EuLisp Foreign Function Interface

Initial Proposal

December 13, 1993

Contents

1

Foreign Function Interface

1.1 Introduction

This document presents an initial proposal for a EuLisp foreign function interface for calling C functions from a EuLisp program and calling EuLisp functions from a C program.

This proposal is derived from part of Talk's foreign function interface, so sections of it may be inappropriate for EuLisp. In particular, the `DEFINTERN` mechanism may not be usable with EuLisp because of a lack of computed access to EuLisp functions by name. Nevertheless, it is presented here as food for discussion.

Also, this proposal allows the definition of foreign functions which return non-Lisp objects, called *pointers*. If this is considered undesirable by the committee, then the relevent sections can be cut, and the `defextern` converter `ptr` can be eliminated in favor of address objects, which are basically boxed foreign pointers. Introducing pointers probably requires a conservative GC, and involves a few extra words in the specifications concerning functions which accept objects and those which accept pointers. More detail available upon request.

Finally, you should realize that the part of the FFI described here is by far the easiest part. All of the hard problems have to do with things like dynamic loading, linking foreign objects with EuLisp modules, OS dependencies, error handling, clean interfaces to existing libraries, etc. In this case, more so than in all other subsystems of the language, the interface is much less important than the implementation. Many important decisions will inevitably be left to implementations, and it is impossible using just the interface described here to write a complete mixed language application[1]. Nevertheless, it seems to us that by standardizing even on this level of interface we can do a great deal of good for users and for EuLisp's image.

1.2 Defining Foreign Functions

`(defextern` name (arg-converter*) result-converter `[`external-name`])` [*defining form*]

Summary Associates a EuLisp function with an external procedure.

[1] For instance, how do you link a C object file with a EuLisp module or runtime?

Converter	EuLisp *type*	C *type*	*Conversion*
`ptr`	ptr	`void*`	none
`<int>`	`<int>`	`long`	drop tag bits
`<string>`	`<string>`	`char*`	none
`<address>`	`<address>`	`void*`	the value of the address is passed
`boolean`	ptr	`long`	none; () is 0 in C.
`<file-stream>`	`<file-stream>`	`int`	the file descriptor for the stream is passed.
`<float>`	`<float>`	`double`	value of the float
`<vector>`	`<vector>`	`void**`	none

Figure 1.1: Argument converters for **`defextern`**.

Arguments	name	`<symbol>`
	arg-converter	`<symbol>`
	result-converter	`<symbol>`
	external-name	`<string>`

This defining form defines an EuLisp function which calls an external language procedure, doing any necessary data conversions. The argument *name* is the name of the EuLisp function to be defined. The optional string *external-name* names the external procedure to be called. This name need not be prefixed by an underscore. If the optional argument *external-name* is not supplied, the name of the function *name* is taken to be the name of the external procedure. The list of external converters *arg-converters* names the converter of each argument to the function. No lambda list keywords are allowed; only functions with a fixed number of arguments are supported. Finally, *result-converter* names the converter of the return value.

There are two kinds of converters: argument converters and result converters. Argument converters are used to convert EuLisp objects into the corresponding C pointer accepted by the C function. In many cases, nothing needs to be done. If the argument converter corresponds to a EuLisp type, a type test will be performed at development level `debug` to ensure that the arguments are correctly typed.

Result converters are used to convert the result of the C function into a corresponding EuLisp object or pointer. In many cases, this conversion will allocate a EuLisp object because the C representation is not rich enough to support EuLisp objects directly.

On some ports, a **`defextern`** form cannot be interpreted due to constraints on dynamic access to the C symbol table.

Figure 1.2 shows a table describing the matching type and the conversion performed for each legal argument converter. Figure 1.2 shows a table describing the matching converter and the conversion performed for each result interface specification. Conversions which allocate EuLisp objects are noted.

For example, the following external procedure is defined in C in the file `ctest.c`:

```
/* This function prints its arguments, exchanges the
   first and second elements of the vector, and returns
   the product of the second and third */
double c_test (char *strg, double nf, long ni, void *vect)
{
  void *p;
  static double result; /* MUST be static to return &result */

  printf("The string is %s.\n", strg);
  printf("The float is %e.\n", nf);
  printf("The integer is %d.\n", ni);
  printf("The vector contains: vect[0]=%p, vect[1]=%p\n", vect[0], vect[1]);
```

Converter	*C type*	EuLisp *type*	*Conversion*
`ptr`	`void*`	ptr	none
`<int>`	`long`	`<int>`	add tag bits
`<string>`	`char*`	`<string>`	allocates a EuLisp string containing the same characters. An error is signaled if 0 is passed.
`<address>`	`void*`	`<address>`	allocates an address holding the result
`boolean`	`long`	ptr	`()` if the result is 0, `t` otherwise
`<float>`	`double`	`<float>`	allocates a EuLisp float containing the same `double`

Figure 1.2: Return value converters for **`defextern`**.

```
  p = vect[0];
  vect[0] = vect[1];
  vect[1] = p;

  result = nf*ni;

  return result;
}
```

Here is the corresponding EuLisp source code:

```
(defextern c-test (<string> <float> <int> ptr) <float> "c_test")
```

`long` `EUL_INT`(*i*) [*C macro*]

Arguments `long` *i*;

`long` `EUL_LONG`(*i*) [*C macro*]

Arguments `long` *i*;

The `EUL_INT` and `EUL_LONG` macros convert signed ints between the C and EuLisp fixed precision integer representations.

