Foreign Interface for PLT Scheme

Eli Barzilay
Northeastern University

Dmitry Orlovsky
Northeastern University

Abstract

Even a programmer devoted to Scheme may prefer using foreign libraries in certain situation. Connecting the two worlds involves glue code, usually using C, which requires significant programming efforts and system expertise. In this paper we describe a PLT Scheme extension for interacting with foreign code, designed around a simple philosophy: *stay in the fun world*, even if it is no longer a safe sand box. Our system relieves the programmer from low-level technicalities while keeping the benefits of Scheme as a better programming environment compared to C.

1 Introduction

Scheme has proved itself as a useful and fun language, good for both general-purpose and domain-specific usages. However, schemers cannot assume a closed system; other languages will always exist, leading to a need for interfacing with functionality that is accessible through foreign libraries. Such libraries come in many different flavors, but the popular 'least common denominator' has been, and still is, plain C libraries[1]. Our goal is to create a mechanism within Scheme for smooth interfacing with such foreign libraries.

1.1 Background

A foreign interface is a piece of *glue code*, intended to make it possible to use functionality written in one language (often C) available to programs written in another (usually high-level) language. Such glue code involves low-level details that users of high-level languages usually take for granted. For example:

- marshaling objects to and from foreign code,
- managing memory and other resources,
- dealing with different calling conventions, implicit function arguments, etc.

[1]Different languages can be used to create foreign libraries, "C" is only used as a generic label.

Foreign function interfaces are subsystems that create such glue code, simplifying an otherwise tedious and error-prone task.

1.2 Foreign Interfaces

There are lots of existing foreign function interfaces; Urban's FFI survey [17], although an incomplete project, provides a good discussion of such systems and relevant issues. Generally speaking, such interfaces can be classified as either static code generators or dynamic foreign interfaces. In principle, the two are quite similar:

- A *static* foreign interface is created and compiled statically, *before* running the program that intends to use it;
- A *dynamic* interface is created at run-time, *while* the application is running.

In practice, the differences are more dramatic:

- A static interface is usually implemented using a C compiler. The advantage of this approach is that it is easy interface foreign code, as most of it is intended to be linked in using a compiler (for example, C header files are used to describe an interface), and since most languages are implemented in C, they provide convenient facilities for calling C functions. Disadvantages of the static approach include being restricted to the pre-compiled interface, requiring either a compiler or a platform dependent binary distribution for such code.
- A dynamic interface is generated at run-time, leading to the obvious advantage of requiring no C compiler or binary distributions. This has a significant effect on dynamic languages like Scheme, where single running REPL can be used to connect to different libraries, supporting exploratory programming in a natural way. The disadvantage of this approach is that it requires more (platform-dependent) low-level work such as stack management and creating stubs (glue functions), while not getting the usual support from a C compiler.

The issues that need addressing are essentially the same ones described in Section 1.1, only the approach differs. The technical issues involved in an interface implementation make static interface generators more popular. It should be noted that it is common to call these systems "foreign *function* interfaces" — in the following text we prefer "foreign interfaces" as these interfaces deal with accessing foreign objects as well as foreign calls.

In both the static and the dynamic cases, it is desirable to have some description of the foreign entities, usually functions, in a way that can help automate the process of generating the glue layer. In this context a "function" can be viewed differently depending on your point of view: from the low-level side, a function is simply a pointer and a description of how it is called; from the high-level Scheme side, it is an object that is expected to have the usual function semantics. *Interface description languages* (IDLs) have a major role in foreign interface systems — these are languages that express arbitrary function behaviors for both of these viewpoints:

- On the C side, there is the type definition of the function, and possibly additional information such as input/output pointers, object ownership, etc.
- In addition to this, there are details that are related to the Scheme side. For example, automatic memory management issues, value marshaling, dealing with aggregates (vectors and structs), and creating new object types.
- On the Scheme side, the result is a plain procedure, like any other Scheme procedure object.

Ideally, the IDL that is used to describe the interface is rich enough to express both views while providing enough information to completely automate the interface generation.

1.3 Implementing a Dynamic Interface

The low-level mechanics of foreign function calls are usually very demanding: managing functions at the binary level is inherently platform dependent, and can even require assembly code or other compiler-specific hacks. Statically, these problems are not too difficult: simply generate C glue code, and let the C compiler do its usual work. Doing this efficiently in a dynamic fashion is difficult, since it is usually not desirable to drag a complete C compiler into your run-time. Dealing with the dynamic aspects of foreign functions is greatly simplified using a library that handles the low-level details: we use `libffi` [11], a library that supports foreign function call-outs and call-backs.

- A call-out is a normal function call. In a dynamic setting, we create a "call-interface" object which specifies (binary) input/output types; this object can be used with an arbitrary function pointer and an array of input values to perform a call-out to the function and retrieve its result. Doing this requires manipulating the stack and knowing how a function is called, these are details that `libffi` deals with.
- A call-back is trickier. Our Scheme implementation has several fixed C-level functions which can implement arbitrary Scheme evaluation. A callback is, however, a simple function pointer — no additional information is available. Modern systems (e.g, Gnome) that use callbacks allow user to register a function pointer together with an arbitrary data pointer, but there is no standard way for this. A proper solution is one that allows creating general "C closures" — combining a function and a data pointer into a single new function pointer. Again, this is technically challenging, as it requires generating stub functions at run-time, which, when applied to some arguments, call the packaged function with the packaged data pointer and the arguments. Again, `libffi` provides the required magic.

`libffi` is maintained and distributed as part of the GCC project, but its goal is to provide a portable library. We use it for all platforms that PLT Scheme targets, including Windows (using a slightly adapted version that works with Microsoft's compiler, courtesy of the Thomas Heller [13]).

1.4 Outline

In Section 2 we state the goal of our work, emphasizing our main design principle. Section 3 describes our implementation, both the C part of the code and the complementing Scheme module. Section 4 demonstrates how our system copes with some of the common and uncommon situations that interface programmers deal with. We conclude with a related work comparison, and outline future plans.

2 Goal: Use Foreign Libraries, Avoid C

Our design follows a simple principle: keep C-level functionality to a minimum. The core of a system for interfacing foreign libraries must itself be written in C, but we try to make such functionality available to Scheme as soon as possible, putting more responsibility on the Scheme level. When dealing with the many details of the interface, mainly type declarations and data marshaling, there is a natural tendency to make a system that is rich in features. We avoid dealing with such complexities in C when possible, providing just enough of an interface that makes it possible to do it in Scheme instead. The combination of a dynamic interface and a minimalistic C-level implementation that should be complemented by Scheme code are the main features that make our approach unique.

Switching more responsibility to Scheme comes with benefits that are familiar to Scheme programmers, but there is an additional advantage that is important in this particular case: the important issue is generating glue code that bridges the gap between foreign libraries and the high-level language. In the static case this involves either complex yet limited C preprocessor acrobatics (e.g., SWIG [1] goes as far as implementing its own C parser). On the other hand, Scheme already comes with a superior syntax system, and PLT Scheme makes this even better with additional language features (syntax objects, module system, etc). This syntax system is much easier for implementing sophisticated glue code with, especially considering our target crowd which undoubtedly feels more at home in the Scheme world.

For example, consider the issue of primitive foreign types that are handled by an interface. Once we can move C integers from Scheme to C and back, we might consider extending the system to deal with C enumerations. This raises a few questions regarding the interface design — how should this C definition:

```
    typedef enum { foo1, foo2, foo3 } foo;
```

be available for Scheme code?

