Proceedings of the

13th European Lisp Symposium

Special Focus on Compilers

Streaming online from Zürich, Switzerland
`https://www.twitch.tv/elsconf`
April 27 – 28, 2020

In cooperation with ACM

Ioanna M. Dimitriou H. (ed.)

ISBN-13: 978-2-9557474-3-8
ISSN: 2677-3465

Cover image by ESA/NASA.
*This image was taken from on board the ISS by ESA astronaut Alexander Gerst, who named it "**Europe at night**".*

Corrected version of April 30, 2020.

Preface

Message from the Programme Chair

Hello, Lispers!

Welcome to the 13^{th}th European Lisp Symposium, online for the first time.

It is a difficult time for all of us, and for that we must show solidarity. It's very sad we don't get to meet in person and indulge in our favourite topic, Lisps. For that we gather virtually, to interact in the only way currently possible. And what an effort this has been from everyone, thank you!

I am grateful to the speakers, for preparing video presentations and being present in the chat to answer questions. To each author for their valuable contributions. To every programme committee member for their time, work, and expertise on reviewing the papers and demos, often giving valuable feedback. I sincerely hope to see all rejected papers of this year build on this feedback, and get published in a future ELS. A big thank you to Nicolas Hafner for quickly adapting his role as local chair and organising a virtual location for us to meet. And of course, to the steering committee, and to Didier Verna, the driving force behind the symposium. I am honoured.

On the programme.

When Didier asked me to chair the ELS 2020, my favourite conference and annual must, I was on my way to A Coruña in Spain to meet my new Igalian coworkers. I had just joined the Compilers Team of Igalia, so this was an easy answer. I told Didier, yes of course and there will be a special focus on compilers.

I am thrilled that most papers we received were directly or indirectly about compilers. We now have invited talks and papers touching almost all focus points on the list of the call for papers. We have practical and theoretical compiler techniques, compiler passes, showcasing of compilers, code generation, compiler optimization, JIT compilers, and even compiler compilers.

But that's not all. The topics go beyond just compilers, beyond even Lisps, and all the way to the topics of privilege and empathy.

I hope you enjoy the virtual meeting and this collection of papers and demos, and I'm looking forward to sitting with you all again in person.

Bonn, April 27, 2020 Ioanna M. Dimitriou H.

Message from the Local Chair

When I attended my first ELS at Goldsmith's in London and met with many similarly minded and enthusiastic people for the first time, a new world opened itself up to me. I had never been surrounded by peers like that before, and I've never missed an ELS since. It was also then that I met with Robert Strandh, who immediately planted the idea in my head to organise an ELS in my home town Zürich sometime.

This idea stuck around in my mind since then, until I finally decided to take a step forward and ask Didier about it. What followed over the past two years were many frantic days of meeting with other people to seek help, looking all over for a suitable venue, organising the dinner, helping out with the website and registration, and so forth. It was a lot of work, and I'm quite sad that the in person conference had to be cancelled due to the pandemic.

Still, it would be even more of a shame to wallow in the burden of this situation. Instead, I'd like to thank all the presenters for their hard work preparing their papers, and I'd like to thank everyone involved in the organisation of the conference for helping with reviews and all the work that goes into the conference as a whole. I hope that, even without a physical presence, we can still create an enjoyable conference for people online and bring some much needed positivity and excitement to the Lisp community.

Finally, I have not fully given up on an ELS in Zürich, and shall try again some other year!

Zürich, April 27, 2020 Nicolas Hafner

Organization

Programme Chair

- Ioanna M. Dimitriou H. – Igalia, Spain/Germany

Local Chair

- Nicolas Hafner – Shirakumo.org, Switzerland

Programme Committee

- Andy Wingo – Igalia, Spain/France
- Asumu Takikawa – Igalia, Spain/USA
- Charlotte Herzeel – Imec, ExaScience Lab, Belgium
- Christophe Rhodes – Google, UK
- Irène Durand – University of Bordeaux, France
- Jim Newton – EPITA Research Lab, France
- Kent Pitman – HyperMeta, USA
- Leonie Dreschler–Fischer – University of Hamburg, Germany
- Marco Heisig – FAU Erlangen-Nürnberg, Germany
- Mark Evenson – Not.org, Austria
- Max Rottenkolber – Interstellar Ventures, Germany
- Metin Evrim Ulu – Middle East Technical University, Turkey
- Paulo Matos – Igalia, Spain/Germany
- Robert Goldman – SIFT, USA
- Robert Strandh – University of Bordeaux, France
- Sky Hester – Independent consultant, USA

Sponsors

We gratefully acknowledge the support given to the 13thth European Lisp Symposium by the following sponsors:

Igalia, S.L.
Bugallal Marchesi, 22, 1º
15008 A Coruña
Galicia, Spain
`www.igalia.com`

RavenPack
Centro de Negocios Oasis Oficina 4
Urb. Villa Parra, Ctra de Cadiz km 176
29602 Marbella Málaga Spain
`www.ravenpack.com`

EPITA
14-16 rue Voltaire
FR-94276 Le Kremlin-Bicêtre CEDEX
France
`www.epita.fr`

SwissDev Jobs
Technoparkstrasse 1
8005 Zürich
Switzerland
`https://swissdevjobs.ch`

Invited Contributions

The Nanopass Framework as a Nanopass Compiler

Andy Keep, Facebook, USA

The nanopass framework is a domain specific language for defining compilers that provides two basic syntactic forms: define-language and define-pass. The define-language form defines the grammar for an intermediate representation and can either define the full language or extend an existing language. Language forms are represented using Scheme records and a parser and unparser can be constructed from the language definition to move between S-expression and nanopass language form representations. The define-pass form defines procedures for operating over these language forms, based on the specified input and output languages.

In addition to these two basic forms, a number of small tools for interacting with languages exist, including tools for extracting the define-language syntactic form for a language, making it easier to see the full language when it was defined as an extension, along with tools for differencing two languages to produce the language extension form, pruning unreachable nonterminals, and defining a procedure for counting nodes in a language form for a given language.

These tools are helpful, but we can imagine wanting more tools, for instance a tool to generate a equivalence procedure over language forms or a tool to generate a procedure for computing a histogram of nonterminal node types in a language form. Unfortunately, each tool must be written with knowledge of the internals of the nanopass framework. What if the nanopass framework instead provided an API for writing these extensions? What if the define-language and define-pass forms, were defined as nanopass languages that could be treated like other nanopass languages? How much of the nanopass framework could be written using the nanopass framework? This talk will explore this experiment.

Andy Keep is a Core System Software Engineer at Facebook AR/VR, working in the area of programming language implementation. He is also one of the maintainers of Chez Scheme, a commercial-grade Scheme compiler that was open sourced by Cisco Systems, Inc., where he was previously a Senior Software Engineer, in 2016. Andy's work with Chez Scheme started as part of his Ph.D. work at Indiana University, where his dissertation, "A Nanopass Framework for Commercial Compiler Development", was focused on replacing the original Chez Scheme compiler with a nanopass compiler. R. Kent Dybvig, the creator of Chez Scheme, was Andy's advisor and aided greatly in this work. Andy continues to work on compiler and related run time systems, and has an active interest in educating the next generation of compiler writers.

Workshop: Mixing Mutability into the Nanopass Framework

Andy Keep, Facebook, USA

Languages defined using the nanopass framework are represented using immutable Scheme records[1], however, it can be useful to have mutable cells with the terminals of a language form. For instance, the Chez Scheme compiler represents each variable as a single Scheme record instance. This means the binding site and all use sites for a given variable all use the same record instance to represent that variable. The variable record contains mutable fields which allow information from variable uses to be visible at the binding site and vice versa. For instance, variable uses can report whether they are referenced, multiply referenced, or assigned to the variable binding site, or the variable binding site can record information needed at the use sites for a variable without constructing an environment within the pass.

This workshop will give a brief introduction to the nanopass framework using an example compiler for a small subset of Scheme, and then look at how this technique is used for converting assigned variables and computing free variable sets in lambda expressions.

Privilege as a technical debt

Amr Abdelwahab, Tourlane.com, Germany

Do you believe political correctness and empathy are buzzwords that limit society rather than contribute to its advancement? Do you think talking about topics like diversity quotas and privilege doesn't make much sense and you would rather spend this time talking about the latest in technology?

In this talk I would like to take the chance to try and add the missing contexts to such terms and arguments, moreover, I will try to go through various examples on how it can impact your product from a very pragmatic perspective.

An African Egyptian native who crossed continents to work with his passion in digital environments. Amr's interests span technology, tech-communities, politics and politics in tech, all enriched through various software engineering roles in Egypt, Hungary, and Germany.

[1]In addition to immutable records, standard (and hence mutable) Scheme lists are used for for representing lists within a language form, but the expectation is that these lists will not be mutated.

On ECL, the Embeddable Common Lisp

Daniel Kochmański, TurtleWare, Poland

Embeddable Common Lisp is a Common Lisp implementation with historical roots dating back to 1985 when Kyoto Common Lisp was released as an open source project by Taichi Yuasa and Masami Hagiya. It is one of the first Common Lisp implementations predating the ANSI standard (CLtL1) and it has influenced its final version. First developed by academia, then by volunteers from the free software community, it still thrives as one of many actively developed Common Lisp implementations.

Thanks to a portable and small core it is possible to embed ECL in other applications as a shared library. This property enables Common Lisp programmers to develop their applications and plugins as an extension to existing software and to use Common Lisp software on platforms like Android and iOS. Executables and libraries built with ECL are small and suitable for writing utilities and libraries used by applications outside of the Common Lisp world.

Maintaining and improving a Common Lisp implementation is a challenging and fun task with many opportunities to learn about software and compilers. During this presentation I'll talk about the past, the present, and the future of ECL. I'll discuss its heritage, then move to its current architecture with its flaws and advantages, and I will finish with my plans for further development.

Daniel Kochmański is a Common Lisp and C hacker and free software proponent. Interested in cognitive science, software development and user experience modelling. Founder of TurtleWare – a consultancy company located in Poland specialized in Common Lisp and embedded systems. In free time passionately reads books and occasionally plays on guitar.

Programme overview

Monday, 27.4.2020

09:15	**Welcome**
09:30–10:30	Andy Keep - The Nanopass Framework as a Nanopass Compiler (ELS keynote)
10:30–11:00	**Coffee break**
11:00–11:30	Robert Strandh - Omnipresent and low-overhead application debugging
11:30–12:00	Frédéric Hamel, and Marc Feeley - An R7RS Compatible Module System for Termite Scheme
12:00–12:30	Marco Heisig - Sealable Metaobjects for Common Lisp
12:30–14:30	**Lunch**
14:30–15:00	Irène Anne Durand - Bidirectional leveled enumerators
15:00–15:30	Max Rottenkolber - Later Binding: Just-in-Time Compilation of a Younger Dynamic Programming Language
15:30–16:00	**Coffee break**
16:00–16:30	Peter Housel - LLVM Code Generation for Open Dylan
16:30–17:00	Jonathan Godbout - Indexing Common Lisp with Kythe
17:00–17:30	Lightning talks

Tuesday, 28.4.2020

09:00–10:00	Andy Keep - Workshop: Mixing Mutability into the Nanopass Framework (ELS invited workshop)
10:00–10:30	Rajesh Jayaprakash - Partial Evaluation Based CPS Transformation: An Implementation Case Study
10:30–11:00	**Coffee Break**
11:00–11:30	Robert Strandh - Representing method combinations
11:30–12:00	Andrea Corallo, Luca Nassi, and Nicola Manca - Bringing GNU Emacs to native code
12:00–12:30	Andrew Lawson - RavenPack in the time of COVID-19 (ELS sponsor)
12:30–14:30	**Lunch**
14:30–15:30	Amr Abdelwahab - Privilege as a technical debt (ELS keynote)
15:30–16:00	**Coffee Break**
16:00–16:30	Alan Dipert - JACL: A Common Lisp for Developing Single-Page Web Applications
16:30–17:00	Marco Antoniotti - Why You Cannot (Yet) Write an "Interval Arithmetic" Library in Common Lisp – or: Hammering Some Sense into :ieee-floating-point
17:00–18:00	Daniel Kochmański - On ECL, the Embeddable Common Lisp (ELS keynote)
18:00–18:30	Lightning talks
18:30	**Conference end**

Paper only

João Távora - A portable, annotation-based, visual stepper for Common Lisp

Technical papers

Peer-reviewed technical papers, with novel content.
The papers appear in the order they appear in the programme.

Omnipresent and low-overhead application debugging

Robert Strandh
robert.strandh@u-bordeaux.fr
LaBRI, University of Bordeaux
Talence, France

ABSTRACT

The state of the art in application debugging in free Common Lisp implementations leaves much to be desired. In many cases, only a backtrace inspector is provided, allowing the application programmer to examine the control stack when an unhandled error is signaled. Most such implementations do not allow the programmer to set breakpoints (unconditional or conditional), nor to step the program after it has stopped.

Furthermore, even debugging tools such as tracing or manually calling `break` are typically very limited in that they do not allow the programmer to trace or break in important system functions such as `make-instance` or `shared-initialize`, simply because these tools impact all callers, including those of the system itself, such as the compiler.

In this paper, we suggest a technique that solves most of these problems. The main idea is to have a *debugger thread* debug one or more *application threads*, with all these threads running in the same image. Tracing and setting breakpoints have an effect only in the debugged thread so that system code running in other threads is not impacted. We discuss several advantages of this technique, and in particular how it can make debugging *omnipresent*, i.e., not requiring recompilation to get its benefits. We describe how to achieve this advantage while keeping the *overhead* low for threads that are not being debugged.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Runtime environments**;

KEYWORDS

CLOS, Common Lisp, Compilation, Debugging

ACM Reference Format:
Robert Strandh. 2020. Omnipresent and low-overhead application debugging. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3747548

1 INTRODUCTION

Good debugging tools are essential for the productivity of software developers. In this paper, we are concerned with *application programmers* as opposed to *system programmers.* The difference, in the context of this paper, is that the techniques that we suggest are not adapted to debugging the system itself, such as the compiler. Instead, throughout this paper, we assume that, as far as the application programmer is concerned, the semantics of the code generated by the compiler corresponds to that of the source code.

In this paper, we are mainly concerned with Common Lisp [1] implementations distributed as so-called FLOSS, i.e., "Free, Libre, and Open Source Software". While some such implementations are excellent in terms of the quality of the code that the compiler generates, most leave much to be desired when it comes to debugging tools available to the application programmer.

Perhaps the most advanced development environment available to application programmers using FLOSS Common Lisp implementations is the one that consists of GNU Emacs[1] (also [4] [6]) with SLIME[2]. Many application programmers consider this development environment to be outstanding. Some even believe that it is one of the best, no matter the programming language under consideration.

However, although this environment does a fairly good job with exploiting the features of the Common Lisp implementations that it supports, limitations of those implementations severely restrict what the application programmer can do. In particular, most of these implementations have only very limited facilities for setting breakpoints (unconditional or conditional) and for stepping.

Even in implementations that allow the programmer to set a breakpoint in some code, the places where it is allowed are necessarily restricted, given how breakpoints are typically implemented. The reason for this restriction is that such a breakpoint would be visible to all callers of the code in which the breakpoint is set. When these callers include important system code such as the compiler, or perhaps the debugger itself, setting such a breakpoint would make the entire system useless. This restriction typically applies also to tracing. Most Common Lisp implementations would either not allow for the programmer to trace important system functions such as `make-instance` or `shared-initialize`, or these functions would be rendered useless with any such attempt. The reason is of course that these functions would be called by the system itself, so that output would be drowned in traces of calls that are unimportant to the application programmer.

In this paper, we suggest a technique that solves these problems. The key features of this technique is that breakpoints and traces take effect only in a thread that is executed

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27–28 2020, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.3747548

[1]https://www.gnu.org/software/emacs/manual/emacs.html
[2]https://common-lisp.net/project/slime/doc/html/

from a special *debugger thread*. Thus, even though a function might contain a breakpoint, when that function is called as a normal part of an application, the breakpoint will not have any effect. Only when that function is called (directly or indirectly) from the special debugger thread is the breakpoint visible.

The technique presented in this paper is yet to be implemented. We have, however, conducted experiments that suggest that it is entirely viable. We plan to make it the default technique used in our system SICL (see Section 3), currently under development.

Throughout this paper, we use the term *user* to mean the person operating the debugger or some debugging-related facility, so as to distinguish this person from the *application programmer*, by which we mean the author of the code being debugged. The two can obviously be one and the same person in two different roles.

2 PREVIOUS WORK

2.1 Process-based debugging

With systems like UNIX, debugging is usually performed as an interaction between two *processes*. The debugger runs in one process and the application in another process. For a breakpoint, the code of the application is modified by the debugger so that the application sends a signal to the debugger when the breakpoint has been reached. For this purpose, the debugger maps the code pages of the application as *copy on write* (or COW). With this technique, instances of the same application that are not executed under the control of the debugger are not affected by the modified code. In particular, with this technique, any application can be debugged, including the debuggger itself.

Some FLOSS Common Lisp implementations suggest the use of this debugging technique, by means of some existing debugger such as GDB[3] (also [7]), in order to set breakpoints. In particular, the CCL (See Section 2.3.) documentation mentions that this technique is possible, and it is also the technique recommended for ECL (See Section 2.4.).

2.2 SBCL

The SBCL Common Lisp implementation[4] has a breakpoint facility. Given a code location, a breakpoint can be set, which results in the code being modified at that location, so that an arbitrary function (given to the constructor of the breakpoint) is called when execution reaches that location.

The only feature that uses the breakpoint facility is `trace`. Furthermore, it is hard for the user to take advantage of the breakpoint facility directly, given that a function such as `make-breakpoint` requires an argument indicating the code location. We are unaware of the existence of a debugger for SBCL that can use the breakpoint facility.

SBCL also has a *single stepper* that the manual says is "instrumentation based". As it turns out, the kind of instrumentation used by the stepper is not that of the breakpoint facility. Instead, when the value of the `debug` optimization quality is sufficiently high compared to the values of other optimization qualities, the compiler inserts code that signals conditions that are specific to the stepper.

2.3 CCL

The CCL Common Lisp implementation[5] does not have the concept of breakpoints.

The CCL `trace` command uses *encapsulation*, meaning that the association between the *name* of a function and the function object itself is altered so that it contains a *wrapper* function that displays the information requested and that calls the original function to accomplish its task.

Currently, CCL does not have a working single stepper.

2.4 ECL

The ECL Common Lisp implementation[6] does not have the concept of breakpoints, so an external debugger such as GDB has to be used for breakpoints. ECL does have a special instruction type in the bytecode virtual machine that is used for stepping.

The `trace` facility uses encapsulation.

2.5 Clasp

The Clasp Common Lisp implementation[7] does not have the concept of breakpoints, nor does it have a stepper. The `trace` facility uses encapsulation.

2.6 LispWorks

The LispWorks Common Lisp implementation[8] provides breakpoints. Breakpoints can be set either from the stepper or from the editor. The first time a breakpoint is set in a definition, the source code of the defining form is re-evaluated with additional annotations that provide information for the stepper.

When a breakpoint has been set, it is active no matter how the code containing it was called. If that code was called outside the stepper, the stepper is automatically started. Thus, breakpoints provide the essential mechanism for the stepper.

Since setting a breakpoint requires access to the source code, and since the source code of the system itself is not supplied, the user can not set breakpoints in system code.

The `trace` facility in LispWorks is accomplished through encapsulation.

2.7 Allegro

The Allegro Common Lisp implementation[9] has the most complete and most sophisticated implementation of breakpoints of all the Common Lisp implementations we investigated.

[3]https://sourceware.org/gdb/current/onlinedocs/gdb/
[4]http://www.sbcl.org/
[5]https://ccl.clozure.com/
[6]https://common-lisp.net/project/ecl/
[7]https://github.com/clasp-developers/clasp
[8]http://www.lispworks.com/products/lispworks.html
[9]https://franz.com/products/allegro-common-lisp/

High-level debugging features are based on a low-level breakpoint mechanism described in a paper by Duane Rettig [5]. In many respects, the low-level mechanism is similar to the one used by UNIX-style debugging, in that it replaces the ordinary machine instruction by one that will *trap*, and thus cause the operating system to send a signal to the Lisp process. However, their mechanism differs in a significant way from the one used by UNIX-style debugging, in that it allows the breakpoint to be handled by the same operating-system process that contains it, with very few exceptions.

