Clasp Common Lisp Implementation and Optimization

Alex Wood, Christian E. Schafmeister
Chemistry Department
Temple University
1901 N. 13th Street
Philadelphia, PA, U.S.A 19122
meister@temple.edu

ABSTRACT

We describe implementation strategies and updates made in the last two years to CLASP,[7] a new Common Lisp implementation that interoperates with C++, uses the *Cleavir*[8] compiler, and uses the LLVM backend[5]. Improvements in CLASP have been made in many areas. The most important changes are: (1) Tagged pointers and immediate values have been incorporated. (2) A fast generic function dispatch approach has been implemented that allows CLASP to carry out generic function dispatch as fast as SBCL, a highly optimized free implementation of Common Lisp. (3) The new LLVM feature "Thin Link Time Optimization" has been added, which speeds up generated code by removing call overhead throughout the system. (4) Type inference has been added to the *Cleavir* compiler, which is part of CLASP. Type inference removes redundant run-time type checks and dead code paths. Type inference currently provides about a 30% speedup in microbenchmarks.[10] (5) Constants and literals have been moved close to the code and "instruction pointer addressing" has been incorporated to speed access to literals and constants. (6) The generic function dispatch memoization approach developed by Robert Strandh[9] is being incorporated and currently shows a 30% improvement in performance of generic function dispatch. (7) The overall build time for Clasp has been reduced from five to eight hours over two years ago to approximately one hour at present.

Keywords

Common Lisp, clasp, cando, c++, llvm, cleavir

1. INTRODUCTION

We describe implementation strategies and updates made in the last two years to CLASP[7] a new Common Lisp implementation that interoperates with C++ and uses the LLVM backend to generate efficient native code for an increasing variety of processors. CLASP is being developed as a scientific programming language and specifically to support *CANDO*.[1] *CANDO* is being designed as a Common Lisp implementation to support the development of new software tools for molecular modeling and molecular design and it extends CLASP by adding hundreds of thousands of lines of C++ and Common Lisp code that implement solutions to computational chemistry problems. *CANDO* is designed to enable interactive programming to develop tools for designing molecules while at the same time generating fast native code for the demanding chemistry and molecular modeling applications.

2. TAGGED POINTERS AND MEMORY MANAGEMENT

CLASP makes use of pointer tagging to indicate the type of a pointer, and also encodes some values directly as immediates. CLASP is currently only targeting 64-bit architectures, and so the following currently applies to the 64-bit implementation. The type of a pointer is indicated using a three bit tag in the least significant bits of a 64-bit word. The meaning of the tags and their current values in binary are fixnum (#B000 and #B100), general pointer (#B001), character (#B010), cons (#B011), vaslist (#B101), and single-float (#B110). #B111 is currently unused.

The immediate types are fixnum, character, and single-float; they are stored in the high bits of the word. Fixnum values are 62 bits wide and they are indicated by the value #B00 in the two least significant bits. This allows fixnum addition and comparison to be carried out without bit shifting. Character values are 32 bits wide, enough to encode Unicode.[11] Single-float values are 32 bit wide, IEEE754 format, corresponding to the C++ 'float' type.

Cons pointers are indicated by the tag value #B011. The cons itself is two sequential 8-byte words. The `consp` test is extremely efficient because the tag is sufficient to indicate a cons. Traversal of lists thus involves only one tag test per element, excluding the ultimate element.

Vaslist is a special CLASP type used to operate on variable-length lists of arguments without allocating an actual Lisp list structure. These can be constructed in Lisp using the &CORE:VA-REST lambda list keyword, similar to e.g. &MORE in SBCL. The vaslist itself is a pointer to a structure incorporating a C/C++ va_list structure, together with a count of arguments remaining. One way we can work with vaslists in CLASP is using the BIND-VA-LIST special operator, which is analogous to DESTRUCTURING-BIND.

All other general object pointers share the pointer tag #B001. General objects consist of one header word followed by data. The header word is used by the garbage collector to identify the layout of the data, and to determine C++ inheritance relationships to avoid the use of the slow C++ template function "dynamic_cast".

There are four kinds of general objects:

1. Instances of C++ classes. These must inherit from Clasp's General_O, and have their layouts known to the garbage collector.

