Fetter Design Document

Rayiner Hashem

August 3, 2005

1 Synopsis

A program will be created to automatically generate foreign function interface (FFI) bindings directly from C++ header files. The program will implement this functionality using GCC-XML to parse C++ header files, giving it the ability to support a wide range of C++ features. The top priorities of the program will robust handling of C++ header files, followed by ease of use, then simplicity of maintenance. It will support multiple FFI back ends, but the first target will be UFFI on CMUCL.

2 Justification

The inability to access modern C++ libraries from Lisp code is an inconvenience for programmers. C++ is the most-used language on SourceForge, featured in 15,607 projects. Among these are premier Open Source platforms like Mozilla and KDE. All of this C++ code is inaccessible from Lisp and represents not only lost opportunities for sharing, but in the case of Mozilla and KDE, large platforms with large user bases that cannot be targeted by Lisp applications. A system that would allow Lisp code to access C++ libraries would thus be quite beneficial to Lisp programmers, particularly to those targeting Open Source platforms. It would not only ease development, but it would widen the potential audience for Lisp applications.

Now, it is reasonable to question the need for another program to automate the generation of FFI declarations. After all, many such programs already exist. However, most of these existing programs do not even attempt to handle C++ header files. Indeed, only a single system, the Sim-

plified Wrapper and Interface Generator (SWIG), offers anything resembling proper support for C++ features. However, even SWIG cannot handle C++ header files that require correct and complete template parsing semantics. In any case, there are no SWIG back ends available for free Common Lisp implementations, and the single Common Lisp back end that does exist, for Allegro Common Lisp (ACL), lacks C++ support entirely. Part of the reason why SWIG back ends for Lisp and Lisp-like languages are so uncommon is that writing a SWIG back end requires quite a bit of knowledge about the SWIG system itself, as well as a deep understanding of the C++ type system. Further, SWIG back ends must be written in C++,! making them less palatable projects for Lisp programmers. For these reasons, a creation of a new program seems justified.

3   Development Process

An incremental development model will be used to create the program. This decision is driven by the author's preference for this model when developing in dynamic languages and by the desire to make several stages of intermediate results available to demonstrate progress. It will be developed with much hindsight, as the author implemented a proof-of-concept of a similar system in 2004. The previous system was written in Python, targeted the Functional Developer Dylan FFI, and was complete enough to generate declarations for C libraries like SDL and OpenGL. Development on the concept was stopped largely because of job-related time pressures.

4   Project Details

The basic structure and operation of the program can be likened to that of a compiler. It will be run with a simple input file specifying the C++ header files and some configuration options. A front end will use GCC-XML to parse the header files, run several transformations on the resulting information, then pass the results via an intermediate file to a specified back end. The back end will then process the output and generate a Lisp package containing the FFI declarations and stub routines required to access the original header files.

  The front end will be responsible for processing the input file, generat-

ing an IR from the specified header files, and transforming the IR to make it readily digestible by the back ends. Generating the IR is fairly simple. When invoked on a header file, GCC-XML generates an XML file containing the header file's class and field declarations. This XML file can then be parsed into an object tree with the same structure. Since the GCC-XML format is very close to a proper semantic representation of the header file, this object tree can be used directly as the IR. Having parsed the XML files and generated the IR, the front end must then simplify the IR before it is passed to the back ends. It must perform transformations such as collapsing namespace hierarchies and nested structures, assigning names to anonymous types, and filling in values for default arguments. These transformations all exist to minimize the complexity of the back ends. After the simplification passes, the objec! t tree can be written to an output XML file and the back end can be started.

The back end will be responsible for taking the simplified IR and generating from it FFI declarations and stub routines. Multiple back ends can exist, each targeting a different FFI. The back ends can be very simple recursive-descent code generators. The front end does most of the work of lowering the IR to an appropriate level of abstraction. This simplicity is an important trait, as it is desired that back ends eventually be written for languages such as Dylan and Scheme. A survey of various FFIs suggests that not all will be able to offer the same level of access to C++ features. Three levels of support have been identified: L0, L1, and L2. L0 access will feature standard C access plus the ability to call methods in C++ classes. L1 access will feature the additional ability to override virtual functions in C++ classes. L2 access will feature the ability to integrate C++ classes into the native object system. A survey of existing FFIs suggests that UFFI can support L0 acce! ss by itself, or L1 access with compiler-specific workarounds for specifying callbacks. Sophisticated FFIs like the one in ACL or Functional Developer can support L2 access.

The runtime library will be a small C support library specific to each C++ compiler. It is necessary because existing Lisp FFIs have no understanding of C++ virtual function tables. The runtime library will encapsulate all the compiler-specific knowledge of virtual table formats, allowing Lisp programs to make virtual function calls and to generate virtual function tables that can be called from the C++ code.

5   Project Time Line

The project shall be divided into three phases, with a milestone at the end of each phase. The phases are outlined below:

- Phase 1 (Mid-June to July 1): Research the target FFI and the IA64 C++ ABI (the C++ ABI for most compilers on *NIX/x86).

- Milestone 1 (July 1): The creation of a detailed design document specifying the structure of the program, the format of the IR, the IR transformations in the front end, the precise semantics of the various levels of access, and the IR to FFI mapping.

- Phase 2 (July 1 to July 15): Create an initial implementation of the front-end and a back end implementation capable of performing L0 access.

- Milestone 2 (July 15): The creation of a Lisp demo using the SDL C++ API.

- Phase 3 (July 16 to August 15): Improve the implementation of the front end and create a back end capable of supporting L1 access.

- Milestone 3 (August 15): The creation of a Lisp demo using the KDE C++ API.

While this plan is a bit aggressive, experience with the previous concept suggests that it is realistic. The previous concept, as described, was implemented in two weeks.

6   Project Deliverables

As a proof of completion, the following items will be delivered:

- A front end capable of parsing input from GCC-XML 0.60.

- A back end capable of allowing L1 access through UFFI on CMUCL.

- A runtime library for GCC 3.x on Linux/x86.