Same-process debugging is made possible by their mechanism that allows existing breakpoints to be *installed* or not. Only installed breakpoints correspond to replaced instructions, whereas uninstalled breakpoints are remembered by the system and can be installed according to the kind of debugging that the higher-level tool implements. The clever aspect of their mechanism is to have the signal handler start its action by uninstalling all breakpoints. Thus, even if a breakpoint exists in some system code that is also used by the debugger, once the debugger is entered, the breakpoint is no longer active. Had the breakpoints remained installed, issuing commands inside the debugger might have invoked some code with a breakpoint, thereby halting the execution of the debugger itself.

This mechanism allows for instruction-level stepping in a way similar to what is possible in separate-process UNIX-style debuggers. Just as with UNIX-style debugging, any instruction can be replaced by a different one that will trap to the debugger. As a result, it is possible to execute one instruction at a time by simply trapping after each instruction. Crucially, however, this mechanism is then used to build high-level tools such as source-level debuggers, steppers, etc.

3 MAIN FEATURES OF THE SICL SYSTEM

SICL[10] is a system that is written entirely in Common Lisp. Thanks to the particular bootstrapping technique [2] that we developed for SICL, most parts of the system can use the entire language for their implementation. We thus avoid having to keep track of what particular subset of the language is allowed for the implementation of each module.

We have multiple objectives for the SICL system, including exemplary maintainability and good performance. However, the most important objective in the context of this paper is *support for excellent debugging tools.* We think it is going to be difficult to adapt existing Common Lisp implementations to support the kind of application debugging that we consider essential for good programmer productivity.

Another main objective of the SICL system is *safety.* In this context, by this term we mean that the system must always be in a coherent internal state. When a system becomes unsafe, it may *crash*, or (worse) silently produce the wrong answer.

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

There are many situations described in the Common Lisp standard that have undefined or unspecified behavior, such as:

(1) Many times when a standard function is called with some argument that is not of the type indicated by the corresponding dictionary entry in the Common Lisp standard document, the behavior is undefined, allowing the implementation to avoid potentially costly tests for exceptional situations.
(2) When a non-local transfer is attempted to an exit point that has been "abandoned", the standard does not require this situation to be detected, making it possible for the system to crash or (worse) give the wrong result.
(3) When some entity is declared `dynamic-extent`, but some necessary condition for this declaration is violated, the implementation is again not required to detect the problem, again potentially resulting in a crash or an incorrect computation.

Fortunately, most potential situations of this type are not taken advantage of by a typical Common Lisp implementation in order to improve performance, but some are. We think that the spirit of the Common Lisp standard is to have a safe language, and that many of these situations of undefined or unspecified behavior exist only to avoid significantly more work for the system maintainers at the time the standard was established.

For that reason, in the SICL system, we do not intend to take advantage of these situations to make the system unsafe for the purpose of better performance, even though we might have to work somewhat harder in order to maintain good performance in all situations.

Many debugging techniques can make the system unsafe. For example, if the debugger allows the user to arbitrarily change the value of a lexical variable, the new value might violate some assumption made by the compiler for the program point in question. Such a violation is very likely to make the system unsafe. The work described in this paper is designed to keep the system safe.

4 OUR TECHNIQUE

4.1 Two versions of every function body

We provide two different versions of every function body[11]. One version, called the *ordinary* version, and the other one is called the *debugging* version. Each version is provided as a separate *entry point* for the function[12]. The two versions are *similar* (but not identical) copies of the entire function body.

By including both versions in the same function, we make it unnecessary for the application programmer to recompile the code with higher debug settings when it is desirable to have more debugging information than what the compiler would generate by default.

[11]This idea was suggested by Michael Raskin.
[12]This idea was suggested by Frode Fjeld.

The ordinary function body is compiled using every typical optimization technique used by a good compiler, including:

- constant folding,
- dead code elimination,
- common sub-expression elimination,
- loop-invariant code motion,
- induction-variable optimization,
- elimination of in-scope dead variables, and
- tail-call optimization.

Some of these optimization techniques are essential for high-performance code, but many of them can make it significantly harder for the user to understand what the program is doing:

- Common sub-expression elimination and similar techniques for redundancy elimination may make it impossible to set a breakpoint in some part of the code, simply because that code has been eliminated by the compiler.
- When a variable is used for the last time, the compiler typically reuses the place that it occupies for other purposes, even though the variable may still be in scope. This optimization makes it impossible for the user to examine the value of a variable that has been eliminated. A user with a poor understanding of compiler-optimization techniques will find the result surprising.
- Loop-invariant code motion results in code being moved from inside a loop to outside it. Any attempt by the application programmer to set a breakpoint in such code will fail.
- Induction-variable optimization will eliminate or replace variables in source code by others that are more beneficial for the performance of the computation, again making it harder for the user to debug the code.
- Tail-call optimization exploits the fact that the stack needs to reflect only the *future* of the computation, but the *past* can be omitted. For the purpose of debugging, the past of the computation provides essential information to the developer.

To avoid many of these inconveniences to the user, the debugging version of the function body is compiled in a way that makes the code somewhat slower, but much more friendly for the purpose of debugging. Some of the optimization techniques cited above will not be performed at all, or only in a less "aggressive" form. Messages from the compiler (such as when dead code is eliminated) are emitted based on the compilation of the normal version of the function body so that the maximum amount feedback is obtained.

Notice that the existence of the two versions of the function body makes it usually unnecessary for the programmer to explicitly indicate values of the `debug` optimize quality. In fact, forcing the programmer to set this value is often a major inconvenience, since it typically requires the programmer to choose between debugging convenience and execution speed. The `speed` and `compilation-speed` optimize qualities may influence compilation of the the normal version of the body, but not the debugging version. The debugging version will in general be much faster to generate because of the absence of most optimization passes.

Furthermore, the debugging version of the code is compiled so that a small routine is called immediately before and immediately after the execution corresponding to the evaluation of a form in the source code. In order to determine whether a breakpoint is present at that particular location in the source code, this routine performs a query of a table managed by the debugger. While the details of how this table is implemented might evolve, here we give one example of such an implementation, thereby arguing that performing a query is not overly expensive in terms of performance.

As an example of implementation of this table, it might be split into two sub-tables called the *summary table* and the *breakpoint table*. Both these tables are managed by the debugger, in that actions on the part of the user may alter their contents. The application consults these tables, directly or indirectly, to determine whether a breakpoint is present.

The purpose of the summary table is to provide a quick test that almost always indicates that no breakpoint is present. Thus, the summary table is a fixed-size bit vector. The size will typically be a small power of 2, for instance 1024 which represents a modest 16 64-bit words on a modern processor. The application routine computes the value of the program counter modulo the size of the table in order to determine an index. If the entry in the table contains 0, then there is definitely not a breakpoint present at the source location in question. Since there are typically only a modest number of breakpoints in a program, most of the time, the entry will contain a 0, making the routine return immediately, and normal form evaluation to continue. The debugging version of the function body accesses this table early on in order to create a reference to it in a lexical variable. This lexical variable is subject to register allocation as usual.

If the entry in the summary table contains 1, then there is a breakpoint at *some* value of the program counter that, when taken modulo the size of the table, has a breakpoint present. If this is the case, then the routine consults the breakpoint table. In other words, the summary table acts as a Bloom filter [3], in that false positives are possible, but false negatives are not. The size of the table determines the probability of a false positive.

The breakpoint table is a hash table in which the keys are values of the program counter[13] and the values are objects that the debugger uses in order to determine information about the breakpoint in question. When the routine finds that a breakpoint is present at the current source location, it informs the debugger. Details of the communication between the application thread and the debugger are discussed in Section 4.2.

In the ordinary version of the function body, when a function call is made, the caller uses the entry point of the callee corresponding to the ordinary version of the body of the callee. In the debugging version of the function body, on the

[13]In implementations where code can be moved by the garbage collector, this table must be re-hashed after a collection. The tentative decision for SICL is to have all code at fixed locations.

other hand, when a function call is made, the caller uses the entry point of the callee corresponding to the debugging version of the body of the callee. This mechanism thus automatically propagates the information about debugging throughout the call chain.

4.2 Communication between the debugger and the application

Debuggers in UNIX systems have full access to the address space of the application, including the stack and the lexical variables. A UNIX debugger can therefore modify any lexical variable and then continue the execution of the application. Such manipulations may very well violate some of the assumptions made by the compiler for a particular code fragment. For example, if the code contains a test for the value of a numeric variable, the compiler may make different assumptions about this value in the two different branches executed as a result of the test.

Allowing a debugger to make arbitrary modifications to lexical variables, let alone to any memory location, in a Common Lisp application program will defeat any attempts at making the system safe, and safety is one of the objectives of the SICL system as expressed in Section 3. We must therefore come up with a different communication protocol that keeps the system safe.

Our design contains two essential elements for this purpose:

(1) The debugger consists of an interactive application with a *command loop*. An iteration of this command loop can of course be prompted by a user interaction. However, when the application detects a breakpoint by querying the tables described in Section 4.1, it injects a command into the command loop of the debugger, triggering the execution of code in the debugger to handle the breakpoint.
(2) A shared *queue* is used to send messages from the debugger to the application. This queue has a *semaphore* associated with it.
(3) Once the application has informed the debugger about a breakpoint, it attempts to *dequeue* the next message on the queue. If the queue is empty, the application automatically *waits* on the associated semaphore, until the debugger issues an *enqueue* operation with instructions for the application.

The debugger is in charge of taking into account the commands issued by the user. When the user indicates that a certain action should be performed at a particular place in the source code, the debugger populates the two tables mentioned in Section 4.1, and records the particular action the user desires, for example:

- The user might indicate that the application should stop and wait for further actions by the user, after the user has examined the state of application data. In this case, the debugger records this desire in the breakpoint table. When the application then reaches the breakpoint in question, the debugger waits for further user action in its command loop.

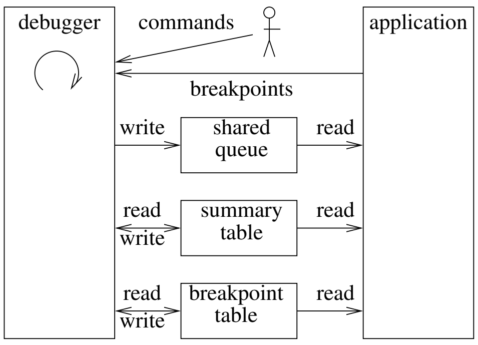


Figure 1: Communication between user, application, and debugger.

- After the user has examined the state of application data as a result of the application having stopped, the user can issue a command that makes the application continue normal execution. The debugger then immediately sends a message to this effect to the application.
- The user might indicate that a *trace message* should be printed without stopping the application. Then, when the breakpoint is reached, the debugger displays the message to the user and sends a message to the application to continue execution.
- The user can also indicate that the execution of the application should be *stepped* in one of several different ways:
  – *next*: Execution stops at the next possible program point.
  – *in*: Execution stops at the beginning of a function being called.
  – *out*: The remaining sub-forms of the form containing the current breakpoint are evaluated, and execution stops immediately after the evaluation of that form.
  – *over*: When a breakpoint is reached that is located immediately before a form is evaluated, the form is evaluated and execution stops immediately after this evaluation.
  – *finish*: Execution of the currently executing function terminates, and stops in the calling function immediately after the call.

  The debugger then sets one or more *volatile breakpoints* (i.e., breakpoints that will be removed once reached) at source locations corresponding to the type of stepping required. It then instructs the application to continue execution as usual.

To allow for the user to examine the state of the application, when the application thread detects a breakpoint, the command it injects into the debugger command loop contains a complete list of live local variables and their values, as well as of special variables bound in the application thread.

Since we intend to provide debugger commands for examining and modifying application data, we must make sure that any such manipulation on the part of the user preserves the integrity of the application.

In particular, any assumptions made by the compiler about the structure or type of some lexical variable must be impossible to violate through the modification of the value of a lexical variable. We obtain this property by making sure that the compiler does not propagate any information about the structure or type of lexical variables between program points that admit breakpoints. Thus, any run-time manipulation that requires this structure or type to be known must be preceded by an explicit test, and the compiler does not generate code that admits a breakpoint between the test and the manipulation.

4.3 Debugger commands available to the user

We have an embryonic implementation of an interactive debugger, called Clordane.[14] We use the McCLIM library for writing graphic user interfaces as a basis for this tool. Currently, Clordane can show the source code of an application (one source file at a time) and an indication of the place of a breakpoint. The application being debugged is then compiled with the SICL compiler, generating high-level intermediate representation (HIR). The HIR code is then interpreted by a small program running in a host Common Lisp implementation.

The communication protocol described in Section 4.2 has been implemented and works to our satisfaction, but only a small subset of interactions have been implemented so far.

We think that the following commands must be implemented in a fully featured debugger:

- The user should be able to point to a location in the source code to indicate a particular action to be taken at that point:
  - Stop the execution of the application and wait for further actions.
  - Print a trace message, possibly containing the values of live variables, and then continue the execution.

  It should be possible to make the action conditional, based on some arbitrary expression to be evaluated in the debugger thread. This expression can contain references to live variables in the application.
- When the application is stopped, the user should be able to examine live variables, and (in some cases, with restrictions) modify their values.
- Also, when the application is stopped, the application programmer should be able to issue one of several types of *stepping* commands, implicitly indicating the next location for the application to stop.

[14]The name Clordane is a deliberate misspelling of "Chlordane" which is a pesticide that was banned in most countries in the 1980s. The misspelling was designed to suggest the Common Lisp language and to make answers by search engines less cluttered.

5 BENEFITS OF OUR TECHNIQUE

Our technique differs both from the tradition of debugging in UNIX-type systems and from the tradition used in FLOSS Common Lisp systems.

5.1 Difference compared to UNIX-like systems

Whereas UNIX-like systems typically run the debugger in a different *process* from that (or those) of the application, with our technique we run both the debugger and the application in the same process.

The main advantage of this organization is that communication between the debugger and the application is greatly simplified. There is no need for a wire protocol to encode and decode data in the form of sequences of octets, simply because with a single process, the address space is shared between the debugger and the application. Instead, we can send data in the form of arbitrarily complex data structures between the two.

5.2 Difference compared to most FLOSS Common Lisp systems

Most FLOSS Common Lisp implementations have a history that started before multi-threading was common. As a result, features such as breakpoints and tracing are often implemented as modifications to the code.

For example, in SBCL, the user can choose to *trace* a function in two different ways. One way is by means of *encapsulation*, meaning that the function is not modified, and instead wrapped in a small routine that then replaces the function as associated with the function *name*. The function being traced is not modified. The other way is by means of a breakpoint; that is, the code of the function being traced *is* modified.

However, in both cases, every caller of the function being traced is affected, barring a caller that is in possession of the function object itself, rather than its name. As a result, it is very likely impractical to trace system functions that may be used internally by the system. For example, tracing `find` or `position` (if at all possible) is likely to generate so much information from callers that are irrelevant to the user as to make the information impossible to exploit. And tracing functions such as `print`, `format`, or `write` would be entirely impossible, since the trace output would very likely call these functions in order to generate the output information meant for the user.

With our suggested technique, tracing a function does not create an encapsulation and does not modify the code of the function. Instead, the existing code communicates with the debugger, and the debugger, running in a different thread, is in charge of displaying information to the user. As a direct consequence, there are no restrictions such as those indicated above. The only possible restriction has to do with inlining, though it may very well turn out to be possible to propagate debugging information with inlining, thereby making it possible to trace, or to set breakpoints in any function such as `car` or `+`. However, it may turn out that the inclusion of debugging code in such low-level functions would be prohibitive in terms of performance of code run under the control of the debugger.

Finally, a significant advantage to our technique is that the application programmer does not have to choose between compiling the code for debugging or for performance. In most existing systems, in order for it to be possible to benefit from all the debugging information possible, the programmer has to compile the code with a combination of values of the existing `debug` qualities that is not optimal for performance. This limitation means that it is often necessary to recompile the application for one of the two purposes. With the technique presented in this paper, no such choice is required, since both versions of the application are always available.

6 DISADVANTAGES OF OUR TECHNIQUE

Perhaps the most obvious disadvantage of our technique is that the size of the code will more than double. The debugging version of the function body must implement the same functionality as the non-debugging version, but in addition to that functionality, it must also contain code for communicating with the debugger. Furthermore, since fewer optimizations are applied to the non-debugging version, even without the code for communication, the debugging version would be larger than the non-debugging version.

While the additional code will impact the memory footprint of the system, we do not think it will have any negative influence on caching. The two versions of the body are kept separate, and the same version is typically executed repeatedly.

Feedback on draft versions of this paper indicate that many readers are worried about the possibility of the behavior of the different versions of the function body described in Section 4.1. This worry is based on experience, as this situation is common, especially with implementations of programming languages other than Common Lisp. As we see it, there are two possible causes for such difference in behavior:

(1) A defect in the compiler can result in native code that does not correspond to the semantics of the source code, and the resulting code can be different in the different versions.
(2) The compiler is exploiting undefined or unspecified behavior, probably in order to improve performance of the resulting code, and it exploits such behavior in different ways in the two different versions.

We briefly addressed the first cause in Section 1, by specifically targeting application programming, assuming that the compiler is essentially free of defects.

The second cause was addressed in Section 3, where we indicated that the SICL system does not intend to take advantage of undefined situations that will introduce any such differences in application behavior.

Finally, the fact that the technique proposed in this paper is incompatible with the way most Common Lisp systems work, makes it unlikely that existing systems will be able to use it. We are convinced, however, that our technique will represent a major advantage in terms of productivity for the application programmer.

7 CONCLUSIONS AND FUTURE WORK

We believe that a decent development environment for Common Lisp must include a very feature-full debugger for application programs, and we firmly believe that the best way of accomplishing such an environment is to have the debugger execute in the same process as the application, but to have the application react to debugging operations only when executed under the control of the debugger. Here "debugging operations" include the tracing of function calls, since that mechanism is based on the general breakpoint mechanism described in this paper.

While it may seem like a valid objection that the application programmer, as a result of detecting a defect, must rerun the application from the debugger in order to benefit from the features proposed by our technique, we disagree with this objection. We simply think that there is usually no valid reason to run the application outside the debugger during the development phase. The only exceptions we can think of would be applications with extreme performance requirements, or applications where response time is part of the specification. While our technique requires the programmer to *execute* the application code from within the debugger, *developing* the code can be done using any tool.

The validity of the technique described in this paper has been somewhat verified, in that we have an embryonic implementation of an interactive debugger, and an implementation of a small subset of the communication protocol between the application and the debugger. However, our current environment does not allow us to verify the ultimate performance of application code run under the control of the debugger. We firmly believe that the performance loss for code without any breakpoints in it will be acceptable, and that the additional cost when breakpoints are involved is to be expected by the application programmer.

We have not yet implemented the two-version body idea (See Section 4.1.) in SICL, mainly because the only implementation of SICL that currently works is the interpreter for intermediate code. This interpreter was written for bootstrapping purposes only and performance is not an issue. For that reason, we have included only the debugging version of function bodies.

In addition to producing a native version of the SICL system, we also need additional work on the Clordane debugger, and on the additional components of the communication protocol between the debugger and the application.

We think it would be desirable for existing Common Lisp implementations to incorporate the technique described in this paper, so as to allow for a much more complete development environment for users. However, we are convinced that the modifications that would be required to those implementations would be prohibitive in terms of time invested by maintainers. For that reason, we unfortunately do not hold out much hope for this possibility, and we intend to concentrate our efforts on making our technique work for the SICL system.

8 ACKNOWLEDGMENTS

We would like to thank the following people for providing information about breakpoints, tracing, and stepping in various Common Lisp implementations: Martin Simmons (LispWorks), Michał "phoe" Herda (CCL), Alex Wood (Clasp), Daniel Kochmański (ECL), Duane Rettig (Allegro).

We would like to thank Frode Fjeld for giving feedback on early versions of this paper, and for suggesting the use of multiple entry points for each function.

We would like to thank Michael Raskin for suggesting that two versions of each function should be provided, thereby making it unnecessary for the programmer to choose the level of debugging.

REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp*. American National Standards Institute, 1994.
[2] *Bootstrapping Common Lisp using Common Lisp*, April 2019. Zenodo. doi: 10.5281/zenodo.2634314. URL https://doi.org/10.5281/zenodo.2634314.
[3] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. ISSN 0001-0782. doi: 10.1145/362686.362692. URL https://doi.org/10.1145/362686.362692.
[4] Debra Cameron, James Elliott, Marc Loy, Eric S. Raymond, and Bill Rosenblatt. *Learning Gnu Emacs, Third Edition*. O'Reilly Media, Inc., 3 edition, 2004. ISBN 9780596006488.
[5] D. Rettig. Instruction-Level Breakpoint Stepping in the Current Process. Technical report, Franz Inc., Oakland, California, USA, 1999.
[6] Richard Stallman. *GNU Emacs Manual*. GNU Press, Cambridge, MA, USA, 2018.
[7] Richard Stallman, Roland Pesch, Stan Shebs, et al. *Debugging with GDB*. GNU Press, Cambridge, MA, USA, 2020. ISBN 978-0-9831592-3-0.

An R7RS Compatible Module System for Termite Scheme

Frédéric Hamel
Université de Montréal
Montréal, Québec, Canada
frederic.hamel@umontreal.ca

Marc Feeley
Université de Montréal
Montréal, Québec, Canada
feeley@iro.umontreal.ca

ABSTRACT

The Termite Scheme language is an existing extension of Gambit Scheme that has features well suited for programming heterogeneous distributed systems using a message passing style. The language supports sending messages containing procedures and continuations, which simplifies migrating tasks between nodes during their execution.

A longstanding issue with the original implementation of Termite is that compiled procedures and continuations can only be sent to other nodes if the compiled code is already loaded in the program receiving the message. This is tedious to arrange in the typical case, and hard or impossible for hot code updates which are an important use case (updating a service without interrupting its execution).

Our work has implemented a solution to this problem: an R7RS compatible module system that automates the distribution of compiled code. The module system uses a version control system to manage module versions and provide a way to distribute code from network accessible repositories. Modules are identified uniquely using the repository location and version number. This allows multiple versions of the same module to coexist in a program, an essential feature to support hot code updates.

We explain the implementation of our module system and how it solves various issues related to Termite Scheme and programming distributed systems. Through an experimental evaluation we have observed speed improvements for RPC of close to one order of magnitude.

CCS CONCEPTS

• **Software and its engineering** → **Modules / packages**; **Modules / packages**; **Distributed systems organizing principles**;

KEYWORDS

Distributed systems, Concurrency, Remote Procedure Call, Mobile code, Modules, Scheme

ACM Reference Format:
Frédéric Hamel and Marc Feeley. 2020. An R7RS Compatible Module System for Termite Scheme. In *Proceedings of the 13th Europeen Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.5281/zenodo.3742443

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27 28, 2020, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.3742443

1 INTRODUCTION

Distributed systems consist of a set of interconnected computational nodes. Nodes interact by sending and receiving messages from other nodes over a communication network. Each node may have a special purpose or there can be more or less duplication of their function. The World Wide Web is a notable example that is good to keep in mind to understand some of the issues. It has server and client nodes, they typically don't run the same server and client programs, and the nodes are not centrally managed.

The implementation of a distributed system consists of developing the programs installed on the nodes that perform the coordination of the nodes' actions with those of other nodes. In a sense, the set of the nodes' programs constitutes a global program that must be correct. The challenging development issues we consider in this paper are the following:

- **RPC:** How is a remote procedure call (RPC) implemented when the program in the sending and receiving nodes haven't been designed together?
- **Code update:** How is a node's program updated when a bug is fixed or a better version is available?
- **Task migration:** How is a service moved to a new node when the underlying platform must be changed (operating system update, hardware upgrades, reboot, ...)?
- **Continuous operation:** How are service interruptions avoided in the above situations?

The Termite Scheme language [19] has been designed as an extension of Gambit Scheme [14] to simplify programming distributed systems and provide solutions to these issues. It borrows concepts from the Erlang programming language [4], but using Scheme syntax and semantics. A particularly interesting feature in our context, not found in Erlang, is the ability to send messages containing continuations.

The Gambit system on top of which Termite is implemented offers many features useful to program distributed systems. It generates portable C code that can be compiled and run on any OS and architecture (32/64 bits, little/big endian, ...). The serialization and deserialization of objects are independent of their machine representation, allowing the transmission of most objects between nodes of a distributed system with different architectures. In particular, procedures and continuations can be serialized. Moreover interpreted and compiled code can be freely mixed in the same program. The serializable procedures allow using a higher-order programming style across nodes, which is useful for implementing RPC. The serializable continuations allow capturing the state of a process and sending it to another node to resume it there, which is useful for code update and task migration.

Unfortunately, the original implementation of Termite has shortcomings when serializing closures and continuations. When the receiving node has no knowledge of the code it receives, it must run the code interpreted which is typically much slower than if it was compiled. In this case, essentially the source code file's AST is serialized, making messages larger. Moreover, this large structure will be sent again if another instance of the closure is sent. When the code is compiled, the messages are compact, but the code must be available in compiled form on the receiving node. This requires a tedious and error-prone setting up of the nodes' programs that would be unsustainable in the context of large independently evolving distributed systems, such as the Web, and difficult to use for RPC, code update and task migration.

Our work aims to allow sending code between nodes that both run native compiled executables. This is not a simple task considering nodes may have different operating systems and architectures, such as ARM, i386 or x86_64, so compiled procedures cannot, in general, be sent as machine code.

To reach this goal our approach delegates to the receiving node the compilation of the modules. Code can be transmitted between machines of different architectures without the usage of cross-compilers, which can be challenging to setup robustly. The code will work on all platforms supported by Gambit such as Linux, macOS, Windows, etc.

An essential property of the module system is that modules must have a globally unique name that includes their version. This allows the coexistence of multiple versions of modules in a program, a situation occuring when a node is updated with an improved version of its code without interruption. Our approach benefits from the use of a version control system to manage the versions of modules in a disciplined way.

The implementation of code migration has been done in different programming languages such as Java [18], JavaScript [22], Tcl [20], Erlang [16, 21, 23], and Scheme [3, 9, 13, 17]. Our work distinguishes itself from these efforts in allowing compiled code to be migrated transparently between nodes regardless of their architecture and operating system, and without the destination node having prior knowledge of the code.

Section 2 is a brief tutorial of the Termite Scheme language. Section 3 explains existing features of Gambit used in the implementation of our module system, which is the subject of Section 4. In Section 5 we evaluate the performance experimentally. Finally, Section 6 discusses related work.

2 TERMITE SCHEME LANGUAGE

Termite [19] applications consist of multiple Termite `nodes` exchanging data in a message-passing style similar to Erlang [15]. A `node` is an abstraction of a computing device that is distinct from the physical nodes (machines) of the distributed system. In practice a `node` corresponds to an operating system process and the physical nodes of the distributed system may contain a single or multiple Termite `nodes`.

Within each `node` there are multiple running threads. Threads are uniquely identified across the distributed system with a `upid` that indicates its location (i.e. a `node` and a sequence number within that `node`).

Each `node` is identified with an IP address and port. The procedure `make-node` is the constructor of `node` identifiers. The `node-init` procedure starts the `node`'s TCP server and registers built-in services (*spawner*, *linker*, *publisher*, etc.) The `node`'s TCP server allows clients to connect to it remotely. Without this the `node` would not be visible on the network.

Services in Termite are nothing more than threads receiving messages in a loop, performing pattern matching on them and executing actions according to the result of the matching. A service is created with the `spawn` procedure which takes a thunk to be executed and an optional local name for the service and returns a thread object. Each thread has a mailbox that buffers the messages it receives. By default services created with `spawn` are only visible to the threads in the current `node`. A service can be published globally (to other nodes) by the procedure `publish-service` that registers the thread object under a specific name in a dictionary within the *publisher* service. That service resolves services by name.

Communication between `nodes` is performed with the following procedures:

- `(! dest msg)`
- `(? [timeout [default]])`
- `(?? pred? [timeout [default]])`
- `(!? dest msg [timeout [default]])`

The procedure `!` sends the message *`msg`* to the mailbox of the *`dest`* thread. The procedure `?` waits for a message to appear in the mailbox and retrieves it, with the optional parameter *`timeout`* specifying how long to wait before returning the *`default`* value. If no default value is given and the timeout expires, an error is raised. The procedure `??` is similar to `?` but filters the received message using the predicate *`pred?`*. For the common "send request then receive response" communication pattern there is the `!?` procedure that is a composition of `!` and `?`, which also automatically adds a unique tag to the messages to match the response with the request.

A message can also be received with the `recv` form which pattern matches the message. This form is typically used to dispatch the messages in the implementation of a service.

The following example shows a typical use of these forms to implement a local service computing the square of a number and a client requesting to square 5 (both in the same `node`):

```
(define square-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag 'square x)  ;; message pattern
           (! from (list tag (* x x))))
          (msg
           (warning "Ignored message " msg)))
        (loop)))))

(!? square-server (list 'square 5)) ;; => 25
```

```
;; ping.scm (running on node1)
(declare (block))
(import (termite))

(define pong-server
  (remote-service 'pong-server node2))

(define new-server
  (spawn
    (lambda ()
      (let loop () ;; code that will be migrated
        (recv
          ((from tag 'clone)
           (call/cc
             (lambda (k)
               (! from (list tag k)))))
          ((from tag 'ping)
           (! from (list tag 'pong)))
          (('update k)
           (k #t)))
        (loop)))))

(node-init node1)

(!? pong-server 'ping) ; => gnop
(! pong-server (list 'update (!? new-server 'clone)))
(!? pong-server 'ping) ; => pong
```

```
;; buggy-pong.scm (running on node2)
(declare (block))
(import (termite))

(define pong-server
  (spawn
    (lambda ()
      (let loop ()
        (recv
          ((from tag 'ping)
           (! from (list tag 'gnop))) ;; BUG!
          (('update k)
           (k #t)))
        (loop)))))

(node-init node2)

;; publish the pong server
(publish-service 'pong-server pong-server)
```

Figure 1: Hot code update example

Here the created thread loops forever receiving lists of the form `(`*from tag* `square` *x*`)` where *from* is the source thread, *tag* is a unique tag created for this request/response, and *x* is the number to square. The thread responds with a message of the form `(`*tag result*`)` where *result* is the square of *x*.

2.1 Hot code update

In many distributed system implementations, updating the code of a node requires restarting the program running on that node. Performing hot code updates while a program is running is useful to change its behaviour without interrupting the service. This can be to fix a bug or to extend the service. In Termite, this is possible in part because both procedures and continuations are serializable.

A basic ping-pong example is enough to unveil issues of the original implementation of Termite. In this example there are two `node`s each running a thread; one that sends `ping` and the other that replies `pong`. The Termite Scheme code in Figure 1 demonstrates how to perform a hot code update of the server without interrupting its service. The actors in this scenario are the server (`buggy-pong.scm`) and the client (`ping.scm`). Note that the server's implementation has a deliberately introduced bug: it replies to a `ping` request with the message `gnop` instead of `pong`.

The client application designed to fix the server is composed of a thread that contains the new behaviour of the server. The thread must handle the same messages as the buggy pong server to be compatible. Additionally, it is distinguished in two ways, first it fixes the response message to `pong` when receiving `ping`, second it handles the message `clone` which captures the continuation of the client thread which will be sent to the buggy server to fix it.

The client starts by creating a local service with `spawn`. It pings the buggy pong server and prints the (incorrect) result, creates and sends the continuation of the `new-server` to the buggy pong server in an `update` message. Then, it re-pings the server and prints the (correct) result. In the original implementation of Termite this works correctly when the code is run interpreted, but it fails when compiled. This is because the message sent by the client contains a continuation that refers to return points in compiled code that do not exist on the server that receives the message. Our module system offers a mechanism solving this problem.

2.2 RPC/RMI

Remote procedure call (RPC) and remote method invocation (RMI) are both mechanisms that allow remote execution of code on a remote computer. The essential difference is that RMI is object-oriented while RPC is not. Java RMI supports direct transfer of serialized Java classes and distributed garbage collection. A remote call[5] can be described as the following sequence of events:

(1) The client calls a local stub with parameters passed to it in a normal way.

(2) The client stub packs the parameters into a message. This is called marshalling.
(3) The client sends the message to a server on the remote node.
(4) The server stub unpacks the parameters of the message. This is called unmarshalling.
(5) Finally, the server stub invokes the procedure with the arguments. The result is marshalled then sent back to the client.

In Termite Scheme and Erlang, RPC servers can be implemented by creating a service that dispatches the messages to the right procedure. The square service example given earlier is a RPC server allowing a single procedure to be called. In general, the message dispatch used in the server constrains the procedures that can be executed to the ones handled by the dispatcher.

Termite has the `on` procedure to circumvent this constraint by allowing the execution of a thunk on any `node`. This procedure takes as parameters a `node` and a thunk and returns the result of calling the thunk on that `node`, for example `(on node2 (lambda () (directory-files)))`. This simplifies the implementation of RPC servers to a simple `node` initialized with the `node-init` procedure.

The procedure `remote-spawn` is similar to `spawn` but the thread it creates is on the `node` specified as a parameter. This thread can then be used to execute specific code. No interface code is required because the client explicitly sends the thunk to the destination `node`.

The power of these features rests on the transparent unrestricted code migration mechanism possible with our module system.

3 EXISTING GAMBIT FEATURES

The implementation of our module system uses the following existing Gambit features.

3.1 Symbolic paths

When a filesystem path begins with `~~`*`name`* it expands to the path bound to that name in the *symbolic path dictionary*. This extension to the filesystem path syntax is convenient for accessing directories whose location depends on the system configuration or command line arguments. For example `~~lib` is bound to the directory containing the builtin Gambit libraries and `~~userlib` is bound to the directory containing user installed libraries.

3.2 Module loader

A source code file may contain the declaration `(##supply-module` *`module-id`*`)` to identify the file as the module *`module-id`*. It can also contain a set of `(##demand-module` *`module-id`*`)` forms indicating dependencies on functionality provided by the modules identified by *`module-id`*. When the file is compiled, these properties are embedded in the generated code and available to the module loader. When the compiled file M is loaded the Gambit runtime system will ensure that M's required modules are loaded first. The required modules are searched using the *module-id* in a set of directories known as the module search order, which by default is `~~userlib` followed by `~~lib`.

3.3 Object serialization

The serialization of objects in Gambit is done with the procedure `object->u8vector` which takes as parameters the object to serialize and optionally a *transform* procedure which is called on every sub-object inside the object in order to customize the serialization process. The result is a `u8vector` (vector of bytes).

The serialization of most objects is straightforward. A first byte indicates in the upper bits the type of the object, and possibly some basic property such as its length in the remaining bits of the byte (if it fits otherwise in the following bytes). This is followed by the serialization of each field of the object. For example the vector `#(1 2 3)` is serialized to the four bytes `#x23 #x51 #x52 #x53`. The first byte indicates the vector type and a length of 3, and the remaining bytes the type and value of the small integers 1, 2 and 3.

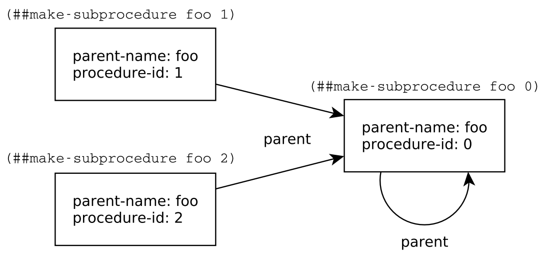
Circular references and shared objects are handled by keeping track of the position of serialized objects in the byte stream and using a special type that refers to a previously serialized object by its position in the stream.

The machine independent serialization of compiled procedures, closures and continuations is based on the serialization of control points. There are control points for procedure entry points, closure entry points and non-tail call return points.

The serialization of control points is the key to serialize closures and continuations. Gambit uses a flat closure representation [7]. A closure is a vector-like object containing the free variables and a reference to a control point (the closure's entry point). Gambit uses the *incremental stack/heap strategy* [10] for managing continuations. A continuation is a chain of continuation frames, either stored on the stack or the heap (the details of the representation are given in [13]).

Continuation frames, like closures, are vector-like objects containing values and a reference to a control point (the return point of a non-tail procedure call). These objects are serialized similarly to vectors. Each of their fields needs to be serialized, the only particularity is that one of the fields is a control point.

For historical reasons Gambit uses the term *subprocedure* for control points. Each subprocedure has a *parent* which is the toplevel procedure that contains it. The subprocedures contained in a given parent are assigned a machine independent integer index identifying that control point in the parent: its *id*. Each toplevel procedure has itself as a parent and an id of 0. These attributes can be obtained with the procedures `(##subprocedure-parent` *`subproc`*`)` and `(##subprocedure-id` *`subproc`*`)` which access meta-information maintained by the runtime system. The inverse operation, namely the retrieval of the subprocedure with a given *parent* and *id*, is performed by the procedure



```
(define foo
  (lambda (x)    ;; id = 0 (entry of foo)
    (lambda (y)  ;; id = 1 (entry of closure)
      (+ (log y) ;; id = 2 (return point)
         x))))
```

Figure 2: Machine independent control point identification

`(##make-subprocedure` *`parent id`*`)`. Figure 2 shows a procedure containing three control points, their logical relationship, and how they can be retrieved with `##make-subprocedure`.

The last issue to address is the machine independent identification of the parent. Gambit has a block compilation declaration, i.e. `(##declare (block))`, that tells the compiler that all global variables defined in the file are not mutated in other files. When this declaration is used, the Gambit compiler assigns a name to each toplevel procedure, which is typically the name of the global variable used in the toplevel `define`, or a name derived from the filename when the lambda-expression is not used in a toplevel `define`. The serialization of parent procedures uses this name, which is obtained with the procedure `(##subprocedure-parent-name` *`subproc`*`)`. Deserialization needs to recover the reference to the parent given its name, and this is done with the procedure `(##global-var-primitive-ref` *`name`*`)`. This procedure fails when a toplevel procedure with that name does not exist in the receiving node's program. Our module system catches this case to trigger the dynamic loading of the missing compiled code with a mechanism explained in Section 4.1. The main point is that the name of the toplevel procedure contains enough information to find the corresponding source code, compile it and load it.

3.4 Namespaces

To avoid clashes of global variables and toplevel macros between modules, Gambit partitions names into namespaces. A name is *qualified* when it contains a `#`, such as `math#pi`, and is *unqualified* when it does not, such as `sqrt`. This notation is inspired by Curtis' *et al* module system for Scheme [11]. A namespace is a prefix that is added to unqualified variable and macro names to make the name qualified. For example, the `math#` namespace applied to the unqualified `sqrt` results in the qualified name `math#sqrt`. A namespace is either the special empty namespace or a name that ends with a `#`.

```
(##namespace ("math#")
             ("" (def define) if < * -))

(def (fact n)
  (if (< n 2) 1 (* n (fact (- n 1)))))
```

Figure 3: Namespace declaration example

Gambit's `##namespace` declaration controls which namespace is added to unqualified names in the scope of the declaration (which is the rest of the file if at toplevel). Three forms of this declaration exist (to simplify we have used the `ns#` namespace):

(1) `(##namespace ("ns#"))`
(2) `(##namespace ("ns#"` *`name1`*`...))`
(3) `(##namespace ("ns#" (`*`name1 alias1`*`)...))`

In the first form, all unqualified identifiers in the scope of the declaration will be augmented with the `ns#` prefix. In the second form only the unqualified names *`name1`*... are augmented. In the third form the unqualified names *`name1`*... are renamed to *`alias1`*... and then the `ns#` prefix is added.

A toplevel `##namespace` declaration placed at the beginning of a file can map the names used in the rest of the code to appropriate namespaces. Figure 3 shows a sample use to implement a small math library. The declaration maps `def` to `define` (in the empty namespace), and everything else except `if`, `<`, `*`, `-` to the `math#` namespace. Consequently this code actually defines the `math#fact` procedure. If the code contained other definitions they too would define names in the `math#` namespace. Another module could access this procedure directly with the qualified name `math#fact`, or with `fact` if in the scope of a `(##namespace ("math#" fact))`, or with `factorial` if in the scope of a `(##namespace ("math#" (factorial fact)))`.

