LLVM Code Generation for Open Dylan

Peter S. Housel
housel@acm.org

ABSTRACT

The Open Dylan compiler, DFMC, was originally designed in the 1990s to compile Dylan language code targeting the 32-bit Intel x86 platform, or other platforms via portable C. As platforms have evolved since, this approach has been unable to provide efficient code generation for a broader range of target platforms, or to adequately support tools such as debuggers, profilers, and code coverage analyzers.

Developing a code generator for Open Dylan that uses the LLVM compiler infrastructure is enabling us to support these goals and modernize our implementation. This work describes the design decisions and engineering trade-offs that have influenced the implementation of the LLVM back-end and its associated run-time support.

CCS CONCEPTS

• **Software and its engineering → Compilers**; **Runtime environments**.

KEYWORDS

compilers, dylan programming language

ACM Reference Format:
Peter S. Housel. 2020. LLVM Code Generation for Open Dylan. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.5281/zenodo.3742567

1 INTRODUCTION

The Dylan programming language [8] is a member of the Lisp family of languages designed to combine much of the dynamicity of other Lisp dialects (such as Common Lisp) with features that enable efficient compiled code and support application delivery using stand-alone executables and shared libraries. One aspect of the language design that enables these goals is library-centric compilation. Program code is organized into individual libraries, and all definitions for a library are submitted at once to the compiler. Information about all of the source definitions in the library allows the compiler to make use of Dylan's *sealing* feature. Sealing a class or a generic function guarantees to the compiler that it will not be extended (through subclassing of classes, or adding methods to a generic function) beyond what is in the library being compiled. Dylan compilers can use sealing guarantees to statically enumerate subtypes and applicable methods at compile time, enabling type inference and optimizations such as method inlining and more specific method dispatch.

ELS'20, April 27–28 2020, Zürich, Switzerland

ACM ISBN 978-2-9557474-4-5.
https://doi.org/10.5281/zenodo.3742567

The structure of the Open Dylan[1] compiler, called DFMC (the Dylan Flow Machine Compiler) is shown in Figure 1. When DFMC needs to compile a Dylan library from a set of source files, processing goes through the following phases:

- The Reader parses the input (using a state-based lexical analyzer and a LALR parser) into an abstract syntax tree based on syntactic fragments. Interleaved with parsing, the Macroexpander rewrites the AST according to macro definitions visible in the source file's lexical scope. When the compiler parses a `define library` definition, it may load the library databases of any referenced libraries so that macro (and other) definitions become visible.
- The Object Modeling and Conversion phases build compile-time representations of bindings, objects, and code. After conversion, code from methods and other definitions is expressed using a Dylan-centric intermediate representation called Dylan Flow Machine or DFM. The DFM representation is effectively Static Single Assignment (SSA), though any variables that are assigned are converted into stack-allocated or heap-allocated value cells rather than being split into distinct SSA values.
- The Optimization phase iteratively transforms DFM functions, performing (among others) type inferencing, tail-call elimination, constant folding, common subexpression elimination, dead code removal, and method inlining.
- The Linking and Emitting phases use one of the selectable back-ends to write out modeled objects and code for final code generation and linking, as discussed in detail below.
- After compilation, the compiler writes out a database file to persist information about the library for import into other libraries.

Before the LLVM back-end described in the present work was implemented, DFMC provided two selectable back-ends:

- The C back-end transforms the DFM intermediate representation into C language source code so that the final machine code generation task can be passed on to a platform C compiler. Run-time support for this back-end is also written in C, and the Boehm-Demers-Weiser conservative garbage collector [2] is used to provide memory allocation.
- The HARP back-end, which was the primary one used for the commercial Dylan product, generates 32-bit Intel x86 machine code for the Windows, Linux, and FreeBSD operating systems. After transforming the DFM representation into a HARP-specific machine-oriented representation, the back-end does basic x86 instruction selection, graph-coloring register allocation, and simple branch optimization before directly writing out (in the Windows case) COFF object files.

