Clasp - A Common Lisp that Interoperates with C++ and Uses the LLVM Backend

Christian E. Schafmeister
Chemistry Department
Temple University
1901 N. 13th Street
Philadelphia, PA
meister@temple.edu

ABSTRACT

Clasp is an implementation of Common Lisp that interoperates with C++ and uses LLVM as its backend. It is available at `github.com/drmeister/clasp`. The goal of Clasp is to become a performant Common Lisp that can use C++ libraries and interoperate with LLVM-based tools and languages. The first sophisticated C++ library with which Clasp interoperates is the Clang C/C++ compiler front end. Using the Clang library, Common Lisp programs can be written that parse and carry out static analysis and automatic refactoring of C/C++ code. This facility is used to automatically analyze the Clasp C++ source code and construct an interface to the Memory Pool System garbage collector. It could also be used to generate automatically Foreign Function Interfaces to C/C++ libraries for use by other Common Lisp implementations.

Categories and Subject Descriptors

D.2.12 [**Software and its engineering**]: Interoperability; D.3.4 [**Software and Programming Languages**]: Incremental compilers

Keywords

Common Lisp, LLVM, C++, interoperation

1. INTRODUCTION

C++ is a popular, multi-paradigm, general-purpose language. In order to support sophisticated C++ language features (classes, methods, overloading, namespaces, exception handling, constructors/destructors, etc.), the syntax and application binary interface (ABI) of C++ has grown to be quite complex and difficult for other programming languages to interoperate with. In past decades, many software libraries were written in C++ and it would be valuable to make these libraries available to Common Lisp.

2. PREVIOUS WORK

Clasp is inspired by, and borrows code from, Embedded Common Lisp (ECL)[1], a Common Lisp implementation that is both written in and interoperates with C.

3. THE CLASP SOURCE CODE

Clasp consists of about 100,000 logical source lines of C++ code, 50,000 logical source lines of C++ header code, and 33,000 logical source lines of Common Lisp. About 26,000 lines of the Clasp Common Lisp source code are derived from the ECL Common Lisp source code. About 10% of the Clasp C++ code is translated from ECL C code.

4. AUTOMATED ANALYSIS OF CLASP C++ CODE FOR COMPACTING GC

Clasp makes it easy for Common Lisp to interoperate with complex C++ libraries and it uses this capability to integrate a sophisticated compacting garbage collector within Clasp. In normal operation, garbage collection is carried out in Clasp by the compacting Ravenbrook Memory Pool System (MPS) garbage collector library[2]. MPS is a copying garbage collector that treats pointers conservatively on the stack and precisely on the heap. MPS continuously moves objects and compacts memory, however, moving of objects in memory is challenging to reconcile with C++ which provides access to C-style pointers and pointer arithmetic. In order to allow the MPS library to work with Clasp's C++ core, every pointer to every object that will move in memory needs to be updated whenever that object is moved. Clasp fully automates the identification of C++ pointers that need to be updated by MPS, using a static analyzer written in Clasp Common Lisp. The static analyzer uses the Clang AST (Abstract Syntax Tree) and ASTMatcher C++ libraries. The static analyzer uses the Clang C++ compiler front end to parse the 173 C++ source files of Clasp and uses the Clang ASTMatcher library to search the C++ Abstract Syntax Tree in order to identify every class and global pointer that needs to be managed by MPS. It generates about 18,000 lines of C++ code that provides a functional interface to the MPS library to update all of these C++ pointers every time the MPS library carries out garbage collection. The static analyzer generates C++ code that updates pointers for 295 C++ classes and 2,625 global variables that represent Common Lisp types and symbols.