4 OUR MODULE SYSTEM

4.1 `define-library` form

To help with the adoption of our module system we have designed it to be compatible with the R7RS standard [24]. The main form for defining libraries (which is synonymous to modules in this paper) is the `define-library` form which has the following syntax:

`(define-library` *<library name>*
<library declaration> `...)`

The first argument is the library name; a non-empty list of identifiers such as `(scheme base)`. The name is followed by library declarations which can be any of the following forms, of which the last six are extensions to the R7RS syntax offering fine control over the compilation and linking process:

- `(export` *`<export spec>`*`...)`
- `(import` *`<import set>`*`...)`
- `(begin` *`<command or definition>`*`...)`
- `(include` *`<filename>`*`...)`
- `(include-ci` *`<filename>`*`...)`
- `(include-library-declarations` *`<filename>`*`...)`

```
(define-library (github.com/fred hello)
  (export hi)
  (import (only (scheme base) define)
          (rename (scheme write) (display show)))
  (begin
    (define (hi str)
      (show "hello ")
      (show str)
      (show "\n"))))
```

Figure 4: The library (`github.com/fred hello`) exporting the procedure `hi`

```
(define-library (gitlab.com/zoo cats)
  (import (only (scheme base) define)
          (github.com/fred hello @1.0))
  (begin
    (define (main)
      (hi "lion")
      (hi "tiger"))))
```

Figure 5: The library (`gitlab.com/zoo cats`) which depends on version `1.0` of the library (`github.com/fred hello`)

- `(cond-expand` *`<cond expand features>`* `...)`
- `(namespace` *`<namespace>`*`)`: e.g. `(namespace "X11#")`
- `(cc-options` *`<options>`* `...)`: e.g. `(cc-options "-O3")`
- `(ld-options` *`<options>`* `...)`: e.g. `(ld-options "-lX")`
- `(ld-options-prelude` *`<options>`* `...)`
- `(pkg-config` *`<options>`* `...)`: e.g. `(pkg-config "X11")`
- `(pkg-config-path` *`<path>`* `...)`

The library's variable and macro definitions are typically contained in the `begin` declaration. It is also possible to put the definitions in another file that is included with one of the `include` forms. The `cond-expand` form allows conditional activation of the library declarations it contains depending on the system's support for specific features.

The `export` declaration indicates the list of variables and macros defined by the library that are accessible to code importing this library. Like with R7RS the `export` declaration can indicate that specific (internal) names are renamed to other (external) names.

A library declares a dependency to another library with the `import` declaration. This declaration identifies the imported library. The `import` declaration may restrict the subset of exported names that are imported (by default all exported names are imported). The exported names may also be renamed. This is specified in the *<import set>* with the forms `(only ...)`, `(except ...)`, `(rename ...)`, and `(prefix ...)`. We have extended the R7RS syntax for imported library names to allow a trailing `@`*`version`* that indicates the specific version required. Any mobile code libraries imported must have a version indicator for reliable operation of the module system.

Figures 4 and 5 are an example of libraries defined with our extended `define-library`. Figure 4 implements a library that exports the procedure `hi`. It uses an `import` declaration that imports only the name `define` exported from the standard library `(scheme base)` and all the names exported from the standard library `(scheme write)`, but with `display` renamed to `show`. Figure 5 is a library that depends on version `1.0` of the library defined in Figure 4.

The R7RS specifies that the library name "is used to identify the library uniquely when importing from other programs or libraries". In the context of mobile code libraries, our system has the stronger requirement that libraries are identified uniquely across all nodes of the distributed system. This is achieved by requiring mobile code libraries to be in code repositories hosted by version control system servers (which could be a public service such as `github.com` or a privately managed server) and to use the location of the repository in the name of the library. The use of a version control system allows multiple versions of the library to be stored in a single repository; each identified with a specific commit tag. The use of a network accessible repository makes it possible to obtain the code from any node of the distributed system. In our example the library name `(github.com/fred hello)` encodes the location of the repository, i.e. `http://github.com/fred/hello`.

The module system uses the library name and version to construct a unique library identifier. The version is either implicit (the current commit tag of the version control system) or indicated explicitly in the `import` declaration. All identifiers in the library name are concatenated with a `/` separator followed by an `@` and the version identifier (implicit or explicit). A trailing `#` is added to get the unique namespace for the library. Unless the library has a `namespace` library declaration to force the namespace (which is mainly useful for builtin libraries), the namespace derived from the library name is used to construct the qualified names of the (toplevel) variables and macros defined by the library. Due to the guaranteed namespace uniqueness, name clashes between libraries are not possible, including different versions of the same library. In our example, if the version of the library is `1.0`, the definition of `hi` is in reality defining the global variable with the qualified name `github.com/fred/hello@1.0#hi` whereas if the version is `1.5` it is `github.com/fred/hello@1.5#hi` that is defined.

Having the library location and version information in the global variable name provides valuable information to the subprocedure deserialization mechanism. When the procedure `(##global-var-primitive-ref` *`name`*`)` fails because a procedure with that name does not exist on the receiving node the system uses the procedure name to determine where to fetch a copy of the repository for the library containing that procedure, and which version of the repository is needed. A local copy of the repository at the required version is then made and the Gambit compiler is invoked to create the compiled code which is dynamically loaded into the running

```
;; expansion of (define-library (github.com/fred hello) ...)

(##declare (block))

(##supply-module github.com/fred/hello@1.0)

(##namespace ("github.com/fred/hello@1.0#")
             ("" define
                 (show display)
                 write-shared
                 write
                 write-simple))

(define (hi str)  ;; defines github.com/fred/hello@1.0#hi
  (show "hello ") ;; calls display
  (show str)      ;; same
  (show "\n"))    ;; same


;; expansion of (define-library (gitlab.com/zoo cats) ...)

(##declare (block))

(##supply-module gitlab.com/zoo/cats@2.0)
(##demand-module github.com/fred/hello@1.0)

(##namespace ("gitlab.com/zoo/cats@2.0#")
             ("" define)
             ("github.com/fred/hello@1.0#" hi))

(define (main)  ;; defines gitlab.com/zoo/cats@2.0#main
  (hi "lion")   ;; calls github.com/fred/hello@1.0#hi
  (hi "tiger")) ;; same
```

Figure 6: Expansion of version `1.0` of the library (`github.com/fred hello`) and version `2.0` of the library (`gitlab.com/zoo cats`)

program, allowing the deserialization to resume. The compiled code is kept locally to avoid costly recompilations if that version of the library is used again in the future. The local copy of the repository is also kept to avoid fetching it again if another version of the library is required.

4.2 Implementation of `define-library`

The `define-library` form is implemented as a macro that expands into existing Gambit forms. For the libraries shown in Figures 4 and 5, expanding the `define-library` forms produces the code shown in Figure 6, for versions `1.0` and `2.0` of the libraries respectively.

The expanded code starts with a `(##declare (block))` declaration that informs the compiler that variables defined in the library will not be mutated in other libraries (this semantics is part of the R7RS specification). This enables some optimizations by the compiler, such as constant propagation and inlining, and it also causes the compiler to assign to each toplevel procedure a name that includes the library's location and version, necessary for the deserialization process.

This is followed by a `(##supply-module` *`library-id`*`)` that provides the module loader with the identity of the library and version implemented by the code. For each imported library (with a non-empty set of definitions), a `(##demand-module` *`library-id`*`)` is generated to inform the module loader that the specified library must be loaded first. The handling of exported and imported names is done through a generated `##namespace` form that maps the names used in the library's code to the qualified names. The definition of all imported macros is then generated.

After that the library's code is generated in its original form. No further processing is needed by the `define-library` macro because the compiler will use the namespace declarations to map the names appropriately during the compilation process.

4.3 Other features

In this section we explain other features of the module system that are not essential for reaching our goal but that are useful in day-to-day use.

4.3.1 Optional version. During the development phase it is good to have a fast turnaround time when debugging a library. It would be tedious to assign a new version after each change of the code. For this reason the module system distinguishes *installed* libraries from those that are *not installed.* When a local copy of a library has been obtained from a version controlled repository (on the network or the local filesystem) it is *installed* and the different versions can be referenced in `import` declarations. A library that is stored in a local directory, possibly managed by a version control system, is *not installed.* In this case `import` declarations must not refer to a specific version and the current state of the code is used. This improves the workflow as it avoids having to install the library after each change. Nevertheless, if it is in a version controlled repository, it is possible to install it whenever there is a need to assign a version to it.

4.3.2 Module aliases. With the `(define-module-alias` *`lib1 lib2`*`)` form, symbolic names can be defined to refer to libraries and library references can be redirected to other locations. This is useful for the development phase for quickly swapping one library for another. It also allows putting the choice of library versions in a centralized place instead of each and every `import` declaration to be able to upgrade to new versions with a single edit. For example, when the following definitions are in effect:

```
(define-module-alias (gitlab.com/zoo cats)
                     (gitlab.com/zoo cats @2.0))

(define-module-alias (fh)
                     (github.com/fred hello))
```

an `(import (gitlab.com/zoo cats))` will import the library `(gitlab.com/zoo cats @2.0)` and an `(import (fh @1.0))` will import the library `(github.com/fred hello @1.0)`. The module alias definitions that are put at the root of a library's directory in the file `_setup_.scm` of a directory will apply automatically to the libraries in that directory (with lexical scoping rules respecting the nesting of the directories up to the root of the repository).

4.3.3 Library management. Automatically installing a library from the network when it is referenced in a deserialization could be a security issue. The Gambit interpreter allows manual installation of modules through command-line arguments, for example `gsi -install github.com/fred/hello`. Moreover, the runtime system maintains a whitelist of the locations from which libraries are unconditionally installed. By default the whitelist contains only `github.com/gambit`, the Gambit project account. The whitelist can be extended through environment variables and command line arguments. When the library's location is not on the whitelist a confirmation will be asked of the user if a REPL is currently started (the runtime system can be configured to always ask for confirmation, or to always refuse to install such libraries).

5 EVALUATION

The original implementation of Termite Scheme was able to send messages containing code that the receiving node did not have as long as the code was interpreted. Our module system extends this capability to compiled code. In this section we evaluate experimentally the performance gain due to the more compact messages and the faster compiled code.

Three machines with different operating systems and architectures were used in the experiments to exercice the machine independence. All three machines are on the same Gigabit ethernet LAN. The machine $M_{ARM/Linux}$ is a 4-core armv7l with 2GB RAM running Linux 4.19 (Raspberry Pi). The machine $M_{x86/macOS}$ is a 6-core Intel i7-8700B with 32GB RAM running macOS. The machine $M_{x86/Linux}$ is a water-cooled 4-core Intel i7-7700K with 16GB RAM running Linux 4.9. The later machine is the fastest and the execution time measurements are the most stable of the three machines due to the better thermal control. This fast machine is always used as the destination of the code migrations; a situation that is representative of the case where the destination of a RPC is a fast compute server.

Three standard Scheme benchmark programs of different source code sizes (in bytes) were used to see the effect of the code size:

- Puzzle (4K)
- Scheme (40K)
- Compiler (400K)

The internal iteration count of the programs was adjusted so that they would have an execution time when interpreted that is roughly proportional to their size. So 400K has both the largest code size and the longest run time (roughly 10 seconds on $M_{x86/Linux}$).

The programs are adapted to our use case as follows. The program is turned into a library by wrapping it in a `define-library` form that exports the program's main entry point and the library is put in a repository hosted on `github.com`. A separate driver program simply imports the library and then causes the program to be executed on $M_{x86/Linux}$ using Termite's `on` form calling the library's "main". The total execution time is measured and also the execution time on the destination node. The difference between these measures is accounted for by the message transfers, the serialization and deserialization, and when the library isn't currently installed, the installation of the library source code locally from `github.com`, the Scheme to C compilation, and the C compilation. The programs are run 20 times, the top and bottom outliers are removed to account for random variations in the network latency, and the tables of results contain the average and standard deviation for each measure in milliseconds.

Three scenarios are tested:

- **INTERPRETED**: The whole program is interpreted. This represents the performance achievable before our module system was implemented.
- **FIRST-INSTALL**: The compiled program is executed and the destination machine is installing the library for the first time.

$M_{x86/macOS}$

Time (ms)	4K		40K		400K	
Total for RPC	146.4	± 0.6	966.9	± 6.1	10463.8	± 3.3
On destination	131.6	± 0.2	948.7	± 5.9	10381.0	± 2.7

$M_{ARM/Linux}$

Time (ms)	4K		40K		400K	
Total for RPC	179.2	± 1.6	1002.7	± 8.1	10801.1	± 11.0
On destination	132.6	± 0.7	954.6	± 0.0	10390.5	± 2.6

Figure 7: Timings in the INTERPRETED scenario with $M_{x86/macOS}$ or $M_{ARM/Linux}$ as the start node

$M_{x86/macOS}$

Time (ms)	4K		40K		400K	
Total for RPC	15.5	± 0.7	48.5	± 0.6	478.7	± 0.6
On destination	2.2	± 0.0	35.2	± 0.1	463.3	± 0.3

$M_{ARM/Linux}$

Time (ms)	4K		40K		400K	
Total for RPC	26.9	± 1.2	60.4	± 0.6	492.5	± 0.7
On destination	2.2	± 0.0	35.2	± 0.0	462.8	± 0.3

Figure 8: Timings in the STEADY-STATE scenario with $M_{x86/macOS}$ or $M_{ARM/Linux}$ as the start node

$M_{x86/macOS}$

Time (ms)	4K	40K	400K
Total for RPC	1208.6	2460.3	148536.2
On destination	2.2	36.1	464.7

$M_{ARM/Linux}$

Time (ms)	4K	40K	400K
Total for RPC	1159.8	2502.0	153272.6
On destination	2.2	37.1	464.1

Figure 9: Timings in the FIRST-INSTALL scenario with $M_{x86/macOS}$ or $M_{ARM/Linux}$ as the start node

- **STEADY-STATE**: The compiled program is executed and the destination machine has previously installed and compiled the library.

Figures 7, 8, and 9 give the timings for the INTERPRETED, STEADY-STATE, and FIRST-INSTALL scenarios respectively with either $M_{x86/macOS}$ or $M_{ARM/Linux}$ as the start node, and always $M_{x86/Linux}$ as the destination node.

The first observation is that in all the scenarios the speed of the start node has very little bearing on the total RPC time. This is to be expected because most of the work is done on the destination node. The slower times are generally for $M_{ARM/Linux}$. This can be explained by the higher messaging overhead.

The message transfer overhead (network latency and serialization/deserialization) will increase with the size of the messages. In the STEADY-STATE scenario the messages are the parameters of the called procedure and the result. This goes from less than a hundred bytes (for 4K) to tens of kilobytes (for 400K). In this scenario the messaging overhead varies between 13-15ms for $M_{x86/macOS}$ and between 25-30ms for $M_{ARM/Linux}$ (slower processor and network interface). The execution time on the destination is essentially identical. In the INTERPRETED scenario the messages also carry a representation of the code to be executed, which is large for 400K. In this scenario the messaging overhead varies between 15-83ms for $M_{x86/macOS}$ and between 47-411ms for $M_{ARM/Linux}$ (here again the overhead is impacted by the slower processor and network interface). Nevertheless the messaging overhead represents a small fraction of the total RPC time. For the STEADY-STATE scenario, as expected the messaging overhead is at its highest for 4K, the shortest running program, because messaging represents more work than the actual computation on the destination.

To evaluate the speed improvement of RPC calls achieved with our module system, we can compare the INTERPRETED and STEADY-STATE scenarios. The total RPC time for STEADY-STATE is up to 22x faster for $M_{x86/macOS}$ and $M_{ARM/Linux}$ on 400K. The shortest running program, 4K, has the lowest but still considerable speedup of 6x-9x.

Figures 9 shows that the time for the installation and compilation of the library can be quite large for large programs (400K, which is about 10,000 LOC, takes about 150 seconds and the others less than 2.5 seconds). Thankfully, installation only has to be done once per library and version used and libraries are rarely so big, especially when good modular programming practices are used.

6 RELATED WORK

The package management offered by our system supports installing multiple versions of a package, which ensures the same dependencies on all nodes. The package management of the Go [6] programming language supports installing and using multiple versions of a module, but it does not support execution time installation of modules that is needed for hot code update. QuickLisp [27] follows a different approach of only keeping the last installed version of packages, which can break dependencies. The module identification does not include the location as the module names are mapped to their location using a central directory, which means the module names have to be registered to avoid name clashes. Like our system, QuickLisp stores modules on public VCS services and has an automatic installation of modules but it is not tied to deserialization. Nix [12] is a system package manager that shares the same idea of keeping multiple versions of a package to avoid breaking dependencies. However, it is not meant to be used as a language package manager and thus it is not integrated with a specific language. Erlang supports hot code update but because the module identification does not contain the location of the library the installation of modules must be done separately.

Before the R6RS standard was ratified (2007), most Scheme systems had designed their custom module system. Support for R6RS and its module system was added to some systems notably Chez Scheme, Guile, Larceny, and PLT Scheme (now Racket). The R6RS standard includes a `library` form to define libraries which has much syntactic similarity to the R7RS `define-library` form used by our work. A relevant difference is R6RS' support for version information in the library name. Our module system allows version information in `import` declarations but not in the `define-library` form. We believe it is less error prone to obtain this information from the underlying version control system as it avoids possible inconsistencies with the code.

Since the ratification of R7RS (2017), support for its less complex module system is growing among Scheme systems with close to 20 systems supporting it. For the strictly R6RS compliant ones the Akku.scm project [26] has developed a converter from the R7RS `define-library` form to the R6RS `library` form. The more widespread support for R7RS' module system is one of the motivations for adopting it in our work.

Over the years different groups have implemented module systems extending Gambit Scheme. Black Hole [1] is an R5RS compatible module system designed to add as little extra syntax as possible to Gambit Scheme. JazzScheme [8] and Gerbil [25] are more intrusive as they promote a whole new custom program structure and syntax, and add features such as object orientation. The SchemeSpheres [2] project has used a prototype implementation of our `define-library` macro, so it has much similarity to our latest work.

However, none of these systems offer the same combination of features as our work and specifically none offers transparent deserialization of compiled procedures and continuations.

The library naming approach we have used which includes the repository location to identify the library could be used by other R7RS Scheme systems without modifying their implementation. This would help avoid library name clashes and also pave the way for future extensions of those Scheme systems to automatically install the library from a public repository (independent of any support for procedure and continuation deserialization).

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] Black Hole, a R5RS compatible module system for gambit. https://github.com/per-gron/blackhole, 2019. Accessed: 2020-02-21.

[2] Schemespheres. https://github.com/alvatar/spheres, 2019. Accessed: 2020-02-21.

[3] David Alan and David Alan Halls. Applying mobile code to distributed systems. Technical report, 1997.

[4] Joe Armstrong. A history of erlang. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, page 6–1–6–26, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937667. doi: 10.1145/1238844.1238850. URL https://doi.org/10.1145/1238844.1238850.

[5] Andrew Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984. doi: 10.1145/2080.357392. URL http://doi.acm.org/10.1145/2080.357392.

[6] Tyler Bui-Palsulich and Eno Compton. Go reference manual 1.13.5. https://blog.golang.org/using-go-modules, 2019.

[7] Luca Cardelli. The functional abstract machine, 1983.

[8] Guillaume Cartier and Louis-Julien Guillemette. Jazzscheme: Evolution of a lisp-based development system. In *2010 Workshop on Scheme and Functional Programming*, page 50. Citeseer, 2010.

[9] Henry Cejtin, Suresh Jagannathan, and Richard Kelsey. Higher-order distributed objects. *ACM Trans. Program. Lang. Syst.*, 17(5):704–739, September 1995. ISSN 0164-0925. doi: 10.1145/213978.213986. URL https://doi.org/10.1145/213978.213986.

[10] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, Apr 1999. ISSN 1573-0557. doi: 10.1023/A:1010016816429. URL https://doi.org/10.1023/A:1010016816429.

[11] Pavel Curtis and James Rauen. A module system for scheme. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, LFP '90, pages 13–19, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X. doi: 10.1145/91556.91573. URL http://doi.acm.org/10.1145/91556.91573.

[12] Eelco Dolstra, Merijn Jonge, and Eelco Visser. Nix: A Safe and Policy-Free System for Software Deployment. pages 79–92, 01 2004.

[13] Marc Feeley. Compiling for multi-language task migration. *SIGPLAN Not.*, 51(2):63–77, October 2015. ISSN 0362-1340. doi: 10.1145/2936313.2816713. URL http://doi.acm.org/10.1145/2936313.2816713.

[14] Marc Feeley. Gambit v4.9.3. Reference Manual, 2 2019. URL http://www.iro.umontreal.ca/~gambit/doc/gambit.pdf.

[15] Marc Feeley and Philip W. Trinder, editors. *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, 2006. ACM. ISBN 1-59593-490-1.

