CLAUDE – The Common Lisp Library Audience Expansion Toolkit

Nick Levine
Ravenbrook Limited
PO Box 205
Cambridge, CB2 1AN
United Kingdom
ndl@ravenbrook.com

ABSTRACT

CLAUDE is a toolkit for exporting libraries written in Common Lisp, so that applications being developed in other languages can access them. CLAUDE co-operates with foreign runtimes in the management of CLOS objects, records, arrays and more primitive types. Lisp macros make the task of exporting a library simple and elegant; template documentation along with C headers and sample code files relieve some of the burden of explaining such exports to the application programmer.

Categories and Subject Descriptors

D.2.12 [**Software Engineering**]: Interoperability—*Distributed objects*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Software libraries*

1. INTRODUCTION

One the the ways in which most Common Lisp implementations extend the ANSI standard is by providing some means of distributing a lisp application as a pre-compiled executable. Another essential extension is the capability to link into libraries written in other languages. The first of these features greatly extends the potential audience of an application, exposing it to end-users who don't ever want to meet a lisp loader (and indeed have no reason to ever know what language the application was written in). The other gives the application writer access to a wealth of foreign libraries and so leaves them freer to get on with solving their core problems.

Less obvious is the possibility of distributing a lisp program as a pre-compiled library, into which authors of applications written in other languages could link. These authors form a vast talent pool, and their users a massive audience base beside which the number of people running applications written in lisp is regrettably less impressive.

In this paper we will discuss a nascent toolkit intended to address this problem. The *Common Lisp Library Audience Expansion Toolkit* (CLAUDE)[1] includes:

- macros for exporting classes and functions;
- shared memory strategies for the allocation and freeing of objects, records, arrays and strings;
- *callbacks* (which in this context means calls from lisp back into the outside world of the application);
- support for UTF-8; high data throughput; full thread-safety; error handling with backtraces visible to the application programmer;
- tests, and templates for generating documentation and C / Python examples.

Authors of libraries written in lisp, in their search for a wider audience, can use CLAUDE to write a thin interface layer on top of their code; the result is saved as a DLL. They should extend the provided header and sample code files and template documentation, to cover their own exported classes and functionality as well as CLAUDE's. Their distribution to application programmers consists then of the DLL plus these extended files. Although the explicit support is for C and Python, there's no particular reason why applications should confine themselves to these two languages: CLAUDE is totally agnostic about the application's implementation choices. In theory CLAUDE can be even used to link two different lisp implementations (we might then, say, drive the LispWorks *CAPI* windowing environment from SBCL); however CLAUDE's interface is deliberately aimed at lower-level languages than Common Lisp and so in practice talking to other lisps won't be its greatest strength.

We will distinguish two roles:

- the *library writer* uses CLAUDE to export their Common Lisp library; and
- the *application programmer* uses that library as part of their (C / Python / other) application.

[1] `http://www.nicklevine.org/claude/`

In the following we will use Python – and to a lesser extent C – to illustrate the foreign (i.e. application) side of several CLAUDE exports.

Hitherto CLAUDE has been deployed only on LispWorks; some LW specifics are present in the code fragments below, exported from the packages SYS and FLI. Other implementations for which a port is possible should be accessible in return for reasonable effort. Indeed where convenient CLAUDE uses Bordeaux Threads (BT) and the Common Foreign Function Interface (CFFI), to help reduce the cost of future ports. However, and this is most unfortunate, most lisps will not be able support CLAUDE, because the facility to save lisp code as a linkable library is not generally available. We expect additional ports to be limited to Allegro and to the lisps which directly embed into other runtimes (ABCL and ECL).

2. EXAMPLE

The inspiration for the present work was Ravenbrook's *Chart Desktop*: a free, pre-existing, 15k LoC library for the layout and interactive display of large, complex graphs[2]. Here, by way of an introduction to CLAUDE's capabilities, is an example of driving part of the Chart library externally.

