Design of an Efficient Lisp Bytecode Machine and Compiler

Alex Wood
ThirdLaw Molecular
Blue Bell, PA, USA
alex.wood@thirdlaw.tech

Charles Zhang
karlos@berkeley.edu

Christian Schafmeister
Temple University
Department of Chemistry
Philadelphia, PA, USA
meister@temple.edu

ABSTRACT

We present a new virtual machine for Common Lisp, focused on efficiency of compiled code as well as efficiency of the compilation process itself. An extended fix-up mechanism is used to apply some important optimizations without requiring an intermediate representation. The new system performs comparably to or better than existing systems with similar design goals.

CCS CONCEPTS

• **Software and its engineering → Compilers**; *Interpreters*; Just-in-time compilers.

KEYWORDS

virtual machine, compiler, bytecode

ACM Reference Format:
Alex Wood, Charles Zhang, and Christian Schafmeister. 2023. Design of an Efficient Lisp Bytecode Machine and Compiler. In *Proceedings of the 16th European Lisp Symposium (ELS '23).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.7818216

1 INTRODUCTION

We have developed a new virtual machine (VM) for Common Lisp, as well as a compiler targeting it. This design balances the axes of execution speed, compilation speed, and simplicity: the bytecode compiler runs quickly, but performs enough optimization during its one pass translation to let the code execute quickly as well. It is suitable for code that does not need to run often or which does not need special optimization, or as the first pass of a more heavy-duty optimizing compiler.

The compiler constitutes 1600 lines of Lisp, which was simple enough to be ported to 3000 lines of C++. The VM itself is only 500 lines of Lisp or 1500 of C++.

In tests, we have found the VM to meet our needs for speed and compilation speed, It outperforms CLISP's VM and performs comparably to ECL's, and we believe that further optimization work built on the general design here could make it even faster.

ELS '23, April 24–25 2023, Amsterdam, Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.7818216

2 MOTIVATION

Our original motivation was to simplify the bootstrapping procedure of Clasp Common Lisp[6], as well as to provide faster definitions of `eval`, `compile`, etc. Clasp builds itself up from a basic C++ core, which has historically meant having to implement the Lisp standard library in a simplified, "pidgin" Common Lisp. This is difficult to do, and has been a perennial source of bugs. A compiler for all of Common Lisp, simple enough to be written in C++, simplifies the situation considerably, as now at least all of CL's basic semantics are available for Lisp code.

This goal also led to the idea of writing a version of the bytecode compiler in portable Lisp. If the compiler is not reliant on the runtime, it can be run from other Lisps as a cross-compiler, simplifying the build process even further.

The other motivation was speed. Before the introduction of the bytecode compiler described here, Clasp had two ways to run Lisp code:[1] First, a conventionally designed interpreter written in C++. This could execute most code, but did so quite slowly, due to e.g. re-expanding macro forms every time they were encountered. Secondly, the primary compiler. This uses the Cleavir[2] Lisp compiler frontend, which then has its intermediate representation (IR) passed to LLVM[3], which produces machine code. As optimizing compilers, Cleavir and LLVM cons quite a lot of IR, and take time to simplify code, making Clasp's primary compiler noticeably slow.

Repeated profiling of compilation has shown that Clasp's compiler is slow in part because it is slow: It calls itself recursively, for example for `macrolet`. This means performing this slow compilation, even on code that only runs once, or only runs during compilation.

The design goals for a compiler are in conflict for code that runs once or not often. A more sophisticated compiler can generate code that runs more quickly, but the sophistication generally causes the compiler itself to take more time. Conversely, a simpler compiler can save compile time, but the generated code will take longer to run. We weighed these goals and developed a design that works well for our purposes. A simpler bytecode compiler neatly fits in the space between an interpreter and the optimizing compiler.

3 PREVIOUS WORK

Several Lisp implementations have used virtual machines, either as their primary means of executing Lisp code (CLISP) or to supplement a primary compiler (ECL, CMUCL).

CMUCL's bytecode compiler is primarily intended to reduce code size[4, § 5.9], although it does compile faster than the primary

[1] An additional compiler written in pidgin Lisp was used only during bootstrapping.
[2] https://s-expressionists.github.io/Cleavir/

(native code) compiler, Python. Furthermore, it uses multiple components from Python – the initial source-to-IR phase as well as the later assembler phase, neither of which were designed with portability or fast compilation in mind. Hence, CMUCL VM's design goals are distinct from ours, as reflected in its design, so it is not directly comparable to this work.

