Integrating Foreign Libraries in Common Lisp: Best Practices

Rudi Schlatte
Joanneum Research

June 4, 2006

Abstract

We explore some of the challenges in integrating foreign libraries into Common Lisp. While generating library bindings automatically from header files and annotations can take care of some basic tasks, an API that integrates well into the language and provides the user with familiar patterns of usage must still mostly be implemented by hand. We illustrate some best practices and common pitfalls following the example of cl-redland, a CL binding for the Redland RDF library.

1 Introduction

In recent years, Common Lisp (CL) enjoyed an upswing in popularity. Recent developments in software practices have revalidated Lisp's approaches, such as dynamic typing and interactive development. In contract to typical scripting languages, CL sports a robust ANSI standard and multiple implementations (both open-source and commercially supported), most of which have highly optimizing native-code compilers.

Hence, the situation described by Gabriel [4, page 192] has been mostly corrected:

> Lisp was too big and too slow for real applications. Applications needed to comprise a mixture of Lisp and other code, and the membrane between the two languages had to be thin [...]

Moore's law took care of the "too big and too slow" part – a typical Common Lisp system runs very well on typical "consumer" machines and needs fewer resources than, e.g., the Java runtime from Sun. But communication between Lisp and programs or libraries written in other languages is still a concern. Common Lisp the Standard does not prescribe a way to call out to foreign libraries written in the host system's native language and calling conventions (today, normally C).

Nevertheless, no current implementation of Common Lisp is as closed off as reading the standard suggests – all major implementations provide a way to connect to the outside world via the network or calling out to foreign libraries, and various compatibility layers exist that provide a common API for these functions across implementations.

In this paper, we will present some "best practices" that were formulated during the development of cl-redland, a Common Lisp binding of the Redland RDF toolkit [7]. Redland is a library for the RDF (Resource Description Framework) data exchange and description format [6] that forms the basis for the semantic web. Redland is written in portable C and consists of a parser for various RDF syntaxes, an engine for the SPARQL query language and backends for storing RDF triple graphs in-memory, in files and in various database engines.

Section 2 describes the various options of integrating foreign libraries in Common Lisp implementations, while section 3 explores the design issues that need to be solved to give the library user more than the basic, low-level building blocks of an interface.

2 Foreign Function Interfaces

Generally, a "foreign function" is a function that is implemented in a different language than the main program. A *foreign function interface* makes it possible to call foreign functions (and access foreign variables) from Lisp.

Since a foreign function interface is not part of the ANSI standard for Common Lisp, implementations have come up with different ways of calling foreign functions. Luckily, there are compatibility libraries that implement a common layer atop the foreign function interfaces of various CL implementations.

For cl-redland, we chose CFFI [2], a library that supports the majority of Common Lisp systems, both commercial and open-source, and integrates well into the Lisp software ecosystem. It has an asdf [1] system definition, hence can be automatically loaded by other systems using asdf, and can be installed via asdf-install. Another good compatibility layer for FFIs is UFFI [10]. CFFI was chosen because it supports slightly more implementations and, crucially, provides portable interface for hooking into the implementations' garbage collector.

Using a specific implementation's FFI was not considered, since cl-redland aims to be portable and useful across implementations.

2.1 Generated Versus Hand-Written Bindings

The first (but not the only) step in integrating the functionality of a foreign library in a Common Lisp system is writing the bindings for the foreign functions and foreign global variables. This is largely a mechanical task – so

much so that there are programs to generate the code, for example SWIG [9]. Generated bindings can save the programmer some typing work, but the bindings still have to be adjusted manually in most cases (see subsection 2.2 for one reason why).

In this section, we use as an example a small part of the Redland API – the initialisation and termination of the library via the following functions:

```
typedef librdf_world;
librdf_world* librdf_new_world (void);
void librdf_free_world (librdf_world *world);
void librdf_world_open (librdf_world *world);
```

A first, hand-written version of this part of the api looked like this:

```
(defctype librdf-world-pointer :pointer)
(defcfun-with-checked-alloc "librdf_new_world"
    librdf-world-pointer)
(defcfun "librdf_world_open" :void
  (world librdf-world-pointer))
(defcfun "librdf_free_world" :void
  (world librdf-world-pointer))
```

Since redland already uses SWIG to generate bindings to various languages and ships with a `Redland.i` SWIG interface file, it was decided to use the CFFI bindings generated by SWIG as a starting point for cl-redland.

The equivalent bindings, as produced by SWIG:

```
(defcfun ("librdf_new_world" librdf_new_world) :pointer)

(defcfun ("librdf_free_world" librdf_free_world) :void
  (world :pointer))

(defcfun ("librdf_world_open" librdf_world_open) :void
  (world :pointer))
```

Cffi's `defcfun` macro by default generates "lispified" identifiers (`librdf_world_open` becomes `librdf-world-open`). The generated SWIG bindings take extra steps to avoid this conversion; this is a good thing, since this way the low-level lisp-side functions have the same name as their C counterparts.