On the lisp side, we can say:

```
(defclass-external graph (chart:graph)
  ())

(defun-external (new-graph :result-type object) ()
  (make-instance 'graph))
```

In both C and Python there's quite a bit of overhead before we can get started. The CLAUDE distribution includes all the code for that; once this is out of the way, we can now go:

```
claude_handle_t graph;
CHECK(chart_new_graph(&graph));
```

Here `chart_new_graph()` is an export from the library, automatically created by the `defun-external` form; note the mapping between the lisp and foreign function names. `CHECK` is an unedifying C macro provided with CLAUDE for confirming that the call into lisp returned a success code. After this call, `graph` will contain a CLAUDE *handle*: a value which can be passed back into lisp at any time to refer unambiguously to the CLOS object in question.

In Python it's worth establishing a more sophisticated correspondence with the CLOS object than the raw pointers that C provides:

```
>>> from pyChart import chart
>>> chart.Graph()
<Chart Graph handle=0x20053748>
>>>
```

[2]`http://chart.ravenbrook.com/downloads`

Here's the application writer's Python code to drive this. The CLAUDE function `val()` makes the foreign call (i.e. into lisp), checks for a success code, and dereferences the pointer result; `init()` establishes a correspondence (in a Python dictionary) between the new Graph instance and its Chart handle.

```
class Graph(ClaudeObject):
    def __init__(self):
        handle = val(dll.chart_new_graph)()
        ClaudeObject.init(self, handle)
```

Meanwhile, provided the lisp library was configured when it was saved to open a REPL on restarting, we can check:

```
(in-package claude)
(wrapper-object (gethash #x20053748 *wrappers*))
->
#<Chart Graph handle=0x20053748>
```

Note the similarities in printed representations, between the CLOS object and its Python counterpart.

3. DATA MODEL

CLAUDE functions take arguments of the following types:

- integer (signed and unsigned);
- UTF-8 encoded string;
- *handle* (used to denote a CLOS object within the library);
- *record* (in this paper: a sequence of values whose length and constituent types are determined by the context in which it's being passed);
- *array* (in this paper: a sequence of values whose constituent types are all the same, but whose length is not known in advance);
- pointer to any of the above (for return values);
- pointer to function (for callbacks).

There's no particular reason why floats can't be supported; Chart happened not to need them.

It's vital that the responsibilities for allocating and recycling memory are clearly laid out, and that CLAUDE co-operates transparently with foreign memory managers.

Aggregate values – strings, records (for example, a pair of integers representing x-y co-ordinates) and arrays (for example, a list of such co-ordinates, or a list of CLAUDE objects) – may be generated by either the library or the application.

The contents of any aggregate value created by the application, passed into the library and retained there – for example, a string to be drawn inside some display window, or an array of records describing a set of new objects to be added

to the system – are copied and retained by CLAUDE. So the application may immediately recycle any memory associated with such aggregates, after the library call using them has returned.

An aggregate value created within the library and passed to the application as return data will be retained by CLAUDE, until such time as the application declares that it has no further use for this by calling the library function `claude_free()`. If the application frees a record or array which itself contains any aggregate values, then these values will be freed recursively.

```
CHECK(chart_new_nodes(&nodes, graph, node_defs));
node_0 = nodes->values[0].handle;
node_1 = nodes->values[1].handle;
/* Tell Chart it may free the array in which it
   returned the nodes. */
freeable.array = nodes;
CHECK(claude_free(freeable));
```

A handle may only be generated by the library, by making an instance of an external class (one defined by `defclass-external`) and then returning that instance from a call to `defun-external`, as in the `new-graph` or `chart_new_nodes()` examples above. When the application has no further use for a set of handles, it should call `claude_remove_objects()` to invalidate the CLAUDE objects and permit memory to be recycled on both sides of the fence. Here's the corresponding lisp function for that:

```
(defun-external (remove-objects
                 :result-type (array object
                                     :call 'discard))
    ((array (array object)))
  (remove-duplicates
   (loop for object in array
         append (remove-object object))))
```

Note that both the arguments and the return value of `remove-objects` are (CLAUDE) arrays of objects. This interface is designed to allow the library to:

1. take further action when an object is invalidated;
2. cause the invalidation of one object to invalidate others with it (for example, in Chart, if the application removes a node from the graph then the library will ensure that the node's edges are removed as well); and
3. decline to invalidate an object, if it so chooses.