CLISP and ECL, on the other hand, designed their VMs with similar goals in mind to ours. CLISP's bytecode is its main means of evaluation[1, § 37.1]. ECL's exists to execute code without going through the more expensive process of C compilation. Both implementations' bytecode are commonly used for evaluation through `cl:eval`.

In the following, we will make frequent comparisons to the VM in CLISP, and occasionally to the VM used in ECL. Many of our design choices were informed by one of the authors' experience writing a compiler targeting the CLISP VM, where it was noticed that certain parts of the instruction set could be substantially simplified and/or made more efficient. In particular, we have substantially simplified the design of closures, instruction encoding and decoding, and non-local exits.

4 DESIGN

The bytecode is organized into bytecode modules, each of which contains a code vector and a literal vector. The code vector is an array of octets encoding bytecode instructions to be interpreted by the virtual machine. The literal vector is an array of literals referenced in the code with the instruction `CONSTANT`. Code for functions go in the same module if they are compiled together, as is the case with local functions defined with `flet`, `labels`, and `lambda`. This way, branches can relative-offset within the same code vector, as Lisp functions always `go` to or `return-from` exit points which are in functions in the same module.

In addition, we have bytecode functions and closure objects. Bytecode functions are `funcallable` objects which point at the appropriate entry point in the code vector. Bytecode closures are bytecode functions with an extra environment vector which bytecode can reference with the instruction `CLOSURE`. The representation of closures is explained in more detail in section 4.3.

Each instruction consists of an opcode byte, followed by zero or more operands. The number of operands depends on the opcode. Usually operands are encodable with a single byte, but if that isn't sufficient, the `LONG` prefix byte is placed before the opcode byte. The rationale for this encoding scheme is explained in section 4.4.

The virtual machine itself operates as a stack machine. Each function call reserves a fixed number (determined by the compiler) of slots on the stack, usually corresponding to lexical variables in the source code; these can be referenced by the instruction `REF`. On top of this, a function can use a variable amount of stack space, usually for temporaries resulting from the evaluation of intermediate expressions, but also to store multiple values. The virtual machine also contains a program counter and a multiple values vector. Each thread of execution has its own independent virtual machine with its own stack.

4.1 Interoperability

We ensured that the design of the virtual machine would allow bytecode functions to call and be called by non-bytecode functions. This allows the bytecode to be only one of several ways a Lisp implementation can evaluate code. In Clasp, bytecode function objects are equipped with a (shared) machine code entry point that invokes the VM, so that they can be called exactly like machine code compiled functions; similarly, in the portable Lisp version of the VM, bytecode functions are `funcallable-standard-object`, and dynamic and lexical exit tags can operate seamlessly. The bytecode does not have any special way of calling bytecode vs. non-bytecode functions, so a function being compiled to machine code does not necessitate its bytecoded callers to be recompiled.

4.2 Instruction set design

The design of the instruction set aims to translate the semantics of Common Lisp into a small, simple, orthogonal set of instructions in order to simplify the construction of virtual machines and compilers targeting the instructions set. At present, there are no instructions for inlining common functions (`car`, `aref`, etc.). There are 58 instructions, which plus the `LONG` prefix (below) means only 59 of 256 possible opcodes are used.

4.3 Flat closures

We designed the virtual machine to support a "flat closure" representation, as opposed to the more common "linked" closure representation used in many simple interpreters and bytecode compilers. This means the environment part of a closure is "flat": it is simply a vector of all values needed by the function and does not contain links to other environments.

However, one particularity in Lisp that complicates the flat closure strategy is the fact that variables can be mutated with `setq`. This requires closed-over variables that are `setq`'d to be represented with an indirect value cell. The cell is then closed over, allowing assignments to the variable to take effect in each flat closure closing over that variable, as references to the variable indirect through the value cell. The linked environment strategy does not require value cells, because an assignment can simply modify "the" environment vector containing the variable's binding directly.