- Should we provide three integer bindings? If so, how do we deal with name clashes?
- Otherwise, should we use a mapping from strings to integers? Maybe use symbols? What about enumerated values that are or-able bit patterns? How should such a map be implemented: as a linked list? A vector of constant names? A hash table?

Answers to these questions determine the nature of the C implementation; once it is written, trying alternatives lead to significant maintenance costs. Our design keeps such complications away from the C level, pushing them up to the Scheme side where there are better ways to deal with them. For example, the C level part of our interface does not commit to a specific implementation for enumerations — it simply exposes C integers. Different strategies are then implemented in the Scheme part, resulting in easier code maintenance. In addition, some Scheme aspects are less accessible from C, making a Scheme solution even more attractive. for example, implementing enumerations as bindings that use the module system to avoid global name-space pollution, or implementing them as syntax objects (removing run-time lookup costs) are both much harder to implement in C than in Scheme.

Another important factor in the complexity of the C implementation is the issue of safety. Scheme is a *safe* language — as buggy as your code might be, you never expect the Scheme process to crash: if such a crash happens, the blame is in the language implementation. Using C extensions such as the ones that PLT Scheme always had, changes things a little — the code to blame can be either in the language implementation, or in the C extension. The invariant fact is that Scheme code can never be blamed for such crashes — they are exclusively considered a C-level problem. There is therefore a yellow caution tape around code that can be blamed for such crashes: it lies exactly on the language boundaries, C on one side and Scheme on the other.

A dynamic foreign function interface inevitably breaks this property: bad Scheme code that defines an interface to a foreign function can specify an integer argument where a pointer is expected, leading to a crash (at best). Using dynamic interface systems does not seem so bad though — a foreign function definition is written in Scheme, but conceptually it is perceived as part of the C world. Scheme code, with the exception of such definitions, is still as safe as it has always been, the yellow caution tape is moved just a little so it surrounds Scheme definitions of foreign interfaces too. This point drives a dynamic foreign interface system to try to be as safe as possible: if function interfaces are the only things that can lead to crashes, then it is desirable to make the system safe in all other respects. For example, when dealing with pointers (arrays referencing, allocations, garbage collection) safety issues go in the C code, making it much more complicated than it would otherwise be.

In contrast, our implementation extends traditional dynamic interface systems by exposing more 'dangerous' operations. Functionality that had to be part of the C world is now accessible in Scheme, moving the yellow tape again to encompass more Scheme code. The average programmer is not concerned with this extra functionality, but interface implementors can now deal with more foreign code without leaving Scheme. Many design decisions that usually affect the C interface can now be pushed up to the Scheme level.

The issue of safety is now related to the module system: the new foreign interface bindings are enclosed in a module. If a Scheme process crashes, the blame is either on C code, or on Scheme code that uses this module: such code is therefore taken as substituting C code, potentially suffering from C's usual illnesses. Code that does not use this module is expected to be as safe as it previously was.

To summarize, the yellow caution tape surrounds more Scheme code now: it lies at the C/Scheme language border *except* for code that uses the new module which is inside the tape. In essence, using the a Scheme module is similar to a using Modula-3's [12] '`UNSAFE`' keyword to declare unsafe code. Quoting Harbison [12, Section 13.3.1] from the Modula-3 book:

> Modula-3 also provides unsafe features, but it differs from many other languages in isolating those features. The unsafe language features are accessible only in interfaces and modules that are labeled by the keyword `UNSAFE`. [...] When all modules and interfaces are safe, Modula-3 guarantees that there will be no unchecked run-time errors. By introducing `UNSAFE`, the programmer assumes part of that burden.

Our system is slightly different in that Scheme modules can provide additional functionality for interface writers, meaning that they will not provide a safe interface, making them have a status similar to that of the new module. This means that rather than a fixed set of unsafe language features, we have a system where these features can also be extended.

An example of this design philosophy is our use of pointers. First, a new Scheme pointer object is introduced, then low-level functions that deal with pointers are added. These are procedures that allocate memory blocks (using one of several '`malloc`' variants), free blocks (for GC-invisible blocks), reference pointers, and set values at a pointer locations. This new functionality is useful in itself, even when there are no foreign libraries to interface with. For example, the procedural part of SRFI 4 [6] can now be trivially implemented in Scheme. Several foreign interfaces have a similar generic 'pointer' object, but it is usually viewed as a last-resort object when an unknown pointer is returned[2] or when an interface is too lightweight for proper types[3] — this is in contrast to our view, where a pointer object is taken as part of the fundamental framework that makes Scheme a viable C substitute for glue code.

3 Implementation

Our implementation consists of a C part, implementing the low-level functionality, and a Scheme part that builds on top of it. The C part of our interface is available as a built-in '`#%foreign`' module which is part of the MzScheme core of PLT Scheme (it is part of the MzScheme executable). This implements the thin interface, providing just enough to make it possible to fill in the gaps using Scheme. This module is therefore intended to be used only by the Scheme part of our interface: the '`(lib "foreign.ss")`' module which is part of MzLib, serving as a wrapper around the internal bindings. For brevity, we refer to the Scheme module as '`foreign`'.

The '`#%foreign`' functionality that is implemented in C is described in Section 3.1, and the Scheme '`foreign`' module is described in Section 3.2.

3.1 The '`#%foreign`' Module: C-Level Interface

The C implementation can be roughly divided into three parts, described in the following sections. Most of this is unrelated to foreign libraries, but providing the framework that make such interactions possible, and making Scheme rich enough to substitute C.

3.1.1 C Types

C-types[4] lie at the core of our system, as they provide the basic specifications for data that is passed on to and back from foreign libraries. We need some way to specify the correlation between tagged Scheme values and the various C types. This mapping is not one-to-one: a single C type can be interpreted as several Scheme types, and a single Scheme type can be translated to different C types. We implement C type objects for this, available as new first-class Scheme values, accessible through '`#%foreign`' bindings. Each C type object has three main parts:

- The actual C type that it represents (a `libffi` type descriptor),
- Code that translates corresponding Scheme objects to C,
- Code that translates such C values to Scheme objects.

In addition, there is some utility information such as a predicate, byte size and alignment. The translation code for these primitive C types is implemented in C. Table 1 presents a summary of the current built-in primitive types[5].

[2]For example, the SWIG manual uses `malloc`, `realloc` and `free` as a simple interface example which uses pointer objects.

[3]Our `cpointer` type pre-existed for PLT Scheme extensions, and was intended for "extensions with modest needs".

[4]Again, "C" is only used as it reflects binary level objects.

[5]The name convention that we have used is that a type called '`foo`' is available in Scheme as a '`_foo`' binding.

Primitive Type	Usage
_void	returns a Scheme void value when used as an output type
_int8, ..., _int64	integer types in various sizes
_uint8, ..., _uint64	non-negative integers
_byte, _word, _int, _uint	aliases for _uint8, _uint16, _int32, and _uint32 respectively
_long, _ulong	aliases for 32- or 64-bit integers, depending on the meaning of 'long' for the current platform
_fixint, _ufixint, _fixnum, _ufixnum	versions of integers (int and long resp.) that assume fitting into an immediate Scheme fixnum integer
_float, _double	floating point numbers (inexacts)
_bool	booleans (as C integers)
_bytes	byte-strings (plain char strings and memory blocks represented as byte-strings)
_string/ucs-4, _string/utf-16	Unicode string types
_path, _symbol	path strings and symbol names as strings (interned when used as an output type)
_pointer	a 'cpointer' object encapsulating a pointer value and an optional tag, #f is used for a NULL pointer
_scheme	a Scheme_Object* pointer, for any Scheme boxed value, this will be its actual pointer

Table 1. Primitive types

Users can create new types in two flavors:

- User-defined types are made by the 'makectype' primitive, and are analogous to primitive types. To create such a type a programmer has to:
  1. Choose the set S_1 of Scheme objects that the new type should handle. This can be any set — combination of several Scheme types, subsets, or a few random values.
  2. Choose an existing C type T as a base type. This type handles some set S_2 of Scheme objects.
  3. Write two procedures: one that translates an S_1 value to S_2 and one that goes the other way.
  4. Apply 'makectype' on T and the two translators.