[16] Adrian Francalanza and Tyron Zerafa. Code management automation for Erlang remote actors. In Jamali et al. [21], pages 13–18. ISBN 978-1-4503-2602-5. doi: 10.1145/2541329.2541344. URL https://doi.org/10.1145/2541329.2541344.

[17] Matthew Daniel Fuchs. *Dreme: For Life in the Net.* PhD thesis, USA, 1995. AAI9609199.

[18] Stefan Fünfrocken. Transparent migration of java-based mobile agents: Capturing and re-establishing the state of java programs. *Personal Technologies*, 2(2):109–116, Jun 1998. ISSN 1617-4917. doi: 10.1007/BF01324941. URL https://doi.org/10.1007/BF01324941.

[19] Guillaume Germain. Concurrency oriented programming in Termite Scheme. In Feeley and Trinder [15], page 20. ISBN 1-59593-490-1. doi: 10.1145/1159789.1159795. URL http://doi.acm.org/10.1145/1159789.1159795.

[20] Robert S. Gray. Agent tcl: A flexible and secure mobile-agent system. In *Proceedings of the 4th Conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4*, TCLTK'96, pages 9–23, Berkeley, CA, USA, 1996. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267498.1267500.

[21] Nadeem Jamali, Alessandro Ricci, Gera Weiss, and Akinori Yonezawa, editors. *Proceedings of the 2013 Workshop on Programming based on Actors, Agents, and Decentralized Control, AGERE!@SPLASH 2013, Indianapolis, IN, USA, October 27-28, 2013*, 2013. ACM. ISBN 978-1-4503-2602-5. URL http://dl.acm.org/citation.cfm?id=2541329.

[22] A. Lukić, N. Luburić, M. Vidaković, and M. Holbl. Development of multi-agent framework in JavaScript. In *ICIST 2017 Proceedings Vol.1*, pages 261–265, 2017.

[23] Stefan M, Raymond Bimazubute, and Herbert Stoyan. Mobile intelligent Agents in Erlang. In *Fourth International ICSC Symposium on ENGINEERING OF INTELLIGENT SYSTEMS (EIS 2004)*, 2004.

[24] Alex Shinn, John Cowan, and Arthur A. Gleckler. Revised^7 report on the algorithmic language Scheme. 2017.

[25] Dimitris Vyzovitis. Gerbil scheme. https://cons.io/, 2020. Accessed: 2020-02-21.

[26] Göran Weinholt. Akku package management made easy. https://akkuscm.org/, 2019. Accessed: 2020-02-21.

[27] Zach Beane. QuickLisp. https://github.com/quicklisp, 2020. Accessed: 2020-04-02.

Sealable Metaobjects for Common Lisp

Marco Heisig
FAU Erlangen-Nürnberg
Cauerstraße 11
Erlangen 91058, Germany
marco.heisig@fau.de

ABSTRACT

We introduce the concept of sealable metaobjects, i.e., classes, generic functions, and methods, whose behavior is restricted to allow for some static analysis. Then we use these sealable metaobjects to define fast generic functions, a variant of standard generic functions that allow for call site optimization — ranging from faster method dispatch to inlining of entire effective methods. Fast generic functions support almost all features of standard generic functions, including custom method combinations and non-trivial references to the next method. Our benchmarks show that a straightforward implementation of Common Lisp's sequence functions using these fast generic functions is competitive with the corresponding built-in sequence functions of SBCL. Fast generic functions are thus an attractive drop-in replacement for standard generic functions in performance critical codes.

KEYWORDS

Common Lisp, Metaobject Sealing, Generic Function Inlining, CLOS

ACM Reference format:
Marco Heisig. 2020. Sealable Metaobjects for Common Lisp. In *Proceedings of the 13th European Lisp Symposium, Zürich, Switzerland, April 27–28 2020 (ELS'20),* 7 pages.
DOI: 10.5281/zenodo.3743823

1 INTRODUCTION

After three decades, the Metaobject Protocol[7] of the Common Lisp Object System is still the pinnacle of object oriented programming technology. Objects, functions and classes can be created, modified and augmented continuously, without sacrificing either correctness[1] or performance. Because generic functions, methods and classes are themselves objects, they can also be extended. This capability allows programmers to explore a whole space of object oriented paradigms.

Yet there is one rather fundamental limitation of CLOS. Since objects can be redefined at any time, implementations are forced to use at least one indirection when calling a generic function. It is not possible to move any part of a generic function into the call site. This indirection is great for modularity and extensibility, but means that generic functions are inherently ill suited for the representation very lightweight operations such as arithmetic operations on floating-point numbers. In particular, new Lisp programmers are frequently disappointed that they cannot extend standard functions like `+` and `find` to new data types.

In this paper, we show how these limitations can be lifted while preserving most of the flexibility that programmers expect from CLOS. We first define a set of sealable metaobjects with carefully chosen limitations on how they can be extended and redefined. Then, in a second step, we show how these metaobjects can be used to implement *fast generic functions*, a variant of standard generic functions that perform several powerful optimizations. Finally, we show how fast generic functions can be used to define efficient number and sequence functions and present some promising benchmark results. All our work is available on Quicklisp. The library for metaobject sealing is called SEALABLE-METAOBJECTS, and the library that provides fast generic functions is called FAST-GENERIC-FUNCTIONS.

[1]Barring eccentricities like concurrent modification of the class hierarchy.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s). .
DOI: 10.5281/zenodo.3743823

2 RELATED WORK

The idea of sealing metaobjects to speed up certain kinds of programs is not new. Similar functionality can be found both in Common Lisp libraries, and in the Dylan programming language. Our main contribution is to gather these ideas and to make them accessible in a high quality implementation. Therefore, we are indebted to the following prior endeavors:

- Henry Baker has proposed a subset of CLOS called STATIC CLOS[2], where it is not possible to change the class of any object and the class hierarchy is fixed at compile-time. In doing so, Static CLOS allows the compiler to eliminate most of the overhead that is normally associated with calls to generic functions. In contrast to our technique, Static CLOS enforces its restrictions globally, not just for a chosen set of metaobjects, and it bypasses most of the machinery of the Metaobject Protocol altogether.
- The Dylan programming language[9] has a feature called *define sealed domain* that permanently freezes a part of the domain of a generic functions, while still allowing changes outside of the sealed domain. Our technique closely mimics this feature, except for the necessary adaptions to integrate it into Common Lisp.
- The library INLINED-GENERIC-FUNCTION[1] by Masataro Asai provides generic functions that can be inlined into the call site. Our work is directly inspired by this library, but tries to improve upon several issues. The crucial difference is that the inlined-generic-function library works by inlining the dispatch function and all reachable effective

methods into the call site, while our technique will only inline effective methods if they are marked as reasonably cheap, and if the dispatch function can be reduced to a single case. By using more conservative inlining rules, we achieve that users can unconditionally use fast generic functions without worrying about code bloat.
- The library SPECIALIZATION-STORE[4] by Mark Cox provides an alternative to generic functions that trade various features of CLOS — like method combinations, argument precedence ordering and class-based dispatch — for better compile-time optimizations.
- The library STATIC-DISPATCH[6] by Alexander Gutev provides a mechanism of static dispatch that is implemented via compiler macros and shadowing of `defmethod`. However, the library is not extensible and doesn't support method combinations.

3 METAOBJECT SEALING

Common Lisp permits incremental changes to the class hierarchy and to the methods of a generic function. The purpose of metaobject sealing is to restrict these capabilities in order to enable static analysis and optimization.

3.1 Sealable Metaobjects

Sealable metaobjects are classes, generic functions, methods, slot-definitions or method combinations with two states — sealed and unsealed. Once a metaobject is sealed, it remains sealed indefinitely. Furthermore, the following restrictions apply to all sealed metaobjects:

(1) Calling reinitialize-instance on a sealed metaobject has no effect.
(2) It is an error to change the class of a sealed metaobject.
(3) It is an error to change the class of any object to a sealed metaobject.
(4) It is an error to change the class of an instance of a sealed metaobject.
(5) Each superclass of a sealed metaobject must be a sealed metaobject.

Restriction (1) ensures that slots cannot be mutated without going through the corresponding accessors or protocols. The restrictions (2), (3) and (4) allow a compiler to reliably perform static type inference. The restriction (5) ensures that the behavior of a sealed class cannot be changed adding new superclasses to it, or by modifying existing superclasses.

Built-in classes, structure classes, and system classes are sealed metaobjects according to our definition. In addition, we provide a `sealable-metaobject-mixin` class that can be used as a building block for other kinds of sealable metaobjects.

3.2 Sealed Specializers

A sealed specializer is either a sealed class, or an EQL specializer whose object is an instance of a sealed class. Each sealed specializer also denotes a Common Lisp type. For a specializer that is a class, the type is the set of all objects that are of that class. For an EQL specializer, the type is the set of all objects that are EQL to the specializer's object.

3.3 Domains

A domain is the cartesian product of the types denoted by some specializers. A sealed domain is a domain whose constituting specializers are sealed. The domain of a method with n required arguments is the n-ary cartesian product of the types denoted by the method's specializers. We say a method is inside of a domain D if the method's domain is a subset of D, and outside of a domain D if the method's domain is disjoint from D.

A domain designator is either a domain, or a list of specializer designators. A specializer designator is either a specializer, or a class name, or an expression of the form `(eql X)` for some object `X`. Most functions that work with domains will accept arbitrary domain designators and convert them to domains in the obvious way.

3.4 Sealable Generic Functions

A sealable generic function is both a generic function and a sealable metaobject. It may contain any number of sealed domains. Initially, a sealable generic function has zero sealed domains. New sealed domains can be added by the function `seal-domain`, which takes a sealable generic function and a domain designator. The following rules apply to sealed domains of a generic function.

(1) All sealed domains of a generic function must be disjoint.
(2) Each method of the generic function must either be fully inside of the sealed domain, or fully outside of the sealed domain.
(3) Each method inside of a sealed domain must be sealed, and all of its specializers must be sealed.
(4) It is an error to add or remove methods from inside of a sealed domain.
(5) It is an error to create a subclass of a sealed class that would violate any of the previous rules for any sealed generic function.

These rules ensure that the behavior of a sealed generic function whose arguments are provably within any of its sealed domains is fully known at compile time.

3.5 Potentially Sealable Methods

A potentially sealable method is both a method and a sealable metaobject. It is called *potentially* sealable because it can only be actually sealed if all of its specializers are sealable. The main purpose of potentially sealable methods is to use them (or a subclass thereof) as the generic function method class of a sealable generic function.

Potentially sealable methods have one additional feature in that they support method properties. Method properties are declarations that are automatically extracted from the method body and are made available with the generic function `method-properties`. The generic function `validate-method-property` checks whether a method property is valid. Figure 1 shows how one can define and use method properties. The purpose of method properties is to allow annotations in custom methods, e.g., whether the method is lightweight enough for inlining.

```
(defclass my-method
    (potentially-sealable-standard-method)
  ())

(defclass my-generic-function
    (sealable-standard-generic-function)
  ()
  (:default-initargs
   :method-class (find-class 'my-method))
  (:metaclass funcallable-standard-class))

(defmethod validate-method-property
    ((m my-method) (p (eql 'foo)))
  t)

(defmethod validate-method-property
    ((m my-method) (p (eql 'bar)))
  t)

(defgeneric my-gf (x)
  (:generic-function-class my-generic-function))

(defmethod my-gf ((x number))
  (declare (method-properties foo)))

(defmethod my-gf ((x sequence))
  (declare (method-properties bar foo)))

(mapcar #'method-properties
        (generic-function-methods #'my-gf))
;;; => ((bar foo) (foo))
```

Figure 1: Defining and using method properties.

3.6 Automatic Sealing

When a sealable metaobject is sealed, it automatically attempts to seal all its superclasses. When a sealable method is sealed, it automatically attempts to seal all its specializers. Furthermore, the function `seal-domain` seals both the generic function and all its methods that lie inside of the sealed domain. This way, users usually don't have to worry about the distinction between sealable and sealed metaobjects at all.

4 FAST GENERIC FUNCTIONS

So far, we have characterized sealable metaobjects solely by showing what they can't do. In this section, we will show how one can exploit the restrictions that we have introduced to speed up generic function calls. To do so, we define fast generic functions and fast methods as subclasses of sealable standard generic functions and potentially sealable standard methods, respectively.

Fast generic functions behave exactly like standard generic functions, except that their generic function method class is `fast-method`, and that they are sealable. The behavior of a fast method is just like that of a standard method, except that it inherits all the restriction

Type	Prototype
(eql 42)	42
(and integer (not (eql 42)))	5
(and real (not integer))	0.0

Figure 2: Static call signatures for the domain of real numbers, for methods specializing on (eql 42) integer and real.

that are imposed on sealable methods, and that all fast methods must be defined in a null lexical environment.

By enforcing that all fast methods are defined in a null lexical environment, we create several optimization opportunities. In particular, we may inline method functions directly into each effective method, and inline entire effective methods into the call site. Furthermore, knowing that a method is defined in a null lexical environment allows us to safely manipulate the method lambda. We exploit this to introduce a more efficient calling convention for method functions.

The next subsections contain a more detailed description of the individual steps towards faster generic functions.

4.1 Static Call Signatures

Whenever a sealed domain is added to a generic function, we compute its set of static call signatures. A static call signature consists of a list of types and a list of prototypes, each with one element per required argument of the generic function. The lists of types of each static call signature are disjoint and their union covers the entire sealed domain. The list of prototypes is chosen such that each prototype lies within the corresponding type, and not within the corresponding type of any other static call signature. We give an example of static call signatures in Figure 2.

Static call signatures are the backbone of fast generic functions. The compiler checks the types of all static call signatures to determine whether one of them is applicable. If so, our library can pass the prototypes of that static call signature to `compute-applicable-methods` to determine the ordered list of applicable methods. The list of applicable methods can then be used to compute an optimized effective method.

4.2 Call-site Analysis

Once a call to a fast generic function occurs in the source code, we have to check whether a static call signature of that function is applicable. One way to do so is by defining a compiler macro for the fast generic function, and using introspection to figure out the types of the arguments in the current environment. But this approach tends to work only if all required arguments of that generic function are lexical variables with explicitly declared types.

A more robust approach is to use the compiler infrastructure of each Lisp implementation. On SBCL, we generate one IR1 transformation for each static call signature, using `deftransform`.

4.3 Computing the Effective Methods

Once there is a unique applicable static call signature and a corresponding sorted list of applicable methods, we still have to generate an efficient effective method that can be called without going through the discriminating function.

Our initial attempt to do so was to modify fast generic functions such that a special initial method would first check a global variable, and if that variable is set, the initial method would return `#'call--next-method` instead of directly calling the next method. Then, we could invoke this next method function with the new arguments to get the desired behavior, without going through the discriminating function. However, we had to learn that accessing `call-next--method` in such a way disables many transformations that usually make generic functions reasonably efficient. The performance hit from losing these transformations was much larger than the cost of going through the discriminating function, so we had to abandon this approach.

Our current technique is to bypass the last stage of a generic function invocation, and to expand the result of calling `compute--effective-method` ourselves, using our own versions of `make--method` and `call-method`. This technique has the advantage that we can use a very efficient calling convention for method functions. Each method function receives one required argument per binding in its original lambda list, and has no keyword, optional and rest arguments. All parsing is performed by the effective method function. Another benefit of this technique is that it still allows fast generic functions to use arbitrary method combinations.

4.4 Call-site Optimization

Once we have determined that a fast generic function can be optimized at a particular call site, and have computed a suitable effective method, we still have to invoke the effective method somehow. To do so, we support three options:

(1) Inlining of the entire effective method.
(2) Inlining of keyword parsing only.
(3) Only bypassing the generic function dispatch.

Option (1) can increase the size of the generated code dramaticaly, so it is only chosen when all applicable methods have the `inlineable` method property. Option (2) can be chosen when the fast generic function expects keyword arguments. It generates an inline lambda function that performs all necessary keyword parsing, and then calls a (not inlined) custom variant of the effective method with required arguments only. Option (3) is the default when none of the two previous options applies. It can still lead to faster performance than a regular generic function call because there is zero dispatch overhead, and because we generate effective methods that are faster than those of most implementations[2].

5 LIMITATIONS

In this section, we outline the quirks and limitations of our technique.

No Redefinability. Once a domain of a generic function has been sealed, there is no way to update any of the methods therein. This means that it is not possible to retroactively fix bugs in any sealed method. We are thinking about adding a `unseal-domain` function for development. But currently, the only portable way of fixing a bug in a sealed function is by loading the fixed code into a new Lisp image.

[2]This is not to say we are better than, say, PCL. We just exploit the additional restrictions that we have imposed on fast generic functions and methods.

```
(defun class-subclasses (class)
  (when (symbolp class)
    (setf class (find-class class)))
  (let ((table (make-hash-table)))
    (labels
        ((subclasses (class)
           (unless (gethash class table)
             (setf (gethash class table) t)
             (cons
              class
              (mapcan
               #'subclasses
               (closer-mop:class-direct-subclasses
                class))))))
      (subclasses class))))

(defmacro replicate-for-each-vectoroid
    (symbol &body body)
  `(progn
    ,@(loop for class in (class-subclasses 'vector)
        unless (subtypep class '(array nil))
           append (subst class symbol body))))

(replicate-for-each-vectoroid #1=#:vectoroid
  (defmethod first-elt ((vector #1#))
    (elt vector 0)))
```

Figure 3: An example of specializing on various subclasses of `vector`.

No Inlining of Discriminating Functions. We deliberately do not inline discriminating functions, primarily to avoid code bloat. This means that users have to make sure that the compiler can statically compute the list of applicable methods for all relevant cases. To compute this list, the compiler has to be able to distinguish whether any sealed method is applicable or not. This can be problematic when a method has EQL specializers or very narrow types (If the compiler cannot prove that an argument is either matches that specializer or doesn't match that specializer, there will be no optimization at all). For example, users should not specialize the same argument both on `list` and `null`, or the compiler cannot optimize calls to objects that might be either conses or NIL.

Non-Portable Specializers. Specializers are either classes or EQL specializers, not types. This is problematic in our case, since the Common Lisp standard doesn't mandate that each primitive type has a corresponding class. An example is the type `single-float`, where the standard only guarantees the existence of a `float` class. Another desirable case is that of specializing on subtypes of `vector`, which would allow for dispatching on the element type. The good news is that doing so works in practice, and can even be made portable by using a macro that replicates a method template for all subclasses of a supplied class. The list of subclasses can be obtained using CLOSER-MOP[3]. An example of such a macro is given in Figure 3.

6 EXAMPLES

In this section, we show how fast generic functions can be used in practice.

6.1 Extensible Number Functions

This first example is about using fast generic functions to define an extensible version of the function +. To do so, we first define a single fast generic function for two argument addition[3] in Figure 4.

```
(defgeneric binary-+ (x y)
  (:generic-function-class fast-generic-function))
```

Figure 4: A generic function for two-argument addition.

With this definition in place, we can add individual methods. Initially, a single method like the one shown in Figure 5 will suffice. More methods would be needed if `cl:+` weren't so flexible already.

```
(defmethod binary-+ ((x number) (y number))
  (+ x y))
```

Figure 5: A method for two-argument addition of numbers.

At this point, neither the generic function `binary-+`, nor its methods are sealed. They can be treated just like standard generic functions and methods. To enable optimization, we have to seal a domain by calling `(seal-domain #'binary-+ '(number number))`. The generic function `seal-domain` will automatically seal the generic function and all methods within the sealed domain. It would also signal an error if some methods were only partially within the sealed domain. But the most important step is that it installs a compiler transformations for calls whose arguments are provably within the sealed domain, and which provably lead to a fixed list of applicable methods.

Before we actually use our new addition function, we define an auxiliary function `generic-+` to extend the behavior of `binary-+` to any number of arguments, as shown in Figure 6.

The compiler macro is necessary, because we don't expect a compiler to be able to inline such a call to `reduce`. Without this compiler macro (or inlining of `reduce`, if it would succeed) there wouldn't be any compile-time type information at the call site of `binary-+` and, consequently, no optimization.

Now, with everything in place, we can use our new addition operator to define other functions. Figure 7 shows such a function, and the corresponding assembler code. We see that each call to the generic function `binary-+` can be reduced to a single assembler instruction. This observation is especially remarkable when we consider the size of the code that has been inlined, as shown in Figure 8.

But the real power of our generic addition operator is that we can still extend it outside of its sealed domain. This capability is illustrated in Figure 9.