Nonetheless, the flat closure approach has many desirable features:

(1) The representation is **safe for space**[5]: bindings that are lexically apparent but not actually used by any live closure are not kept alive. This is in contrast to the linked environment representation, where all bindings in an outer scope are kept alive even if the only bindings actually used by a live closure are in an inner scope, causing a memory leak.
(2) Closure variable access is constant-time, since it entails only one lookup in the flat environment. There is no need to crawl up through nested environments to find the one containing a given variable's value.
(3) The instructions used in the virtual machine to support flat closures are substantially simpler: We only need to support a single `CLOSURE` instruction taking a single operand (the index into the environment) to reference a closed-over variable or exit tag, and three operand-less instructions `MAKE-CELL`,

CELL-REF, and CELL-SET to support variable assignment by manipulating value cells. This is in contrast, for example, to the plethora of closure access and non-local exit instructions used in CLISP[1], which must specify at least two indices: one to specify the scope depth and one to index into the environment. Hence, flat closure instructions are more compact and take up less opcode space in comparison to equivalent instructions used to support linked closures.

ECL's bytecode system uses yet another approach. The virtual machine maintains the current lexical environment as a simple linked list of values at runtime.[2, § 4.6.3] When a closure is created, it simply includes the state of that list at the time the function is closed over. An advantage is that instructions only require a single operand to index into the environment, as with flat closures. However, it is not safe for space, as all bindings in the lexical scope are closed over and kept alive, like with the "linked" environment strategy. Additionally, accessing closure values is even slower than with the linked environment strategy used by CLISP. A variable access entails traversing the environment represented as a linked list, which takes linear time with respect to the number of total bindings in the environment. This is in contrast to linked closures, where a linked list *of only scopes* is traversed followed by a fast vector reference of the found environment.

We see then that from the perspective of run-time representation, the flat closure strategy is the clear winner: It allows for the simplest instructions, avoids memory leakage, and accesses values the fastest. However, its use is usually avoided in simpler compilers with fewer or no optimization passes. The problem is that, as explained, the flat closure strategy in Lisp sometimes requires indirect value cells. Avoiding value cells when possible is crucial for performance: choosing the value cell representation for a variable entails a cell allocation when the variable is bound, and an extra indirection for every reference and assignment to the variable. For a sophisticated compiler, a separate environment analysis pass can be done on IR to figure out exactly which variables are closed-over and mutable, so that the decision to use value cell representations is made before any code is emitted. In contrast, a one-pass compiler does not have that luxury: by the time the compiler recognizes that a variable is setq'd and closed over, it may have already emitted references or assignments to that variable. Thus, the only choice is to assume every variable needs a value cell, even those which end up being local and never setq'd!

Despite this issue, we were nonetheless able to choose the flat closure representation while solving the main drawback for our one-pass compilation strategy. The compiler optimistically emits instructions for the best (and most common) case scenario of not needing cells at all into the assembly, while noting fix-ups in the stream. During the final link step, by which time it is known whether the variable in question needs a value cell or not, the necessary indirection instructions are then emitted. The fix-up annotations are used to ensure no "holes" and "gaps" result in the final assembly. These steps are needed anyway to do necessary things like resolving labels for assembly, so the overall compilation strategy is not complicated. We describe the fix-up algorithm and the way we generalized the data-structures used to achieve this in more detail in section 4.6.

To illustrate the difference between the flat environment and linked environment strategies, consider the following closures:

```
(lambda (a b)
  (print b)
  (lambda (c)          ; env_1
    (setq c 9)
    (lambda (d e)      ; env_2
      (print e)
      (print c)
      (lambda ()       ; env_3
        (+ a d)))))
```

Figure 1: Linked closure strategy (used in CLISP)

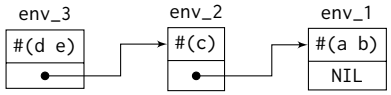
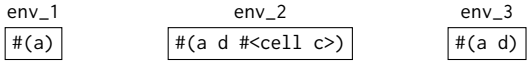


Figure 2: Flat closure strategy



With linked environments, we see that all bindings in the lexical environment are kept alive, even those which are never used by the innermost lambda. Hence, an unbounded amount of garbage could be retained.

4.4 long instruction prefix

Since we target a bytecode machine, it is important to make the actual encoding of instructions into bytes compact and fast. Because of the small number of opcodes, it is possible to represent all of them in a single byte.[3] However, operands for some instructions may exceed the size of a single byte, especially for control flow instructions.

We chose to use different-sized versions of each control flow instruction and a LONG prefix scheme for the other instructions: An instruction whose operand exceeds the size of one byte has the opcode prepended with a LONG prefix byte. This prefix indicates that the instruction's operand is instead two bytes wide, allowing indexing from 0-65535, which seems to be enough for all "reasonable" code. [4] Instructions with more than one operand that require a long version each have special interpretations as to which operand receives a wider interpretation according to what makes sense. This scheme allows the common case of few variables/constants/etc to be encoded compactly and decoded trivially, while only the rarer extended case entails overhead.

