Foreign Functions and Common Lisp

Harlan Sexton
Lucid, Inc., 707 Laurel St.
Menlo Park, CA 94025

Abstract

The language Common Lisp is a standard dialect of Lisp which has been implemented on a wide range of machines by a variety of commercial and academic groups. One serious flaw in the Common Lisp standard, at least to many Common Lisp users on "general-purpose" hardware,[1] is the lack of an defined foreign function interface, or FFI. The subject of this note is a discussion of FFI's for three different Common Lisp systems on engineering workstations and some thoughts on what foreign function interfaces ought to look like.

1 Introduction

To users of Common Lisp on engineering workstations, mini-computers, etc., the need for a foreign function interface (FFI) is usually quite clear. Without it, the user is cut off from the defined interface for the operating system (OS), and forced to get along with the very limited OS services provided by Common Lisp. The user is also unable to make use of non-Lisp code developed by co-workers or provided by system libraries. In short, a Common Lisp system without an FFI on such hardware is very poorly integrated into its environment, and its usefulness may be fatally impaired.

The purpose of this note is to look at some implementations for foreign function interfaces on engineering workstations, and to extrapolate from these what a standard Common Lisp FFI might be. We begin by mentioning informally some of the more obvious problems for Common Lisp implementations of foreign interfaces. We then describe the FFI's in some Common Lisp implementations, and conclude with a general description of an "ideal" FFI.

There are two basic problems in developing a Common Lisp FFI. The first problem is to figure out what is feasible. At one extreme, it should certainly be possible to invoke something like a Fortran routine for convolving two simple-vectors of type float and placing the result in a third such vector (assuming that there is a Fortran compiler for the machine in question). At the other extreme, requiring a Common Lisp implementation to be able to load the binary object files for another Lisp or a Prolog and then be able to call to and return from that other programming system as a foreign function seems a bit extreme. Also, support for call by value for arguments of type fixnum is

[1] "General-purpose" hardware is a marketing term that means computing hardware not specifically designed to run a specific language such as Lisp, rather than something like a combination computer, all-terrain vehicle, and washer-dryer.

quite reasonable. But, given that many Common Lisps implement fixnum objects as immediates, can we required an FFI to support call by reference[2] for such arguments? As a final example, it should certainly be possible for Lisp functions to somehow access foreign data-structures, but requiring the sequence functions to apply to foreign arrays is a bit excessive, so how much integration between Lisp and foreign data is enough?

The second basic problem is to decide how to express the connections between Common Lisp and foreign code. For example, on a UNIX[3] workstation, having a "foreign-type" system that is analogous the `struct` mechanism of the C programming language is natural and convenient for most OS needs, but such a design doesn't seem generally applicable to other languages.

2 Current Implementations

In this section we shall discuss the designs of the Common Lisp FFI's from Franz ExCL, DEC Vax Lisp, and Lucid CL. In order to keep the discussion simple, the examples are drawn only from implementations for engineering workstations from these companies. The choices here were motivated by three considerations, that the implementations be familiar, that they have generally useful features, and that the number of implementations discussed be small. The discussion is not intended to be and should not be considered a *product* comparison; no attempt has been made to compare features on concurrently available implementations for the same hardware.[4]

The features of these foreign interfaces fall, more or less naturally, into three rough categories: foreign-calling, foreign-loading, and foreign-types. We discuss how our representative Lisps implement each of these categories in turn.

FOREIGN-CALLING - By the **foreign-calling** system we mean the mechanism used by the FFI to allow Lisp code to execute foreign code that is present in the address space of the Lisp process, including being able to specify call-discipline, conversion, and type-checking for the functions arguments. There is generally little difference in the basic structure of the foreign-calling systems for these FFI's, primarily because this aspect of an FFI is constrained by the requirements of the calling-disciplines of the languages being called. All of the foreign-calling systems support type-checking of arguments and equivalent kinds of conversions of Lisp arguments to foreign formats.

---

[2]By call by reference I mean that the foreign and Lisp code share the same "value cells", not just, *e.g.*, the ability to pass fixnum arguments to Fortran functions; requiring support for call by reference for a Lisp data-type would make it essentially impossible to implement this type as an immediate type.

[3]UNIX is a trademark of AT&T Bell Laboratories.