[3]A real-world protocol would probably include additional generic functions for implicit coercion and dealing with commutativity.

```
(defun generic-+ (&rest things)
  (cond ((null things) 0)
        ((null (rest things)) (first things))
        (t (reduce #'binary-+ things))))

(define-compiler-macro generic-+ (&rest things)
  (cond ((null things) 0)
        ((null (rest things)) (first things))
      (t (reduce (lambda (a b) `(binary-+ ,a ,b))
                 things))))
```

Figure 6: Extending `binary-+` to any number of arguments.

```
(defun add3 (x y z)
  (declare (single-float x y z))
  (generic-+ x y z))
```

```
mov RAX, [R13+16]
mov [RBP-8], RAX
movaps XMM1, XMM4
addss XMM1, XMM3 ; floating-point addition 1
addss XMM1, XMM2 ; floating-point addition 2
movd EDX, XMM1
shl RDX, 32
or DL, 25
mov RSP, RBP
clc
pop RBP
ret
```

Figure 7: A function using `generic-+` and the corresponding x86-64 assembler code on SBCL.

```
(lambda (#:x-7 #:y-8)
  (let ((.gf. #'binary-+))
    (macrolet ((call-method
                 (method &optional next-methods)
                 ...))
      (flet ((next-method-p () nil)
             (call-next-method ()
               (no-next-method
                .gf.
                (specializer-prototype
                 (find-class 'fast-method)))))
        (funcall
         (lambda (x y)
           (block binary-+ (+ x y)))
         #:x-7 #:y-8)))))
```

Figure 8: The (slightly simplified) inline lambda generated for a single call to `generic-+` on two single floats.

```
(defmethod binary-+ ((x string) (y string))
  (concatenate 'string x y))

(generic-+ "foo" "bar" "baz")
;;; => "foobarbaz"
```

Figure 9: Extending binary-+ to strings.

6.2 Extensible Sequence Functions

In this example, we will show how generic functions can be used to define an extensible sequence protocol. Our approach differs from the extensible sequence protocol by Rhodes [8] in that the sequence functions themselves are generic, with the exception of methods inside of their sealed domains.

Our exemplary sequence protocol consists of just three functions – `generic-length`, `generic-elt` and `generic-count`. The full source code of these functions is shown in Figure 10. Each generic function is extensible except for the domain of vectors. A more realistic protocol would provide more functions, additional keyword arguments, and an additional sealed domain for lists, but none of this is relevant for our explanation. As soon as the generic functions are defined, we use the trick from Figure 3 to generate specialized methods for all subclasses of `vector`.

The main point we want to make with this example is that fast generic functions can be nested. The definition of `generic-count` consists of calls to `generic-length` and `generic-elt`. Nevertheless, all this indirection disappears within a sealed domain, a fact that can be seen by looking at the assembler code for an exemplary function call as in Figure 11.

7 BENCHMARKS

Even though the assembler code in Figure 7 and Figure 11 looks promising in terms of performance, we have also conducted some further real-world benchmarks. We decided to compare the performance of the sequence function `cl:find` on SBCL with an equivalent definition that uses fast generic functions that we wrote for SICL[10], based on some earlier work by Durand and Strandh[5]. For our version, we present timings both with effective method inlining enabled and without. Furthermore, show timings for various element types and numbers of supplied keywords. These keywords have been chosen so that they supply the same values as the corresponding default values, i.e., we supply `#'eql` as the test function, `#'identity` as the key function, the sequence's length as the value of `:end` and *false* as the value of `:from-end`. Our benchmark results are shown in Figure 12.

The benchmark results are very promising. In particular, they show that eliminating the method dispatch and inlining the keyword argument parsing step can yield substantial benefits when working with short sequences. In almost all cases, our implementation of `find` is on par or even slightly faster than that of SBCL. The only cases where SBCL's implementation has a small lead is that of processing lists, and the case of having zero static information at the call site. We hope that we can speed up these cases in the future, too.

```
(defgeneric generic-length (sequence)
  (:generic-function-class fast-generic-function))

(defgeneric generic-elt (sequence index)
  (:generic-function-class fast-generic-function))

(defgeneric generic-count (item sequence &key test)
  (:generic-function-class fast-generic-function))

(replicate-for-each-vectoroid #1=#:vector
  (defmethod generic-length
      ((vector #1#))
    (declare (method-properties inlineable))
    (length vector))

  (defmethod generic-elt
      ((vector #1#) (index integer))
    (declare (method-properties inlineable))
    (elt vector index))

  (defmethod generic-count
      (item (vector #1#) &key (test #'eql))
    (declare (method-properties inlineable))
    (loop for index below (generic-length vector)
          for elt = (generic-elt vector index)
          count
          (funcall test item elt))))

(seal-domain #'generic-length '(vector))
(seal-domain #'generic-elt '(vector integer))
(seal-domain #'generic-count '(t vector))
```

Figure 10: A simple protocol for working with sequences and the corresponding inlineable methods for the domain of vectors.

But the important message for library authors is that a straightforward implementation using fast generic functions is at least as good as any highly specialized hand written dispatch mechanism, yet offers almost all the flexibility and convenience of standard generic functions. This was a very important goal for us, because it means programmers are not forced to choose between maintainability and performance anymore — they can have both.

8 CONCLUSIONS AND FUTURE WORK

We have presented a library for metaobject sealing that requires little more than access to the Metaobject Protocol of CLOS. In a second step, we have demonstrated how these metaobjects can be used to design fast generic functions. Finally, we have provided examples how these fast generic functions solve an infamous problem of Common Lisp — defining generic functions that are extremely efficient for a set of primitive types, yet extensible in general. We are especially proud that fast generic functions support the full set of

```
(defun generic-count-user (vector)
  (declare (simple-vector vector))
  (declare (optimize (safety 0))) ; simplify asm
  (generic-count 5 vector :test #'equal))
```

```
    mov RAX, [R13+16]
    mov [RBP-8], RAX
    mov RDI, [RSI-7]      ; (length vector)
    xor EAX, EAX          ; (setf index 0)
    xor ECX, ECX          ; (setf count 0)
    jmp L2
    nop
L0: mov RBX, [RSI+RAX*4+1]; (generic-elt vector index)
    cmp RBX, 10           ; (equal elt 5)
    jeq L3
L1: add RAX, 2            ; (incf index)
L2: cmp RAX, RDI          ; (= index (length vector))
    jl L0
    mov RDX, RCX
    mov RSP, RBP
    clc
    pop RBP
    ret
L3: add RCX, 2            ; (incf count)
    jmp L1
```

Figure 11: A function using `generic-count` and the corresponding x86-64 assembler code on SBCL. Thanks to effective method inlining, the compiler was able to eliminate even the keyword parsing step and could turn the call to `equal` to a single `cmp` instruction.

features of standard generic functions, including arbitrary method combinations and calls to `call-next-method` with arguments.

One issue we still have to address is performance portability. So far, the full set of fast generic function optimizations is only available on SBCL. At the very minimum, we'd like to port these optimizations to CCL and ECL. We are also willing to support closed-source implementations, given that someone can tell us how to access the necessary mechanisms in the compiler.

The other issue we are still working on is that of inheritance from multiple sealable classes. We have to ensure that doing so does not merge two previously disjoint domains of some generic functions. One way around this issue would be to allow only single inheritance of sealable classes, but we are trying to find a less restrictive technique.

This work was born out of our work on extensible sequence functions for the new Common Lisp implementation SICL[10]. Our hope is that what is useful to us is also useful to others. Experience reports and feedback are most welcome!

9 ACKNOWLEDGMENTS

We want to thank everyone on the #sicl IRC channel for their valuable feedback - especially Jan Moringen and Robert Strandh. Furthermore, we want to thank Pascal Costanza for writing and maintaining the invaluable CLOSER-MOP library.

element		1 Element			50 Elements		
type	*k*	SBCL	SICL	Inline	SBCL	SICL	Inline
*	0	30	32	32	447	342	343
*	1	36	60	60	454	371	372
*	2	39	87	87	454	397	396
*	4	51	140	140	466	507	490
single-float	0	20	17	2	422	360	181
single-float	1	20	18	6	444	354	213
single-float	2	21	18	9	445	354	305
single-float	4	21	21	9	436	406	474
list	0	15	15	5	404	424	175
list	1	17	17	7	422	422	263
list	2	17	21	9	402	585	224
list	4	18	23	9	574	696	337

Figure 12: Benchmark results comparing SBCL's built-in implementation of `find` with our implementation in SICL. All timings are given in nanoseconds. The first column describes the statically known vector element type, the second column describes the number of supplied keyword arguments, and the following two groups of three columns describe the results for processing a single element and 50 elements, respectively. Therein, the SBCL columns show the results on SBCL 2.0.1, the SICL columns show the results of our implementation that uses fast generic functions and no inlining, and the Inline columns show the results of our implementation but with inlining enabled.

REFERENCES

[1] Masataro Asai. The inlined-generic-functions library. https://github.com/guicho271828/inlined-generic-function, 2015.
[2] Henry G. Baker. Clostrophobia: Its etiology and treatment. *SIGPLAN OOPS Mess.*, 2(4):4–15, October 1991. ISSN 1055-6400. doi: 10.1145/126983.126984.
[3] Pascal Costanza. The closer-mop library. https://github.com/pcostanza/closer-mop, 2005.
[4] Mark Cox. The specialization-store library. https://github.com/markcox80/specialization-store, 2015.
[5] Irène Durand and Robert Strandh. Fast, maintainable, and portable sequence functions. In *Proceedings of the 10th European Lisp Symposium on European Lisp Symposium*, ELS2017. European Lisp Scientific Activities Association, 2017.
[6] Alexander Gutev. The static-dispatch library. https://github.com/alex-gutev/static-dispatch, 2018.
[7] Gregor Kiczales. *The art of the metaobject protocol.* MIT Press, Cambridge, Mass, 1991. ISBN 978-0262610742.
[8] Christophe Rhodes. User-extensible sequences in Common Cisp. In *Proceedings of the 2007 International Lisp Conference*, ILC '07, pages 13:1–13:14, New York, NY, USA, 2009. ACM. ISBN 978-1-59593-618-9. doi: 10.1145/1622123.1622138.
[9] Andrew Shalit. *The Dylan reference manual : the definitive guide to the new object-oriented dynamic language.* Addison-Wesley Developers Press, Reading, Mass, 1996. ISBN 978-0201442113.
[10] Robert Strandh. SICL: Building blocks for implementers of Common Lisp. In *Proceedings of the 4th European Lisp Symposium on European Lisp Symposium*, ELS2011. European Lisp Scientific Activities Association, 2011.

Bidirectional leveled enumerators

Irène Durand
irene.durand@u-bordeaux.fr

ABSTRACT

An enumerator E outputs the elements of a sequence $\widehat{E} = e_0, e_1, \ldots$, one at a time. The sequence $\widehat{E}$ may be finite $(e_n)_{n \in [0,c[}$ or infinite $(e_n)_{n \in \mathbb{N}}$.

The `Enum` package is part of the `TRAG` system which is written in `Common Lisp`. The code can be found at https://idurand@bitbucket.org/idurand/trag.git and a web interface at trag.labri.fr. The first version of the `Enum` package was presented at ELS 2012 in Zadar (Croatia).

The first version of the package offered the possibility of creating basic enumerators (inductive or from a list) and combining them using operations like products, sequences, filters, mapping. The topic of this paper is the enumeration of the tuples of the general cartesian product $T^p = \widehat{E^1} \times \widehat{E^2} \ldots \times \widehat{E^p}$ of the sequences associated with p enumerators $E^1, \ldots, E^p$. The first property that one would want is *fairness*: each one of the enumerators will regularly move forward. The fairness property was already achieved by the diagonal product enumerator of the 2012 version. In this paper we address an additional property which concerns the distance between two enumerated tuples. A 2-ordering of T^p is such that the distance between two consecutive enumerated tuples is at most 2. The binary diagonal product of the 2012 version had the 2-ordering property. But recursive application of this binary product to obtain T^p does not give a 2-ordering for $p \geq 3$. In this paper, we define bidirectional *leveled* enumerators and a binary product with these enumerators such that recursive application of the binary product gives a leveled 2-ordering which is as desired a 2-ordering.

ACM Reference Format:
Irène Durand. 2020. Bidirectional leveled enumerators. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3740554

1 INTRODUCTION

Since the beginning of the 21st century, the topic of enumeration has become more and more widespread both in theoretical and practical computer science.

An enumerator E outputs the elements of a sequence $\widehat{E} = e_0, e_1, \ldots$, one at a time. The $\widehat{E}$ may be finite $(e_n)_{n \in [0,c[}$ or infinite $(e_n)_{n \in \mathbb{N}}$.

In combinatorics, the problem of enumerating objects comes as a generalization of counting objects. Many recent books deal with theoretical questions raised by enumeration problems [2, 8, 10].

Practically, enumerating sets of objects is essential when the sets are too large (or even infinite) to be computed in extenso. Typical examples are database queries, extraction problems for large data collections like the web, and answers to constraint satisfaction problems [12].

Enumerators[1] are also very useful for programming. In `Python2`, the function `range(n)` computes and returns the list

```
[0, 1, ..., n - 1].
```

People used to write "`for i in range(n):`" to perform an iteration on the elements of the list `[0, ..., n - 1]`. The iterator of the list was provided by `xrange(n)`. In order to avoid the computation of the list the programmer could write

```
  for i in xrange(n):
```

Now in `Python3` (starting from 2008), `xrange` has become the basic operation and has been renamed `range`. In order to obtain the behavior of `range(n)` of `Python2` one must write `list(range(n))`

```
Python 3.6.9 (default, Nov  7 2019, 10:44:02)
>>> range(3)
range(0, 3)
>>> for i in range(3):
...     print(i, end=' ')
... print(' ')
0 1 2
>>> list(range(3))
[0, 1, 2]
```

Some programming languages (`Java`, `Lisp` [7, 13]) provide libraries for enumerating simple objects such as lists, arrays, strings or hash tables, but no operation to combine these simple enumerators in order to build enumerators for more complex user defined objects. The Sage [11] software is a free open-source mathematics software system licensed under the General Public License. It combines the power of many existing open-source packages into a common Python-based interface. It implements enumeration for complex objects like *graphs, posets,* etc.

However, Sage does not handle terms and term automata which are the objects we were interested in for the framework of the `TRAG`[5, 6] system which is almost[2] entirely written in `Common Lisp`. Also, we would like a Lisp implementation rather than a Python one. That is why we initially started developing the `Enum` package.

The `SERIES Lisp` package by Richard C. Waters[9] also deals with finite or infinite sequences. However, these series are not enumerators in the sense that a series or a finite consecutive part of it is treated as a blackbox. There is no explicit cursor that moves along the elements of a series and which is accessible to the user. However, such cursors must exist in the implementation. Otherwise an operation like computing the cartesian product of two series would not be possible.

Many basic features provided by our enumeration package are essentially the same as the ones provided by the `SERIES` package. The essential difference is that an enumerator points to a current element of its underlying sequence. A series is a functional object

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27–28 2020, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.3740554

[1] There are sometimes called *iterators* in the programming context.
[2] The web interface code contains some `JavaScript`.

while an enumerator is a state machine (a non functional object). In other words, there may be several enumerators, pointing to different elements of the same (possibly virtual) sequence while there is just one series for a given sequence.

Our package also gives an object-oriented version of these concepts while the SERIES package does not use generic functions nor standard classes. The use of CLOS gives object-oriented extensibility that we do not have with the SERIES package. We make great use of this extensibility in TRAG and in the implementation presented in this paper.

The infinite lists of Haskell[1] offer the same kind of possibilities as the SERIES package.

Our Enum package is part of the TRAG[3] system[5, 6] which is written in Common Lisp. The code can be found at https://idurand@bitbucket.org/idurand/trag.git. The first version of this package was presented at ELS 2012 [4] in Zadar (Croatia). That first version of the package offered the possibility of creating basic enumerators (inductive or from a list) and combining them using operations like products, sequences, filters, and mapping.

Our enumerators may be useful for Lisp programmers and Lisp implementors as shown by the following implementation of the map function from the Common Lisp HyperSpec. This function must deal with sequences of heterogeneous types: some are lists, some are vectors. By creating an enumerator for each sequence, we obtain an homogeneous list of enumerators which can be put in parallel to obtain the tuples to which we apply the function fun passed as second parameter. In case the result-type is not NIL, collect-enum collects the final elements of the sequence into a list which is converted to the desired result-type.

```
(defun map (result-type fun &rest sequences)
  (let ((enumerator
          (make-parallel-enumerator
           (mapcar
            (lambda (s)
              (make-sequence-enumerator s))
            sequences)
           :fun (lambda (tuple)
                  (apply fun tuple)))))
    (if (null result-type)
        nil
        (coerce (collect-enum enumerator) result-type))))
```

The topic of this paper is the enumeration of the tuples of the general cartesian product $T^p = \widehat{E^1} \times \widehat{E^2} \ldots \times \widehat{E^p}$ of the sequences associated with p enumerators $E^1, \ldots, E^p$. A 2-ordering of T^p is such that the distance between two consecutive enumerated tuples is at most 2. The binary diagonal product of the 2012 version had the 2-ordering property. But recursive application of this binary product to obtains T^p does not give a 2-ordering for $p \geq 3$. In this paper we define bidirectional *leveled* enumerators and a binary product with these enumerators such that recursive application of the binary product gives a leveled 2-ordering which is as desired a 2-ordering.

[3]trag.labri.fr

2 ENUMERATORS AND BIDIRECTIONAL ENUMERATORS

In the following, the sequence enumerated by an enumerator E will be denoted by $\widehat{E}$.

An enumerator E is a state machine which outputs the elements of a sequence $\widehat{E} = e_0, e_1, \ldots$ one at a time. The sequence may be finite $(e_n)_{n\in[0,c[}$ or infinite $(e_n)_{n\in\mathbb{N}}$.

All function names ending with -p are are predicates.

2.1 General enumerators

In the Enum package, each enumerator E has at least the following elementary operations:

- next-element-p (E): does there exist a next element?
- next-element (E): move to the next element.

For the implementation, we also need

- init-enumerator (E): put E in its initial state
- copy-enumerator (E): return an independent copy of E

Examples with a finite enumerator.

```
ENUM> (setq *abc* (make-list-enumerator '(a b c)))
=> #<LIST-ENUMERATOR {100ADDAAC3}>
ENUM> (next-element *abc*) => A
ENUM> (next-element *abc*) => B
ENUM> (next-element-p *abc*) => T
ENUM> (next-element *abc*) => C
ENUM> (next-element-p *abc*) => NIL
ENUM> (collect-enum *abc*) => (A B C) ; only if finite
```

The function collect-enum may be used on a finite enumerator E and returns the elements of the sequence $\widehat{E}$ as a list.
All the upcoming code will implicitly take place inside the ENUM package.

Examples with an infinite enumerator.

```
(setq *naturals* (make-inductive-enumerator 0 #'1+))
=> #<INDUCTIVE-ENUMERATOR {100AEA9653}>
(next-element *naturals*) => 0
(next-element *naturals*) => 1
(next-element *naturals*) => 2
(init-enumerator *naturals*)
(next-element *naturals*) => 0
(next-element *naturals*) => 1
(next-element-p *naturals*) => T ; always true
;; collect the first 9 values
(collect-n-enum *naturals* 9) => (0 1 2 3 4 5 6 7 8)
(collect-n-enum *abc* 9) => (A B C)
```

The function collect-n-enum may be used on any enumerator E and a natural integer n and returns the first n elements of the $\widehat{E}$ as a list (or all the elements of $\widehat{E}$ if n is less than the number of elements of $\widehat{E}$).

2.2 Bidirectional enumerators

A *bidirectional enumerator* B is based on an underlying non bidirectional enumerator E. The bidirectional enumerator has additional code in order to move backwards as well as forwards. In addition to the operations defined for all enumerators, B has a way (+1 to move forwards, -1 to move backwards), an initial-way and the following operations to handle ways:

- `initial-way (B)`: return initial way
- `way (B)`: return current way
- `invert-way (B)`: invert `way` of `B`

together with the following operations:

- `way-next-element-p (way B)`:
  does there exist a next element in this `way`?
- `way-next-element (way B)`:
  move to the next element in this `way`.
- `latest-element (B)`: return latest element enumerated.