A simpler way of dealing with longer operands is to use a 16-bit code rather than an 8-bit code as we do here, so that all opcodes as well as all operands are 16 bits long rather than 8. This is the

[3]There is quite a bit of opcode space left over as well, which could be used for compressed instructions, as in CLISP. No decision has been made as to which compressions would be most profitiable yet.

[4]To exceed this limit for the ref instruction, for example, a function needs to bind 65536 lexical variables live at the same time.

approach taken by ECL. While it is simpler to encode and decode than our scheme, it nearly doubles code size in the common case.

An alternate scheme is used by CLISP's virtual machine. It uses a variable length encoding scheme for instruction operands: If the most significant bit of an operand byte is set, the operand continues into the next byte. This may save some space in cases where only one operand of several needs to be long, but considerably complicates encoding and decoding, and reduces the range of the simple one-byte case to 0-127.

This last aspect makes the long prefix scheme almost always more compact in practice compared to the variable-length encoding scheme, as functions typically have less than 255 live locals or 255 constants. In fact, for instructions with single operands, our scheme is more compact than the variable-length encoding scheme up to 383 locals or constants, since one extra byte is already needed for values 128-255 in the variable-length encoding scheme.

The design of the instruction set as a whole also makes the `LONG` encoding scheme more attractive. As alluded to already, the presence of many instructions with multiple operands can make the variable-length encoding option more uniform and simple for the decoder. In CLISP, a variable reference may require an instruction with several operands. For example, `LOADIC`, Load Indirect Closure, has four operands. Under the `LONG` prefix scheme, choosing to extend all operands would waste space if only one operand needs an extension, and on the other hand only selectively choosing which operand to extend would complicate the virtual machine decoding step, sacrificing speed. However, this is not a real drawback given the rest of our instruction set design: thanks in part to our choice of flat closure representation, all instructions but one take at most two operands.[5] If an instruction has only one operand, the variable-length encoding scheme's advantage is completely negated, and with two operands we are wasting one byte at most.

Control flow instructions have operands which represent signed-relative offsets into the code. As multi-byte relative offsets are very common and there are very few control flow instructions, they are not handled using the prefix scheme: Each branch instruction has 1-, 2-, and possibly 3-byte offset variants. This is much faster for branching and jumps than the variable length encoding scheme, while only having a small impact on opcode space given the small number of control flow instructions.

4.5 Non-local exits

One of the trickiest parts of implementing Common Lisp is the correct and efficient handling of the dynamic and lexical exit constructs, namely `catch`, `throw`, `block`, `return-from`, `tagbody`, `go`, and `unwind-protect`. Used within a function, lexical exits can usually be implemented simply by restoring the dynamic environment (containing e.g. special variable bindings and `unwind-protect` handlers) that was in effect before the execution of the corresponding `block` or `tagbody` form, and then doing a normal control transfer. However, lexical exit tags in Common Lisp can be closed over as well, although they still have only dynamic extent. Implementing a non-local exit to a closed over tag requires some coordination with the closure strategy: the non-local exit needs information about how to restore the dynamic environment from a different stack frame, and this information needs to be invalidated in safe code as well so that out-of-extent exits can be checked.

At first, we based our instructions for non-local lexical exits on the design of the instructions used in CLISP. In CLISP, there are separate instructions to handle `block` and `tagbody`: `BLOCK-OPEN` and `TAGBODY-OPEN` save the current dynamic environment *and* the program counter(s) to return to. `BLOCK-CLOSE` and `TAGBODY-CLOSE` invalidate the information required to restore the dynamic environment. `RETURN-FROM` and `GO` are each responsible for unwinding and restoring the saved dynamic environment and transferring control to the saved program counter. Finally, there are also `RETURN-FROM-I` and `GO-I` instructions which do the same thing but for saved dynamic environment information only accessible in an outer lexical environment. In particular, `TAGBODY-OPEN` takes a variable number of operands, one for each label corresponding to a `GO` tag, and one for the number of labels. The corresponding exit instructions then encode an index of which label to go to, which the virtual machine must then resolve to the actual label before actually jumping to it. ECL uses a similar scheme as well.