2. Instances of Common Lisp classes, i.e. standard-objects, funcallable-standard-objects, conditions, and structure-objects. These are implemented as the C++ class Instance_O (Table 1). These consist of a "Signature" (a description of the object's slots used for obsolete instance updating), a pointer to their class, and a *rack* of slots. Funcallable instances include more data (implemented as FuncallableInstance_O, Table 2), but keep the class and rack at the same positions as non-funcallable instances, to facilitate uniform access.

3. Instances of the C++ Wrapper<T> template class, which wraps C++ pointers and keeps track of their types. These can be used as pointers to C++ objects outside of managed memory, so that such objects can be operated on from Lisp.

4. Instances of the C++ Derivable<T> template class, which is used to create Common Lisp classes that inherit from C++ classes.

Lisp's other built in classes, such as symbols, complex numbers, and arrays, are implemented as C++ classes, i.e. the first kind.

Because the header does not include information about Lisp classes, among other things, it is not totally sufficient for Lisp type checking. It can however be used for rapid discrimination of instances of built in classes.

General objects and conses are stored on the heap and managed by the memory manager. CLASP can be built to use one of two automatic memory management systems: the Boehm garbage collector[2], or the Memory Pool System (MPS)[6].

Boehm is a conservative collector originally designed for C programs, meaning that it treats objects as opaque blobs of memory, identifies memory words as pointers without using type information, and does not move objects. This allows it to allocate and collect quickly, but as it does not compact memory it can leave the heap fragmented, impacting long-running CLASP programs and ones that allocate large objects.

The Boehm build of CLASP is required to run a special C++ static analyzer, written in CLASP, that determines memory layouts for all C++ classes in the CLASP C++ source.

The Memory Pool System is a precise collector more suitable for Lisp. It requires information per-type about where pointers are located; this is derived from the static analyzer, indexed using the general object headers. MPS can move and compact memory, using the least significant two bits of the header word to indicate forwarding pointers and padding. The MPS will run in a fixed memory footprint, making it suitable as the default memory manager for CLASP for scientific programming.

Table 1: Memory layout of Instance_O.

Offset	C++ type	Field
0	T_sp	Signature
8	Class_sp	Class
16	SimpleVector_sp	Rack

Table 2: Memory layout of FuncallableInstance_O.

Offset	C++ type	Field
0	std::atomic<claspFunction>	entry
8	Class_sp	Class
16	SimpleVector_sp	Rack
24	std::atomic<T_sp>	CallHistory
32	std::atomic<T_sp>	SpecializerProfile
24	std::atomic<T_sp>	DispatchFunction

3. ARRAYS

Common Lisp vectors and arrays are implemented using the structures shown (Table 3 and Table 4). The TYPE of a simple vector can be specialized within the Clasp C++ code to be any C++ type or class. The types that are currently supported are T_sp (the general Common Lisp T type), fixnum, double, float, and signed and unsigned integer types of length 8, 16, 32 and 64 bits. Simple bit vectors are implemented by manipulating bits in unsigned 32 bit words. The offset of the Length field in simple vectors and the FillPointerOrLengthOrDummy field of complex arrays is the same so that the length or fill-pointer can be accessed quickly for both simple and complex vectors. For multi-dimensional arrays FillPointerOrLengthOrDummy is ignored. The Flags field stores whether the array has ARRAY-FILL-POINTER-P and whether the array is displaced.

Array operations can be complex, and they are inlined for both simple vector and complex array operations. CLASP does this by undisplacing the array to the range of memory of the simple vector that ultimately stores the array contents and then indexes into that simple vector. Inlining is performed for SVREF, ROW-MAJOR-AREF, SCHAR, CHAR, ELT and AREF.

4. FAST GENERIC FUNCTION DISPATCH

CLASP implements the fast generic function dispatch approach developed by Robert Strandh.[9] Fast generic function dispatch is important in CLASP because it uses the Cleavir compiler (also written by Robert Strandh)[8], which makes heavy use of generic functions in its operation.

To enable the dispatch technique, all objects have a 64-

Table 3: Memory layout of simple vectors, the SimpleVector_O class.

Offset	C++ type	Field
0	size_t	Length
8	TYPE	Data[length]

Table 4: Memory layout of complex arrays, the MDArray_O class.