[4]I am *much* more familiar with Lucid's FFI implementations that with any other vendors, since I have been primarily responsible for the latest version Lucid's of foreign interfaces. The features described in this note as being from Lucid CL are not all in any one version of a released product at the moment due to the usual problems of development and release schedules not staying perfectly synchronised. Unfortunately, my lack of similar information on the other implementations may have led me to leave out some of their new and interesting ideas.

The only notable difference between any of these three systems is that the Vax Lisp foreign-calling system supports the call by value return[5] discipline. We illustrate the two different foreign-calling styles by slightly artificial examples.[6]

The Franz/Lucid style of foreign-calling supports only call by value and call by reference[7], and use a single macro named (something like) `define-foreign-function`. A declaration for a foreign-function with two integer arguments and an integer return-value would look like:

```
(define-foreign-function foreign-proc ((arg1 :integer) (arg2 :integer))
  :return-type :integer)
```

Evaluation of this form would define an ordinary Lisp function named `FOREIGN-PROC` of two arguments returning one value. For example, this Lisp function might be associated with a Pascal function of the following form:

```
function foreign_proc (arg1: integer; arg2: integer): integer;
```

This Lisp function acts as a "stub" function that sets up the proper calling context for the Pascal code, calls this Pascal function, restores the Lisp context, and coerces the returned value to a Lisp format. This Lisp function may be used just as any other Lisp function.

When type-checking is enabled, which may be through a keyword argument to the `define-foreign-function` macro, through a declaration, or by some other means, this function will verify that the arguments passed it are integers.

The DEC style of foreign-calling supports both call by value and call by value return. It also uses a single macro referred to as `define-foreign-call`. A declaration for a foreign-function with two integer arguments, the first by value and the second by value return, and returning an integer value, would look like:

```
(define-foreign-call foreign-proc-2
    ((arg1 :integer :value) (arg2 :integer :value-return))
  :return-type :integer)
```

Rather than defining a Lisp function `foreign-proc-2`, it creates an internal function that is indexed by the symbol `FOREIGN-PROC-2`, and is used as an argument to the macro `call-foreign`. This Lisp code might be used to call the following Pascal function:

```
function foreign_proc_2 (arg1: integer; var arg2: integer): integer;
```

---

[5] **Call by value return, also known as call by value result, is the call-discipline in which the formal parameter in the calling code is identified with a local variable in the called procedure. This local variable is initialised by evaluating the argument corresponding to this formal parameter at call-time, and then on return from the procedure the argument (if it is a variable or other lvalue) is assigned the value of the local procedure variable. If the argument is not a variable, then this assignment does not occur, and depending on the compiler or language this may be an error or a reduce to an instance of call by value. The semantics differ somewhat from call by reference, but are similar, and it seems the best simulation of this discipline for calls between functions that use dissimilar formats for storing data.**

[6] **The forms we shall use to describe the two styles are not precisely the ones used by any of our three implementations, but they are representative.**

[7] **Call by reference is supported for many non-immediate types such as vectors; this is easy, since Lisp objects of these types are actually the (slightly modified) addresses of their contents.**

Vax Lisp simulates call by reference using call by value return by expanding the form `(call-foreign foreign-proc-2 var-1 var-2)` to:

```
(let ((.foreign-function. (get-foreign-function 'foreign-proc-2))
      .return-value. .temp-var1.)
  (multiple-value-bind (.return-value. .temp-var1.)
      (funcall .foreign-function. var-1 var-2))
  (setf var-2 .temp-var1.)
  .return-value.)
```

(We used variable names like `.temp-var1.` in this example instead of gensyms to make the "intent" of these variables clearer.) This style of foreign-calling has the advantage over the "functional" style that the natural form of call by value return can be used from Lisp; that is, the calling form has side-effects on its parameters.

FOREIGN-LOADING - By the **foreign-loading** system we mean the mechanism used by the FFI for "loading" foreign code and data objects into the Lisp address space and determining the addresses of these objects. There is also great similarity in the basic structure of the foreign-loading systems for these FFI's, since the function of this type of system is limited. The slight differences that do exist seem to be primarily due to some differences in implementation strategies.