We moved away from this strategy because our decision to use flat closures and our decision to put all code compiled together into the same module enables a much simpler, more efficient, and more elegant design for the lexical exit instructions. First, flat closures allow us to simply use a `CLOSURE` instruction to reference any closed over dynamic environment information, so there is no need to have separate instructions to do closure indirection. Second, we can avoid closing over program counters altogether since the place to transfer control to is lexically known at each lexical exit. Because functions compiled together share the same code vector, we only need to encode a relative offset to the place to `return-from` or `go` to in the exit instruction, exactly as with an ordinary `JUMP` instruction. Finally, we see that after the above changes, `tagbody` and `go` can now be handled exactly the same as `block` and `return-from`, so we obliterate the distinction. We are then left with three simple fixed operand instructions:

(1) `ENTRY`: Allocates and pushes an object with information about the current dynamic environment onto the stack.
(2) `EXIT` l: Pops an object off the top of the stack and unwinds to the dynamic environment in that object, exiting to label l.
(3) `ENTRY-CLOSE`: Pops an object off the top of the stack and invalidates the dynamic environment information in that object, preventing future (out-of-extent) unwinds.

This is a clear improvement over the eight instructions used in CLISP, in both opcode space usage and performance. Note that the value returning semantics of `return-from` are handled orthogonally by instructions pertaining to multiple values.

However, when a lexical exit is to a tag defined in the same function, we can avoid consing an object which saves the dynamic environment altogether, since the actions needed to restore the dynamic environment can be determined statically. We can then also avoid the cost associated with dynamically unwinding the stack on exit, so that we can use a simple `JUMP` instruction instead. Most Lisp functions implicitly define blocks, which are usually unused or only used within the same function, so making this case fast is important. Allowing the compiler to recognize when doing such an

[5] The exception, `PARSE-KEY-ARGS`, is only used in handling lambda lists with `&key` arguments, and so is not a performance bottleneck.

optimization is possible is quite similar to the logic for eliding value cells for mutable closure bindings, so we again fold this optimization into the fix-up process described in the next section.

4.6 Fix-ups

Compilers and assemblers which emit to machine code or bytecode need to deal with the fix-up problem: How do you emit labels, which represent other locations in code, as offsets in the byte stream, before the position of those locations are known? The problem is further complicated by the fact that the labels instructions refer to can take up variable amounts of space, which can in turn affect the offsets of other labels! The label's offset can even be affected by its own size, in the case of backward branches.

The standard solution for sophisticated compilers and assemblers is to use fix-ups and resizer data structures. Fix-up annotations are accumulated when instructions are first emitted. These annotations include information such as best-case/worst-case size, current size, original position, and current position. As more instructions are emitted, the fix-ups are continually updated, until a final linking step creates the final vector of bytes.

We chose to use the optimistic version of this solution, where the smallest possible label sizes are assumed at first, as opposed to some assembly algorithms which work pessimistically, perhaps for faster convergence. Furthermore, because we compile and assemble in the same single pass, there is no rigid distinction between the two concepts, in contrast to many other compilers. This facilitates generalizing the fix-up data structures to handle other simple cases of "variable-length group of bytes". For example, fix-ups can adequately represent the decision to use a value cell or not, which in a heavier duty compiler is handled as part of optimizations on a distinct IR. This way we can avoid building and constructing separate IRs, and spending time in multiple passes. Because we need to emit and resolve labels using fix-ups anyway, we can save a significant amount of memory and time (as well as compiler complexity) by folding such optimizations into the fix-up step.

Most instructions can be emitted with fixed-size operands right off the bat. Conceptually, we can think of labels as a temporarily variable-length operand, and this is what fix-ups usually deal with. However, by generalizing the idea to variable-length sequences of bytes to be emitted, we can use fix-ups to emit or not emit entire instructions. When the compiler encounters a lexical variable or exit tag, it optimistically assumes that a cell is not needed, and generates bytecode that does not generate a cell. It also creates a "fixup" object, which is stored along with the bytecode being generated. Once the compiler finally resolves all fix-ups, it can now decide which variables or tags *do* need a cell, and treats this "variable-length group of bytes to be emitted" like a label and adjusts all other fixups by the appropriate number of bytes. The final link step, responsible for concatenating the bytecode for individual functions into a module, then copies the right bytes into the final module.

The generalized algorithm is also optimistic, so it always produces the best possible code. Labels are as small as possible, and no NOPs need to be left in the assembly stream to support the emission or non-emission of cell allocation and accesses.