The calls to `remove-object` allow the library to update its structures and specify secondary removals; supplying `'discard` against the `:call` keyword invokes the following method on each of the dead objects, to recycle lisp memory:

```
(defmethod discard ((self object))
  (let* ((wrapper (shiftf (object-wrapper self)
                          nil))
         (address (sys:object-address wrapper)))
    (setf (wrapper-object wrapper) nil)
    (remhash address *wrappers*)
    wrapper))
```

Finally, `remove-objects` returns to the application the full set of invalidated handles. The CLAUDE objects corresponding to each of these handles have been turned over for garbage collection; the application should no longer communicate with the library about those objects and any attempt to do so will signal a CLAUDE error. If the application has been mapping the handles into its own objects then these should now be recycled:

```
def _discard(self):
    handle = self.handle
    del _objects[handle]
    self.handle = None
```

4. ARGUMENT TRANSLATION

Every CLAUDE object has a *wrapper* which is a statically allocated structure, doubly linked to the object:

```
(defstruct wrapper
  object)

;; Debugging aid (mostly)
(defvar *wrappers* (make-hash-table))

(defmethod initialize-instance :after
    ((self object) &key)
  (let* ((wrapper (sys:in-static-area
                   (make-wrapper object)))
         (address (sys:object-address wrapper)))
    (setf (gethash wrapper *wrappers*) address
          (object-wrapper self) wrapper)))
```

The external representation of an object is its wrapper's memory address, guaranteed never to change. The generic-function `box` which retrieves this address is cautious about possible errors (as is its counterpart `unbox` which converts incoming addresses back into the corresponding CLAUDE objects):

```
(defmethod box ((self object)
                &key &allow-other-keys)
  (if (slot-boundp self 'wrapper)
      (let ((wrapper (object-wrapper self)))
        (if wrapper
            (sys:object-address wrapper)
          (error "~a doesn't have a handle."
                 self)))
    (complain "~a has not yet been initialised."
              self)))
```

```
(defmethod box ((self null) &key allow-null)
  (if allow-null
      0
    (call-next-method)))

(defmethod box (self &key &allow-other-keys)
  (complain "~a is not a ~a object and cannot ~
             be boxed."
            *library* self))
```

Note the use of zero to denote null objects, when the library's API permits this.

Boxing and unboxing are handled behind the scenes, by a LispWorks *foreign-converter* object. Such converters make the argument and result handling of `defun-external` elegant (and indeed quite invisible to the library writer).

```
(fli:define-foreign-converter object
    (&key allow-null type) (object address)
  :foreign-type 'uint
  :foreign-to-lisp `(unbox ,address
                           :allow-null ,allow-null
                           :type ,type)
  :lisp-to-foreign `(box ,object
                         :allow-null ,allow-null))
```

We suggest that if possible the application should mirror the above process, if not its elegance, so that it too is dealing with objects rather than raw pointers:

```
_objects = {0:None}

class ClaudeObject(object):
    def __init__(self, handle):
        _objects[handle] = self
        self.handle = handle

    def init(self, handle):
        self.__init__(handle)

    def box(self):
        return self.handle

def unbox(handle):
    return _objects[handle]
```

Depending on the level of introspection exposed by the library, the application's `unbox` might be used interactively as a debugging aid. For example in Chart, faced with the following error...

```
pyChart.invoke.ChartError: Source and destination
are the same node (#<Chart Node handle=0x20053838>),
which is not permitted.
```

...`unbox` can identify the object with handle `0x20053838`, and then `chart_describe_nodes()` can ask the library to provide further information about it:

```
>>> unbox(0x20053838)
<Chart Node handle=0x20053838>
>>> chart.describe_nodes([_])
[(<Chart Graph handle=0x20053748>,
  [], u'Hello World!')]
>>>
```