[1] Before it was released as open-source software, the Open Dylan implementation was initially known as Harlequin Dylan, and later as Functional Developer

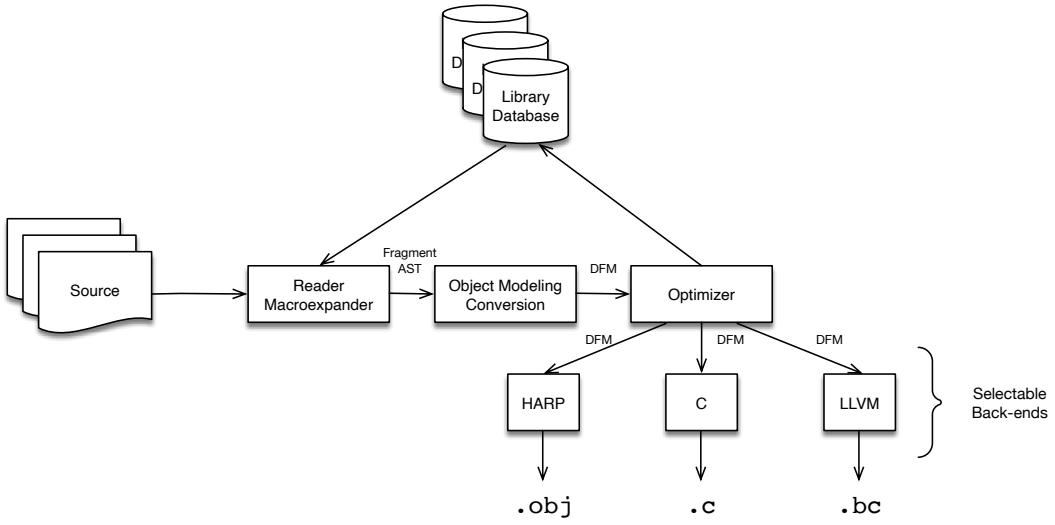

Figure 1: DFMC Compiler Structure

Microsoft CV4 Symbolic Debug Information format is also written into the object file output.

Most of the run-time support for this back-end is written in the HARP machine representation, made available using a stand-alone runtime generator tool. The remainder of the run-time is in C. Either the Boehm-Demers-Weiser conservative collector, or the Memory Pool System [3] (which supports incremental generational collection) can be used, selectable at build time.

2 THE LLVM BACK-END

The LLVM back-end was developed with the following goals in mind:

- Support debug information on platforms other than Windows.
- Expand support to other architectures while minimizing the inefficiencies incurred by compiling via C code.
- Take advantage of optimizations provided by the LLVM compiler infrastructure.
- Eventually support integration with non-conservative garbage collectors such as the Memory Pool System.

The following sections discuss details of the design decisions implemented in the back-end.

2.1 Back-end Intermediate Representation

LLVM defines an SSA-based representation for code, along with an extensive set of architecture-independent and architecture-specific intrinsic functions. Once code is compiled into the LLVM IR, the LLVM analysis and optimization phases and code generators can output assembly or machine code for a variety of target architectures and platforms. Most compilers using the LLVM infrastructure construct the IR representation by interfacing with the LLVM libraries (either directly from C++ or indirectly via C bindings). Other alternatives include writing out the LLVM assembly language representation and letting the infrastructure parse it, and directly writing out the IR in LLVM's compact binary "bitcode" format.

Though Open Dylan provides a foreign-function interface that would have allowed using the LLVM libraries through their C bindings, it was judged that it would be easier long-term to use a Dylan-native intermediate representation. This has allowed the interface to fit better stylistically with the surrounding code. For instance, the LLVM libraries construct IR objects such as types, constants, and instructions within a *context* so that (through hash-consing) semantically equivalent objects are actually identical. Since the DFMC back-end does very little code analysis at the LLVM IR level, this property is not as useful there, making the burden of always making a context available less worthwhile.

Given a Dylan-native IR, the back-end could either write out textual LLVM assembly language or bitcode. In addition to the overhead of writing out and parsing textual representations, the assembly language has historically not provided many compatibility guarantees as the LLVM language has evolved. Though the documentation for the bitcode format representation of LLVM IR is somewhat incomplete, often requiring reverse-engineering, the relative stability of the format has made this effort worthwhile.

Though the native IR does not attempt to unify equivalent objects such as LLVM types or constant expressions, the bitcode representation requires that they be unified and enumerated in order to represent references. The Dylan LLVM bitcode writer does this at bitcode output time, collecting all referenced IR objects into equivalence classes and performing partition refinement before assigning indices and writing out bitcode records for each item.

2.2 Type Representation

LLVM uses a typed intermediate representation. This means that unlike the HARP back-end (but like the C back-end), the LLVM back-end must take care to use the right type representation within

generated code and in run-time support routines, and to insert cast operations when necessary.