Some small adjustments had to be made to the generated bindings, but as a whole the decision to use SWIG instead of hand-written low-level bindings has paid off and saved the implementor some hours of typing boilerplate code.

2.2 Dealing with Pointer Ownership

A problem that can be solved neither by automated generation of bindings nor by mechanically transcribing C header files is the ownership of pointer return values of C functions. C function declarations do not provide information on whether the caller is expected to call `free` on the returned pointer.

For example, here is the C function to obtain the textual representation of a Redland URI object:

```
unsigned char* librdf_uri_to_string (librdf_uri* uri);
```

And here is the C function to obtain the literal from a node:

```
unsigned char* librdf_node_get_literal_value
  (librdf_node *node);
```

One of these functions returns a freshly-`malloc`'d string that the caller must free, the other one returns a shared string that must not be modified or deallocated by the caller.

The documentation or C source code must be consulted to determine the needed behaviour of the caller of a specific C function. Except with some sort of source code annotation, this information can not be known by inspecting the header files alone.

With CFFI, return values that must be deallocated by the caller can be handled by defining another datatype. The following datatype was used in cl-redland:

```
(defctype caller-owned-string :pointer)

(defmethod translate-from-foreign
    (value (name (eql 'caller-owned-string)))
  (prog1 (foreign-string-to-lisp value)
    (unless (null-pointer-p value)
      (foreign-free value))))
```

With this datatype, the `librdf_uri_to_string` foreign function is defined as follows:

```
(defcfun ("librdf_uri_to_string" librdf_uri_to_string)
    caller-owned-string
  (uri :pointer))
```

3 Implementing a Lispy API

Creating FFI bindings for a C library is only the first step. Without higher-level Lisp constructs, the code using the library will be as prone to exhibit memory leaks and crashes due to programmer errors as the equivalent C code. The library user should not have to worry about deallocating memory by hand or about type-safety issues. This section describes how the low-level C-style constructs should be wrapped to produce an API that is pleasing to the Lisp programmer.

3.1 Exception Handling

Typically, foreign functions signal failure via a special return value. The Lisp-style API should check for failed foreign function calls and raise an appropriate exception. It is recommended that the library build its own

hierarchy instead of just calling error with a string argument – otherwise the library user would have to resort to string comparison if he wants to handle a specific condition.

```
(define-condition redland-error (error)
  ())

(define-condition allocation-error (redland-error)
  ((object-type :reader object-type-of :initarg :object-type))
  (:report
   (lambda (condition stream)
     (format stream "Failed object allocation for ~A."
             (object-type-of condition)))))

(defun allocation-error (object-type)
  (error 'allocation-error :object-type object-type))
```

Where possible, the Lisp bindings should provide restarts so that the library user can handle errors programatically.

Beyond these hints, we will not dwell on Common Lisp's condition system; for a good introduction see for example [8, chapter 19].

3.2 Library Initialization

Some libraries need to be initialized once or have a global handle that is passed to various foreign functions. If the selfsame object is passed to all or most functions, it need not be made part of the lisp-side API. Instead, it can be initialized and stored at library-load time.

Herre is how cl-redland initializes the Redland foreign library, using the low-level bindings from section 2.1:

```
(defvar *world*
  (make-instance
   'world
   :pointer (let ((world (librdf_new_world)))
              (when (null-pointer-p world)
                (redland-alloc-error "world"))
              (librdf_world_open world)
              world))
  "The global Redland ẅorld¨ object.")
```

If the library only needs to be initialized but does not have a global state object, the following variation can be employed:

```
(defvar *library-initialized-p*
  (progn (initialize-library) t))
```

In both cases, the variable holding the library state handle needs not be exported or documented, since it is an implementation detail not of interest to Lisp-side library users.

3.3 Using Objects Instead Of Raw Pointers

Each resource that is created C-side should have a corresponding Lisp object that can be inspected, used in generic function dispatch and that can handle the lifetime of the C-side resource. Take, for example, the part of the Redland API that deals with creating, copying, printing and deallocating URI objects:

```
librdf_uri* librdf_new_uri (librdf_world *world,
    const unsigned char *uri_string);

librdf_uri* librdf_new_uri_from_uri (librdf_uri* old_uri);

void librdf_free_uri(librdf_uri *uri);

unsigned char* librdf_uri_as_string (librdf_uri *uri);
```

The Lisp-side URI object just stores the C-side pointer, with no state of its own:

```
(defclass uri ()
  ((pointer  :initform nil :initarg :pointer
             :reader pointer)))
```

The remainder of this subsection shows how to implement the behaviour for wrapper objects that lets them integrate seamlessly into a Lisp system.