5 RESULTS

5.1 Clasp build performance

Integration of the VM into Clasp allowed for Clasp's build procedure to be simplified substantially. Before the VM was used, a compiler in "pidgin" Common Lisp was interpreted, and this compiler was used to compile the Cleavir-based compiler. With the VM, the Cleavir-based compiler could be built directly by the C++ core. This simplification greatly improved build times: Clasp from just before the new VM build system was merged in took 150 minutes of CPU time to build, while the 2.0 release with the new system took 85 minutes.

5.2 VM Performance

In order to check the performance of the system, we used the `cl-bench` system[6], modified so as to avoid file compilation, and with extra machinery to test compile times. The results are displayed in Table 1. The benchmarks named with "CMP" prefixes represent the time taken to compile all the other benchmarking code in that group, five hundred times.

"VM" is the version of the system described here used in Clasp; that is, the C++ implementations of the bytecode VM and compiler. The results for CLISP and ECL were measured using their bytecode systems as well.

We also measure the performance of SBCL with its native compiler. SBCL, having an optimizing native code compiler, is not closely comparable to any of the three virtual machine systems exhibited here. It is included only to demonstrate the difference between such a compiler and VM systems generally. SBCL strongly outperforms the VMs on almost all runtime benchmarks, while exhibiting much longer compile times in the CMPARRAY and CMPCRC40 benchmarks.

Interpretation of these data is complicated by the fact that the virtual machines and compilers could not be compared in isolation. Each implementation's library influences its timing; a more tightly written `gensym` can influence macroexpansion and thus compile time, while other functions like `+` and `aref` play an important role in run times. Still, we believe these results indicate something about our system's efficiency.[7]

Our system outperforms CLISP in almost all tests. Comparison to ECL is more ambiguous: we do worse on some metrics, but better on others. Part of this is probably attributable to the differing implementations of the standard library functions rather than the operation of the virtual machines themselves, but this is difficult to determine as it is not possible to run ECL with Clasp's functions or vice versa.

It is clear that our system exhibits performance comparable to ECL and better than CLISP in most instances. Compile times, while still much better than those of a native code compiler, are generally worse than ECL's or CLISP's. While we believe the general organization of the compiler described here is efficient, more work could be done on optimizing its performance.

[6] https://gitlab.common-lisp.net/ansi-test/cl-bench

[7] While an implementation of our VM in portable Lisp exists, it cannot use low-level runtime tricks that C++ and C code integrated into these implementations can, and so is much slower. We do not compare it here.

Benchmark	VM	Clisp	ECL	SBCL
CMPARRAY	0.560	0.426	0.222	22.659
1DARRAYS	0.254	0.648	0.232	0.0108
2DARRAYS	9.535	27.527	7.330	0.0765
3DARRAYS	21.484	64.128	15.408	0.281
BITVECTORS	0.0118	0.566	0.467	0.0184
STRINGS	0.136	2.865	1.250	0.512
STRINGS/ADJ	13.987	41.333	20.253	0.613
STRING-CONCAT	30.738	*	42.021	5.940
SEARCH-SEQ	3.997	5.945	1.978	0.383
CMPCRC40	0.0637	0.0839	0.0310	1.111
CRC40	5.279	21.927	12.377	0.152
CMPGABRIEL	8.555	3.670	3.400	61.627
BOYER	*	*	172.960	0.543
BROWSE	1.091	2.181	1.149	0.0359
DDERIV	1.444	3.909	2.629	0.0626
DERIV	2.908	3.146	2.311	0.0493
DESTRUCTIVE	1.315	4.322	1.188	0.0401
DIV2-TEST1	0.972	3.778	0.924	0.0274
DIV2-TEST2	2.558	3.180	2.197	0.0420
FFT	5.210	12.973	3.619	0.0185
FRPOLY/FIX	1.701	5.336	4.155	0.0547
FRPOLY/BIG	1.928	5.974	5.033	0.148
FRPOLY/FLOAT	1.699	5.716	3.964	0.0825
PUZZLE	8.417	28.327	6.134	0.101
TAK	0.404	2.380	1.861	0.0122
CTAK	1.998	0.800	0.621	0.0100
TRTAK	0.401	2.398	1.886	0.0122
TAKL	2.941	11.180	11.633	0.0840
STAK	*	5.917	0.378	0.0523
FPRINT/UGLY	0.481	0.117	0.179	0.627
FPRINT/PRETTY	5.354	0.530	2.876	0.212
TRIANGLE	1.541	5.367	1.850	0.0518

Table 1: Benchmark results (times in seconds). "*" indicates that the Lisp could not run the benchmark due to control stack exhaustion.