The approach taken by Vax Lisp and Franz ExCL is to make use of the linker/loader provided by the operating system, and so these two implementations differ relatively little. In contrast, the Lucid foreign-loading system (on UNIX hardware) includes a loader written specifically for this purpose, and this allow the Lucid system to have complete access to the foreign "name-space". The primary user-visible difference between this system and the others is a consequence of this special loader, and it is that the Lucid system can dynamically redefine individual foreign symbols, *e.g.*, `foreign_function_1`, by "remembering" all references to `foreign_function_1` and linking in the new address for this symbol at each of these references. This means that the Lucid foreign loading system can dynamically redefine individual foreign functions.

Another difference between these implementations is in support for **callback**, which is to allow foreign functions to "call back" to Lisp code from a foreign context. Callback is not supported by Vax Lisp, while the other two implementations have very similar versions. We illustrate these versions with more slightly artificial examples. Both Franz ExCL and Lucid CL would declare a C callback "function" `CALLBACK-FUNCTION-1` of two arguments, a double-float and an integer, and which returned a double-float, using the macro `defun-callback` in essentially this manner:

```
(defun-callable (callback-function-1 :return-type :double-float)
    ((arg-1 :double-float) (arg-2 :integer))
  (+ arg-1 (float arg-2)))
```

This macro creates a Lisp function indexed with the name `CALLBACK-FUNCTION-1` that is suitable for being treated as a C function with arguments of type `double` and `int` and returning a value of type `double`. The only difference between these two implementations is how the connection between the Lisp function and the C code would be established.

If we suppose that the C function is denoted in C by the name `callback_function_1`, then the C code to use the Franz ExCL callback function looks something like:[8]

```
double (*callback_function_1)(); /* declare callback function pointer */
  ...
/* use system provided C_callback_init to initialize callback function
   pointer AFTER defun-callable macro is evaluated */
callback_function_1 = C_callback_init("callback_function_1");
  ...
/* call the initialized function */
temp_float_1 = (*callback_function_1)( temp_float_2, temp_int_3);
```

The C code to do the same thing in Lucid CL looks like:

```
/* declare callback function pointer; this function will not be defined
   before defun-callable macro is evaluated */
extern double callback_function_1();
  ...
/* call the function */
temp_float_1 = callback_function_1( temp_float_2, temp_int_3);
```

The reason for the difference between the two methods for calling back into Lisp from C is that the Lucid system is able to *introduce* new symbols into the foreign name-space while the Franz system does not support this feature.[9]

FOREIGN-TYPES - By foreign-types we mean how the FFI assigns some attributes to an area in memory (foreign-storage) that determine how the bits stored in this area are to be interpreted. The primary function of a foreign-type system is to define a correspondence between some Lisp types and low level foreign-types so that areas of "foreign-typed" memory may be accessed and set from Lisp. For example, suppose a foreign-type system defines a Lisp to C type correspondence of **float** to **double**. This would mean that a given 64 bits of foreign-storage that was assigned the foreign-type **double** would have defined Lisp access and set functions to convert these bits to and from Lisp objects of type **float**. It is normal for a foreign-type system to define correspondences between types that are naturally related, and all of the systems we describe here do considerably more than this.

Another important function of a foreign-type system is to allocate and manage storage used by foreign code. The areas of foreign-storage for most systems must protected from scanning by the Lisp system's garbage collector (GC), since most data for foreign-code may contain "arbitrary bits" confusing to the GC. Another part of this storage-management function is to make the assignment of a foreign-type to an area of foreign-storage; that is, to create typed foreign-storage. Sometimes this assignment is implicit, in that the only legal foreign-storage is that explicitly created by the system with a specific

[8]The initialisation method used by Frans ExCL is different from that described below; the method shown here seems easier to understand from a "picture" than the slightly more complex and flexible method actually used by ExCL.

[9]This capability can usually be implemented in Lisp systems that use OS provided loaders, although having a native Lisp loader makes it easier.

foreign-type, but in other systems foreign-storage may be assigned any foreign-type at any time.

To be somewhat pedantic then, we may define **foreign-storage** as any area in memory that may be legitimately used as data-storage by foreign code, and **typed foreign-storage** is any area that has been assigned a foreign-type and Lisp access and set functions.[10] The two major functions of a foreign-type system are:

(1) to define the correspondences between Lisp types and foreign-types, provide the conversion functions from Lisp types to foreign-types, and

(2) to allocate and maintain areas of foreign-storage, and to control the assignment of foreign-types to these areas.