Alternatively, a little cut-and-paste allows us to access the object on the lisp side:

```
CLAUDE 17 > (unbox #x20053838)
#<Chart Node handle=0x20053838>

CLAUDE 18 > (describe *)

#<Chart Node handle=0x20053838> is a CHART:NODE
WRAPPER           #<WRAPPER Node 2005383B>
GRAPH             #<Chart Graph handle=0x20053748>
LABEL             "Hello"
EDGES             NIL

CLAUDE 19 >
```

A record is passed as a raw sequence of values; an array is implemented as a record whose first member is the array's length, and whose remaining members are the array's values. Corresponding to `box` and `unbox` for objects, we have `construct` and `deconstruct` for records, and `pack` and `unpack` for arrays.

```
def set_callbacks(object, callbacks):
    c_tuples = map(lambda((name, function)): \
                   (c_string(name), function or 0),
                   callbacks)
    records = map(construct, c_tuples)
    array = pack(records)
    boxed = object.box() if object else None
    lib.set_callbacks(boxed, array)
```

In LispWorks both of these aggregate types are handled by foreign-converters, and once again the library writer is isolated from any unpleasantness. For example (decoding of incoming record addresses):

```
(fli:define-foreign-converter record
    (types &key allow-null) (objects address)
  :foreign-type 'uint
  :foreign-to-lisp `(deconstruct ,address ',types
                                 :allow-null
                                 ,allow-null)
  :lisp-to-foreign `(construct ,objects ',types
                               :allow-null
                               ,allow-null))

(defun deconstruct (address types &key allow-null)
  (let ((pointer (cffi:make-pointer address)))
    (if (cffi:null-pointer-p pointer)
        (unless allow-null
          (complain "Null pointer for record ~
```

```
                    which expected ~a"
                   types))
      (loop for type in types
            for index from 0
            collect
            (follow pointer index type)))))

(defun follow (pointer index type)
  (let ((address (cffi:mem-aref pointer 'uint
                                index)))
    (ecase (follow-type type)
      ((array)   (unpack address (cadr type)))
      ((boolean) (not (zerop address)))
      ((object)  (unbox address :type type))
      ((uint)    address)
      ((int)     (if (<= address
                         most-positive-fixnum)
                     address
                   (+ (logand address
                              most-positive-fixnum)
                      most-negative-fixnum)))
      ((record)  (apply 'deconstruct
                        address (cdr type)))
      ((ustring) (from-foreign-string
                  (make-string-pointer
                    address))))))
```

Note incidentally that the CLAUDE package shadows the symbol `ARRAY`.

One particular use of arrays is for reducing the stack switching overhead of making very many calls into or out of lisp, which can otherwise become significant. We strongly recommend that wherever possible the data for such calls should be packed into arrays and the interface desgined so as to enforce fewer calls across the language boundary. For example, Chart supports `chart_new_nodes()` rather than `chart_new_node()`[3].

5. FUNCTION CALLS

The macro `defun-external` defines the library's functional interface. It hides everything that really matters but the programmer never wants to see: argument unmarshalling, checking and result marshalling; foreign name translation; memory management; error handlers. It installs an entry point into lisp (as if by `cffi:defcallback`), arranging for the corresponding C-name to be exported from the library and thus made visible to the application programmer. By hacking into the `defcallback` form's macroexpansion it is able to wrap error handlers around argument deconstruction as well as the body of the function; if for example an argument of the wrong type shows up then CLAUDE can explain the problem to the application:

```
>>> chart.describe_nodes([g])
Traceback (most recent call last):
  [...]
  File "pyChart\invoke.py", line 63, in invoker
    check(func, ctypes.byref(pointer), *args)
  File "pyChart\invoke.py", line 54, in check
    raise ChartError()
pyChart.invoke.ChartError: #<Chart Graph handle=
0x20053748> is a graph, but a node was expected.
>>>
```

The types of all arguments and the result (if any) must be specified, to allow for automatic conversion on the way in and out of the function. For example, this function:

```
(defun-external (display-graph
                 :result-type (object
                               :allow-null t))
    ((display display))
  (chart:display-graph display))
```

takes a display and returns a CLAUDE object, possibly null; and this one:

```
(defun-external set-node-locations
    ((display display)
     (nodes-and-locations (array
                           (record
                            (node
                             (record
                              (int int)
                              :allow-null t))))))
  (loop for (node location) in nodes-and-locations
        do
        (setf (chart:node-location node display)
              (when location
                (apply 'chart:make-location
                       location)))))
```

takes a display and a sequence of (node location) pairs, where the location is either a pair of integers or null, and the function does not return a result.