  When the new C type is used to send values to foreign code (function arguments, or setting pointers), the first translator is used and processing continues with T, and when receiving values from foreign code (return values or pointer references), T is used first and the second translator is then applied. The implementation of user types does not involve libffi, which only sees primitive types.
- New struct types are created from a list of existing types using the 'makecstructtype' primitive. This is mainly implemented by libffi since it describes a new low-level data type with new size and alignment information. On the Scheme side the resulting primitive type is similar to a _pointer, but when it is used to send or receive values, the contents of the pointer is copied rather than the pointer itself.

No additional functionality is implemented at the C level for these types except trivial accessors and size/alignment information. Additional abstraction layers like enumerations and struct constructors and deconstructors are implemented in Scheme. As a result, we don't have to commit to a specific marshaling scheme at the binary level (in fact, the Scheme part of the interface implements two different marshaling schemes for each of these cases).

3.1.2 Pointers

As mentioned above, pointers are an integral part of our interface, exposed as useful Scheme objects. A Scheme pointer object encapsulates the actual pointer value (adding an extra level of indirection), and a 'tag' which is an arbitrary Scheme object. C functionality is limited to a usable minimum: allocating memory blocks (using various allocator functions — either through the garbage collector, or raw malloc), referencing and setting pointed values (given a type), and pointer equality.

Again, functionality implemented by the C level is kept to a minimum. For example, the tag values that are attached to pointers can be used to enforce a type for referencing and setting a pointed value, but such a design can be better implemented and enforced in Scheme, so these tags are *ignored* by the C part of the interface.

3.1.3 Interfacing Foreign Functionality

So far, all C-level functionality is useful by itself, extending Scheme so it can handle machine-level raw data. The final piece of the C part of our interface is the one that actually deals with foreign libraries. First, there is functionality for opening a dynamic library and pulling out objects. These objects can be used as pointer objects, so it is possible to both reference and change their values (useful for libraries that contain user-modifiable customization hooks).

Dealing with function values is separated into function calls that we can do ("callouts") and calls from foreign code to our functions ("callbacks"). This is where libffi makes the implementation much easier. Two Scheme-accessible procedures, fficall and fficallback, are in charge of converting C functions to Scheme (callouts) and Scheme procedures to C (callbacks) respectively. At the Scheme level, these procedures are used by a new _cprocedure type constructor, which provides a symmetric 'marshaling' interface for both ways of this conversion, so users are not aware of any differences in the underlying translation mechanism.

Bindings that are implemented by the C part of our implementation and made available through the '#%foreign' module are listed in Table 2. This, together with Table 1, is a complete summary of the C-level implementation. Again, '#%foreign' is not intended for use outside of our 'foreign' implementation (described next), but many of these procedures are re-exported by 'foreign'.

3.1.4 Garbage Collection Issues

There are some important memory management issues that should be mentioned at this point: a moving garbage collection, such as the one used by the precise PLT Scheme version (mzscheme3m) complicates things considerably when foreign code interacts with objects on the (GC-visible) Scheme heap. There are certain objects that should not move in memory, most notably, the callable function pointers generated by libffi to implement C closures must not move, so we need to take extra care in allocating these using plain malloc, where the garbage collector does not touch them. Callbacks are especially fragile in this aspect: when C code calls Scheme code the garbage collector might be triggered and any GC-visible pointers that the C function might use will inevitably be invalidated. This problem does not have an easy solution — either memory is managed by a non-moving collector possibly managing different memory regions using different collectors (this solution is impossible with PLT Scheme's precise GC), or doing manual management. The C implementation takes care of this when deal-

Primitive Bindings	Usage
ffi-lib, ffi-lib?, ffi-lib-name	open a foreign library and related functionality
ffi-obj, ffi-obj?, ffi-obj-lib, ffi-obj-name	get a foreign object pointer from a library and related functionality
make-ctype, make-cstruct-type, ctype?, ctype-basetype, ctype-scheme->c, ctype-c->scheme, ctype-sizeof, ctype-alignof	Handling C type descriptor objects (see Section 3.1.1)
cpointer?, cpointer-tag, set-cpointer-tag!, ptr-ref, ptr-set!, ptr-equal?	Handling C pointer objects (see Section 3.2.2)
malloc, end-stubborn-change, free, make-sized-byte-string, register-finalizer	Interface for the standard C malloc and other allocators that are used in MzScheme, and related memory management functions
ffi-call, ffi-callback, ffi-callback?	creating a call-out object (a Scheme procedure that calls a foreign function when applied) from a C pointer and creating callbacks (objects that can be passed onto foreign functions as function pointers) from Scheme procedures, both functions accept an input type list and an output type

Table 2. Primitive '#%foreign' bindings

Defined Type	Usage
_string/utf-8, _string/locale, _string/latin-1	various C strings, using different encoding
_string	uses one of the existing string types, depending on the value of the default_stringtype parameter; '#f' is used as a NULL value
_file	similar to the _path type, except that path names are resolved using expandpath
_string/eof	similar to _string, but in case of #f (NULL), an end-of-file object is returned
_enum, _bitmask	these are actually functions that consume a list of symbols, and create an integer-mapping type that translates a single symbol (_enum) or a list of symbols (_bitmask) to an integer

Table 3. Simple types defined by the Scheme module

ing with libffi objects, but nothing else. If a movable pointer is passed on to a C function which can use Scheme callbacks or otherwise retain it, then it is the responsibility of the Scheme level to deal with copying these values to non-movable memory (using the system's raw malloc which is accessible in Scheme). The Scheme part of our interface simplifies some of these issues, but there is no general solution when (potentially misbehaved) foreign code is involved, since such code is ignorant of any memory management issues for objects it does not "own".

A related issue is dealing with pointers that can be contained in other objects. The Scheme-visible 'malloc' function uses atomic allocation by default except for allocating a _pointer- or a _scheme-based type. User-created struct types are, however, problematic because they can hold both pointers and other values. Our implementation uses only atomic memory blocks for these, which works as long as there are no GC-able pointers in structs, which so far was not a problem. We have a plan for dealing with such pointers, in case a solution is needed: expand new struct types with a map of contained GC-able pointer offsets. In any case, users should be aware of the fact that memory blocks are moved and use raw-malloced pointers as necessary when callbacks or library references are involved.

3.2 The foreign Module: Scheme-Level Interface

At the Scheme level, we have added a new '(lib "foreign.ss")' module to MzLib. Scheme programmers should use this module which complements the built-in '#%foreign' module. The purpose of this module is to re-export some useful parts of '#%foreign' with an additional degree of sanity and convenience. For example, 'getffiobj' is a convenient procedure that combines 'ffilib' to open a library, 'ffiobj' to retrieve a pointer, and 'ptrref' to convert it into a Scheme value. In addition, it builds a layer of additional functionality using the built-in module, varying from new types, through an IDL, to memory management issues.