`EUL_DEFINTERN0-19`(*cfunction-name*, *lisp-name*) [*C macro*]

Arguments *cfunction-name*;
`char` **lisp-name*;

These C macros define a C function *cfunction* which accepts *i* (typed implicitly as `void*`) arguments, calls the EuLisp exported function *lisp_name*, and returns the value of that call. The string *lisp_name* is the printed representation of an EuLisp symbol naming an exported function.

In this example, we implement the well-known Fibonacci function using mutually recursive EuLisp and C functions. This example contains one EuLisp module which is linked with the C code compiled from `cfib.c`.

Here is `cfib.c`:

```
#include <eulisp.h>

static EUL_DEFINTERN1(lisp_integer_fib,"lisp-fib")

/* Lisp fib with conversions done */
```

```
long lisp_fib (long n)
{
  return EUL_LONG(lisp_integer_fib(EUL_INT(n)));
}

/* C implementation of the well-known Fibonacci function */
int c_fib (n)
int n;
{
  if (n<2)
     return 1;
  else
     return (lisp_fib(n-1) + lisp_fib(n-2));
}
```

Here is the EuLisp source code file `lispfib.t`:

```
(defextern c-fib (<int>) <int> "c_fib")

(defun lisp-fib (n)
  ;; EuLisp implementation of the well-known Fibonacci function.
  (if (< n 2)
      1
    ;; Recursive calls go to C.
    (+ (c-fib (- n 1))
       (c-fib (- n 2)))))
```

1.3 Addresses

`<address>` `<object>` [*class*]

Summary The class of addresses.

This is the class of address structures, which are generally used to box foreign structure pointers. Index arithmetic using the generic arithmetic functions can be performed on addresses. When using arithmetic on addresses, integer increment or difference values are in terms of bytes.

(`addressp` *ptr*) [*function*]

Summary Determines if an object is an address.

Arguments *ptr* `ptr`

Result `boolean`

This function returns `t` if the object *ptr* is an address structure and `()` otherwise.

```
? (addressp #@5)
= t
```

`#@` `hexnum` [*literal syntax*]

Summary Reads an address in hexadecimal format.

This literal syntax reads in the next token as a base 16 address. The integer can be up to the word size of the machine.

```
? #@66da10
= #@0066DA10
```

`(ptr-address` *pointer*`)` [*function*]

Summary Boxes a non-object pointer into an address.

Arguments *pointer* `ptr`

Result `<address>`

Allocation `<address>` 1

This function builds and returns an address which points to *pointer*.

```
? (ptr-address ())
= #@00000000
? (ptr-address 'foo)
= #@004D6710
```

See also `objectp` (5)

`(address-ptr` *address*`)` [*function*]

`(setf (address-ptr` *address*`)` *new-value*`)` [*setf form*]

Summary Dereferences an address containing a pointer.

Arguments *address* `<address>`

Result `ptr`

This function returns the pointer contained in the address *address*. You should be extremely careful with the result, since it may not be a valid EuLisp object and may thus cause unexpected errors, such as **`segmentation violation`**, when passed to certain EuLisp functions. When modified, it changes the pointer contained in *address* to be the new value, which can be any pointer.

```
? (eq () (address-ptr #@00000000))
= t
? (address-ptr #@00000013)
= 4
```

See also `objectp` (5)

1.4 Pointers

`(objectp` *pointer*`)` [*function*]

Summary Determines if a pointer is a EuLisp object.

Arguments *pointer* `ptr`

Result `boolean`

This function returns `t` if *pointer* is a EuLisp object, and `()` otherwise.

```
? (objectp 'foo)
= t
? (objectp 3)
= t
? (objectp (address-ptr #@1))
= ()
```

(ptr-string *pointer*) [*function*]

Summary Copies memory contents into a fresh string.

Arguments *pointer* `ptr`

Result `<string>`

Allocation `<string>` 1

This function creates and returns a string containing the same sequence of bytes as found at location *pointer* up to the first null byte. This function is useful when dealing with C `char*`, since it creates an equivalent EuLisp string.

(ptr-float *pointer*) [*function*]

Summary Copies memory contents into a fresh float.

Arguments *pointer* `ptr`

Result `<float>`

Allocation `<float>` 1

This function creates and returns a float containing the same 8-byte floating point value (a `double` in C) which is at location *where*. This function is useful when dealing with C `double*`, since it creates an equivalent EuLisp float.

```
? (ptr-float 3.14)
= 3.14
```