Passing a variable number of arguments across a foreign function interface can get ugly and we've chosen not to support it. The arguments to an external function are always fixed; the library is free to accept nulls for arguments whose specification was in some sense optional; and an array argument can have variable length.

By default each external call runs in its own thread. This allows for a last-ditch defense against errors in the library (or CLAUDE, or for that matter the lisp implementation): if the thread dies then the call is necessarily over but the program is not[4]. This precaution might not seem very important once the library is believed to be working properly, but during development it's invaluable.

[3] A simple experiment indicates that if `chart_new_node()` were made available, it would generate 1000 nodes almost an order of magnitude slower than its array counterpart.

[4] LispWorks executes all incoming calls in its "thread created by foreign code"; if this thread is killed (say by an unguarded `"Quit process"` restart) then no further incoming calls can be handled in that lisp session.

```
(let ((result +error+)
      (name (format nil "Safe call to ~s"
                    function)))
  (flet ((safe-run ()
           (when (with-simple-restart
                     (abort "Abandon call to ~a"
                            function)
                   (let ((*debugger-hook*
                          'debugger-hook))
                     (apply function arguments))
                   t)
             (setf result +success+))))
    (bt:join-thread (bt:make-thread #'safe-run
                                    :name name))
    result))
```

Function calls from the CLAUDE library into the application are referred to as *callbacks* (we're describing life from the application writer's perspective). Their uses range from notification of mouse events in a graphical toolkit to the handling of asynchronous errors. In lisp they are equivalent to invocations of `cffi:foreign-funcall-pointer`; LispWorks has a declarative macro `fli:define-foreign-funcallable` for handling this, a call to which is assembled and explicitly compiled first time a callback is invoked.

Callbacks are established and removed by calls to `claude_set_callbacks()`; they can be associated with a specific object and/or with none at all.

```
@ctypes.WINFUNCTYPE(ctypes.c_void_p,
                    ctypes.c_uint,
                    ctypes.c_uint)
def advise_condition(foreignRef, report):
    lib.raise_error(report)

set_callbacks(None, (("claude_advise_condition",
                      advise_condition),))
```

If the library wishes to invoke a callback on an object and none is currently established – either on that object or non-specifically – then nothing happens (i.e. it's not an error). It's assumed that the application programmer doesn't care to be informed about this particular event.

```
(invoke-callback
 '(:void
   (display (display :allow-null t))
   (address uint))
 display
 'advise-condition
 (when (and display (object-wrapper display))
   display)
 report)
```

6. ERROR HANDLING

The functions which CLAUDE exports all return a success/fail code (and so any "result" which functions wish to communicate must be via a pointer argument for the application to dereference).

```
enum {
  CLAUDE_RES_OK   =  0,        /* success */
  CLAUDE_RES_FAIL = -1         /* failure */
};
```

If the application spots a fail code, it can call `claude_last_error()` to find out more.

```
(defun-external (last-error :result-type ustring)
    ()
  (shift-last-error nil))

(defun debugger-hook (condition encapsulation)
  (declare (ignore encapsulation))
  (shift-last-error condition)
  (abort))

(defun shift-last-error (condition)
  (shiftf *last-error*
          (when condition
            (with-output-to-string (report)
              (report condition report)))))
```

There are methods on `report` for handling errors, warnings, and *complaints* (application pilot errors spotted by the library). The method on errors produces a full lisp backtrace; the library can choose whether to pass the whole thing on or to chop it (after the first line, say); if the application does anything intelligent with this string and its end users are sufficiently well trained then the backtrace could even be forwarded to the library writer for assistance. CLAUDE contains an elementary patch loader (which has been described elsewhere[5]) and so field patching of the lisp library is generally possible. Note that in LispWorks, at least, it is not possible to export new functions from the library without resaving the image, and so the introduction of additional `defun-external` forms is one thing that cannot be patched.