Like many Lisp implementations, Open Dylan uses tagged pointers, with the lower two bits indicating whether a Dylan value is a heap object, a (fixed-size) integer, or a character. At present the LLVM generic byte pointer type `i8*` is used to represent Dylan `<object>` values; in the future an alternative address space marker may be added to the pointer type in order to mark garbage-collectable pointers for the LLVM generator of static GC root information.

When necessary, casts to other types are inserted at the point of use. Operations on tagged integers or characters require an `inttoptr` cast, along with (potentially) a shift to remove the tag. Accesses to heap objects require a `bitcast` to a `struct` pointer type reflecting the heap layout of the object. Dylan heap objects are represented using a single header word, a pointer to a garbage-collector "wrapper" structure (also used to provide concrete type information at run time), followed by one or more pointer-sized slots.

Objects may also have an optional "repeated slot" used to implement various types of containers. Repeated object pointer slots are used for generic container types such as `<simple-object-vector>`, and repeated "raw"-typed slots for specialty containers such as `<string>`.

In addition to ordinary object pointer types that belong to the Dylan value type hierarchy, Open Dylan supports a number of raw types for values such as untagged bytes, machine words, and floating-point values. These normally have a straightforward mapping to LLVM primitive types and are used to implement higher-level operations and as part of the foreign-function interface.

2.3 Primitive Functions

The Open Dylan compiler defines a set of intrinsic "primitive" functions, which are used in the implementation of the base `dylan` library and other low-level libraries to represent operations such as pointer equality, object memory allocation, numeric format conversion, or raw memory access. The HARP and LLVM compiler back-ends divide these primitive functions into three categories: those that are expanded in-line when they are called, those that generate a call to a run-time support routine, and those that generate a call to an implementation written in C. Many of these primitives are called with or return raw-typed values.

The following demonstrates the implementation of the arithmetic `+` operation on `<single-float>` using calls to primitive functions that unbox the `<single-float>` values as raw values, perform the addition as another raw value, and then box the result. DFMC optimizations can make use of the fact that the boxing and unboxing primitives are inverses of each other and allow them to cancel each other out when calls to this method are inlined.

```
define sealed inline method \+
    (x :: <single-float>, y :: <single-float>)
 => (z :: <single-float>)
  primitive-raw-as-single-float
    (primitive-single-float-add
       (primitive-single-float-as-raw(x),
        primitive-single-float-as-raw(y)))
end method;
```

Since the HARP back-end works primarily with word-size values, and the C back-end is able to take advantage of C type promotion rules, much of the Open Dylan code base was somewhat "loose" with the types of arguments to primitives. The LLVM intermediate language, being strictly typed, requires explicit integer widening and narrowing operations, so the translation of primitive calls frequently had to take this into account by adding automatic conversions. In some cases, explicit calls to cast primitives had to be added to convert between pointer and (integer) address types.

2.4 Run-Time Support Routine Generation

The definition of the Dylan language requires that a base library named `dylan` be provided so that programs can make use of the language's built-in macro syntax, classes, and functions by importing the `dylan` module that it exports. The Open Dylan implementation of this base library uses some "bootstrap" definitions found within the source of the compiler, but with the bulk of the library written as ordinary Dylan source files.

Included with the shared library generated for the `dylan` library are the run-time support routines needed by all libraries written in Dylan. These routines include implementations of the primitive functions, helper routines that implement function entry points, and interfaces with system facilities such as the garbage collector or arithmetic trap handling.

To build the run-time support routines we use a specialized generator tool based on many of the libraries that make up DFMC. This includes the reader, macroexpander, and enough of the modeling phases that the tool can process the source for the `dylan` library. This is desirable because many of the primitive functions and entry points need to reference classes and functions defined in the base library. Parsing the same source means that there is a single "source of truth" for these definitions.

Listing 1 illustrates a compile-time expander for a simple primitive function. The run-time support generator tool locates all of the run-time primitive definitions such as this one and executes them, causing the IR for support routines to be generated for output. When necessary, these routines can access the compile-time models for definitions found in the `dylan` library, giving information such as the size and layout of `<double-integer>` class instances. Calls to `ins` routines in the body of this definition insert LLVM basic blocks and instructions into function definitions, which are then written out (as bitcode) to be included in the run-time support.