3.3.1 The `print-object` method

Each object should have a corresponding `print-object` method that shows helpful, short information about the object to the library user. This is especially important for wrapper objects, because much of the object state will reside in the foreign library where it is unavailable to Lisp-side introspection – for example, inspecting a wrapper object in the debugger of the Lisp implementation will likely only show an opaque pointer.

```
(defgeneric  uri-to-string (uri)
  (:method ((uri uri))
    (librdf_uri_to_string (pointer uri))))

(defmethod print-object ((uri uri) stream)
  (print-unreadable-object (uri stream :type t :identity t)
    (format stream "~A" (uri-to-string uri)))
  uri)
```

With the `print-object` method in place, the printed output of uri objects contains some useful info.

```
CL-USER> (rdf:uri "http://example.com/")
#<CL-REDLAND:URI http://example.com/ {11F5D0D9}>
CL-USER>
```

`describe-object` can be specialized in a similar way.

3.3.2 Wrapper Object Construction Protocol

Typically, creating a wrapper object means observing a more or less elaborate allocation protocol to create the wrapped foreign resource at the same time.

Keene [5, page 25] recommends the creation of constructor functions, for example `make-uri` for the class `uri` to enforce mandatory instance initialization steps. An alternative way, chosen for cl-redland, is to define an `:after` method on the generic function `initialize-instance`, to make sure the user cannot construct uninitialized wrapper objects by calling `make-instance` directly[1].

```
(defmethod initialize-instance
    :after ((instance uri) &key uri-string old-uri
            &allow-other-keys)
  (let ((uri-pointer
         (cond
           (uri-string (librdf_new_uri (pointer *world*)
                                       uri-string))
           (old-uri (librdf_new_uri_from_uri
                     (pointer old-uri)))
           (t (error "Can't create URI without one of ~
uri, uri-string."))))))
    (when (null-pointer-p uri-pointer)
      (allocation-error "uri"))
    (setf (slot-value instance pointer) uri-pointer)
    (cffi:finalize
     instance (lambda () (librdf_free_uri uri-pointer)))))
```

The method on `initialize-instance` enforces the invariant that each `uri` object has a corresponding foreign object and allows the user to construct `uri` objects from other `uri` objects and URI strings via `make-instance`.

Whether to implement constructor functions is a matter of taste; cl-redland does not provide them because the additional boilerplate code and documentation effort was not considered worthwhile.

3.3.3 Designator Functions

Common Lisp has the concept of *designators*, i.e. objects that denote other objects. For example, in Common Lisp strings designate pathnames, so all functions handling pathnames are prepared to handle strings as well.

[1] The author's (unsubstantiated) guess is that the object initialization protocol was not yet finalized when Keene's book was written, hence the need for constructor functions in her work.

Typically, a function with the same name as the type exists that converts all designators to the designated type – for example, the function `pathname` takes a pathname designator and returns a pathname object.

Where appropriate, the library author should provide designators to the user. In `cl-redland`, strings are URI designators via the following function:

```
(defun uri (thing)
  (etypecase thing
    (uri thing)
    (string (make-instance 'uri :uri-string thing))))
```

Most of the functions in cl-redland that operate on URI objects also accept strings, making them more pleasing for the programmer to work with.

3.3.4 Destruction and Resource Handling

To avoid memory and resource leaks, the foreign datastructures must be deallocated when no longer needed. Typically, Lisp implementations provide a way to hook into the garbage collector so that user-supplied code is executed when a Lisp object is collected. Here are the relevant parts of the `initialize-instance` method again – at the end, Lisp is informed what to do when garbage-collecting a `uri` object.

```
(defmethod initialize-instance
    :after ((instance uri) &key uri-string old-uri
            &allow-other-keys t)
  (let ((uri-pointer [...]))
    [...]
    (cffi:finalize
     instance (lambda () (librdf_free_uri uri-pointer)))))
```

Although it should be general knowledge by now, we wish to emphasize that relying on garbage collector finalization methods for resource deallocation, especially for non-memory resources, is inherently unsafe! This is because the garbage collector runs at unpredictable intervals and might not collect objects for surprisingly long times, especially when memory pressure is low in the Lisp system. Hence, means of explicitly deallocating foreign objects (and invalidating the corresponding Lisp wrappers) should be provided at least for non-memory resources, for example connections to databases.