An asynchronous error can't be handled as above: the option of returning a `CLAUDE_RES_FAIL` isn't available when we're not actually inside a `defun-external` call. So it's advised that all threads which the library creates should install an error handler which invokes the `claude_advise_condition` callback, and that the application should handle this callback. When the application is notified of an asynchronous error, it can either deal with this directly, or make a synchronous call to `claude_raise_error()` and allow the usual error handlers to take over:

```
(defun-external raise-error ((report ustring))
  (shift-last-error report)
  (abort))
```

7. THE EXPORT PROCESS

After downloading the toolkit, the first step in exporting a library is to call `claude-setup:configure`, supplying the library's name and a working directory. For example, with a library called `wombat`, the following files and subdirectories are established:

[5]See `http://www.nicklevine.org/play/patching-made-easy.html`

```
wombat/
  build/save-dll.lisp
  claude/
    claude.asd
    src/...
  examples/C/...
  include/wombat.h
  manual/...
  pyWombat/...
  src/defsys.lisp
  wombat.asd
```

Even though no library is yet present we can easily try out the vanilla framework, by using the build script `save-dll.lisp` to save a DLL and then running some tests through the Python interface:

```
C:\home\wombat> lisp -init save-dll.lisp
  [...]                          ; saves wombat.dll
C:\home\wombat> python
Python 2.7.1
>>> from pyWombat import wombat
>>> wombat.Wombat()
<Wombat Wombat handle=0x200538b0>
>>> wombat.objects.communications_test()
True
>>>
```

Source files defining the library's external interface can later be added to the `src/` directory; these files along with dependencies for loading the library itself should be listed in the LispWorks `defsys` file (or, for those who'd rather, in the equivalent `.asd` file; the build script should then be set to load this instead). As each external class or function appears in lisp, its counterpart should be added to the `pyWombat` package. Whether or not the library is being targetted at Python application writers, Python is a reasonable test environment for it. Only once features are known to be working is it worth adding them to the C header and test files and finally to the manual.

The library, and in particular its CLAUDE layer, should be developed using lisp images saved to include a full lisp IDE (for instance, the LispWorks environment, or SLIME). CLAUDE supports this, via command-line arguments supplied when `save-dll.lisp` is loaded. With an IDE in place all the standard lisp development tools are immediately available. In particular it's straightforward to edit and recompile library code; depending on the framework being used to test the library, it may (Python) or clearly will not (C) be possible to interactively update the calling code without frequent halts. As already noted, one other cause for closing down and resaving the lisp image is addition of exported functions (new calls to `defun-external`).

8. ANOTHER EXAMPLE

Let's see what's involved in exporting another library: one not written by the present author. We choose Edi Weitz's ".NET layer for Common Lisp": *RDNZL*. We don't have space for the whole thing here, so we pick the following short example, for accessing web pages, out of RDNZL documentation and work on exporting that.

```
(import-types "System" "Net.WebClient")

(defun download-url (url)
  (let ((web-client (new "System.Net.WebClient")))
    [GetString (new "System.Text.ASCIIEncoding")
               [DownloadData web-client url]]))
```

Note the reader-macro on `#\[`, which case-preserves the next symbol and expands into a call to `rdnzl:invoke`. Two minor problems are presented, and we'll have to work around them:

- the function `rdnzl:new` returns an instance of the *structure* type `rdnzl:container`; and
- neither the argument types of `invoke`'s variable number of arguments nor that of its return value are preordained.