As running example for the foreign-types section, consider the following C `struct` definitions:

```
/* C types examples */
typedef struct flat_str {
  long flat1;
  long flat2;} flat_struct;

typedef struct eight_str {char string[8];} eight_chars;

typedef struct compound_str {
  long        compound1;
  long        compound2;
  eight_chars compound3;
  flat_struct compound4;} compound_struct;
```

These examples are chosen to be simple and so that the alignments of the slots are unambiguous. In each of the Lisp systems we examine we'll look at how foreign-types corresponding to these C types would be defined.

The foreign-type system for Franz ExCL (for UNIX workstations) is the simplest of the three we are considering, and the easiest to use for many applications. It is modeled on the `struct` mechanism of the C programming language (the foreign-types it defines are referred to as **Cstructs**) and provides a straightforward means of translating UNIX OS system structure-types into Lisp.

The mechanism only provides correspondences between Lisp types and the standard low level types from C, it only allows the user to define foreign-types that are analogous to structs in C, and it is not possible to access "slots" in a foreign-structure unless the

[10]These definitions are somewhat simple-minded. First, the idea of foreign-storage as being limited to the area where foreign code can legitimately read or write is somewhat limited if we want to use foreign code to examine and modify Lisp objects. Second, one can imagine assigning a foreign-type to areas of memory where it is only possible to read the bits (such as the `text` area in a UNIX process), or to areas of memory where it is only meaningful to write (special bus addresses for output devices), so the definition of typed foreign-storage is somewhat cramped, as well.

slot's type is one of these low level types. The system will, however, allow the user to create compound types and allocate storage for them that can be modified by foreign functions; there is, apparently, no way provided by the system for Lisp code to examine or modify compound structures. In this system the only legal foreign-storage is explicitly created.

The C example types could be readily defined as follows:

```
(defcstruct flat-struct (flat1 :long) (flat2 :long))
(defcstruct eight-chars (string 8 :char))
(defcstruct compound-struct
  (compound1 :long)
  (compound2 :long)
  (compound3 eight-chars)
  (compound4 flat-struct))
```

In these examples, both slots of the `flat-struct`, all of the entries of `eight-chars`, and the first two slots of `compound-struct` may be accessed and set from Lisp. The last two slots of `compound-struct` are not accessible from Lisp. In the terms we defined above, all of these C structures reside in foreign-storage, but the area comprised of the last two "compound" slots are not typed foreign-storage.[11] Foreign-storage created using the function `make-compound-struct` is actually part of a Lisp object of type **(simple-array (unsigned-byte 32) *)**, as is all foreign-storage.

The foreign-type system for DEC Vax Lisp provides the tightest integration with the Common Lisp type system of any of the foreign-type systems we are discussing here. It is modeled on the lisp **defstruct** system (it creates types referred to as **alien structures**), and the types defined by this mechanism automatically become Common Lisp types. The mechanism also automatically defines related functions analogous to those defined by **defstruct**.

It is the most verbose to use, since it is up to the user to specify the details of the layout of the foreign-type's components, but this allows almost complete control over the structure of a foreign-type. The mechanism only provides correspondences between Lisp types and a set of generic low level types, and it is not possible to define compound types. In this foreign-type system the only legal foreign-storage is that explicitly created by the system.

In Vax Lisp the C example types could be defined by:

```
(define-alien-structure flat-struct
  (flat1 :signed-integer 0 4)
  (flat2 :signed-integer 4 8))
(define-alien-structure eight-chars
  (string :string 0 8))
(define-alien-structure compound-struct
  (compound1 :signed-integer 0 4)
  (compound2 :signed-integer 4 8)
  (compound3 :text 8 16)
  (compound4 :text 16 24))
```

[11] Obviously, if it were important to access the sub-slots of compound4, it would be easy to define a flat version of `compound-struct` by using the ability to make structure slots into arrays.

In these examples the foreign-storage and typed foreign-storage areas are identical to the examples above.[12] Foreign-storage created by the function `make-compound-struct` is identified with a Lisp object of Lisp type **compound-struct**, which is a subtype of type **alien-structure**, as is all foreign-storage.