Offset	C++ type	Field
0	size_t	FillPointerOrLengthOrDummy
8	size_t	ArrayTotalSize
16	Array_sp	Data
24	size_t	DisplacedIndexOffset
32	Flags	Flags
40	size_t	Rank
48	size_t	Dimensions[Rank]

Table 5: Generic function call time.

Implementation	gf call (s)	accessor call (s)
Clasp(553e35a)	1.34	1.14
SBCL(1.3.10)	1.36	1.14
ECL(16.0.0)	27.24	7.26
ccl(1.11-store-r16714)	3.82	3.65

bit value called the *stamp*, unique to the class it was defined with. For general objects, the stamp is within the header word for instances of C++ classes, but it is stored within the object otherwise.

The fast generic function dispatch approach works by compiling discriminating functions that dispatch to precompiled effective methods based on the stamps of their arguments (Figure 1). This reduces discrimination to a series of integer comparisons, making it very efficient. The "slow path" of dealing with actual classes and calling the MOP-specified generic functions only comes into play if the integer comparisons fail to branch to a known effective method.

If a Lisp class is redefined, its stamp is changed and existing discriminators have their dispatching for the old stamp removed. Calls to discriminators involving obsolescent instances can therefore update instances only in the slow path. This in particular, is a major improvement over ECL, which checks object obsolescence for all calls to accessors.

CLASP incorporates a small extension to the dispatch technique. Calls involving EQL specializers cannot be memoized normally, because they imply the necessity of tests more specific than stamp tests. However, CLASP does memoize some calls involving EQL specializers, for standard generic functions that cannot have more behavior on them due to MOP. The critical condition is that any argument in an EQL-specialized position does match an EQL-specialized method, and not only class-specialized methods. For example, if a generic function of one parameter has one method specialized on the symbol class, and one EQL-specialized on a particular symbol, calls involving the latter will be memoized while the former will not be. This speeds the most common uses of EQL specializers while preserving correctness.

In a test where two generic functions that accept two arguments were called 100,000,000 times - the timing values (in seconds) were obtained (Table 5).

The performance is remarkable given that clasp started out using the ECL CLOS source code and reimplemented the ECL generic function dispatch cache. With the Strandh fast generic function dispatch, CLASP is equivalent to the performance of SBCL, a highly optimized implementation of Common Lisp.

There is a "warm-up" time associated with this fast generic function dispatch method as it is currently implemented within CLASP. Discriminating functions are compiled lazily when the generic function is called and the first compilation forces a cascade of compilation of about 1,200 functions. This takes tens of seconds of real time. This only happens once at startup and thereafter discriminating functions are invalidated and recompiled only when methods are added or removed or classes are redefined.

5. COMMON FOREIGN FUNCTION INTERFACE

CLASP has now incorporated an implementation of the Common Foreign Function Interface (CFFI). This gives it access to many C libraries that have been exposed to Common Lisp using CFFI. This is in addition to CLASP's built in C++ interoperation facility *clbind*.

6. LINK TIME OPTIMIZATION

A new LLVM feature called "Link Time Optimization" (LTO) has been incorporated into CLASP. With LTO, all code that is compiled with COMPILE-FILE and all of the CLASP C++ code is compiled to the LLVM intermediate representation (LLVM-IR) and written to files as LLVM "bitcode". The link stage of building then does a final round of optimization, wherein LLVM-IR from Common Lisp code and LLVM-IR from C++ code for internal functions with symbols that are not exported are inlined within each other and those that are not inlined are converted to the "fastcc" calling convention, which passes arguments as much as possible in registers. The overall effect is to reduce the overhead of calls within CLASP.