6 EXAMPLE OF COMPILED CODE

6.1 Basic code

To illustrate how the bytecode looks in practice, here is what our system compiles the Common Lisp function

```
(lambda (x)
  (let ((y 5))
    (print y)
    (lambda () (+ y x))))
```

into:

```
check-arg-count-= 1
bind-required-args 1
const 0 ; '5
set 1
```

First the function checks that it was provided exactly one argument. Then it binds that one argument into the register file at positions starting at 0 and below 1, i.e. just 0.

To bind y, the constant 5 is pushed to the stack, then popped from the stack and placed in register 1.

```
fdefinition 1 ; 'PRINT
ref 1
call 1
```

This is the `(print y)` call. The definition of `print` is looked up and called on the value we just placed in register 1, i.e. y.

```
ref 1
ref 0
make-closure 2 ; '#<BYTECODE-FUNCTION {100C2D803B}>
```

A closure over x and y is allocated for `(lambda () (+ y x))`, and pushed to the stack. Note that 2 is just the index in the constants vector for the closure's code; the number of values being closed over is not encoded in the instruction, since that information is encoded in the function object.

```
pop
return
```

The closure just allocated is popped from the stack into the multiple values vector. The multiple values are then returned.

6.2 Non-local exit example

We can demonstrate our non-local exit and dynamic environment instructions with the bytecode for a loop. This code binds a dynamic variable, calls a global function, then calls another global function with a closure that can initiate a non-local exit. If this closure is called, the loop exits. Otherwise, the dynamic variable binding is undone, and then the loop repeats.

```
(lambda (x y)
  (block nil
    (tagbody
     loop
       (f)
       (let ((*z* x))
         (g (lambda () (return y)))
         (go loop)))))
```

The outer function compiles to the following:

```
check-arg-count-EQ 2
bind-required-args 2
ref 1
make-cell
set 1
entry 2
save-sp 3
```

In the prologue, the outer function processes its arguments and makes a cell for y, which is referenced from the closure. Then, it both creates and saves an "entry" object (containing information to restore the dynamic environment at that point in time) at location 2 and saves the stack pointer at location 3. The entry is used for restoring the dynamic environment in a real non-local exit, while the stack pointer is used when no function boundary needs to be crossed, since the action of restoring the dynamic environment can be done statically.

```
L0:
  fdefinition 'F
  call 0
  ref 0
  special-bind '*Z*
  fdefinition 'G
  ref 1
  ref 2
  make-closure '#<BYTECODE-FUNCTION {1004100D1B}>
  call 1
  unbind
  restore-sp 3
  jump-8 L0
```

This is the body of the loop. The variable is bound by `special-bind`, and the closure is created and passed to `g`. Note that the closure references both stack slots 1 and 2. 1 is the cell for `y`, but 2 is the entry created by the earlier `entry` instruction.

After the call, the loop continues. Rather than execute a true non-local exit with dynamic unwinding, the compiler has statically determined what part of the dynamic environment needs to be undone - the special variable binding - and inserts an instruction to do that. `restore-sp` then sets the stack pointer back to where it was, and `jump-8 L0` transfers control back to the top of the loop.

```
  unbind
  nil
  pop
```

These instructions would be executed when the tagbody form's end is reached normally. This cannot occur in the example code, but our compiler is not smart enough to determine this.

```
L1:
  entry-close
  return
```

Finally, upon an abnormal return, the non-local entry object for the `block` is invalidated, and the outer function finally returns.

The label `L1` is not used in this function's code; it is referenced in the inner closure's code, but the label is still assembled into a relative offset due to the fact that functions compiled together share the same bytecode vector. This means the destination does not need to be recorded in the entry object, or determined dynamically by the unwinder. The code of the lambda is disassembled here:

```
  check-arg-count-LE 0
  closure 0
  cell-ref
  pop
  closure 1
  exit-8 L1
  return
```

The function loads `y` from its cell, in location 0 of the closure vector, and prepares to return it. Then, it loads the entry object for the non-local exit from closure slot 1, and `exit-8 L1` transfers control to label `L1` of the outer function using the information in that object. `exit-8` is responsible for dynamically determining what actions need to be taken to unwind correctly; in that case that will include unbinding `*z*`, and also undoing any dynamic binding established by the function `g`, which cannot be statically determined by the compiler. If the entry object is found to have been already invalidated, the unwinder throws an appropriate error.