The foreign-type system for Lucid CL is the most general of the foreign-type systems we are discussing here. It is also modeled on the lisp **defstruct** system, but foreign-types do not become Common Lisp types. The mechanism also automatically defines creation, access and modification functions analogous to those defined by **defstruct**, but no others.

It is slightly less verbose to use than Vax Lisp even though the user is allowed to specify the details of the layout of the foreign-type's components, since there are "reasonable" default values for most things. It is possible to specify not only the layout of the components of foreign-storage of a defined type, but also the address alignment requirements of the foreign-storage for that type. This is needed for hardware implementations that, for example, require loads and stores of double-floats to be double-word aligned.

The mechanism has built in correspondences between Lisp types and a set of generic foreign types, and it is possible to define arbitrary compound foreign-types (except that array types must have their dimensions completely specified). In this foreign-type system any part of the address-space may be treated as foreign-storage; this correspondence is implemented through Lisp objects of type **foreign-pointer**, which have as attributes an address and a foreign-type. The Lisp inspector also understands objects of type **foreign-pointer**.

In Lucid CL the C example types could be defined by:

```
(def-foreign-struct flat-struct
  (flat1 :type :signed-32bit)
  (flat2 :type :signed-32bit))
(def-foreign-struct eight-chars
  (string :type (:array :character (8))))
(def-foreign-struct compound-struct
  (compound1 :type :signed-32bit)
  (compound2 :type :signed-32bit)
  (compound3 :type eight-chars)
  (compound4 :type flat-struct))
```

In these examples all of the foreign-storage is also typed foreign-storage. Foreign-storage created by the function `make-compound-struct` is identified with a Lisp object of Lisp type **foreign-pointer**, as is all foreign-storage. A foreign-pointer so created would have foreign-type `(:POINTER COMPOUND-STRUCT)`, and applying the function `compound--struct-compound3` to this pointer would return another foreign-pointer of type `(:POINTER EIGHT-CHARS)`, and this object has slots that are immediate types. Thus, the Lucid system is able to define a correspondence between any foreign-type and some Lisp type by the expedient of taking all of the "hard" foreign-types and assigning them to a new Lisp type called **foreign-pointer**.

[12] The same remarks about flattening the compound structure definition apply here as in the Frans ExCL example.

SUMMARY - To review, we see the FFI implementations that we considered tend to share many features. The category for which there were the most significant differences was foreign-types, which is probably a result of the wide gulf between the type systems for statically typed languages such as C and (dynamically typed) Lisp. The variations in the foreign-loading category are mostly due to implementation decisions, since the requirements for such a system are quite straightforward. Finally, the almost complete coherence between the different versions of foreign-calling systems described here is due to the high degree of standardization in supported languages such as Pascal, C and Fortran.

3 Thoughts on a Standard FFI

The categories described above, foreign-calling, foreign-loading, and foreign-types, seem a sufficiently natural way to break down the requirements for a foreign function interface that we shall use them in describing our thoughts on what an FFI ought to include. We discuss the foreign-loading and foreign-call categories first, and since the requirements for these are fairly simple, we shall be brief. We then give a fairly lengthy discussion of what a foreign-type system ought to contain. In each of these cases the discussion stops far short of providing a complete list of user functions; the goal is to provide an outline for a Common Lisp FFI, not a specification. In particular, the precise names and syntax of macros and functions proposed here is not central to the discussion, and would be spelled out at length by such a specification.[13]

FOREIGN-CALLING - The foreign-calling mechanism proposed here is a hybrid of the two different styles we have seen above; we shall describe it in terms of two new macros `defun-foreign` and `defun-foreign-callback`. (The second macro will require some support from the foreign-loading mechanism, and this will be discussed below.)

The `defun-foreign` macro takes two required arguments, NAME-AND-OPTIONS and ARGLIST, with an optional documentation string following the ARGLIST argument. The NAME-AND-OPTIONS argument specifies the function name and the values of all relevant additional information such as which foreign language is being called, the type of the function's return-value, and the function's foreign name. The ARGLIST is a list of argument-specifiers and the ampersand keywords `&optional`, `&rest`, and `&key`. (Use of both `&rest` and `&key` in the same arglist is not allowed.) An argument-specifier contains both a formal parameter (to be used as part of the Lisp function's arglist) and keywords specifying foreign-type and call-discipline. Foreign-types such as `:double-float` or `:signed-32bit` correspond to Lisp types such as float and integer, respectively, and both call by value or call by value return must be legal for arguments of these types. Call by reference for arguments of foreign-types such as `:string` and `:array` must also be supported, but support for call by value for such arguments is implementation-dependent. The macro `defun-foreign` defines an ordinary Lisp function, as in the Franz/Lucid example above. Call by value return must be supported by having foreign functions with

[13]This is not to minimise the importance of the names and syntax; the formal differences between the various implementations of these foreign interfaces are more than enough to infuriate users, and a well-specified standard would be very welcome. This is just an inappropriate forum for that level of detail.