The operations `next-element-p` and `next-element` can be written in terms of `way-next-element-p` and `way-next-element`:

```
(defun next-element-p (B) (way-next-element-p (way B) B))

(defun next-element (B) (way-next-element (way B) B))
```

The implementation of a bidirectional enumerator uses a slot `latest-element` to store the latest enumerated element, and two slots `past-elements` and `future-elements`, the first one containing a stack of already enumerated elements that occur before the latest enumerated element and the second a stack of the elements that occur after.

- If the enumerator is moving backwards:
  the top element of `past-element` is popped and moved to `latest-element` while the former `latest-element` is pushed on `future-elements`.
- If the enumerator is moving forwards:
  if `future-elements` is empty the underlying enumerator `E` is called and the result is pushed on `future-elements`; then the top of `future-elements` is popped and moved to the slot `latest-element`.

In both cases, `latest-element` is returned.

2.2.1 Creation and initialization of a bidirectional enumerator. Given a nonempty enumerator `E`, enumerating $e_0, e_1, \ldots$, one can obtain its bidirectional version `BE` with the operation:

`make-bidirectional-enumerator (E &key (initial-way 1))`

Let `BE =`
`(make-bidirectional-enumerator E :initial-way initial-way)`.

In `BE`, one has access to `E`, the *underlying enumerator*, by `(enum BE)`.

At initialization, if `initial-way` is `-1`, we move `(enum BE)` forwards, so towards the first element of `E`, e_0, in order to go back to this element at the next call of `next-element`. Consequently, the first call `(next-element-p BE)` will return `true`, the first call `(next-element BE)` will return the first element of `(enum E)` that is e_0; then `(next-element-p BE)` will return `NIL` as long as its `way` remains `-1`.

2.2.2 Example of creation and use of a bidirectional enumerator.

```
(setq *B-NATURALS*
  (make-bidirectional-enumerator *naturals*))
=> #<BIDIRECTIONAL-ENUMERATOR {100B59FEB3}>
(next-element *B-NATURALS*) => 0
(next-element *B-NATURALS*) => 1
(next-element *B-NATURALS*) => 2
(way *B-NATURALS*) => 1
(invert-way *B-NATURALS*) => -1
(way *B-NATURALS*) => -1
(next-element *B-NATURALS*) => 1
```

```
(describe *b-naturals*) =>
#<BIDIRECTIONAL-ENUMERATOR {100B61AF13}>
  [standard-object]

Slots with :INSTANCE allocation:
  ENUM                            =
  #<INDUCTIVE-ENUMERATOR {100B61AED3}>
  INITIAL-WAY                     = 1
  WAY                             = -1
  LATEST-ELEMENT                   = 1
  LATEST-ELEMENT-P                 = T
  PAST-ELEMENTS                    = (0)
  FUTURE-ELEMENTS                  = (2)
; No value

(next-element *B-NATURALS*) => 0
(next-element-p *B-NATURALS*) => NIL
(invert-way *B-NATURALS*) => 1
(next-element-p *B-NATURALS*) => T
(next-element *B-NATURALS*) => 1
(next-element *B-NATURALS*) => 2
(latest-element *B-NATURALS*) => 2
(way-next-element -1 *B-NATURALS*) => 1
```

3 ENUMERATION OF CARTESIAN PRODUCTS

Let $E^1, \ldots, E^p$ be nonempty enumerators (finite or not) such that each E^i enumerates

- $e^i_0, e^i_1, \ldots$ if E^i is infinite
- $e^i_0, e^i_1, \ldots, e^i_{c^i-1}$ where $c^i = card(E^i)$ otherwise.

Let $T^p = \widehat{E^1} \times \widehat{E^2} \ldots \times \widehat{E^p}$ be the cartesian product of the sequences associated with the E^i. It consists of all the tuples $t = (e^1_{j^1}, e^2_{j^2}, \ldots, e^p_{j^p})$ such that $\forall i \in [1, p], e^i_{j^i} \in E^i$.

If every E^i is finite, $card(T^p) = \Pi^p_{i=1} c^i$ otherwise T^p is infinite.

There are multiple ways of ordering T^p so multiple possible enumerators of T^p. A random ordering would be possible but we would have to store all the previously enumerated tuples in order to avoid enumerating them again. We now discuss some interesting properties which an ordering may have.

3.1 Fairness property

The necessity of a diagonal ordering arises when at least one of the components is infinite. We would like to avoid being blocked at a given value of some component while enumerating an infinite other component. We call this property *fairness*. For instance in the example above, when enumerating `*naturals*` × `*abc*`, we would like to avoid enumerating: `(0 A) (1 A) (2 A) (3 A) ...` as shown in Figure 1 and never switch to the `B` value of the `*abc*` enumerator.
We would rather want something like

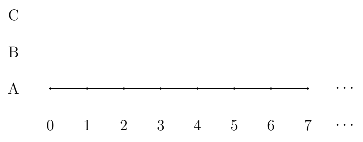


Figure 1: Unfair ordering `*naturals*` × `*abc*`

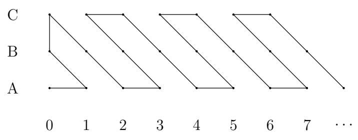


Figure 2: Diagonal ordering of `*naturals*` × `*abc*`

```
(collect-n-enum
  (make-product-enumerator (list *naturals* *abc*) 10)) =>
((0 A) (1 A) (0 B) (0 C) (1 B) (2 A) (3 A) (2 B) (1 C) (2 C))
```

where we move forwards regularly on all enumerators as shown in Figure 2.

For simplicity, because there is a bijection between the indices $0, 1, 2, \ldots$ and the elements of a sequence $e_0, e_1, e_2, \ldots$, we will use examples where each E^i is either $\mathbb{N}$ or some finite interval $[0, c-1] \subset \mathbb{N}$.

Figure 3 shows a diagonal-ordering of $[0, 1] \times [0, 1]$ and Figure 4 a diagonal ordering of $[0, 2] \times [0, 2]$. This diagonal ordering was the one implemented in [4]. However, this ordering lacks an interesting property that we present below.

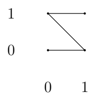


Figure 3: Diagonal-ordering of $[0, 1] \times [0, 1]$

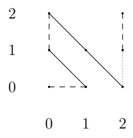


Figure 4: The diagonal-ordering of $[0, 2] \times [0, 2]$

3.2 Bijective enumerators

To simplify the definitions of the next section, we assume that each enumerator E^i is bijective (all its elements are distinct). If this is not the case, it is easy to transform a non bijective enumerator into a bijective one by making it run in parallel with a bijective one like `*naturals*` which enumerates $\mathbb{N}$. This transformation is illustrated below.

```
(setq *e* (make-list-enumerator '(a b c) :circ t))
=> #<LIST-ENUMERATOR {101680B8F3}>
(collect-n-enum *e* 10)
=> (A B C A B C A B C A)
(defparameter *bijective*
  (make-parallel-enumerator (list *naturals* *e*)))
=> #<PARALLEL-ENUMERATOR {101680C4E3}>
(collect-n-enum *bijective* 8)
=> ((0 A) (1 B) (2 C) (3 A) (4 B) (5 C) (6 A) (7 B))
```

Note the `:circ` keyword parameter that makes `*e*` an infinite enumerator cycling on the elements of the list `(A B C)`.

3.3 d-orderings

We are going to define a concept of distance between two enumerated tuples. The aim will be to minimize the maximum distance between two consecutively enumerated tuples.

Definition 1. The *distance* between two tuples $t_j = (e^1_{j^1}, \ldots, e^p_{j^p})$ and $t_k = (e^1_{k^1}, \ldots, e^p_{k^p})$ of the cartesian product T^p is defined by

$$d(t_j, t_k) = \Sigma^p_{i=1} \mid k^i - j^i \mid .$$

This definition works well if every E^i is bijective, that is to say that if $p \neq q$ then $e^i_p \neq e^i_q$; otherwise we would not have a mapping from tuple of indices to tuples of elements. If some E^i is not bijective, we may transform it into a bijective one as described in Section 3.2.

Definition 2. A pair of consecutive elements of an enumerator E is called a *step*. If there is a concept of distance between the elements (for instance E enumerates tuples), the *size of a step* (e_j, e_{j+1}) is the distance between the two elements: $d(e_j, e_{j+1})$.

Definition 3. An ordering of T^p is a d-ordering if d is the maximum size of a step.

Our aim is to have a 2-ordering of T^p.

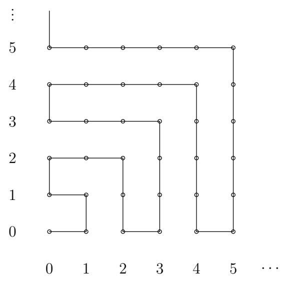


Figure 5: A 1-ordering of $\mathbb{N} \times \mathbb{N}$

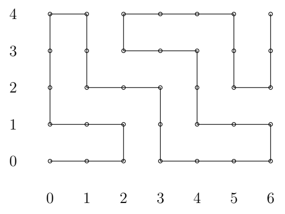


Figure 6: A 1-ordering of $[0, 6] \times [0, 4]$

Figure 5 shows a 1-ordering of $\mathbb{N}\times\mathbb{N}$. This ordering is not feasible in our setting for arbitrary (finite or non finite) enumerators because we would need to know one step in advance whether we have reached the end of a finite enumerator to turn around before it is too late. So the ordering depends on the parity of the size of the finite enumerators as shown in Figure 6 and Figure 7. [3] explores this idea in detail. Unfortunately, we just have the `next-element-p` predicate to know whether there is a next element but no way to know whether there are two upcoming elements.

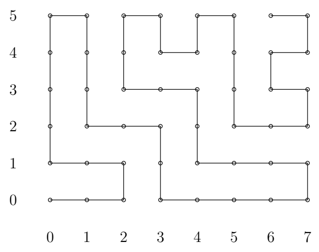


Figure 7: A 1-ordering of $[0,7] \times [0,5]$

In [4], the ordering of T^2 given by the binary diagonal product `DP(E1, E2)` is a 2-ordering. But recursive use of this ordering does not preserve the 2-ordering property as shown below. Figure 3 shows a diagonal ordering of $[0,1] \times [0,1]$ which is a 2-ordering. However using the same ordering again to obtain an ordering for $[0,1] \times [0,1] \times [0,1]$, we obtain the ordering shown in Figure 8 which is not a 2-ordering but a 3-ordering because of the step $(e_3, e_4) =$ (`(0 0 1)`, `(1 1 0)`) which is of size 3. More generally, we can show that repeated use of this ordering gives a p-ordering of $[0,1]^p$.

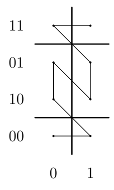


Figure 8: The diagonal ordering of $[0,1]^3$

3.4 Leveled ordering of T^p

The concept of leveled ordering is necessary to achieve our goal of a 2-ordering of $T^p = \Pi_{i=1}^p \widehat{E^i}$. A level will be the subset of tuples having identical height. We now need the notion of *height of a tuple* in the cartesian product.

Definition 4. The *height* of a tuple $t = (e^1_{j^1}, e^2_{j^2}, \ldots, e^p_{j^p}) \in T^p$, is the sum of the indices of the elements in the $\widehat{E^i}$:

$$h(t) = \Sigma_{i=1}^p j^i$$

Note that in the case where each E^i is either $\mathbb{N}$ or $[0, c-1] \subseteq \mathbb{N}$, the height of a tuple is the sum of its elements.

Definition 5. The l^{th}*-level* of T^p is the set of tuples with height l.

The l^{th} level (finite) of T^p is denoted by L_l,

$$L_l = \{t \in T^p | h(t) = l\}$$

If T^p is finite, it has a finite number of levels and can be written as the partition of its levels:

$$T^p = \bigcup_{l=0}^{\Sigma_{i=1}^p (c^i - 1)} L_l$$

If T^p is infinite, it has an infinite number of levels:

$$T^p = \bigcup_{l=0}^{\infty} L_l$$

Definition 6. An ordering of T^p is *leveled* if it satisfies the following constraints:
$\forall l > 0, \forall j > 0$, if $t_j \in L_l$, we have either $t_{j-1} \in L_l$ or $t_{j-1} \in L_{l-1}$.

In other words a leveled ordering traverses the levels $L_0, L_1, \ldots$ in the increasing order of levels $L_0, L_1, \ldots$ (without constraint so far on the order of enumeration inside a level).

Definition 7. A step giving a change of level is called *major step*. A step inside a level is called a *minor step*.

In addition, leveled enumerators have the predicate:

- `minor-step-p (E)`

which returns `T` if the next step (`next-element`) does not change the level (`NIL` otherwise). In other words, it returns `false` when we are done with the enumeration of the current level.

4 IMPLEMENTATION OF BIDIRECTIONAL LEVELED ENUMERATORS

Definition 8. A *bidirectional leveled* enumerator is a leveled enumerator which, in addition, is bidirectional (it has a `way` and an `initial-way`).

When going forwards (`way = +1`), it enumerates the levels in increasing order: $L_0, L_1, \ldots$ When going backwards (`way = -1`), it enumerates the levels in decreasing order: $L_l, L_{l-1}, \ldots$ while keeping the forward order inside each level.

4.1 Diagonal product of a bidirectional enumerator with a bidirectional leveled one

Let `X` be a bidirectional enumerator and `Y` be a bidirectional leveled enumerator, which when going forwards, enumerates the levels $Y_0, Y_1, \ldots$

Below, we define `D = BL(X, Y)`, the *leveled bidirectional product* of `X` and `Y`.

When `D = BL(X, Y)` is created, the initial way of `X` is set to `+1` and the initial way of `Y` is set to `-1`.

Definition 9. A *minor step on level* is a step that changes the level of `X` in a way and the level of `Y` in the opposite way but not the level of `D`.

In addition to the usual operations, we have the following accessors:

- `enumX(D)`
- `enumY(D)`

to access `X` and `Y` respectively.

The other operations are shown in Figure 9.

The call `(sliding-step X Y 1)` corresponds to a *jump-up* (move to higher level) `(sliding-step X Y -1)` corresponds to a *jump-back* (move to lower level).

In the case where neither `X` nor `Y` can move in their current way and the enumeration is not finished, we are in a case called *corner step* which may happen only when at least one of the enumerators is finite (otherwise there is always a possible *sliding step*). In the

```
(defun latest-element (D)
  (cons (latest-element(enum-x D)
        (latest-element (enum-y D)))))

(defun minor-step-p (D) ; precondition (next-element-p D)
  (and (next-element-p (enum-y D))
       (or (next-element-p (enum-x D))
           (minor-step-p (enum-y D)))))

(defun way-next-element-p (way D)
  (or (way-next-element-p (way D) (enum-x D))
      (way-next-element-p (way(D) (enum-y D)))))

(defun way-next-element (way D)
  (let* ((enum-x (enum-x enum))
         (enum-y (enum-y enum))
         (next-x (next-element-p enum-x))
         (next-y (next-element-p enum-y)))
    (cond
      ((and next-y (minor-step-p enum-y))
      ;; lower-level minor step
       (next-element enum-y))
      ((and next-y next-x) ; minor-step on level
      ;; each one makes a major in its way
       (next-element enum-x) (next-element enum-y))
      ;; major step
      ((not (or next-x next-y))
       (corner-step enum-x enum-y way))
      (t (sliding-step enum-x enum-y way))))
  (latest-element enum))

(defun sliding-step (X Y way)
;; precondition: X or Y can move in its way
  (if (next-element-p Y)
      (way-next-element way Y)
      (way-next-element way X))
  (invert-way X)
  (invert-way Y))
```

Figure 9: Code for `D = BL(X,Y)`

corner step case, we invert the way of the enumerator which goes in the opposite direction of `way` (of the product enumerator) and move it to the next level according to `way`. If `way = 1`, we move to the higher level. If `way = -1`, we move to the lower level. The other enumerator changes way (it could not contribute to the level change because it is blocked in the direction `way`).

```
(defun corner-step (X Y way)
;; change the way of the enumerator
;; which goes in opposite direction
;; to way and move it; the other enumerator changes way
  (when (plusp (* way (way X)))
    ;; put in X the one that goes in direction -way
    (psetf X Y Y X))
  (invert-way X)  ; X will move in direction way
  (next-element X) ; Y will move in direction -way
  (invert-way Y))
```

Note that the diagonal ordering and the leveled ordering coincide in the binary case ($p = 2$).

In Figure 10, one may see a corner step (dotted line) and sliding steps (dashed line).

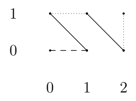


Figure 10: Diagonal (also leveled) ordering of $[0, 2] \times [0, 1]$

4.2 Properties of `BL(X, Y)`

We will show that if Y defines a leveled ordering so does `BL(X, Y)`.

Let `X` be a bidirectional enumerator that enumerates $x_0, x_1, \ldots$ and `Y` a bidirectional leveled enumerator that enumerates the levels $Y_0, Y_1, \ldots$

Let $L_0, L_1, \ldots$ be the levels of `D=BL(X, Y)`.

The distribution of an element among all tuples of a sequence is denoted by $*$:

$1 * (0, 1), (0, 2), (1, 1) = (1, 0, 1), (1, 0, 2), (1, 1, 1)$

and the concatenation of lists of tuples by $+$

We will show that

$$L_0 = x_0 * Y_0$$
$$L_1 = x_1 * Y_0 + x_0 * Y_1$$
$$L_2 = x_0 * Y_2 + x_1 * Y_1 + x_2 * Y_0$$

and more generally that for $l \geq 0$:

$$L_{2l} = x_0 * Y_{2l} + x_1 * Y_{2l-1} + \ldots + x_{2l} * Y_0$$
$$L_{2l+1} = x_{2l+1} * Y_0 + x_{2l} * Y_1 + \ldots + x_0 * Y_{2l+1}$$

$\forall l \geq 0$, we note

$$Y_l = \{y_{l,0}, y_{l,1}, \ldots, y_{l,k_l-1}\}$$

where k_l is the number of elements in the level Y_l.

Lemma 10. `X` and `Y` move in opposite directions.

Proof. Initially, `X` moves forwards and ready to enumerate x_0 and `Y` moves backwards ready to enumerate $y_{0,0}$. In the code, we see that `X` and `Y` change direction simultaneously. □

Lemma 11.

(1) `X` moves forwards when enumerating level 0 and backwards when enumerating level 1.
(2)
$$L_0 = x_0 * Y_0 \quad (a)$$
$$L_1 = x_1 * Y_0 + x_0 * Y_1 \quad (b)$$

Proof. Initially, `X` moves forwards ready to enumerate x_0 and `Y` moves backwards ready to enumerate $y_{0,0}$. So 1., and 2. hold.

At the first call to `next-element(D)`, we have `(and next-x next-y)` and we do a `minor` step `on level` with

`(next-element X)` and `(next-element Y)`;

the enumerators remain in the their current way. The enumerated element is $(x_0, y_{0,0})$ and `X` moves forwards.

The next calls will do `minor` steps `on Y` in which only Y moves forwards: $(x_0, y_{0,1}), (x_0, y_{0,2}) \ldots (x_0, y_{0,k_0})$. After k_0 calls, we will have enumerated $x_0 * Y_0$. So 2.(a) holds. Point 1. remains true.

At the next call (next-element D), the call (minor-step-p Y) will return NIL because we have finished level 0 on Y. We will do a sliding_step.
As (next-element Y) returns NIL, X will be the one that moves forwards (moves to level 1 because X is moving forwards) and both change direction so X will move backwards and Y forwards. The enumerated element is $(x_1) + y_{0,0}$ (1. holds). Then we will have minor steps on Y until the end of level 0 of Y, $(x_1) + y_{0,1}, \ldots, (x_1) + y_{0,k_1}$. So we have done $x_1 * Y_0$.

The next step will be a minor step on level (X and Y change directions but not D); we enumerate $(x_0) + y_{1,0}$ and X goes forwards (1. holds). Then minor steps on Y: $(x_0) + \ldots (x_0) + y_{1,k_1}$ which gives $x_0 * Y_1$ (1. still holds); so we have done $x_1 * Y_0 + x_0 * Y_1$ (2.(b) hold). After $2k_0 + k_1$ steps, we will have enumerated $x_0 * Y_0 + x_1 * Y_0 + x_0 * Y_1$. □

Proposition 12.

(1) X moves forwards when enumerating an even level.
(2) D is a bidirectional leveled enumerator whose levels $L_0, L_1, \ldots$, are such that for $l > 0$,

$$L_{2l} = x_0 * Y_{2l} + x_1 * Y_{2l-1} + \ldots + x_{2l} * Y_0$$
$$L_{2l+1} = x_{2l+1} * Y_0 + x_{2l} * Y_1 + \ldots + x_0 * Y_{2l+1}$$

(3) In the negative way, D enumerates $L_l, L_{l-1}, \ldots$,.

