Extensions to the Franz, Inc.'s Allegro Common Lisp Foreign Function Interface

John Keane
Department of Computer Science
Rutgers University
New Brunswick, NJ
keane@cs.rutgers.edu

January, 1996

Abstract

As provided by Franz, Inc., the foreign function interface of Allegro Common Lisp has a number of limitations. This paper describes extensions to the interface that facilitate the inclusion of C and Fortran code into Common Lisp systems. In particular, these extensions make it easy to utilize libraries of numerical subroutines (such as those from Numerical Recipes in C) from within ACL, including those routines that take functions as arguments. A mechanism for creating Lisp-like dynamic runtime “closures” for C routines is also described.

Acknowledgments

The research presented in this document is supported in part by NASA grants NCC-2-802 and NAG2-817. This research is also part of the Rutgers-based HPCD (Hypercomputing and Design) project supported by the Advanced Research Projects Agency of the Department of Defense through contract ARPA-DABT 63-93-C-0064.
1 Motivation

Allegro Common Lisp, from Franz, Inc., provides a means for Lisp routines to utilize subroutines written in C and Fortran, called the foreign function interface. [Franz, 1993]. Though adequate for most purposes, the foreign function interface is not as easy to use as might be desired, and does not support certain kinds of interface with foreign code as straightforwardly as we might wish. In particular, we found it difficult to define and use foreign code that takes functions as arguments, such as a Newton-Raphson root finder, because of the way in which C-callable Lisp functions are defined.

Additionally, ACL (version 4.2 and later) utilizes two different mechanisms for loading foreign code into the Lisp image, one for SVR4 and Posix operating systems, and one for BSD operating systems. Under SVR4 systems, all foreign code must be first linked into shared-object (.so) libraries. While this provides significant improvement in the speed of loading foreign code, and reduces the size of the Lisp image, it requires that the shared-object libraries be defined and built ahead of time. The two load mechanisms place an additional burden on Lisp user programs to be aware of the system under which they are running, and act accordingly, if they are to be portable.

In writing these extensions, we wished to make using foreign code as transparent to the user as possible. We built on top of the solid foundation of the existing foreign function interface. These extensions do not replace functionality in ACL, they complement it.

2 The Foreign Function Extensions Calling Model

Figure 1 illustrates the model that is used for all foreign function interactions defined through the extensions. In the example, the Lisp function foo wishes to call the C routine bar, which wishes to call the Lisp function baz. These calls are not made directly, but are made through “helper” routines that are automatically generated when the functions are defined in Lisp. These helper routines add a small amount of additional overhead to each foreign function call, but provide important functionality in providing parameter management.

In Figure 1 the Lisp function foo calls the Lisp stand-in function bar.
This function takes the arguments of the C routine `bar`, which it then re-packages into a parameter control block, and calls the C front-end routine `bar_c_fe`. This routine takes the parameter control block, and calls the target C routine `bar` after recovering the parameters.

The C routine `bar` calls the C stand-in routine `baz`. This function takes the arguments of the Lisp function `baz`, which it then re-packages into a parameter control block, and calls the Lisp front-end routine `baz-lisp-fe`. This function takes the parameter control block, and calls the target Lisp function `baz` after recovering the parameters.

3 Defining Foreign Functions

Two new functions have been added to define foreign functions. These are:
Both functions have a required argument, the `header`. This specifies the calling parameters and return values of the function, using C data types. The header can be automatically created by parsing the C header for the function.\(^1\)

The `typecheck` argument specifies whether argument type checking code is to be included in the interface functions. The `write-files` argument specifies whether the generated interface code is to be written out. The `compile` argument specifies whether the interface routines are to be compiled. The last two arguments give the target directories for the interface code.

These routines differ from the ACL `ff:defforeign` function in that they do not need to be invoked every time the foreign code is included into a system. If the interface routines are written out and compiled, only the load function below need be used.

### 4 Defining C-callable Lisp Functions

\[
\text{ffx:define-external-callable (header &key (callback nil) (write-files t) (compile nil) (interface-lisp-dir "./")) (interface-c-dir "./"))}
\]

\(^1\)C headers can be generated for Fortran routines by using the widely-available `f2c` Fortran to C compiler.
The arguments to this routine are much the same as for the previous two routines, except that the function specified in the header is a Lisp function. If the callback argument is t, the interface is generated as a callback, which will allow C routines to call dynamic lambda expressions.

5 Loading Foreign Functions

\texttt{ffx:load-external-system} (&key routines
\begin{verbatim}
  files
  libraries
  (interface-lisp-dir "./"
  (interface-c-dir "./"
  solib-name
  (solib-dir "./"
  (rebuild-solib t))
\end{verbatim}

The routines argument is a list of strings containing the names of routines defined by \texttt{ffx:define-C-routine}, \texttt{ffx:define-Fortran-routine}, or \texttt{ffx:define-external-callable}. The files argument is a list of strings containing the names of object files to load. The libraries argument is a list of strings containing the names of system libraries (or local libraries) to load with the foreign code. Under SVR4 systems, the user must specify solib-name, the name of the shared-object (.so) lib. If \texttt{rebuild-solib} is T the .so lib will be re-built before loading.