3.2.1 Additional Types

The Scheme module, like the C part, revolves mainly around types. First, there are several simple types that are implemented in the Scheme module, summarized in Table 3. Adding these types is simple, as described in Section 3.1.1, for example, the _file type is intended to make it easy to interact with foreign functions that expect a file name — making it possible to use names like "~foo/bar". The definition in 'foreign' involves using expandpath when going from Scheme to C, and leaving the path as is when going from C to Scheme:

```
(define _file
  (make-ctype   ; create a new type,
   _path        ; based on _path
   expand-path  ; expand-path when sent out
   #f))         ; receive: same as _path
```

Since this part of the implementation is in Scheme, we can now develop better solutions than we could if we used only C. For example, note that _enum and _bitmask are not type objects, but *functions* that create type objects — they are *type constructors*. Also, note that there are multiple string types, since our system is integrated into the development version of PLT Scheme which uses Unicode for its strings — the _string type is therefore an 'identifier syntax' that expands into a usage of the 'default_stringtype' parameter. Both of these would take a much heavier implementation if they were implemented in C.

3.2.2 Pointer Types

Section 3.1.2 mentions that Scheme pointer objects have an arbitrary 'tag' value associated with them, and that these tags are ignored by the C part of the interface. The 'foreign' module provides a _cpointer function that, when given some Scheme value, constructs a new _pointer-based type which tags pointer objects when they arrive from the foreign side, and raises an error when passing a pointer with the wrong (non-eq?) tag from Scheme. This functionality might be extended in the future to use the tag value in some more meaningful way, for example, make it be another type object and make pointer dereferencing use it instead of taking a type argument, or use it to imitate inheritance where a pointer can be used in places where an ancestor pointer kind is expected. In addition to the _cpointer function, there is a definecpointertype syntax:

(_define-cpointer-type ⟨*_id*⟩
    [⟨*type-or-*#f⟩ ⟨*scm→c*⟩ ⟨*c→scm*⟩])

which defines such a type using "⟨*id*⟩" as a tag, together with a '⟨*id*⟩?' predicate and a '⟨*id*⟩-tag' binding for the tag value.

The optional type and translation arguments can be used to specify the base type in case it is not _pointer (for example, if it is a struct type), and translation procedures. Such arguments are also available for _cpointer.

3.2.3 Vector Types

Exposing C functionality in Scheme makes it possible to use arbitrary blocks of memory to hold data. Allocating such a block is even simpler with the provided list->cblock and cblock->list, both implemented in Scheme, but the result is just a bare pointer object. It is therefore useful to encapsulate such a memory block with the type of objects it uses and the number of objects contained in it. Using this we benefit from no per-item storage overhead as well as making some foreign interfaces easier to deal with, and at the same time ensure that there are no violation of the vector bounds. Interacting with these vectors is intentionally similar to using plain Scheme vectors:

```
> (define v (make-cvector _int 10))
> (cvector-length v)
10
> (cvector-set! v 5 55)
> (cvector-set! v 15 55)
cvector-ref: bad index 15 for cvector bounds of 0..9
> (cvector-ref v 5)
55
```

These vectors can be used as inputs to foreign functions via the _cvector type.

SRFI 4 [6] defines similar structures, except that there are different Scheme types (therefore different function names) for each kind of vector, making it limited to numeric vectors. Our 'foreign' interface adds a complete re-implementation of SRFI 4, which will replace the C-based module that is currently a part of PLT Scheme[6].

3.2.4 Struct Types

The C part of our implementation provides limited support for defining struct types: we get a 'makecstructtype' function which constructs a new kind of primitive type given a list of existing types. This new type can be used with Scheme pointer objects, which will cause copying the structure *contents* rather than the pointer value when marshaling data. Accessing these objects is left for the Scheme side, which uses the information given by the ctypesizeof and the ctypealignof functions to compute the offsets into the contained values.

This functionality is sufficient for the 'foreign' module to make C structs accessible from Scheme. Two interfaces are provided:

1. _list-struct is a type constructor: given a list of type objects, it constructs a matching C struct type, and wraps the result in a yet another type that translates values contained in such a C struct value to and from a Scheme list of values. Using this type is simple, but it involves extra allocations which is an extra overhead some users will want to avoid.
2. definecstruct is a new syntax, similar to PLT Scheme's 'definestruct', except that slots have an associated type. Values of this new type are kept as a pointer object that references the memory block holding the binary data. Again, this simplifies interfaces: there is no overhead involved as we are dealing with the raw data. A simple example of using such a struct type follows:

```
> (define-cstruct _foo
    ((x _int) (y _double)))
> (define x (make-foo 1 2.3))
> (foo? x)
#t
> (list (foo-x x) (foo-y x))
(1 2.3)
> (set-foo-y! x 4.5)
```

[6]The current implementation does not deal with the external syntax specified in SRFI 4.

3.2.5 Simple Function Types

Finally, the core functionality that allows interactions with foreign libraries is enabled by the _cprocedure type constructor. This constructor creates a function type when given a list of input types and an output type. Like all other C type objects, the resulting function type has two translation procedures: one going from C to Scheme and one going back. For these function types, the first translator generates a *callout* object that can be used as a new Scheme primitive, and the second generates a callback object that can be sent to C code allowing it to invoke a Scheme procedure. This internal function is implemented via the primitive 'fficall' and 'fficallback' functions (see Table 2), it's definition is (roughly[7]):

```
(define (_cprocedure itypes otype)
  (make-ctype _pointer
    (lambda (x) (ffi-callback x itypes otype))
    (lambda (x) (ffi-call x itypes otype))))
```

This means that from the user's point of view, a simple type specification like '(_cprocedure (list _int _int) _int)' can be used as either an input or an output type, and it can properly nest (negative function type occurrences generate callbacks and positive occurrences generate callouts). For example, the following contrived higher-order C function:

```
int foo_ho_ho_func(int x, int(*(*f)(int))(int)) {
  return (f(x+1))(x-1);
}
```

can be used (interactively!) in Scheme in a straightforward way:

```
> ((get-ffi-obj "foo_ho_ho_func" "foo.so"
     (_cprocedure
       (list _int
             (_cprocedure
               (list _int)
               (_cprocedure (list _int) _int)))
       _int))
   3
   (lambda (x) (lambda (y) (+ y (* x x)))))
18
```

3.2.6 Complex Function Types: IDL Features

The _cprocedure can generate simple interfaces, but it is insufficient in cases where the foreign function needs an additional layer of interface when arguments and/or the return value on the Scheme side don't match those of the foreign side. A common example of

[7]The actual implementation accepts another optional argument that can be used to tweak the resulting primitive procedure. This is described in the following section.

this is a foreign function that expects a pointer and a size indicator, which correspond to a single Scheme object that encapsulates both. For example, the standard C '`read`' function expects a string buffer and its size in two input arguments. A simple `_cprocedure`-generated interface inevitably exposes the additional argument, so the interface programmer needs to wrap it by additional glue code. For this, `_cprocedure` has an extra optional argument that is expected to be a procedure that wraps the resulting foreign function[8]:

```
(define c-read
  (get-ffi-obj "read" "libc.so.6"
    (_cprocedure (list _int _string _int) _int
      (lambda (prim)
        (lambda (fd buf)
          (prim fd buf (string-length buf)))))))
```

Another common example is the use of 'output pointers' by foreign code to return multiple values. Again, a naive `_cprocedure` interface will be awkward to use from Scheme code, and the interface programmer needs to use a wrapper that makes the foreign function more Scheme-friendly:

```
(define c-modf
  (get-ffi-obj "modf" "libc.so.6"
    (_cprocedure (list _double _pointer) _double
      (lambda (prim)
        (lambda (d)
          (let* ([p (malloc _double)]
                 [r (prim d p)])
            (values (ptr-ref p _double) r)))))))
```

More forms of wrappers are needed in other situations: additional argument dependencies, input- and output-pointers, different allocation strategies, implicit 'self' pointers, etc. In general, we need a way to combine arbitrary wrappers that operate on arbitrary arguments. Such wrappers cannot be implemented as new C types, since such types can add layers of processing on each value independently, rather than the required interaction among multiple arguments and output values. What we need here is some form of an interface description language (IDL). The requirements for an appropriate IDL are:

- it should be easy to write and easy to read,
- it should be rich enough to express interactions such as the two demonstrated above as well as others,
- it should not lead to an expensive performance hit,
- it should be easy to extend when facing new situations.