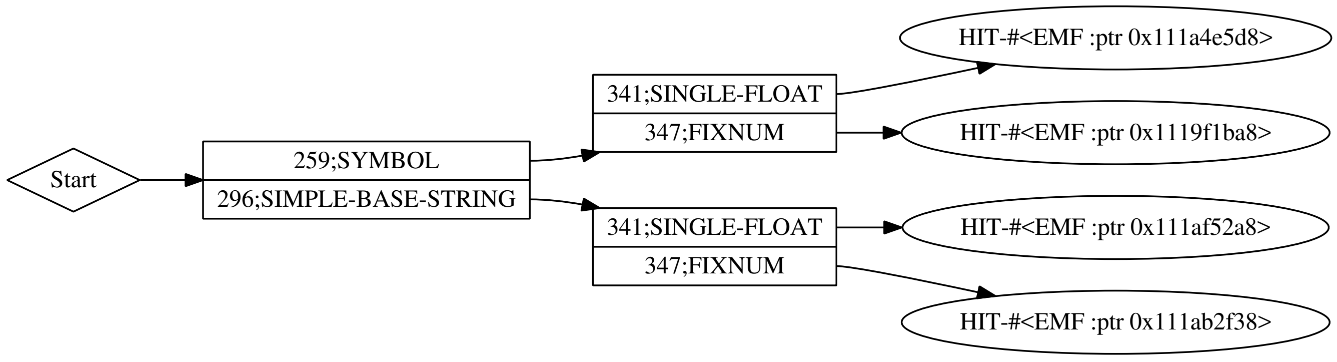
7. C CALLING CONVENTION

Currently, CLASP only supports the x86-64 Application Binary Interface (ABI) because it makes non-portable references to x86-64 ABI C calling convention details to improve performance. The key detail that is important is that the x86-64 C calling convention passes up to six integer word arguments in registers and returns up to two integer word arguments in registers and this used by CLASP to use registers as much as possible when making function calls. CLASP passes arguments with the following C calling signature: (void* closure, size_t number-of-arguments, void* arg0, void* arg1, void* arg2, void* arg3, ...). So, up to four general objects are passed in registers and when the lambda list for a function uses &rest or &key arguments then the ABI details of C calling convention "var-args" facility is used to spill the register arguments into a register save area on the stack. A C va_list is then "rewound" to point to the *arg0* argument so that all arguments can be accessed one at a time using the CLASP "vaslist" facility.

8. INSTRUCTION POINTER ADDRESSING

CLASP uses the LLVM library, which defines C++ classes for Modules, Functions, BasicBlocks, Instructions and GlobalVariables. Code generated by LLVM cannot currently be managed by the memory manager and must live outside of the managed memory space, at fixed locations. Functions

Figure 1: A generic function specialized on two arguments. The left box represents stamp values that are matched to the first argument stamp and the right boxes represent stamp values matched to the second argument depending on the first argument.


Start
259;SYMBOL
296;SIMPLE-BASE-STRING
341;SINGLE-FLOAT
347;FIXNUM
341;SINGLE-FLOAT
347;FIXNUM
HIT-#<EMF :ptr 0x111a4e5d8>
HIT-#<EMF :ptr 0x1119f1ba8>
HIT-#<EMF :ptr 0x111af52a8>
HIT-#<EMF :ptr 0x111ab2f38>

therefore accumulate in memory as CLASP runs. Memory depletion has not been a problem because modern computer memories are large, but it does make referencing Common Lisp objects from LLVM generated code problematic. To deal with this, each LLVM module has a block of garbage collector roots. These roots point to all non-immediate constants referenced by the functions in the module. These constants can then exist in memory managed space without being collected inappropriately.

9. STACK UNWINDING

Stack unwinding is achieved in CLASP using C++ exception handling to allow CLASP to inter-operate with C++ code. C++ stack unwinding on x86-64 uses the Itanium ABI Zero-cost Exception Handling.[4] It is “Zero-cost” in that there is no runtime cost when it is not used but actual unwinding the stack is quite expensive. Timing of a simple CATCH/THROW pair demonstrates that unwinding the stack in Clasp is about 90x slower than it is in *SBCL* and thus stack unwinding must be done judiciously.

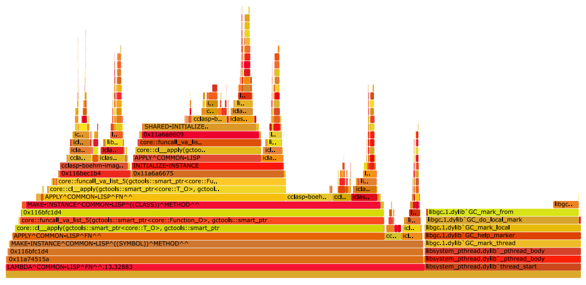
10. INLINING OF ARITHMETIC FUNCTIONS

Arithmetic can now be done without function calls in common cases. When the operands to a binary operation such as addition are of the same kind - fixnum, single float, or double float - the operation is carried out without a function call. This also occurs if one operand can be easily coerced to a value of the other type, e.g., a fixnum plus single float operation becomes a single float plus single float operation once a single float corresponding to the fixnum value is produced.