This function handles the entire foreign load process, under both SVR4 and BSD systems. It issues the required \texttt{ff:defforeign} calls, and invokes the foreign function loader. It also takes care of registering the externally-callable Lisp functions, and passing the call pointers to the associated C interface functions. If the interface routines have been previously generated and compiled, this function is the only one required to load external code into the Lisp system.
6 Passing Lisp Functions to Foreign Functions

Many C routines take a C function pointer as an argument. If the user wishes to pass a named Lisp routine to a C function, she may use the function `ffx:get-C-function-pointer`, which takes the name of a Lisp function previously defined with `ffx:define-external-callable` and returns the pointer to the C stand-in interface routine for that function.

Suppose that the user wishes to pass a dynamically created lambda expression instead, such as a function closure. Because of their dynamic nature, it would be highly inefficient to have to generate a separate unique interface for each dynamic closure. Instead, the user may use the function `ffx:bind-lisp-callback`. This function takes as arguments a lambda expression, and the name of a callback previously defined with `ffx:define-external-callable` taking the same formals as the lambda. A C function pointer is returned that, when dereferenced by a C routine, will cause the closure to be called through the callback interface code. Any number of dynamically generated closures can be bound to a single callback, and a unique C function pointer will be created for each.

The function `ffx:bind-lisp-callback` works by creating a tiny machine-language program called a *function vector*. Each function vector is about 64 bytes long, and is malloced in Lisp foreign function space. The function vector contains the address of the C stand-in interface routine for the callback, the address of a designated storage location in Lisp foreign function space, and a unique Lisp index to the closure. The address of the function vector is what is returned by `ffx:bind-lisp-callback`. When control is transferred to the function vector, it stores the closure index into the designated storage location, and transfers control unconditionally to the C stand-in interface routine. Since the stack is not touched, the C stand-in interface routine finds all the arguments it is expecting, just as if it had been called directly by the C routine. It then builds the parameter block, and calls the Lisp front-end for the callback. The front-end function looks at the designated storage location, finds the index to the closure, and calls it.

We note in passing that this mechanism for function vectors can also be used within systems written entirely in C to give the effect of Lisp function closures, without the need to pass additional information through global
variables. This allows many problems to be avoided that occur when global variables are overwritten during nested function calls.

7 Example

This is an example of the foreign function extensions applied to the function newt, from *Numerical Recipes in C* [Press et al., 1986]. This routine takes two function pointers as arguments, the function to solve for the roots of, and a function to compute the Jacobian of the first function. Here is the definition of newt and the callbacks for the Lisp routines:

;;; This routine is part of the Numerical Recipes in C library

(setf *newt-header* (make-instance 'header
   :routine-name "newt"
   :return-type 'void
   :c-arglist '((|x| double array)
                  (|n| int none)
                  (|check| int pointer)
                  (|fixits| int none)
                  (|vecfunc| void function)
                  (|jacfunc| void function))
   :c-header
   '(("void newt(double x[], int n,
      int *check, int fixits,
      void (*vecfunc)
      (int, double [], double []),
      void (*jacfunc)
      (int, double [], double [],
      double **,
      void (*vecfunc)
      (int, double [], double []))
    )))

(ffx:define-c-routine *newt-header* :typecheck *lisp-c-interface-typecheck*
   :interface-lisp-dir *lisp-c-interface-lisp-dir*
(setf *newt_vecfunc-header* (make-instance 'header
  :routine-name "newt_vecfunc"
  :return-type 'void
  :c-arglist '((|n| int none)
                (|x| double array)
                (|f| double array))
  :c-header
  '("void newt_vecfunc(int n, double x[],
     double f[])")))

(ffx:define-external-callable *newt_vecfunc-header* :callback t
  :interface-lisp-dir *lisp-c-interface-lisp-dir*
  :interface-c-dir *lisp-c-interface-c-dir*
  :write-files t
  :compile t)

(setf *newt_jacfunc-header* (make-instance 'header
  :routine-name "newt_jacfunc"
  :return-type 'void
  :c-arglist '((|n| int none)
                (|x| double array)
                (|fvec| double array)
                (|df| double array)
                (|vecfunc| void function))
  :c-header
  '("void newt_jacfunc(int n, double x[],
     double fvec[],
     double **df,
     void (*vecfunc)
     (int, double [],
     double [])"))))

(ffx:define-external-callable *newt_jacfunc-header* :callback t
:interface-lisp-dir
*lisp-c-interface-lisp-dir*
:interface-c-dir
*lisp-c-interface-c-dir*
:write-files t
:compile t)
Here's how the routines are loaded into Lisp:

```
(ffx:load-external-system :routines '
  "newt"
  "newt_vecfunc"
  "newt_jacfunc")
:files
  (list
    (concatenate 'string *recipes-directory* "nrutil")
    (concatenate 'string *recipes-directory* "newt")
    (concatenate 'string *recipes-directory* "fmin")
    (concatenate 'string *recipes-directory* "fdjac")
    (concatenate 'string *recipes-directory* "ludcmp")
    (concatenate 'string *recipes-directory* "lubksb")
    (concatenate 'string *recipes-directory* "lnsrch"))
:interface-lisp-dir *lisp-c-interface-lisp-dir*
:interface-c-dir *lisp-c-interface-c-dir*
:libraries "("m")
:solib-name "numrecipes.so"
:solib-dir *default-solib-dir*
:rebuild-solib *rebuild-solib-flag*)
```

8 Conclusion

We hope that these extensions will make it easier for Lisp programmers to use existing subroutine libraries in C and Fortran. Though we have not formally benchmarked the performance of the interface, we have been using it for some time now, and have not noticed any substantial performance difficulties.

The code for these extensions is available to anyone who wishes to use them. Please contact the author at his e-mail address on the title page.
References