:value-return arguments return multiple values. Specifically, for a foreign function with N arguments passed by value return, the foreign function returns $N+1$ values. The first value is the foreign code's return value, and the rest of the values are the "terminating" values for the value return arguments (in the same left to right order).

The defun-foreign-callback macro takes three required arguments, NAME-AND-OPTIONS and ARGLIST and BODY, with optional documentation string and declarations following the ARGLIST argument. The NAME-AND-OPTIONS and ARGLIST are similar to those for defun-foreign, except that no keywords are permitted in the ARGLIST. The BODY form is evaluated in an implicit progn, and this value is returned to a foreign context as the value of the function.

FOREIGN-LOADING - The only feature not consistently supported by Common Lisp systems to be included here is the ability to dynamically insert *new* foreign-symbols into the foreign name-space. This feature would be used to support the more straightforward callback style of Lucid CL, described above.

It will be up to the implementation whether or not dynamic redefinition of individual foreign-functions is supported. (Requiring this feature could force implementors to write their own linker/loader, and this is infeasible on systems with undocumented object-file formats.)

FOREIGN-TYPES - The foreign-type mechanism proposed here is a hybrid of the three styles we have seen above with some additional features. The mechanism is defined in terms of three underlying ideas: a new Lisp type call foreign-storage, "primitive" foreign-types, and a macro for defining foreign structures named defstruct-foreign.

Sizes of foreign-types discussed below will be those which would be natural for machines that are byte-addressable with 8bit bytes and 32bit words, but this is just for purposes of exposition. The foreign-type system must support the native addressing-units for whatever machine it resides on.

In this foreign-type system, all foreign-storage looks to Lisp like objects of type foreign-storage, and all foreign-types are Lisp subtypes of this type; this is analogous to Vax Lisp alien-structures. The attributes of an object of type foreign-storage are its address, size in bytes, and its type. The first two would be returned by the function foreign-storage-address and foreign-storage-size, respectively, and the third by the Common Lisp function type-of. If the type of a foreign-storage object is foreign-storage and not some proper subtype of this type, its size would be NIL.[14] Instances of storage of a given foreign-type (other than foreign-storage) would have a well-defined, positive size; variable-size foreign-storage, if needed, would be supported by mechanisms distinct from those discussed here.[15]

All foreign-types have the following attributes: :alignment comprised of :modulus and :remainder, :size, and print method. Saying that a foreign-type ft has :modulus

[14]Objects of type foreign-storage correspond to a normal usage in C of the type-cast (char *). That is, they correspond to areas of foreign-storage where the type is indefinite, but the address is not.

[15]Some support for variable-size foreign-types is a good idea for such applications as databases, but incorporating variable sizes into the design described here would complicate matters substantially, and the need for such generality is unclear.

M and :remainder R means that any foreign-storage of type ft with address A is required to satisfy $A = Mx + R$.

A "primitive" foreign-type is, roughly, one that is the foreign equivalent of Lisp type. For example, a foreign-type of :double-float can be regarded as being 64 bits of foreign-storage that contains the foreign equivalent of a Lisp object of type float. More precisely, a primitive foreign-type is a foreign-type for which direct conversions to and from Lisp objects of a given "non-foreign" type is defined. The system must provide at least primitive foreign-types corresponding to signed and unsigned integers of sizes 8, 16, and 32 bits, single and double floats, characters, strings of specified length, and bit-fields of sizes 0 to 32 bits. The system also must provide exported versions of the conversion methods for these, so that users can build their own primitive foreign-types. Finally, the system must provide the function define-primitive-foreign-type, for making user-defined primitive foreign-types. Its required arguments are NAME, INPUT-METHOD, OUTPUT-METHOD, and SIZE. It also takes keyword arguments for, :modulus, :remainder, and :print-method.