One way that we have tried to tackle this issue is by providing the necessary abstractions as a collection of procedures, each performing a single task, and have interfaces use combinators to build the required argument interactions. This approach has a major drawback: it leads to complex expressions which are hard to write and harder to read. Using this approach, code that converts a Scheme string argument to buffer-size and pointer arguments might use a '`string+len`' function together with combinators that arrange to swap the arguments, for example:

```
(define foo
  (get-ffi-obj "foo" "foo.so"
    (_cprocedure (list _int _string) _int
      (compose prim:1+2->2+1
               (prim:1->1+2 string+len)))))
```

[8]Actually, our interface is part of the new version of PLT Scheme, which has a new *byte-string* type for raw (non-Unicode) character sequences. We use strings in the following examples for simplicity.

It is obvious that this code is hard to read — for example, inspecting the types reveals that there is a bug in this code[9]. In addition, such procedures will often be higher order for customization, making things even worse. Another drawback of this approach is the number of procedure applications that are involved in each call: any time overhead involved in foreign calls might be critical, and we don't want programmers to move to inferior tools because of it.

The approach that our system takes uses Scheme's syntax abstraction capabilities instead. We define a new type combinator, `_fun`, which is actually a syntax transformer. Usages of `_fun` generate the appropriate wrapper code, and use `_cprocedure` with it to create the function type.

Simple usages of `_fun` are similar to `_cprocedure` except that the types need not be put in a list, and an infix '`->`' marker separates the input types from the output type. For example, using `_fun` for the higher-order C example from Section 3.2.5:

```
> ((get-ffi-obj "foo_ho_ho_func" "foo.so"
     (_fun _int (_fun _int -> (_fun _int -> _int))
           -> _int))
   3
   (lambda (x) (lambda (y) (+ y (* x x)))))
18
```

In its simple form, the `_fun` type constructor has this syntax:

(`_fun` ⟨*f-type*⟩* `->` ⟨*f-type*⟩)

which covers simple function interfaces in a slightly more convenient form than `_cpointer`. In its full form, `_fun` is extended to deal with common argument interactions like most IDLs and more — rather than fighting with a limited preprocessor or re-implementing a C parser, we have a real (meta) language to help us. Using syntactic abstractions in Scheme, we achieve a powerful IDL through `_fun`, one that can be extended to handle all possible situations.

The full form of the `_fun` syntax has two optional parts, and each ⟨*f-type*⟩ subform can have an optional identifier and/or expression:

(`_fun` [⟨*args*⟩ `::`] ⟨*f-type*⟩* `->` ⟨*f-type*⟩ [`->` ⟨*expr*⟩])

⟨*f-type*⟩ ::= ⟨*t-expr*⟩ | ([⟨*id*⟩ `:`] ⟨*t-expr*⟩ [`=` ⟨*expr*⟩])

⟨*t-expr*⟩ ::= *expressions that evaluate to a type value*

The sequence of ⟨*f-type*⟩s in their full form behave like a sequence of '`let*`'-bindings, each with an associated type and a value (both plain Scheme expressions). As with '`let*`', value expression can refer to previous identifiers for their values. Omitting an identifier makes the corresponding value inaccessible for subsequent expressions; omitting a value expression means that the resulting wrapper function will expect a corresponding argument. For example, in this definition:

```
(define c-read
  (get-ffi-obj "read" "libc.so.6"
    (_fun _int
          (buf : _string)
          (_int = (string-length buf))
          -> _int)))
```

there are three arguments that are passed on to the foreign function:

- The first uses the short form: it has no value so it will receive the first value passed on to '`c-read`', and it has no name so its value can not be used in following expressions.
- The second argument has no value too, making it get the sec-

[9]A type checker will help avoiding such errors, but will not make things easier to read and write.

ond 'c-read' argument, and its value is bound to 'buf'.

- The third argument has a value expression so the value that is passed on to the foreign function is always the length of the second (string) argument.

'c-read' is therefore a Scheme procedure that expects two arguments and returns an integer, by arranging for properly calling the foreign 'read'.

In some rare cases, an interface needs to have better control of the wrapper's argument list — which is the purpose of the optional '⟨*args*⟩ ::' prefix: it specifies the arguments to the resulting wrapper function. For example, if 'read' were to expect the buffer size first, we would use this _fun type:

```
(_fun (fd buf) ::
      (fd : _int)
      (_int = (string-length buf))
      (buf : _string)
      -> _int)
```

Note that identifiers are important here, as they connect the foreign inputs with the wrapper's inputs. The ⟨*args*⟩ part can also be used to specify normal Scheme argument lists, including optional arguments.

A second '->' marker denotes a result expression different than the one that the foreign function returned. This expression can use any bound values and arguments, as well as the foreign result value (if given an identifier). For example, the 'modf' interface given above is better written with _fun as:

```
(define c-modf
  (get-ffi-obj "modf" "libc.so.6"
    (_fun _double (p : _pointer = (malloc _double))
          -> (r : _double)
          -> (values (ptr-ref p _double) r))))
```

The fact that we can insert any Scheme expression for the return value makes it easy to change such definitions so they use alternative ways for assembling the return values, for example, changing 'values' to 'cons' in the above. If this was implemented in C, such changes would require more work.

The similarity between the _fun syntax and 'let*' is not incidental: _fun assembles a wrapper function that contains a single 'let*' expression, which evaluates the various expressions, binding the results to specified identifiers. For example, the usage of _fun in the last example expands to:

```
(_cprocedure (list _double _pointer) _double
  (lambda (ffi)
    (lambda (tmp15)
      (let* ((p (malloc _double))
             (r (ffi tmp15 p)))
        (values (ptr-ref p _double) r)))))
```

This satisfies the efficiency requirement: only one extra function call is wrapped around the foreign call.

3.2.7 Additional IDL Features: Custom Function Types

The _fun facility handles some common cases where we need to bridge a gap between the foreign function and Scheme code that uses it, but there are additional cases that are not addressed. For example, the 'modf' interface code above represents such a common situation – output pointers that are used by foreign code to return multiple values. We therefore extend the _fun syntax further, by making it interact with special 'custom function types' that are themselves syntaxes — such types can install pieces of code that are used before and after the foreign call, possibly modifying the corresponding value. In the case of output pointers we want to allocate some memory before the foreign call and dereference it afterward, a task that is achieved by the _ptr custom type. _ptr is a syntax with usages that has the following form:

(_ptr ⟨*mode*⟩ ⟨*type-expr*⟩)

⟨*mode*⟩ ::= i | o | io

The ⟨*mode*⟩ specifies an input, output, or input/output pointer. In the 'modf' case, we use an output pointer:

```
(define c-modf
  (get-ffi-obj "modf" "libc.so.6"
    (_fun _double (p : (_ptr o _double))
          -> (r : _double) -> (values p r))))
```

The code that is generated by this _fun syntax is similar to the previous code,

```
(lambda (tmp15)
  (let* ((p (malloc _double))
         (r (ffi tmp15 p))
         (p (ptr-ref p _double)))
    (values p r)))
```

but notice that we don't need to explicitly allocate a double or dereference the pointer.