Let's deal with the RDNZL structure instances first, wrapping externalised objects around them[6]. In general `new` takes a variable number of arguments, but that feature won't be needed for this cut-down example.

```
(defclass-external container ()
  ((container :reader container-container
              :initarg :container)))

(defun contain (container)
  (make-instance 'container :container container))

(defun-external (new :result-type container)
  ((type ustring))
  (contain (rdnzl:new type)))
```

The next two functions are straightforward. Each RDNZL container has a *type-name* which is used in its `print-object` method, and it's worth exporting that; `rdnzl:import-types` takes a list of strings and doesn't return a value.

```
(defun-external import-types
  ((types (array ustring)))
  (apply 'rdnzl:import-types types))

(defun-external (type-name :result-type ustring)
  ((container object))
  (rdnzl::get-type-name
   (container-container container)))
```

The only real annoyance is `rdnzl:invoke`. Its first two arguments are a (RDNZL) container and a string, but after that all bets are off. A simple (if inelegant) solution for handling this flexibility is to export a number of functions, one for each signature that's needed. Here are the two we want. We have to translate between RDNZL's structure objects and our externalised CLOS instances, but otherwise the going is clear.

[6]CLAUDE does support the direct export of structure-objects, via its `defstruct-external` macro, but using that here would involve modifying RDNZL source, so we'll investigate this other option.

```
(defun-external (invoke-container-string
                 :result-type container)
  ((container object)
   (method ustring)
   (arg ustring))
  (contain
   (rdnzl:invoke (container-container container)
                 method arg)))

(defun-external (invoke-string-container
                 :result-type ustring)
  ((container object)
   (method ustring)
   (arg container))
  (rdnzl:invoke (container-container container)
                method
                (container-container arg)))
```

Now for the Python. We start by writing a stub for simplifying access to each of the external functions, for instance:

```
def new(type):
    return lib.rdnzl_new(type)
```

We can then implement a higher-level interface. Omitting here some of the details (string conversion functions; and a `__repr__` method which uses `rdnzl_type_name()` to generate printed representations), we have:

```
def import_types(types):
    c_strings = map(c_string, types)
    lib.import_types(objects.pack(c_strings))

class Container(objects.RdnzlObject):
    def __init__(self, type=None, handle=None):
        if handle is None:
            handle = lib.new(type)
        objects.RdnzlObject.init(self, handle)

    def invoke_container_string(self, method, arg):
        invoke = lib.invoke_container_string
        handle = invoke(self.box(),
                        c_string(method),
                        c_string(arg))
        return Container(handle=handle)
```

and similarly for `invoke_string_container()`.

Finally, we can try it all out, stepping our way through the `download-url` function with which we started:

```
>>> from pyRdnzl import rdnzl
>>> rdnzl.import_types(["System", "Net.WebClient"])
>>> c=rdnzl.Container("System.Net.WebClient")
>>> d=c.invoke_container_string("DownloadData",
      "http://nanook.agharta.de/")
>>> e=rdnzl.Container("System.Text.ASCIIEncoding")
>>> (c,d,e)
(<Rdnzl Container System.Net.WebClient
      handle=0x20053748>,
 <Rdnzl Container System.Byte[] handle=0x20053838>,
 <Rdnzl Container System.Text.ASCIIEncoding
      handle=0x200538b0>)
>>> print e.invoke_string_container("GetString", d)
<html>
  <head>
    <meta http-equiv="refresh"
          content="0;url=http://weitz.de/">
  </head>
  <body>
  </body>
</html>
>>> rdnzl.objects.remove_objects([c,d,e])
>>>
```

9. CONCLUDING REMARKS

This toolkit has only been tried on one lisp implementation and exercised by one individual. Indeed it came about as a quite unforseen consequence of publishing Chart and it's clearly yet to meet its full general potential.

Preparing a Common Lisp library for use in the outside world means abandoning some of CL's grace: no `with-interesting-state` macro can cross the divide; interfaces which depend on any number of keyword arguments will have to be rethought, as will multiple return values, closures, and undoubtedly various other features among CL's great strengths. If these restrictions are not excessive then libraries can be exported in exchange for very reasonable effort. CLAUDE is an abstraction layer whose task is to simplify that effort.

Our audience can be expanded.