It is often convenient for the run-time support routines written in C (such as the interface to operating system thread and synchronization primitives) to have access to the object layouts of a few select Dylan types. To facilitate this, the run-time support generator tool also writes out a C language header file with type definitions corresponding to the definitions in the `dylan` library.

2.5 Entry Points and Calling Conventions

The Open Dylan implementation of multi-method dispatch [1] has a number of different ways of generating code for function calls. When the exact method to be called is known to the compiler due to type inference and sealing rules, then DFMC can either inline the method call or invoke the method's internal entry point

Listing 1: Sample Run-Time Primitive Function Definition

```
define side-effect-free stateless dynamic-extent
  &runtime-primitive-descriptor primitive-wrap-unsigned-abstract-integer
    (x :: <raw-machine-word>) => (result :: <abstract-integer>);
  let word-bits = back-end-word-size(be) * 8;
  let maximum-fixed-integer
    = generic/-(generic/ash(1, word-bits - $dylan-tag-bits - 1), 1);

  // Check for greater than maximum-fixed-integer
  let cmp-above = ins--icmp-ugt(be, x, maximum-fixed-integer);
  ins--if (be, cmp-above)
    // Allocate and initialize a <double-integer> instance
    let class :: <&class> = dylan-value(#"<double-integer>");
    let double-integer = op--allocate-untraced(be, class);
    let low-slot-ptr
      = op--getslotptr(be, double-integer, class, #"%%double-integer-low");
    ins--store(be, x, low-slot-ptr);
    let high-slot-ptr
      = op--getslotptr(be, double-integer, class, #"%%double-integer-high");
    ins--store(be, 0, high-slot-ptr);
    ins--bitcast(be, double-integer, $llvm-object-pointer-type)
  ins--else
    // Tag as a fixed integer
    let shifted = ins--shl(be, integer-value, $dylan-tag-bits);
    let tagged = ins--or(be, shifted, $dylan-tag-integer);
    ins--inttoptr(be, tagged, $llvm-object-pointer-type)
  end ins--if;
end;
```

(IEP) directly. In this case, because the exact arity of the called function is known and any keyword arguments are already split into separate arguments, the call is able to use the LLVM `fastcc` calling convention, ensuring that as many arguments as possible are passed in registers.

The compiler generates an internal entry point for each compiled method. Following the formal arguments, artificial arguments representing the next applicable method(s) (used to implement `next-method` calls) and the `<method>` object itself (so that closed-over values stored in closure objects can be accessed). When these values are not used, the caller passes LLVM `undef` values, allowing the LLVM code generator to avoid emitting code to set them.

At the other extreme, generic functions about which nothing is known are called using the external entry point (XEP) convention. Because the number of arguments the function will accept is unknown, the caller passes the `<function>` object and the number of arguments at the head of the function arguments. Since the function arity is not guaranteed to match between the caller and the entry point, and because the entry point may need to collect `#rest` arguments or keyword arguments into a stack-allocated vector, LLVM `ccc` (C calling convention) is used.

The XEP entry points are pre-generated as part of the run-time support, using LLVM intermediate representation builders similar to those used for primitive functions. Several external entry point variants are available, for monomorphic (single-method) functions with or without optional arguments, for object slot accessors, and for generic functions that need to use the dispatch machinery. For each variant, 20 different variants are generated, one for each possible required argument arity. The compiler initializes the `xep` slot of each `<function>` object according to the function signature and number of methods.

When polymorphic method dispatch is known to be required, either at the call site or within the generic function's external entry point, then a decision tree of objects called *engine nodes* is built. Every engine node has an engine node entry point, also generated as part of the run-time support, most with multiple variants. For example, a discriminator engine point is responsible for checking the argument value of a single argument position and then choosing with which child node discrimination should continue. Some of these node types make use of dispatch code written in Dylan to make discrimination decisions before chaining to the next node's entry point. Engine node entry points are passed the engine node object, a reference to the dispatch "head", and all of the function's required arguments. These are also able to use the LLVM `fastcc` convention.

Once the applicable method is located, when it can, the leaf engine node will call into its internal entry point directly. For methods with keyword parameters or other optional arguments, the method entry point (MEP) is used instead. The MEP scans through the optional arguments and determines the values of each keyword argument (explicitly passed or defaulted) and then chains to the IEP with a tail-call.