5. EXPOSING C++ TO COMMON LISP

Clasp was designed to ease interoperation of Common Lisp with foreign C++ libraries. To facilitate interoperation, Clasp incorporates a C++ template library (called *clbind*) that allows C++ library functions and classes to be exposed within Clasp by binding Common Lisp symbols to C++ functions, classes, and enumerated types. This approach is very different from the typical Foreign Function Interface (FFI) approach found in other Common Lisp implementations. Binding a C++ function to a Common Lisp symbol requires one C++ function call at Clasp startup, providing the name of the global Common Lisp symbol to bind and a pointer to the C++ function to bind to it. A C++ wrapper function that performs argument and return value conversions is automatically constructed by the C++ template library. The programmer can add additional value conversion functions. C++ pointer ownership is controlled by additional template arguments to `def`. The *clbind* library is based on the *boost::python*[3] and *luabind*[4] C++ template libraries.

To illustrate how a C++ class, constructor, method and function are exposed to Clasp Common Lisp, within the package `VEC`, the following example is provided.

```
// Exposing a C++ class and function to Clasp
#include <stdio.h>
#include "clasp/clasp.h"
class Vec2 {
public:
  double x, y;
  Vec2(double ax, double ay) : x(ax), y(ay) {};
  double dotProduct(const Vec2& o) {
    return this->x*o.x+this->y*o.y;
  };
};
void printVec(const std::string& s, const Vec2& v) {
  std::cout <<s<<"("<<v.x<<","<<v.y<<")";
}
extern "C" {
  void CLASP_MAIN() {
    using namespace clbind;
    package("VEC") [
      class_<Vec2>("vec2",no_default_constructor)
      .  def_constructor("make-vec2"
               ,constructor<double,double>())
      .  def("dot-product",&Vec2::dotProduct)
      ,def("print-vec2",&printVec)
    ];
  }
}
```

A sample Clasp Common Lisp session that uses the exposed C++ class and function is shown below:

```
>;;; A sample Common Lisp session.
>;;; "els.bundle" is the library built
>;;; from the source file above.
> (load "els.bundle")
T
> (defvar *a* (vec:make-vec2 1.0 2.0))
*A*
> (defvar *b* (vec:make-vec2 4.0 5.0))
*B*
> (vec:print-vec2 "a" *a*)
a(1,2)
> (vec:dot-product *a* *b*)
14
```

Notice how two Vec2 GC managed instances are created. The *a* object is printed and then the dot-product is calculated between the two vectors. An interface constructed this way between Clasp and a C++ library does not require either Clasp or the C++ library to be recompiled.

6. C++ RAII AND NON-LOCAL EXITS IN COMMON LISP

C++ code makes heavy use of a technique called "Resource Acquisition Is Initialization" (RAII). RAII requires that stack unwinding be accompanied by the ordered invocation of C++ destructors. To support interoperation with C++ and RAII, C++ exception handling is used within Clasp to achieve stack-unwinding and to implement the Common Lisp special operators: `go`, `return-from`, and `throw`. C++ exception handling is "zero runtime cost" when it is not invoked but can be quite expensive when used. For this reason the Clasp compiler uses exception handling only when necessary.

7. CONCLUSIONS AND FUTURE WORK

Towards the goal of developing Clasp as a performant Common Lisp compiler, the Cleavir[5] compiler, which is part of the SICL Common Lisp implementation developed by Robert Strandh, has been incorporated into Clasp. We are currently extending Cleavir/Clasp to generate source location debugging information (as DWARF metadata) and to incorporate optimizations that enhance performance of the generated code. We are also working on utilizing immediate tagged pointers for `fixnum`, `character`, and `cons` Common Lisp types and garbage collection of LLVM generated code.

8. LICENSE

9. ACKNOWLEDGMENTS

Thanks to Robert Strandh for providing Cleavir and for many fruitful discussions.

10. REFERENCES

[1] *Attardi, G. (1994)* The embeddable Common Lisp. In Papers of the fourth international conference on LISP users and vendors (LUV '94). ACM, New York, NY, USA, 30-41. http://doi.acm.org/10.1145/224139.1379849

[2] *Richard Brooksby. (2002)* The Memory Pool System: Thirty person-years of memory management development goes Open Source. ISMM02.

[3] *Abrahams, D.; Grosse-Kunstleve, R.W. (2003)* Building Hybrid Systems with Boost.Python. C/C++ Users Journal

[4] http://www.rasterbar.com/products/luabind.html

[5] https://github.com/robert-strandh/SICL