7 FUTURE DIRECTIONS

7.1 Trucler integration

The Lisp implementation of the compiler uses the Trucler environment protocol, a CLOS based update and expansion of the environment-related operators described in CLTL2.[7] This allows it to access functions and macros from the host implementation's global environment, or to use an alternate first-class global environment. First class environments facilitate using the VM for cross-compilation or for sandboxing - for example, untrusted "script" code could be byte-compiled in an environment in which dangerous operators like `(setf fdefinition)` and `read` are not available, or have restricted definitions.

However, the compiler uses its own environment structures internally rather than host environments, so host definitions of complex macros like 'loop' that use `cl:macroexpand` do not work. If the compiler was rewritten to use Trucler internally rather than its own environments, and if Trucler support on the Lisp implementation is sufficient, it would be possible for the VM to be smoothly usable within an implementation as a drop-in replacement for the implementation's `cl:compile` and/or `cl:eval`.

7.2 File compilation

The bytecode compiler itself works as `cl:compile` or `cl:eval`, not implementing the complex semantics of file compilation. However, it can be run in such a way that it doesn't actually produce a module or functions, or resolve 'load-time-value', etc., and instead simply returns enough information to construct a module. This can be used by a suitable file compilation mechanism.

We are working on such a file compiler, and accompanying FASL format. The ultimate goal of this project, besides providing a drop-in `cl:compile-file` implementation, is to allow one Lisp implementation to produce *portable* FASLs that can then be loaded successfully in a completely different Lisp implementation. Our main motivation is to use this for bootstrapping a primitive Lisp with FASLs produced by a full Lisp, but we believe it could be more generally useful.

7.3 Conversion to IR

The bytecode produced by the compiler is a fairly direct reflection of the source code, but with macros expanded, and no internal reliance on environment information. These properties make it suitable for conversion to IR for an optimizing compiler. We are planning to write a system to convert the bytecode into Cleavir's IR. This would allow the bytecode compiler to act as a frontend to a smarter compiler.

One change that would need to be made is having the compiler record more information about the code. For example, it would be important to record source information for debugging, and various declarations such as of types for optimization (as the bytecode compiler is too simple to use them itself).

With this system set up, it would be possible to use the VM to facilitate just-in-time compilation of Lisp code. Code could be at

first compiled quickly into bytecode, and then only if necessary, compiled further into optimized machine code.

In conjunction with a portable FASL format, this would allow the bytecode to serve as a portable post-`read` code interchange format, somewhat like Java VM bytecode. Optimizations depend on the specific nature of the target machine, such as those relating to arithmetic, can be done by a specific implementation. There would be a separation of concerns between the frontend and the backend of the language system, and it would be possible to distribute code without either dumping an entire monolithic Lisp image or relying on the end user to deal with all the complexity of compiling Lisp source.

8 CONCLUSION

Our bytecode system can compile Common Lisp code quickly, and run it with reasonable efficiency. Performance is comparable or superior to that of other Lisp virtual machines. The fix-up mechanism allows the compiler to apply several important optimizations without requiring a complex and slower IR.

REFERENCES

[1] Bruno Haible, Michael Stoll, and Sam Steingold. Implementation notes for gnu clisp, 2010. URL https://clisp.sourceforge.io/impnotes/index.html. Last accessed 14 February 2023.

[2] Daniel Kochmański, Marius Gerbershagen, Tomasz Kurcz, and Juan Jose Garcia Ripoll. Ecl manual, 2016. URL https://ecl.common-lisp.dev/static/manual/. Last accessed 18 February 2023.

[3] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *CGO*, pages 75–88, San Jose, CA, USA, Mar 2004.

[4] Robert A. MacLachlan. Cmucl user's manual, 2016. URL https://cmucl.org/downloads/doc/cmu-user/. Last accessed 18 February 2023.

[5] Zoe Paraskevopoulou and Andrew W. Appel. Closure conversion is safe for space. *Proc. ACM Program. Lang.*, 3(ICFP), jul 2019. doi: 10.1145/3341687. URL https://doi.org/10.1145/3341687.

[6] Christian A Schafmeister and Alex Wood. Clasp common lisp implementation and optimization. In *Proceedings of the 11th European Lisp Symposium on European Lisp Symposium*, pages 59–64, 2018.

[7] Robert Strandh and Irène Durand. A clos protocol for lexical environments. In *Proceedings of the 15th European Lisp Symposium*, ELS '22, pages 20–26, 2022.