FFIGEN Manifesto and Overview*

Lars Thomas Hansen
lth@cs.uoregon.edu

February 6, 1996

Abstract

FFIGEN (Foreign Function Interface GENerator) is a program suite which facilitates the writing of translators from C header files to foreign function interfaces for particular language implementations.

On a more general level, FFIGEN is a statement about how such translators should be structured for maximum usability, namely as a single translator from C to a rational intermediate language and as multiple translators from the intermediate language to separate FFI translations. In the present document I motivate this two-level structure by arguing that the many policy questions inherent in choosing a mapping from one language to another cannot be accomodated in a single translator, and that the two-level structure promotes significant code reuse. Companion documents present the program suite itself.

1 Manifesto

Many language implementations have mechanisms which provide support for call-outs to other, typically more primitive, languages. In particular, implementations of very-high-level languages like Scheme, Common Lisp, Standard ML, and Haskell support call-outs to system-level languages, typically C. Other examples include the support for call-outs to C and assembly language in C++, the EXTRINSIC directive in HPF, and the `<*EXTERNAL*>` pragma in DEC SRC Modula-3. Mechanisms to call-out to other languages are typically called *foreign function interfaces* (FFIs). The purpose of an FFI is often to gain access to functionality which is not (efficiently) expressible in the language itself; other times the FFI is used to allow the program to interface to existing libraries.

FFIs are only rarely part of the language definition; the only examples I can think of are the support for C and assembly in C++ and the EXTRINSIC directive in HPF. More typically, each language implementation has its own idiosyncratic and often ad-hoc mechanism for supporting foreign data types, functions, and variables. The mechanisms

*This work has been supported by ARPA under U.S. Army grant No. DABT63-94-C-0029, “Programming Environments, Compiler Technology and Runtime Systems for Object Oriented Parallel Processing”.

are not standardized probably because they depend to a large extent on the calling conventions of the procedure being called, the operating system on which the program is running, the architecture of the machine, the data types of the language being called, the version of the compilers for the host and foreign languages, and so on. (In the following I will refer to a point in the space made from the product of the preceding attributes as a *target*.) Since the system dependencies are considerable, it is unlikely that a fully general and portable FFI can be defined for a language, and in addition, an interface that works with all targets is likely to be neither functional nor convenient. The chances for any portable, standardized language to adopt a non-trivial FFI therefore seem slight. This is not to say that an adequate job can't be done in many cases–for example, Franz Allegro Common Lisp sports a sophisticated FFI which supports C and Fortran seemingly very well–only that no *standard* and *general* solution is likely to emerge.

Based on these observations, an approach to inter-language calling would be to accept the fact that FFIs are implementation-dependent and instead concentrate our effort on a higher level of abstraction: that of the library interface. Even if the FFI is target-dependent, most of the time the interface to a library is not (which is the beauty of an interface in the first place). If, for each library, there existed a reasonable definition of its interface, then a program could take that definition and generate FFI code for the library for a given target. This is the approach advocated by the creators of the ILU system (see section 3).

However, manufacturers of libraries are *not* distributing reasonable definitions of the interfaces to their libraries. All you usually get is a C or C++ header file. A header file is not a reasonable definition of the interface because of the baggage it carries: nested include files, preprocessor macros, conditional compilation, syntactic peculiarities, implementation language target dependencies, and so on. In the best of all worlds, the manufacturer would distribute the interfaces in an interface definition language like the Object Management Group's IDL or ILU's ISL, and maybe one day that will be common. In the mean time, we must fend for ourselves.

What we must do is to provide a translator which takes as its input not a reasonable definition but instead a C or C++ header file or set of header files, and produces as its output the FFI code for the library for a given target. However, such a program is likely to be complicated and there will be one version for each target. Maintaining all these translators will be an unpleasant task. We could of course have one translator, to IDL or ISL, and translators from the interface language to the FFI, and as we will see, this is a variation on the mechanism implemented by FFIGEN.