For non-memory foreign resources, facilities similar to Common Lisp's `with-open-file` should be implemented by the library. For example, Redland provides *storage* objects that deal with storing triples in memory, a file or a database. Here is cl-redland's (somewhat long) `with-storage` macro:

```
(defmacro with-storage
    ((storage &key (storage-name "memory")
              (name "test") (options-string ""))
     &body body)
  "Evaluates BODY with STORAGE bound to a triple store. Also
binds *STORAGE* to the same triple store for the extent of
the form. The triple store object to which STORAGE is bound
has dynamic extent; its extent ends when the form is exited."
  (let ((created (gensym "CREATED-")))
    `(let* ((,created nil)
            (,storage (make-instance
                       'storage :storage-name ,storage-name
                       :name ,name
                       :options-string ,options-string))
            (*storage* ,storage))
       (unwind-protect (progn (setf ,created t)
                              ,@body)
         (when ,created
           (let ((pointer (redland-pointer ,storage)))
             (librdf_storage_sync pointer)
             (librdf_storage_close pointer)
             (librdf_free_storage pointer))
           (setf (slot-value ,storage 'redland-pointer) nil)
           (sb-ext:cancel-finalization ,storage))))))
```

It is still possible to create storage objects with indefinite extent via `make-instance` normally, but when the lifetime of the object is known, `with-storage` takes care to properly dismantle the storage object immediately and deregister it from the garbage collector after running the supplied code block.

Cl-redland does not contain explicit means of deallocating foreign memory-only resources at the moment. If it is discovered that finalizer methods do not free foreign memory at a sufficient rate, implementing a generic function `free-object` is considered. However, the author considers this as a measure of last resort because `free-object` would leave an "empty" wrapper object still accessible to the application, thereby breaking a useful invariant.

3.4 Provide Iterators And Result Lists

Because of the sparseness of C, operations that return a set of results are usually implemented via *iterators* (also called *cursors* in database binding libraries). For the sake of brevity, this describes solutions in an abstract way, positing a foreign `abstract-handle` and functions to get the result objects from it.

Lisp-side, there are some ways of giving result sets to the library user.

A `get-result-stream` / `get-next-result` combination is useful when a large number of results is expected; it is used like this:

```
(loop with results = (get-result-stream foreign-handle)
  for result = (get-next-result results)
  while result
  do (process result))
```

If `nil` can be a valid result, have `get-next-result` return two values: the result and a boolean that indicates whether there are more results waiting.

The `get-all-results` method of collecting iterator data accumulates all results in a list and is used in the following way:

```
(dolist (result (get-all-results foreign-handle))
  (process result))
```

This is more concise than the previous form, but has the disadvantage that it collects all results beforehand. For large result sets or result objects, this will increase memory pressure somewhat.

Another form of working with iterator results is to provide a custom looping macro `do-results`:

```
(do-results (result foo)
  (process result))
```

This is as concise as the `dolist` form and potentially as memory-efficient as the `loop`, but not all users are comfortable with new iteration constructs.

A reasonable approach might be to provide all of the above forms, especially since each one is trivially implementable on top of the others.

3.5 Integrate With Other Libraries

The system should come with a system definition file for `asdf` [1] or `mk-defsystem`, so that it can be automatically loaded by other systems. Also consider making the library automatically installable via `asdf-install` [3]. Using this infractructure means that users can install at least the Lisp parts of the library with minimal effort.

4 Conclusion

Creating bindings to foreign libraries can be a worthwhile alternative to re-implementing existing code in Common Lisp. Seamless integration of foreign code is possible, and the results can be as pleasing to use as a wholly-native library. Binding generators such as SWIG can take care of low-level interface code with some help, but the implementation of an object-oriented API must be done by hand. Further work is needed for seamless distribution – at the moment, the foreign library must be manually installed, since Lisp-level tools like `asdf-install` presently do not (and cannot be expected to) support the

automatic installation of arbitrary system-specific library code. Still, using a foreign library is a good way to get access to needed functionality in Lisp in a short time.

References

[1] Daniel Barlow. Asdf manual. `http://constantly.at/lisp/asdf.pdf`. (14 Apr. 2006).

[2] Cffi – the common foreign function interface. `http://common-lisp.net/project/cffi/`. (26 Feb. 2006).

[3] Daniel Barlow et al. asdf-install. Tutorial at `http://cvs.sourceforge.net/viewcvs.py/*checkout*/cclan/asdf-install/doc/index.html`. (17 Apr. 2006).

[4] Richard P. Gabriel. *Patterns of Software*. Oxford University Press, 1996. `http://dreamsongs.com/NewFiles/PatternsOfSoftware.pdf` (22 Jun. 2006).

[5] Sonya Keene. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, 1989.

[6] Rdf primer. `http://www.w3.org/TR/rdf-primer/`. (25 June 2006).

[7] Redland rdf application framework. `http://librdf.org/`. (26 Feb. 2006).

[8] Peter Seibel. *Practical Common Lisp*. Apress, 2005. `http://gigamonkeys.com/book/` (22 Jun. 2006).

[9] Simplified wrapper and interface generator. @urlhttp://www.swig.org/. (26 Feb. 2006).

[10] Uffi: Universal foreign function interface for common lisp. `http://uffi.b9.com/`. (26 Feb. 2006).