The macro defstruct-foreign, which is very similar to the Common Lisp defstruct[16] macro, is used to define foreign structures. The primary differences between the structure options for the macros defstruct and defstruct-foreign are that, for the "foreign" version:

:type The :type structure option specifies what kind of Lisp storage is allocated for this type of foreign-storage by the default constructor function. It may be at least one of :dynamic, meaning that the foreign-storage is susceptible to being move or reclaimed by the GC, and :static meaning that the foreign-storage may not be reclaimed or moved.

:alignment There is a structure option named :alignment that allows the user to specify alignment requirements for foreign-storage of this type. The default values are system-dependent.

:include The structure option :include is not allowed.

:named The structure option :named is not allowed.

:initial-offset The structure option :initial-offset specifies a minimum number of bytes to skip over before allocating storage for the structure slots.

The slot options for defstruct-foreign would differ from those of defstruct as well. The primary differences for these options are:

:type The :type slot option is *required.* It must be an already defined foreign type, although recursive types may be defined using types of :pointer.

:offset The precise offset of the slot in bytes may be specified.

:union A slot may be defined to be an element of a union of some previous slot. This is analogous to union in C structure types or case in Pascal record types.

[16]See *Common Lisp, The Language*, by Guy Steele, Chapter 19.

The details of how `defstruct-foreign` constructs the layout of a foreign structure are fairly tedious. Generally, the slots in the structure are layed out in the order specified in the body of the macro, as compactly as possible while satisfying alignment requirements. Slots with specified offsets are layed out independently of all the other slots. Component slots of a `:union` are treated as being part of an "amalgamated" type whose size is the maximum size of each component type and whose alignment is the smallest alignment satisfying all the component constraints[17].

The combination of user-definable primitive foreign-types and user definable foreign structures will give most of the functionality of a **def-C-struct**, at least for compilers that lay out structures "from left to right" using simple alignment criteria. Trying to handle the most general case is hopeless, and individual compilers of interest may be modeled by a macro defined in terms of `defstruct-foreign` that sets the `:offset` value for each slot.

4 Conclusion

In this note we have looked at some basic problems faced by Common Lisp FFI's and described the foreign interfaces in some common implementations. We closed with a fairly conservative set of suggested features to include in a standard FFI.

The basic principles used in selecting these features were:

(a) Consider only features that are feasible within the current "standard" implementations of Common Lisp.

(b) Include features that are of proven usefulness, or that make the overall design more coherent.

(c) Defer on features whose correct design is not yet clear. (Premature specification of a feature might restrict future growth of the FFI in some unfortunate way.)

(d) Defer on ideas of unproven value, or those that may require an inordinate amount of effort to implement.

It is worth applying these principles to a few potential FFI features that have not been included here. For example, there is no suggested means of providing a way for users to provide support for new languages. This idea has been rejected on the grounds of (d), with some concern that (a) might apply, too. There are also no proposals for support for foreign structure-types that are not constructed "left to right" or for making user-defined primitive types be automatically integrated into the foreign-calling mechanism.[18] These are definitely type (d) features, too.

[17] That is, the amalgamated type's alignment is given by the smallest solution to the simultaneous congruences specified by the alignment of each component type. If a solution does not exist, then it is an error.

[18] That is, defining a new primitive type called `:signed-64bit` would not automatically make it legal to declare an integer argument to a foreign function to be of this type.

There is no support proposed for variable sized foreign-types. This is a type (c) feature. It is pretty likely that some such support is needed, but the precise requirements are not yet clear. And finally, there is no inherited "analogy" between compound types; that is, even though `:signed-16bit` and (signed-byte 16) are regarded as being analogs, the same does not apply to arrays of these respective types. This is also type (c), since the only clear use for such support is to allow the use of Common Lisp sequence functions on Lisp arrays displaced to foreign-arrays, and it is unclear that this is more than a minor convenience.

Common Lisp as currently defined has no requirements for a foreign function interface, but almost all implementations provide some sort of FFI; in fact, such interfaces may be critical for the usefulness of a Common Lisp in "general-purpose" environments. Since some of the most widely used Common Lisps have evolved functionally quite similar FFI's, choosing a generally acceptable set of features for a foreign function interface should be straightforward.