2.6 Multiple Return Values

Like Common Lisp, the Dylan language allows functions to return zero or more values. Most standard calling conventions are not designed to support variable numbers of return values, making this a challenge to support. DFMC solves this with a vector of 64 values in thread-local storage (part of a Thread Environment Block structure). This storage is effectively a large register file, one that sometimes needs to be spilled to stack and restored. The primary (zeroth) value is returned in the main function return register, and (in the HARP and C back-ends) the return value count is placed in thread-local storage.

The LLVM back-end takes advantage of the fact that most current architecture ABIs support returning a structure of up to two words in registers. IEPs and the generated entry points return a type defined as:

```
%struct.mv = type { i8*, i8 }
```

placing the count of return values in the second word.

When a function returning multiple values is inlined, the intermediate return values may be available as local SSA values. Though the DFM representation does not distinguish between different kinds of multiple-value temporaries, the LLVM back-end makes an effort to ensure that a local SSA representation is used rather than forcing them to go through thread-local storage. These two strategies reduce the overhead of working with multi-value functions.

2.7 Foreign Function Interface

Support for interfacing with C and other languages has been a goal of most Dylan language implementations. Using the raw types to represent scalar values, along with facilities for modeling C structures, variables, and functions, Open Dylan is able to interoperate with a variety of C language application programming interfaces. Some support for calling Objective C methods is also available.

The LLVM tools' support for link-time optimization means that code from Open Dylan and other languages using the LLVM intermediate representation can be optimized or inlined across language barriers.

One challenge for the LLVM back-end is that while the LLVM intermediate representation is able to isolate front-end compilers from most of the specifics of the calling convention, it does not hide many of the details of passing and returning aggregate values such as structures and arrays. Providing platform-specific support for transforming aggregate argument and return values into function signatures that LLVM will support, just as the Clang compiler does, is an area for future work.

2.8 Non-Local Exit and Unwind-Protect

The Dylan language supports stack-unwinding non-local exit and forced cleanups during unwinding with the `block` construct and its `cleanup` clause.

The HARP code generator implements this by building a chain of bind-exit and unwind-protect frames on the stack. Non-local exits traverse this chain, executing unwind-protect cleanups and then restoring the final frame and instruction pointers.

The LLVM back-end reduces the overhead of constructing bind-exit frames by using the Itanium C++ ABI facilities for "zero-cost" exception handling, which optimizes for the case where the exception is not taken. When starting a bind-exit block, only a single word uniquely identifying the exit block needs to be stored into the bind-exit frame. Function calls within the block that might potentially cause an unwind contain branches to LLVM `landingpad` blocks that handle the unwind or cleanup. LLVM code generation builds tables that can locate these exception landing pads with the help of a Dylan-specific "personality function" included in the run-time support routines.

In addition to its low overhead in the usual case, this scheme also has the potential to interoperate well with C++ exception handling. The disadvantage, however, is that when non-local exits are frequent the cost can be quite high, mostly due to the overhead of locating the unwind tables using the system dynamic linker. The Gabriel `ctak` benchmark [4] is an example of a program that performs poorly with this scheme.

2.9 Thread-Local Storage

Open Dylan supports module variables that are thread-local. In the LLVM back-end, these are implemented using the `thread_local` storage model for global variables. This requires that when a new thread is started, thread-local variables in all loaded libraries be set to their initialization values, and that the storage locations be added to the set of roots known to the garbage collector. Dynamically loading a new library can also cause new thread-local storage to be added.

2.10 Debugging Support

The LLVM intermediate representation can express source-level debugging information, including source code locations, local and global variable locations, and types. This information is translated into platform-specific debug information, such as DWARF or Microsoft CodeView format. The Open Dylan LLVM back-end can generate this debug metadata, so that profilers, code coverage analyzers, and other tools can work transparently with Dylan libraries.

The LLVM project's LLDB debugger does require that it recognize a language type before it will make use of local variable and type information. The encoding of local variable and type metadata was designed to be compatible with that generated by the Clang compiler, and so we were able to submit a patch to the LLDB developers to explicitly support the Dylan language. Most debugging tasks can be handled using this support.

The Open Dylan programming environment includes a debugger that operates on a remote process. In addition to supporting breakpoints, stepping, and reading local variables, the debugger can also compile definitions and dynamically load them into the running remote process for execution, implementing a REPL for Dylan. Redefinitions are handled by compiling libraries in "loose" mode, which suppresses sealing and other kinds of optimizations, making the compiled code rely on dynamic typing and introspection operations. While this has long been supported on the Windows platform using the HARP code generator, we are currently expanding it to work with LLVM-generated code, integrating the LLVM debugger as a component to handle low-level and platform-specific debugger functionality.