Custom Type	Usage
_ptr	input, output, or input/output pointers
_box	similar to an input/output _ptr, but modifies the Scheme box contents (PLT Scheme has a mutable box type. Note that we don't need to associate Scheme boxes with 'shadow' pointers: either copy values, or use a _pointer instead of a box)
_list, _vector	marshal lists and vectors as C pointers
_bytes	uses Scheme byte-strings (raw, non-Unicode strings)
_?	a special non-type intended for saving intermediate interface results

Table 4. Simple types defined by the Scheme module

The custom function types that are provided by the 'foreign' are listed in Table 4. Further details on these types can be found in our user manual.

As mentioned above, Custom types are implemented as syntaxes. _fun tries to expand each type expression it encounters, and if an expansion is identified as a custom type, then it has certain forms that contain the relevant pieces of code. A custom type expansion is a '(⟨*key:*⟩ ⟨*val*⟩ ...)' sequence where all of the ⟨*key:*⟩s are from a short list of known keys. Each key interacts with generated wrapper functions in a different way, which affects how its corresponding argument is treated:

type: specifies the foreign type to be used (#f can be used to make this not participate in the foreign call).

expr: specifies an expression to be used for arguments of this type, removing it from wrapper arguments.

bind: specifies a name that is bound to the original argument if it is required later (e.g., _box needs to refer to the original box).

1st-arg: specifies a name that can be used to refer to the first argument of the foreign call (good for common cases where the first argument has a special meaning, e.g., for method calls).

prev-arg: similar to 1st-arg:, but refers to the previous argument.

pre: a pre-foreign code chunk that is used to change the argument's value.
post: a similar post-foreign code chunk.

The following is the implementation of the _ptr custom type from the 'foreign' module. It is provided to roughly demonstrate how this is done; again, complete details are given in the user manual.

```
(define-syntax _ptr
  (syntax-rules (i o io)
    [(_ i  t)
     ;; input: malloc a pointer, set its value from the argument
     (type: _pointer
      pre:  (x => (let ([p (malloc t)]) (ptr-set! p t x) p)))]
    [(_ o  t)
     ;; output: malloc a pointer on entry, dereference on exit
     (type: _pointer
      pre:  (malloc t)
      post: (x => (ptr-ref x t)))]
    [(_ io t)
     ;; input/output: like output, but set its contents on entry
     (type: _pointer
      pre:  (x => (let ([p (malloc t)]) (ptr-set! p t x) p))
      post: (x => (ptr-ref x t)))]))
```

All of the special custom types provided by 'foreign' are defined this way.

To conclude: our _fun satisfies all requirements mentioned above for a good IDL: it is easy to read and write, it can express all wrapper interactions that other IDLs can express and more, it is efficient, and extensible by the ability to add new custom types that handle new kinds of processing. As expected from a syntax transformer that performs some substantial work, it carries some conceptual overhead, but we believe that overall it is better than the C processing alternatives since Scheme is superior in its syntactical abstraction capabilities.

4 Usage Examples

With the implementation of our system, we provide a few (mostly Linux) library interfaces. This was used to test the implementation, motivating the overall design. We now describe a few examples of using our system, all based on these interface implementations.

Syntactic Abstractions

C provides some (limited) degree of syntactic abstraction, whereas Scheme truly shines in this area. When a complete library interface is desired (rather than pulling out a few useful functions), repetition is common. Writing interfaces in Scheme makes such problems almost non-existent — for example, our ImageMagick interface uses a simple macro:

```
(define-syntax defmagick
  (syntax-rules (:)
    [(_ id : x ...)
     (define id
       (get-ffi-obj 'id libwand (_fun x ...)))]))
```

to make interface definitions easier.

Defining new syntaxes can help in other, less common situations. For example, KSM [4] has a clang:sym form that exposes a foreign library variable as a Scheme binding. Using PLT Scheme macros, we can achieve this functionality in Scheme using a macro that defines the C 'variable' as a macro[10]:

```
(define-syntax defcvar
  (syntax-rules ()
    [(_ var lib type)
     (define-syntax var
       (syntax-id-rules (set!)
         [(set! var1 val1)
          (set-ffi-obj! 'var lib type val1)]
         [(var . xs)
          ((get-ffi-obj 'var lib type) . xs)]
         [var (get-ffi-obj 'var lib type)]))]))
```

and verify that it is working properly:

```
> (defcvar z "x.so" _int)
> z
0
> (set! z 123)
> z
123
> ((get-ffi-obj "getz" "x.so" (_fun -> _int)))
123
```

where the C code that is compiled into "x.so" is:

```
int z = 0;
int getz() { return z; }
```

Using Types

C types in our system are somewhat lighter than expected: there is only a loose correlation between these types and Scheme object types. A type in our context can simply mean a different way of marshaling Scheme values to/from C, for example, the _file type from Section 3.2.1 is simply a different way to marshal MzScheme path objects which are normally used with _path. No C-level support is needed for such cases: there are no new binary tags involved, and no new object representations at the implementation level, meaning that it is extremely cheap to create such type descriptors. A common usage of types is therefore as a simple mechanism to add hook on the translation process.

For example, the ImageMagick library specifies a 'MagickWand' type, which is always being manipulated as a 'MagickWand*' pointer. There are functions that return a pointer to a newly created 'MagickWand' object, and these objects must be destroyed with the 'DestroyMagickWand' function. To do this automatically, we define a _MagickWand type using _pointer and providing a new translation when going from C to Scheme, one that uses 'registerfinalizer' to make the GC use 'DestroyMagickWand' when reclaiming the pointer object[11]:

```
(define _MagickWand
  (make-ctype _pointer
    #f ; Scheme->C translation is the same as _cpointer
    (lambda (ptr)
      (if ptr
        (begin (register-finalizer ptr destructor) ptr)
        (error '_MagickWand "got a NULL pointer")))))
```

We can make this even better with a new cpointer type which uses an appropriate tag to identify these pointers and make sure that we don't confuse pointers to internal ImageMagick objects of different types. The following definition uses 'definecpointertype' (see Section 3.2.2) to create a type that tags all pointers when they are

[10]From the MzScheme [9, Section 12.1] manual: The 'syntax-idrules' form has the same syntax as 'syntaxrules', except that each pattern is used in its entirety (instead of starting with a keyword placeholder that is ignored).

[11]This assumes that there is no way to get a second pointer object that refers to the same 'MagickWand' object, so care should be taken with functions that can create such aliases.

moved from the foreign side to Scheme, and check the tag when sending a Scheme pointer object out to foreign code.

```
(define-cpointer-type _MagickWand #f #f[12]
  (lambda (ptr)
    (if ptr
      (begin (register-finalizer ptr destructor) ptr)
      (error '_MagickWand "got a NULL pointer"))))
```

A different example of using a new type comes from our TCL interface: the `Tcl_Eval` function returns a status integer, indicating a possible error. In our implementation, we define `evaltcl` as:

```
(define eval-tcl
  (get-ffi-obj "Tcl_Eval" libtcl
    (_fun (interp : _interp = (current-interp))
          (expr : _string)
          -> _tclret)))
```

using the following `_tclret` definition:

```
(define _tclret
  (make-ctype (_enum '(ok error return ...))
    (lambda (x) (error "tclret: only for returning"))
    (lambda (x)
      (when (eq? x 'error)
        (error 'tcl (get-string-result
                      (current-interp))))
      x)))
```

which effectively translates a TCL error into a Scheme exception.

