions during a heap export in a manner analogous to how we support them during a heap dump. Their FFI avoids much of the complexity that we show in our lower layer because it does not support callbacks.

[Huelsbergen(1995)] presents an FFI between SML/NJ and C that employs a copying policy for marshaling (as opposed to a policy of data-level interoperability). It works by generating C code that compiles and links into the SML runtime (they state replacing this static linkage with a dynamic one based on dynamically linked libraries is straightforward). Their system supports callouts and callbacks; they deal with migration of callback target closures by registering the closure’s address in the callback as a root with the collector. We instead introduce a level of indirection between the machine code bytevector and the target closure.

[Urban(2004)] provides a broad (though incomplete) survey of FFI systems and implementations. The current draft ends with the suggestion that that values should be passed only by value (not by reference), to avoid any use of foreign pointers, which appears at odds with a policy of data-level interoperability. It is interesting that even as late as 2004 there is not an obviously “right” choice for this design axis.

6.2 FFI’s for Scheme

[Rose and Muller(1992)] present a Scheme system centered around integrated development with C. All C types are mapped into some class of data on the Scheme side, allowing seamless transfer of data between the two sides. This design goal led to a number of design constraints, such as using a “hyperconservative” garbage collector and a calling convention for Scheme compatible with that of C. In contrast, we layer an FFI on top of a high-performance Scheme runtime: we extended the runtime with new primitives, but the FFI does not compromise the main design goals of Larceny. Larceny has precise garbage collectors and a specialized calling convention for the MacScheme machine.

[Kelsey and Sperber(2003)] propose an interface for writing glue code in C. It provides a “lowest common denominator” approach to interfacing with foreign libraries: you can hook into arbitrary libraries, but you have to develop C code to do it.

[Barzilay and Orlovsky(2004)] present the FFI for PLT Scheme. Their philosophy of “stay in the fun world” agrees with our own; we have taken that philosophy further by using Scheme to generate our callout and callback trampolines. The PLT Scheme FFI uses.

\footnote{Urban’s conclusion is based in part on his view that Lisp code for interacting with foreign memory is even less readable than C code; perhaps tools such as those provided by our upper layers would address this concern.}
a GNU library, libffi [Green(2008)], to support callouts and callbacks; they point out that the libffi generated structures are allocated via malloc to circumvent the garbage collector, but do not provide further detail on how movement of callback targets is handled. It would be interesting to see if Larceny could also use libffi to avoid the need to develop ABI-specific code in the FFI lower layer; but the effort of hooking into libffi may exceed the effort of maintaining our construction of callout and callback trampolines. The PLT Scheme FFI has a sophisticated extensible syntax for generating wrapper code; we hope to adopt some of their ideas in a future revision of the middle layer of the Larceny FFI.

6.3 Extracting information from header files

[Rose and Muller(1992)] describes an interface extractor tool to scan header files and store information in Unix object files that their Scheme system can later load. They extract a large amount of data from the headers, converting definitions of macros with arguments into dynamic functions and definitions of types into first class Scheme values. We are much more limited in what we can extract, because we do not parse the header files directly.


Future work includes improving Unicode interface support, adding the ability to marshal structure parameters between the middle and lower layers, and adopting a more expressive interface DSL along the lines of [Barzilay and Orlovsky(2004)]. We also want to acquire experience interfacing to other foreign libraries, such as OpenGL.

References


7. Conclusion and Future Work

We have presented the layers of the Larceny FFI, from the low level details of callouts and callbacks and up to the high level syntactic forms used to write abstract interfaces.

Our FFI supports advanced features such as relinking foreign functions on heap reload. The FFI design is robust: we dynamically generate ABI-specific machine code in our trampolines, but that code is completely independent from the the MacScheme machine model used for compiled Scheme code.

Our higher layer libraries provide define-c-info, a tool that extracts information from C header files without reinventing the wheel of a C parser. This special form provides the basis for a high-level portable interface to C struct and enum types.

The interface of the PLT FFI is not directly portable to Larceny; in particular, their strategy for extensibility requires procedural macros to be able to expand subexpressions and inspect the results in a local manner, which Larceny does not currently support.