3 BUILD SYSTEM INTEGRATION

The Open Dylan compiler uses the system linker to link compiler output into shared libraries; for the LLVM and C back-ends, external tools are also used for the final machine code generation task. Because different platforms, different toolchains, and even individual installations can vary widely, it is helpful to have a way to configure how Open Dylan invokes these external tools without requiring changes to the compiler. To allow this, the Open Dylan compiler implements an interpreted domain-specific language for toolchain builds, Christopher Seiwald's Jam [7], re-implemented in Dylan for ease of integration. Toolchain-specific build scripts provide Jam functions that can define build steps and establish dependency relationships between build products. When the Open Dylan compiler has finished compiling all libraries needed for a project, it invokes these build script functions to determine final code generation and linking steps. These steps are then executed in parallel as dependencies and CPU resources allow.

Taking this approach to configuring build tools has made it possible to support all three compiler back-ends and a variety of external toolchains on Windows, Linux, BSD, and macOS platforms with a minimum of effort.

4 RELATED WORK

The CLASP[5, 6] implementation of Common Lisp is also designed to compile a Lisp using the LLVM compiler infrastructure. As such, there are many similarities in implementation techniques, including the (external) calling convention, multiple-value return, and stack unwinding. Dylan does have the advantage of being able to take advantage of sealing information; for example, inlining of arithmetic and other frequent operations can be handled in a general way rather than with special-casing in the compiler.

5 CONCLUSION AND FUTURE WORK

The LLVM back-end to the Open Dylan compiler demonstrates a number of techniques for building a Lisp-family compiler using the general-purpose language implemetation infrastructure provided by the LLVM project.

Future work will likely include adding support for LLVM type-based alias analysis metadata, allowing the LLVM optimizer more flexibility in reordering memory operations when it can infer that different object pointers do not modify the same object. We also hope to adapt our code generator to make use of LLVM's garbage collection safepoints facility, which generates GC root stack map information for run-time garbage collection, and explicitly models at compile time the relocations that a garbage collector may perform. This would allow us to use a relocating garbage collector such as the Memory Pool System, expanding our GC options beyond our current use of conservative collectors such the Boehm-Demers-Weiser collector. Completion of these features will allow us to satisfy the original goals we set for the LLVM back-end.

6 ACKNOWLEDGEMENTS

The Open Dylan developers gratefully acknowledge Functional Objects for making the Functional Developer code base available as open-source software, as well as the many developers who have continued to improve it since.

REFERENCES

[1] Jonathan Bachrach and Glenn Burke. Partial dispatch: Optimizing dynamically-dispatched multimethod calls with compile-time types and runtime feedback. Technical report, 1999. URL https://people.eecs.berkeley.edu/~jrb/Projects/partial-dispatch.htm.

[2] Hans-J. Boehm. Simple garbage-collector-safety. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, PLDI '96, page 89–98, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 0897917952. doi: 10.1145/231379.231394. URL https://doi.org/10.1145/231379.231394.

[3] Richard Brooksby and Nicholas Barnes. The memory pool system: Thirty person-years of memory management development goes open source. Technical report, 2002. URL https://www.ravenbrook.com/project/mps/doc/2002-01-30/ismm2002-paper.

[4] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, Cambridge, Mass., 1985. ISBN 0-262-07093-6.

[5] Christian E. Schafmeister. Clasp - a common lisp that interoperates with c++ and uses the llvm backend. In *Proceedings of the 8th European Lisp Symposium*, ELS2015, pages 90–91. European Lisp Scientific Activities Association, 2015.

[6] Christian E. Schafmeister and Alex Wood. Clasp common lisp implementation and optimization. In *Proceedings of the 11th European Lisp Symposium*, ELS2018, pages 59–64, Marbella, Spain, 2018. European Lisp Scientific Activities Association. ISBN 9782955747421.

[7] Christopher Seiwald. Jam: Make(1) redux. In *Proceedings of the USENIX Applications Development Symposium Proceedings on USENIX Applications Development Symposium Proceedings*, UNIX'94, pages 79–88. USENIX Association, 1994. URL https://www.usenix.org/legacy/publications/library/proceedings/appdev94/seiwald.html.

[8] Andrew Shalit. *The Dylan Reference Manual.* Addison-Wesley Developer's Press, Reading, MA, 1996.