Note that the TCL interface uses a Scheme parameter '`current-interp`' as the value of the first argument to '`TCL_Eval`'. We can make this implicit by defining a new custom type syntax, using the '`expr:`' keyword:

```
(define-syntax _cur-interp
  (syntax-id-rules ()
    [_ (type: _interp expr: (current-interp))]))
(define eval-tcl
  (get-ffi-obj "Tcl_Eval" libtcl
    (_fun _cur-interp (expr : _string) -> _tclret)))
```

Using Custom Types

Custom types are intended to be used in situations where simple independent processing of each argument is insufficient. For example, many functions in the ImageMagick interface return a 'status' integer that indicates if there was an error. If an error has occurred, the main object involved in the function invocation should be used to retrieve the error message and severity. One way to deal with this situation is to save the object in a place accessible right after the foreign call, like a parameter. This is essentially what the TCL interface does, where `_tclret` uses a parameter to get the error message. The ImageMagick interface is different — instead of a single implicit context parameter, it fits more an object-oriented style, where each method call happens in its object's context.

As a result, a good interface must be able to provide a relation between different arguments, namely the result value (to be checked for an error) and the first argument (providing the current object context). This is done using the `1st-arg:` keyword of a custom type which specifies an identifier that will be bound to the first argument:

```
(define-syntax _status
  (syntax-id-rules (_status)
    [_status
     (type: _bool
      1st-arg: 1st
      post: (r => (unless r
                    (raise-wand-exception 1st))))]))
```

Memory Management

Usually, there are important aspects of the library interface that are not fully specified. Memory management issues often fall under this category. For example, a naive interface might behave in a surprising way:

```
> (define crypt
    (get-ffi-obj "crypt" "libcrypt"
      (_fun _string _string -> _string)))
> (define a (crypt "foo1" "23"))
> a
"23.kLNfMwUW0Q"
> (define b (crypt "foo4" "56"))
> b
"568.5HohJYC0g"
> a                     ; a is modified!
"568.5HohJYC0g"
> (string-set! a 0 #\X) ; verify that a and b
> (list a b)            ; are the same string
("X68.5HohJYC0g" "X68.5HohJYC0g")
> (eq? a b)             ; ...but not quite the same
#f
```

Using a simple SWIG interface, made using the C prototype declaration for '`crypt`':

```
extern char *crypt(const char *key, const char *salt);
```

suffers from this problem too. The reason for this strange behavior is that both our interface implementation and SWIG's generated code use MzScheme's '`make_string_without_copying`' function, which simply wraps an existing C string in a Scheme string object. The standard Unix `crypt` function returns a pointer to its own static string, making the above interaction create two Scheme string objects that point to this static buffer — but the Scheme objects are still different. This can be dangerous as it breaks an implementation assumption, so some solution is required. Changing the implementation to use '`scheme_make_string`' would not be acceptable in the general case since it leads to an expensive overhead. In addition, there are other foreign functions (e.g., `getcwd`) that can allocate a return string, and blindly copying it will cause a memory leak (the allocated string is not in GC-controlled memory).

Using our system simplifies such a solution since we don't have to break out of Scheme, we can simply use a new type[13]:

```
(define _string/copy
  (make-ctype _string #f
    (lambda (x) (string-append x #""))))
```

We can solve numerous problems in a similar way, for example, using semaphores to avoid problems with the single crypt buffer, or creating a new `_string/free` that copies a string and freeing the previous GC-invisible one.

[12]Use `_cpointer` as a base type, no extra translation when going to from Scheme to C, and register the destructor on the way back.

[13]Note that this is not relevant now, since our system is part of the Unicode-enabled MzScheme, so Scheme strings are stored in Unicode format, meaning that they are always copied.

5 Related Work

The first and foremost advantage that our foreign interface has over existing implementations, is the fact that it is truly dynamic. This means that functionality that traditionally is available only via C code is available to Scheme programmers, which makes for a compiler- and architecture-independent system. Furthermore, the dynamic aspect of the system allows for playing with foreign extensions dynamically, modifying and debugging the interface at run-time[14]. Exploratory programming is therefore possible, hence the overall development cycle becomes much lighter.

A second advantage comes from the fact that we use Scheme. Using a language with robust syntactical abstractions makes it possible to provide an IDL-like interface for interface programmers, with features that can go beyond capabilities of conventional IDLs [18, 16]. Having syntactic abstractions in the language makes it possible for users to extend their own code using new constructs, including ones that are unique to a single library, in contrast to fixed IDLs that are either fixed, or used through a primitive facility like the C preprocessor.

Dynamic interfaces are not as common as static interfaces. Existing dynamic systems, for example the Allegro CL foreign function interface [10] and Python's ctype module, do not provide the low-level C-substitute features that we do. Urban's FFI survey [17], although a little out-dated, provides an excellent overview on existing systems and implementation issues. It is interesting to note an SML interface system [2] as another, somewhat similar system to ours. Similar to our design, the main idea is *data-level interoperability* [8] — making raw C data available to the high-level language, but our system differs in a few important aspects:

- Our design is built around the idea of enabling arbitrary C-like unsafe code — whereas Blume's system uses SML's type system to enhance interaction with foreign code.
- Our system goes one step further in giving users more power. "If you can do it in C, then we will let you do it in Scheme" rather than "Some C-level operations are useful enough that we let you use them".
- Blume's system is limited to SML's syntactic framework, where we use Scheme's capabilities for creating IDL-like syntax.

We focus our comparison on static interface generators such as GreenCard [14], G-Wrap [3], and SWIG [1]. There are Scheme systems that fall under this category too by providing support for combining Scheme and C code, for example, Gambit-C[15] [5] and KSM [4]. Most notably, SRFI 50 [15] attempts to standardize this approach, possibly making it possible for different Scheme implementations to share C code. These systems make it possible to write Scheme code that is converted to C code, so it is easy to write such 'Scheme' code that calls C functions as if they were plain function calls. Some of these systems lack a code generation component that is derived by an IDL or some equivalent, but they can all be seen as static code generators.

We now focus on SWIG as a popular system that can be used for multiple high-level languages. A simple translation using SWIG requires the user to compile (through the SWIG parser) a C header file with a SWIG interface file, resulting in C code that is then, yet again, compiled using a C compiler, to produce a C module that is finally imported into Scheme. In contrast to the static approach, our '`foreign`' library makes it possible for a Scheme developer to quickly open up a C library, pull out a few procedure objects and start an interactive development session.

[14] As long as no fatal errors occur.

[15] Some parts of this were ported to PLT's MzC compiler.

func.	Glue Type	CPU	Real	GC
`crypt`	SWIG	38%	4%	-34%
	Handwritten C glue	53%	49%	0%
`sqadd`	SWIG	55%	57%	0%
	Handwritten C glue	60%	61%	0%

Table 5. Comparison of overhead time

It could be argued that a simpler, more user friendly system comes at a price of expensive overhead, leading to an inherent sacrifice of performance. Testing out two simple benchmarks, we found that the interface overhead of our system is just slightly slower as a compiled interface that was generated by SWIG, which itself has an almost identical overhead to hand-written glue code.

