Verrazano Project Summary

Rayiner Hashem

September 2, 2005

1 Abstract

A project was proposed to create a bindings generator to make C++ libraries accessible from Lisp code. A program was created to generate bindings for C++ libraries, and basic bindings for Cairo and FLTK were created, along with short demos utilizing those bindings. The initial goals of the project were met with varying degrees of success. Given the time-scale of the Summer of Code, the author feels that the project was reasonably successful in meeting its immediate goals.

2 Analysis of Goals

The initial goals of the project were to create a program that would:

- Offer robust handling of C++ header files.
- Be easy to use.
- Be simple to maintain.
- Offer support for multiple back-ends.

These goals were all met to some degree. Verrazano's handling of C++ header files is fairly robust. Verrazano is not at a point where one can expect a given library to "just work". Due to the complexity of the C++ standard and the GCC C++ ABI, reaching that level of robustness will require much more extensive testing than could be done within the time-scale

of the Summer of Code. However, Verrazano is already quite useful with a minimal level of tweaking. Verrazano was able to handle Cairo, FLTK, and libldap with minimal fixes to the generator. Verrazano's ease of use is its primary shortcoming. It's simple to use, but it can be somewhat tedious. In particular, the symbols to export from a binding must be specified manually. While most parts of Verrazano do not have to deal with errors, as the GCC-XML output is assumed to be correct, there are a few places in the front-end that require better error-checking and reporting. Simplicity of maintenance is Verrazano's primary strength. The code is compact and easy to develop. Much effort was spent on defining proper abstractions within the frontend, and refactoring the code continuously to keep it clean. As a result, new features, particularly those relating to transformations on the intermediate representation or modifications to the code generation, can be added quite quickly. In the author's opinion, the current code is a good base to build upon for the next steps in the program's evolution. Verrazano's support for multiple backends is quite good in the theory, but presently hampered by some structural issues. Fixing these issues is a simple matter of rearranging some of the build infrastructure and package definitions to better separate the frontend from the backends.

3 Analysis of Deliverables

The originally-proposed deliverables were the following:

- A frontend capable of parsing input from GCC-XML 0.60.
- A backend capable of allowing L1 access through UFFI on CMUCL.
- A runtime library for GCC 3.x on Linux/x86.

These deliverables were provided in spirit, if not in detail. The frontend is currently dependent on GCC-XML 0.7-CVS, as it takes advantage of some features not present in the 0.60 release. The backend does not use UFFI on CMUCL, but rather CFFI on SBCL. These changes were the result of the Hello-C project's decision to develop CFFI instead of UFFI. A compiler-specific runtime library was deemed unnecessary for the time-being, so one was not provided. However, a backend-specific runtime library was found to be necessary, so one was provided. The only material differences between

the proposed deliverables and the provided deliverables was the level of C++ access offered. It was proposed to offer full L1 access, including calling of virtual functions and overriding of virtual functions. This was not completely achieved. Currently, Verrazano does not allow access to virtual functions defined in virtual bases of a class, and does not allow overriding virtual functions. Support for other C++ features, including calling virtual functions, handling of overloaded functions, etc, was provided. These limitations in the level of C++ access were due to time constraints.

4 Future of the Project

Currently, Verrazano is not feature-complete. It is, however, at a level where it should be useful to intrepid developers. It's C support is theoretically complete, and should be usable for real work. It's C++ support is extensive enough that it should be useful in many cases. Given the present state of the project, the following tasks are proposed as the next steps in the development of the program, roughly in the order in which they will be undertaken:

- (Short term) Implement support for overriding C++ virtual member functions from Lisp code.
- (Short term) Implement support for accessing members of virtual bases.
- (Short term) Refactor the code to achieve better separation between the frontend, the cffi backend, and the GCC ABI-specific algorithms.
- (Short term) Define and implement a serialization format for the IR to allow non-Lisp backends to be written.
- (Medium term) Impose a period of stabilization during which the primary work will be to generate bindings for different libraries and to fix any defects exposed by doing so.
- (Long term) Work on better integrating C++ code with Lisp code. In particular, integrating the C++ type system with CLOS. This integration will likely require features not present in C-FFI (as a result of its limited type system), so it is likely that this work will necessitate a new backend. Functional Developer's FFI or clisp's FFI look like very attractive possibilities.