Facilities are now in place to deal with unboxed values. These are special register values, outside of the normal managed memory regime, representing number values. For example, even though 64-bit floats cannot be immediates due to lacking space for a tag, they can be dealt with as unboxed values. Inlined arithmetic essentially consists of ”unboxing” values (e.g. extracting double floats from memory), performing machine arithmetic operations, and then boxing the results. A future compiler stage will remove unboxing operations followed immediately by boxing operations, as are done for arithmetic operations using the results of other operations as operands.

Arithmetic involving nontrivial allocations, such as on bignums or complex numbers, still goes through function calls.

Figure 2: A flame graph generated by profiling CLASP repeatedly calling MAKE-INSTANCE. 19.7% of the time is spent in SHARED-INITIALIZE and 27.2% of the time is consumed by the memory manager.



11. PROFILING

Since all CLASP Common Lisp and C++ code compiles to LLVM-IR, all functions look like standard C functions to standard profiling tools. This allows standard tracing tools like “dtrace”[3] to be used to profile CLASP code (Figure 2).

12. SOURCE TRACKING

CLASP has incorporated source tracking using facilities from the SICL project. CLASP makes extensive use of the nascent SICL project, including the CLEAVIR compiler, and the SICL reader. The CLEAVIR compiler has recently been upgraded to generate “Abstract Syntax Trees” (AST) from “Concrete Syntax Trees” (CST). CST’s are a representation of Common Lisp source code that has source location information attached to every atomic token and every list. One of the purposes of this is to carry source code location information into the AST and then all the way to the final generated machine code to facilitate debugging. Other applications for CST’s include: tools that carry out source-to-source translation and programming tools like syntax highlighting.

13. DEBUGGING USING DWARF

CLASP generates DWARF debugging information using the DIBuilder API of the LLVM C++ library. This will allow CLASP compiled programs to be inspected and debugged using the industry standard debuggers "gdb" and "lldb". DWARF debugging information is used by these debuggers to provide information about source line information and the locations of variables in stack frames. The DWARF source line information that will be generated with the aid of source tracking information provided by Concrete Syntax Trees will greatly facilitate debugging. The uniform use of DWARF debugging information for Common Lisp and C++ code will allow the debugging of CLASP programs that make use of C++, C and Fortran libraries.

14. CONCLUSIONS AND FUTURE WORK

CLASP is a new implementation of Common Lisp that interoperates with C++ and uses the LLVM library as a backend. It supports novel interoperation features with C++ libraries and industry standard profiling and debugging tools. CLASP incorporates the Cleavir compiler that is a platform for exploring new ideas in compiling dynamic languages.

15. ACKNOWLEDGMENTS

We want to thank professor Robert Strandh for his constant patience and guidance which made this work possible.

16. REFERENCES

[1] *Schafmeister, C. E.* "CANDO - a Compiled Programming Language for Computer-Aided Nanomaterial Design and Optimization Based on Clasp Common List" (2016): 75âĂŞ82.

[2] *Boehm, H.* "Simple Garbage-Collector-Safety", Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation.

[3] http://dtrace.org/blogs/about/

[4] http://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html

[5] *Lattner, C. (2005)* "Masters Thesis: LLVM: An Infrastructure for Multi-Stage Optimization", Computer Science Dept., University of Illinois at Urbana-Champaign, http://llvm.cs.uiuc.edu

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

[7] *Schafmeister, C. (2015)* "CANDO - A Compiled Programming Language for Computer-Aided Nanomaterial Design and Optimization Based on Clasp Common Lisp", European Lisp Symposium, 2015.

[8] https://github.com/robert-strandh/SICL/tree/master/Code/Cleavir

[9] *Strandh, R. (2014).* "Fast generic dispatch for Common Lisp", Proceedings of ILC 2014 on 8th International Lisp Conference - ILC 14. doi:10.1145/2635648.2635654

[10] *Wood, A. (2017).* "Type Inference in Cleavir", Proceedings of ELS 2017

[11] *The Unicode Consortium.* "The Unicode Standard." http://www.unicode.org/versions/latest/