An additional important problem is that there is not one but several translations for every target. A given interface can be translated to any of several FFIs depending on the desired *policy* for the translation. For example, consider a function

`char *fgets(char*, int, FILE*).`

What does `char*` translate to? Consider the FFI provided by Chez Scheme version 5. It has a `string` type which in a parameter position causes the address of the first character of the string argument to be passed to the function, but which in the return position causes the characters to be copied from the storage pointed to by the return value (if not `NULL`) into a fresh Scheme string. So if we translate `char*` as `string`, we end up with:[1]

```
(define fgets
  (foreign-function "fgets"
    (string integer-32 unsigned-32)
    string))
```

which is expensive because the string is (needlessly) copied on return. On the other hand, we can treat a `char*` as "just a pointer" and translate as:

```
(define fgets
  (foreign-function "fgets"
    (unsigned-32 integer-32 unsigned-32)
    unsigned-32))
```

but this does not let us access the characters in the buffer using Scheme's string functions, since the buffer is not a string. In the end, it appears that no fixed translation for `char*` is possible; even if a fixed translation (and then: which one of them?) is adequate in most situations, there will be special cases.[2]

The bottom line is, there is a lot of policy that goes into a translation into a specific FFI. Hence we have a slogan (the core of the Manifesto):

> A good foreign function interface is 25% code and 75% policy.

It should be a goal, then, to separate the ardous task of parsing and type-checking C headers and translating them into a rational intermediate form, from the task of translating the intermediate form into a FFI specification for a given target and translation policy.

2 The FFIGEN System

I have written a program, which I call `ffigen`, which takes as its input a C header file and produces as its output a rational translation of the interface defined by the header file. A rational translation is one in which unnecessary or redundant syntax has been removed, preprocessor macros have been expanded, and preprocessor conditionals have been resolved so that definitions have been included or excluded corrspondingly. The exact format of the intermediate code is described in a companion document, the *FFIGEN User's Manual*. `ffigen` functions as the *front-end* of a system which translates C headers into foreign function interfaces.

Each target system will have one or more specific *back-ends* which take the intermediate form and produce translations for particular targets and translation policies.

[1] The `FILE*` is translated as an `unsigned int`.

[2] Arguably, it would have been better for `fgets()` to return a truth value or the number of characters read.

Substantial parts of the back-end code is largely target-independent and can therefore be shared by multiple back-ends.

I have written one back-end to serve as a sample; it produces FFI code for Chez Scheme version 5. It is documented in a companion document, *FFIGEN Back-end for Chez Scheme Version 5.*

3 Related Work

Kenneth B. Russell of MIT has implemented a system called Header2Scheme which translates C++ to the FFI of the SCM Scheme system. FFIGEN and Header2Scheme are fairly different at this point. My goal with FFIGEN was to cover all of ANSI C including the preprocessor in a reasonable way; this is doable because ANSI C is a small, fixed, and fairly simple language. C++, on the other hand, is a very large, changing, and complex language, and Header2Scheme therefore handles only part of it at this time (as of version 1.2, it does not handle preprocessor macros, typedefs, and enums). In addition, my emphasis was on not fixing policy at all, which gives great freedom (and more work) to back-end writers, whereas Russell has mostly fixed the policy. On the other hand, Header2Scheme allows some policy decisions to be expressed in auxiliary files given to the translator, and I have yet to experiment with these mechanisms in FFIGEN. Header2Scheme is available from URL

`http://www-white.media.mit.edu/ kbrussel/Header2Scheme`

A message[3] posted to the Usenet group `comp.lang.scheme` (among others) alleged that Apple has a translator for their Dylan implementation which will take a C header file and generate Dylan FFI glue for it. I know nothing else about this system (but would appreciate hearing about it from anyone who knows).

The ILU (Inter-Language Unification) system from Xerox PARC provides cross-language calling functionality for modules which have interfaces specified in ISL, the ILU interface definition language. ILU will take the interfaces and produce stubs (glue, as it were) for the languages so that they can call each other. The ISL file specifies the interface somewhat abstractly in terms of data types which are meaningful in ISL but which have various mappings in the target languages; again, one mapping is assumed to fit all.

4 Acknowlegements

FFIGEN is based on the `lcc` ANSI C compiler. See the *FFIGEN User's Manual* for full acknowlegements and a copyright notice.

[3] `<1996Jan17.121933.25825@chemabs.uucp>`