Our results are summarized in Table 5. Two functions were used for this analysis — the first is the `crypt` function taken from the standard Unix `libcrypt`: consuming two strings and producing an encrypted string result. The second is a simple C function, `sqadd`, that performs an addition of two integer squares. We measured a million executions of `crypt` and 30 million executions of `sqadd`, performing each test for 16 rounds beginning with a fresh MzScheme process, discarding the 6 extreme timings and averaging the other 10. The percentages are computed as: $\frac{Time_{PLT}-Time_{RawC}}{Time_{SWIG}-Time_{RawC}}-1$ where $Time_{PLT}$ is the averaged running time of our interface, $Time_{SWIG}$ is the average running time of SWIG, and $Time_{RawC}$ is that of an implementation of comparable repetition loops in C. The same computation was used to compare our system against handwritten C glue code.

As Table 5 shows, our system is about 1.5 times slower then SWIG, and, in most cases the handwritten glue code. The biggest performance hit is in the simple arithmetic function, where the actual foreign code does much less than the interface code. Situations like this should rarely occur since the usual case of using a foreign library is when it can do some substantial work that is otherwise hard to achieve in Scheme.

While issues of timing and performance are important, aspects such as implementation complexity and ease of use must also be considered. Comparing our system to SWIG and interfaces that use an IDL, it becomes clear that our implementation is better in at least one aspect. One advantage that our system provides over the static approaches is the ability to specify additional functionality using new user-defined types that involve arbitrary translation code. The main point here is that such translations are written in the high-level language itself rather than dealing with the intricacies of the C implementation.

In addition, regardless of interface design and syntactical complexity, our implementation is better because the interfacing mechanism itself is in a high order language: making it possible to include arbitrary Scheme code as part of the foreign call specification. This is further enhanced by the fact that we use Scheme since it is possible to create new syntactic abstractions to deal with new requirements. Either with SWIG interface files or with an IDL, the interface developer is still confined by C and C-like code with its known shortcomings when it comes to dealing with complex problems.

6 Future Work

C++ Libraries Currently, there is support only for plain C libraries. Depending on implementation details, it can be feasible to interface C++ libraries. This might involve plenty of details regarding object layout, inheritance, virtual function tables, name mangling, etc. Hopefully, these issues can be addressed in Scheme so we might not need any further enhancements to the C part of our implementation.

Parsing C One of the main disadvantage of our system is that it is not using C, so we cannot use C include files as rough interface specifications. We plan to investigate a simple C header-file parser that will parse files into s-expressions, which can be used to automate some aspects of interface generation (A working parser prototype exists). Such a parser does not need to be fast and efficient, since parsing can be done at syntax expansion time, eliminating any run-time speed costs. In addition, note that as usual with other interface generators, this will almost never mean that an interface can be fully automated, as header files do not provide enough information — this situation might improve if we target some IDL language instead (most use similar syntax).

Memory Management Issues Currently, our system works well with both versions of PLT Scheme: the one that uses the Bohm conservative garbage collector and the one that uses a precise moving collector. However, there are still issues that interface writers need to be aware of. In time, we will gain more experience writing interfaces, which will motivate further functionality that will make this easier — our goal is, of course, making GC-related issues as transparent as possible for interface writers.

One aspect of this, is dealing with struct objects that might contain GC-able pointers. We have a plan to deal with this, effectively making it possible to specify in Scheme a map of pointer offsets that the garbage collector should be aware of, making it treat new Scheme-defined structs properly.

Additional Scheme Support There are some areas in which additional Scheme support is needed. For example, an array of structs is hard to deal with — there is no way to get to one such struct and modify it, since accessing it will create a copy. We believe that it is possible to write Scheme code that will make this possible, by not pulling out a struct copy, but rather provide forms that will use nested reference indexes, where some are vector indexes and some are struct field names. If we can make this composable, it would be possible to deal with them in an easy way — without resorting to pointer aliasing[16].

An additional area where additional support is needed, is when dealing with foreign functions that block. MzScheme contains a few hooks that are intended to be used when it is embedded as a library, these hooks can be used for calling blocking foreign functions as well.

Using Contracts PLT Scheme has support for procedure contracts [7] which could be used to enhance the robustness of library interfaces. Specifically, we want to treat contract violations in modules that use the '`foreign`' module as more severe, as these are equivalents of C bugs, which might result in a crash. A module would also need some way of declaring it as a proper interface, meaning that code that uses it should not be blamed for crashes. Alternatively, code that is not intended as an interface (i.e., code that provides functionality for interface modules) should propagate the property of contract violation severity.

[16]The precise garbage collector makes it impossible to get a pointer to the internal part of an allocated block

Assembly Code Generation Working our way to native just-in-time compilation, we plan on adding machine-code generation ability to PLT Scheme. We will interface this functionality via the '`foreign`' module. Furthermore, some of the interface aspects can be implemented in assembly when runtime is important.

7 References

[1] D. M. Beazley. SWIG: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of the 4th USENIX Tcl/Tk Workshop*, pages 129–139, July 1996.

[2] Matthias Blume. No-longer-foreign: Teaching an ML compiler to speak C "natively". In *BABEL'01: First workshop on multi-language infrastructure and interoperability*, September 2001.

[3] Rob Browning. G-Wrap home page. `http://www.nongnu.org/g-wrap/`.

[4] Hangil Chang. KSM-Scheme home page. `http://square.umin.ac.jp/ hchang/ksm/`.

[5] Marc Feeley. Gambit Scheme system. `http://www.iro.umontreal.ca/ gambit/`.

[6] Marc Feeley. SRFI 4: Homogeneous numeric vector datatypes. `http://srfi.schemers.org/srfi-4/`.

[7] Robert Bruce Findler and Matthias Felleisen. Contracts for higher-order functions. In *ACM SIGPLAN International Conference on Functional Programming*, 2002.

[8] Kathleen Fisher, Riccardo Pucella, and John Reppy. Data-level interoperability. Bell Labs Technical Memorandum, April 2000.

[9] Matthew Flatt. *PLT MzScheme: Language Manual*. PLT, August 2004. Version 208.

[10] Franz Lisp. Foreign function interface. `http://www.franz.com/-support/documentation/6.1/doc/foreign-functions.htm`.

[11] Anthony Green. The libffi home page. `http://sources.redhat.com/libffi/`.

[12] Samuel P. Harbison. *Modula-3*. Prentice-Hall, 1992.

[13] Thomas Heller. The ctypes module. `http://python.net/crew/theller/ctypes/`.

[14] Simon Peyton Jones, Thomas Nordin, and Alastair Reid. GreenCard: a foreign-language interface for Haskell. In J. Launchbury, editor, *2nd Haskell Workshop*, 1997.

[15] RIchard Kelsey and Michael Sperber. SRFI 50: Mixing scheme and c. `http://srfi.schemers.org/srfi-50/`.

[16] The Open Group. *CAE Specification, DCE 1.1: Remote Procedure Call*, chapter 4. The Open Group, October 1997.

[17] Reini Urban. Design issues for foreign function interfaces. `http://xarch.tu-graz.ac.at/autocad/lisp/ffis.html`, Last updated at 2004.

[18] A. Vogel, B. Gray, and K. Duddy. Understanding any IDL — lesson one: DCE and CORBA. In *Proceedings of the Third International Workshop on Services in Distributed and Networked Environments (SDNE'96)*, 1996.

Acknowledgments

We would like to thank Matthew Flatt: this work would not be possible without his help, especially with GC-related issues. The comments and suggestions made by the reviewers have been extremely helpful, Mike Sperber was particularly helpful in the process of revising this text.