Proof. By induction on l.
Base case $l = 0$: solved by lemma 11
Induction $l + 1$: Induction hypothesis yields for l:

(1) X moves backwards when enumerating $2l + 1$.
(2)

$$L_{2l} = x_0 * Y_{2l} + x_1 * Y_{2l-1} + \ldots + x_i * Y_0$$
$$L_{2l+1} = x_{2l+1} * Y_0 + x_{2l} * Y_1 + \ldots + x_0 * Y_{2l+1}$$

After enumeration of the last element of level L_{2l+1}, X is going backwards.

The next step yields a sliding step which triggers a major step in Y then a change of direction of both enumerators which gives $x_0 * y_{2(l+1),0}$ the first element of level $2(l + 1)$ with X going forwards (1. hold) and Y going backwards. Then we will have minor steps of level $2(l + 1)$ on Y which will give: $x_0 * Y_{2(l+1)}$.

Then a minor step on level, where X moves forwards and Y moves to the beginning of the lower level $(x_1) + y_{2l+1,0}$, X remains positive.

Then minor steps on Y yielding $(x_1) + Y_{2l+1}$. □

5 DIAGONAL ENUMERATION OF A CARTESIAN PRODUCT

Definition 13. Let Null be the bidirectional leveled enumerator corresponding to the empty product enumerating the singleton set containing a single tuple of length 0.

$$\texttt{Null} = \{()\}$$

This enumerator has only one level $L_0 = \{()\}$.

Lemma 14. Let X be a bidirectional enumerator enumerating $x_0, x_1, \ldots$ $\mathsf{BL}(X, Null)$ is a bidirectional leveled enumerator enumerating $(x_0), (x_1) \ldots$ whose levels are $L_i = \{(x_i)\}$ and having only major steps of size 1.

5.1 BL(X,Y) preserves leveled orderings

Lemma 15. Let $E^1, E^2, \ldots, E^p$ be bidirectional enumerators. The enumerator $\mathsf{BL}(E^1, \mathsf{BL}(E^2, \mathsf{BL}(\ldots, \mathsf{BL}(E^p, \texttt{Null}))))$ is a bidirectional leveled enumerator and a leveled ordering of $T^p = \Pi_{i=1}^{p} \widehat{E^i}$.

Proof. By induction on p.

If $p = 1$ Lemma 14 applied with $X = \mathsf{BL}(E^1, Null)$ yields the desired result.
Induction hypothesis

$$\mathsf{BL}(E^2, \mathsf{BL}(E^3, \mathsf{BL}(\ldots, \mathsf{BL}(E^p, \texttt{Null}))))$$

yields the leveled 2-ordering of $E^2 \times \ldots \times E^p$.
Induction step Let $X = E^1$ and

$$Y = \mathsf{BL}(E^2, \mathsf{BL}(E^3, \mathsf{BL}(\ldots, \mathsf{BL}(E^p, \texttt{Null}))))$$

Let $\mathsf{BL}(X, Y)$ defined as in Section 4.1.
By Proposition 12, we have a bidirectional leveled enumerator giving a leveled ordering

□

5.2 BL(X,Y) preserves 2-orderings

Lemma 16. An enumerator X enumerating $x_0, x_1, \ldots$, can be seen as a leveled enumerator whose levels are the singletons $L_i = x_i$ having only major steps of size 1.

Lemma 17. If all major steps of Y have size 1 then all major steps of BL(X,Y) have size 1.

Proof. During a major step of BL(X,Y) (corner step or sliding step) either X or Y makes a major step. This step has size 1 by hypothesis for X and by Lemma 16 for X. □

Lemma 18. If all minor steps of Y have size 2 then all minor steps of BL(X,Y) have size 2.

Proof. If BL(X,Y) does a minor step, it is either a minor step on Y (of size 2 by hypothesis) or it consists of one major step on X and one major step on Y. The major step on X has size 1 by Lemma 16 and the major step on Y has size 1 by Lemma 17. The total size will again be 2. □

Lemma 19. Let $E^1, E^2, \ldots, E^p$ be bidirectional enumerators. The enumerator $\mathsf{BL}(E^1, \mathsf{BL}(E^2, \mathsf{BL}(\ldots, \mathsf{BL}(E^p, \texttt{Null}))))$ is a bidirectional leveled enumerator and a d2-ordering of $T^p = \Pi_{i=1}^{p} \widehat{E^i}$.

Proof. By repeated application of Lemma 18 and Lemma 17
. □

5.3 Leveled 2-ordering

Proposition 20. Let $E^1, E^2, \ldots, E^p$ be bidirectional enumerators. The enumerator $\mathsf{BL}(E^1, \mathsf{BL}(E^2, \mathsf{BL}(\ldots, \mathsf{BL}(E^p, \texttt{Null}))))$ is a bidirectional leveled enumerator and a leveled 2-ordering of $T^p = \Pi_{i=1}^{p} \widehat{E^i}$.

Proof. By Lemma 15 and Lemma 19. □

6 EXAMPLES

In the examples, we will use only integers so that the level of a tuple is the sum of its elements.

```
(setq *e2* (make-list-enumerator '(0 1)))
(setq *e3* (make-list-enumerator '(0 1 2)))
(collect-enum *e2*) => (0 1)
(collect-enum *e3*) => (0 1 2)
(collect-enum (make-product-enumerator *e3* *e3*))
=> ((0 0) (1 0) (0 1) (0 2) (1 1) (2 0) (2 1) (1 2) (2 2))
(collect-enum
  (make-product-enumerator (list *e3* *e3* *e3*)))
=>
((0 0 0) (1 0 0) (0 1 0) (0 0 1) (0 0 2) (0 1 1) (0 2 0)
 (1 1 0) (1 0 1) (2 0 0) (2 1 0) (2 0 1) (1 0 2) (1 1 1)
 (1 2 0) (0 2 1) (0 1 2) (0 2 2) (1 2 1) (1 1 2) (2 0 2)
 (2 1 1) (2 2 0) (2 2 1) (2 1 2) (1 2 2) (2 2 2))
(collect-n-enum
  (make-product-enumerator (list *naturals* *e3* *e3*)) 45)
=>
((0 0 0) (1 0 0) (0 1 0) (0 0 1) (0 0 2) (0 1 1) (0 2 0)
 (1 1 0) (1 0 1) (2 0 0) (3 0 0) (2 1 0) (2 0 1) (1 0 2)
 (1 1 1) (1 2 0) (0 2 1) (0 1 2) (0 2 2) (1 2 1) (1 1 2)
 (2 0 2) (2 1 1) (2 2 0) (3 1 0) (3 0 1) (4 0 0) (5 0 0)
 (4 1 0) (4 0 1) (3 0 2) (3 1 1) (3 2 0) (2 2 1) (2 1 2)
 (1 2 2) (2 2 2) (3 2 1) (3 1 2) (4 0 2) (4 1 1) (4 2 0)
 (5 1 0) (5 0 1) (6 0 1))
```

The latest example is illustrated by Figure 11.

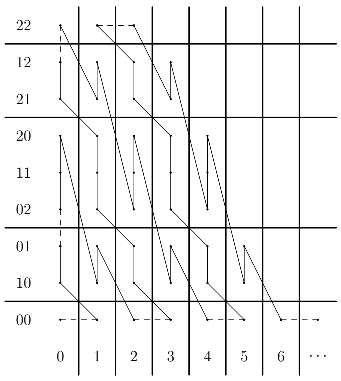


Figure 11: The leveled 2-ordering of $\mathbb{N} \times [0, 3] \times [0, 3]$

The leveled ordering that we have used is not the only possible one. Figure 12 shows another possibility. The code will be almost the same but instead of just inverting the way of Y (which changes the order or enumeration of the levels) we recursively invert the ways of the underlying enumerators of Y which reverses the order of enumeration of Y.

7 CONCLUSION AND FUTURE WORK

We have defined bidirectional leveled enumerators in order to obtain a 2-ordering of a cartesian product of enumerators. The code exists and is available. An iterative implementation that does not use the recursive calls to the binary bidirectional leveled enumerator BL(X, Y) exists but was not discussed in this paper. Although it is

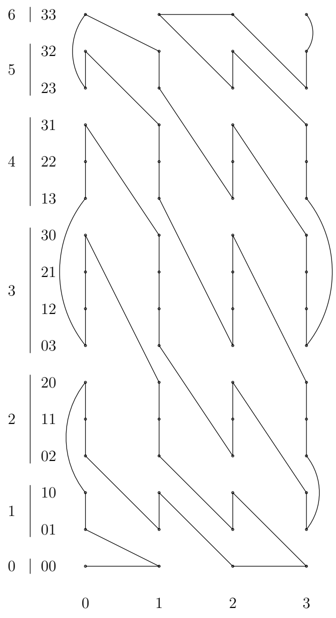


Figure 12: Another leveled 2-ordering of $[0, 3]^3$

in TRAG, the Enum package is self-contained. In the near future we plan to make the Enum package available as an independent system.

REFERENCES

[1] Al. *The Haskell Programming Language.* http://www.haskell.org.
[2] Miklós Bóna. *A Walk Through Combinatorics: An Introduction to Enumeration and Graph Theory Fourth Edition.* World Scientific Publishing Company, 2016. ISBN 9789813148864. URL https://books.google.fr/books?id=uZRIDQAAQBAJ.
[3] Bruno Courcelle, Irène Durand, and Michael Raskin. On defining linear orders by automata. Submitted to Special Issue of Moscow Journal of Combinatorics and Number Theory, November 2019.
[4] Irène Durand. Object enumeration. In *Proceedings of the 5th European Lisp Symposium*, pages 43–57, Zadar, Croatia, May 2012.
[5] Irène Durand. Trag. http://dept-info.labri.u-bordeaux.fr/ idurand/trag, depuis 2015.
[6] Irène Durand. Trag-web. https://trag.labri.fr, depuis 2018.
[7] Enumeration. *Enumeration Package for Common Lisp*, 2012. http://common-lisp.net/project/cl-enumeration/.
[8] I.P. Goulden and D.M. Jackson. *Combinatorial Enumeration.* Dover Books on Mathematics. Dover Publications, 2004. ISBN 9780486435978. URL https://books.google.fr/books?id=ufiJAwAAQBAJ.
[9] Inc. Guy L. Steele, Thinking Machines. *Common Lisp the Language, 2nd edition.* Digital Press, 1990. ISBN 1555580416.
[10] Richard P. Stanley. *Enumerative combinatorics.* Cambridge University Press; 2nd edition, 2000. ISBN 9780521663519.
[11] W. A. Stein et al. *Sage Mathematics Software (Version x.y.z).* The Sage Development Team. http://www.sagemath.org.
[12] Edward Tsang. *Foundations of Constraint Satisfaction.* Academic Press, London and San Diego, 1993. ISBN 0127016104.
[13] Richard. C. Waters. *Series Package for Common Lisp.* http://series.sourceforge.net/.

Later Binding: Just-in-Time Compilation of a Younger Dynamic Programming Language

Max Rottenkolber
max@mr.gy
Interstellar Ventures
Bonn, Germany

ABSTRACT

We examine LuaJIT, an implementation of the dynamic programming language Lua. By using a technique known as *tracing just-in-time compilation* LuaJIT is able to evaluate high-level language features with great efficiency. It does this by using only a conservative set of optimization passes, and without resorting to explicit type declarations, or abandoning type safety. In presenting the implementation's design we consider its strengths and weaknesses. Finally, we propose future directions for dynamic language implementations that wish to leverage this technique.

CCS CONCEPTS

• **Software and its engineering** → **Just-in-time compilers**; *Runtime environments*; **Interpreters**; *Dynamic compilers*; *Extensible languages*; *Functional languages*; *Object oriented languages*.

KEYWORDS

compilers, just-in-time, dynamic languages, object orientation, functional programming, late binding

ACM Reference Format:
Max Rottenkolber. 2020. Later Binding: Just-in-Time Compilation of a Younger Dynamic Programming Language. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.3743226

1 INTRODUCTION

Lua is a minimalist, dynamic programming language with Pascalesque syntax and Schemeish semantics. *LuaJIT* [Pall 2017] is an implementation of a Lua interpreter that uses tracing just-in-time compilation [Bala et al. 2000] to accelerate the evaluation of Lua programs.

A *just-in-time* (JIT) compiler intertwines run-time and compilation-time of the program to leverage run-time information to guide program optimization. Initially, the program is evaluated by a traditional interpreter program [Mccarthy 1960]. But soon enough the interpreter pulls off a magical trick. It considers the program it is evaluating while it is executing it on a given input, and speculatively compiles machine code[1] to perform the remainder of the evaluation on the remaining input.

This trick has interesting implications for the evaluation of dynamically typed, late binding programming languages such as Lua, Smalltalk, and Lisp.[2] In implementations of these languages there tends to be a lot of information about the program available at run-time. However, traditional *ahead-of-time* (AOT) compilers are in many cases not able to leverage this abundance of information to optimize emitted code. At AOT compile-time, information about the types of values and, by the extension, the specialization of methods may be overly conservative due to limits of inference.[3] The result is redundant dispatch on value types during evaluation.

2 MOTIVATING EXAMPLES: COLLAPSING ABSTRACTIONS

Consider this Common Lisp program which computes the sum of the integers $1..n < x$.

```
(defun sum (x)
  (loop for n from 1 to x sum n))
```

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27–28 2020, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.3743226

[1]Machine code here refers to a program for the *Instruction Set Architecture* (ISA) of the computer that runs our interpreter.
[2]Incidentally, Smalltalk hackers pioneered many aspects of modern JIT compilation.
[3]Here we consider implicit type information primarily, (optional) type declarations are a separate *can of worms.*

When compiled with Clozure Common Lisp on x86_64 we can observe the following disassembly for the loop body.

```
L38
    (movq (% save0) (% arg_y))              ;    [45]
    (movq (% save2) (% arg_z))              ;    [48]
    (movl (% arg_y.l) (% imm0.l))           ;    [51]
    (orl (% arg_z.l) (% imm0.l))            ;    [53]
    (testb ($ 7) (% imm0.b))                ;    [55]
    (jne L63)                               ;    [58]
    (cmpq (% arg_z) (% arg_y))              ;    [60]
    (jle L90)                               ;    [63]
    (jmpq L204)                             ;    [65]
L63
    (lisp-call  (@ .SPBUILTIN-GT))          ;    [77]
    (recover-fn-from-rip)                   ;    [84]
    (cmpb ($ 11) (% arg_z.b))               ;    [91]
    (jne L204)                              ;    [95]
L90
    (movq (% save1) (% arg_y))              ;    [97]
    (movq (% save0) (% arg_z))              ;   [100]
    (movl (% arg_y.l) (% imm0.l))           ;   [103]
    (orl (% arg_z.l) (% imm0.l))            ;   [105]
    (testb ($ 7) (% imm0.b))                ;   [107]
    (jne L126)                              ;   [110]
    (addq (% arg_y) (% arg_z))              ;   [112]
    (jno L140)                              ;   [115]
    (lisp-call  (@ .SPFIX-OVERFLOW))        ;   [117]
    (recover-fn-from-rip)                   ;   [124]
    (jmp L140)                              ;   [131]
L126
    (lisp-call  (@ .SPBUILTIN-PLUS))        ;   [133]
    (recover-fn-from-rip)                   ;   [140]
L140
    (movq (% arg_z) (% save1))              ;   [147]
    (movq (% save0) (% arg_z))              ;   [150]
    (testb ($ 7) (% arg_z.b))               ;   [153]
    (jne L174)                              ;   [157]
    (addq ($ 8) (% arg_z))                  ;   [159]
    (jno L196)                              ;   [163]
    (lisp-call  (@ .SPFIX-OVERFLOW))        ;   [165]
    (recover-fn-from-rip)                   ;   [172]
    (jmp L196)                              ;   [179]
L174
    (movl ($ 8) (% arg_y.l))                ;   [181]
    (lisp-call  (@ .SPBUILTIN-PLUS))        ;   [189]
    (recover-fn-from-rip)                   ;   [196]
L196
    (movq (% arg_z) (% save0))              ;   [203]
    (jmpq L38)                              ;   [206]
```

In the disassembly of the compiled loop we quickly see a pattern. Run-time type checks ([55], [107], [153]) followed by a branch to either *fixnum*-specialized code ([60], [112], [159]) or generic run-time dispatch routines (SPBUILTIN-GT, SPBUILTIN-PLUS). Also notably, as characteristic for Lisp, arithmetic overflow is checked for ([115], [163]), and handled by type promotion (SPFIX-OVERFLOW).

Let us look at a similar program written in Lua, and the machine code emitted for the inner loop by LuaJIT.

```
function sum (x)
   local a = 0
   for n=1,x do a = a + n end
   return a
end
```

If the above function was called as sum(100) the following code will end up being executed.

```
->LOOP:
2a53ffe0  xorps xmm6, xmm6
2a53ffe3  cvtsi2sd xmm6, ebp
2a53ffe7  addsd xmm7, xmm6
2a53ffeb  add ebp, +0x01
2a53ffee  cmp ebp, eax
2a53fff0  jle 0x2a53ffe0 ->LOOP
```

Lua uses a single type for all numeric values—typically *double* floats. Hence the accumulator a is held in an SSE floating point register (*xmm*7). LuaJIT managed to infer that the type of the index variable i can be narrowed to a 32-bit integer, held in *ebp*, and converted to a double for arithmetic in *xmm*6. For the actual arithmetic, native integer and floating point addition instructions are emitted (*add*, *addsd*). Notably absent from the inner loop are any dispatches on value types.[4] Any guards needed to ensure correctness of the program—say, what if x is a string?—are hoisted before the loop.

The fundamental difference between the two compilers we just examined is *when* they emit code. Clozure Common Lisp compiles the function ahead of run-time, and emits one code path for all run-time cases possibly encountered during the lifetime of the function. LuaJIT on the other hand emits code at run-time, and only for code paths that are actually executed. Subsequently, emitted code is more narrowly specialized on particular evaluations of the program.

2.1 Object Orientation

Lua has no direct notion of *object oriented* programming. Instead, the built-in *setmetatable* allows programmers to overload the various built-in operators such as indexing (., :) by setting the so-called "metatable" of an object. This mechanism lends itself to all sorts of meta-programming, and enables customizations of the language such as operator overloading, or prototype based object orientation.[5]

```
Acc = {}

function Acc:new ()
   return setmetatable({a=0}, {__index=Acc})
end

function Acc:sum (n)
   self.a = self.a + n
end
```

Again, with a similar program, we look at how LuaJIT compiles a different set of abstractions. Instantiating our accumulator class,

[4] Also absent is handling of arithmetic overflow. However, we argue that this is satisfyingly handled by the underlying hardware's implementation of IEEE double-precision floating point numbers.

[5] As a metatable can itself have a metatable installed, inhertiance comes quite naturally.

and calling its *sum* method in a loop causes the following loop body to be emitted.

```
local a = Acc:new()
for i=1,100 do a:sum(i) end
```

```
->LOOP:
2a53ffe0  xorps xmm6, xmm6
2a53ffe3  cvtsi2sd xmm6, ebp
2a53ffe7  addsd xmm7, xmm6
2a53ffeb  movsd [rax], xmm7
2a53ffef  add ebp, +0x01
2a53fff2  cmp ebp, +0x64
2a53fff5  jle 0x2a53ffe0 ->LOOP
```

To our satisfaction, the emitted code is almost unchanged. The only differences are the store of our accumulator (2a53ffeb) not being forwarded beyond the loop body, and the loop limit being emitted as a constant literal instead of being held in *eax*. Notably, the *sum* method has been inlined, hence there is no function call overhead.

2.2 Functional Abstractions & Polymorphism

Lua supports closures and higher-order functions. Let us try higher-order functions next, and add some gratuitously explicit polymorphism, too.

```
function make_acc ()
   local a
   return function (x)
      if x == nil then
         return a
      elseif type(x) == 'number' then
         a = (a or 0) + x
      elseif type(x) == 'string' then
         a = (a or "") .. x
      end
   end
end
```

For the last example, we create an accumulator closure. We want to see how LuaJIT inlines the closure into the emitted loop body code.

```
local acc = make_acc()
for i=1,100 do acc(i) end
```

```
->LOOP:
2a53ffd0  xorps xmm6, xmm6
2a53ffd3  cvtsi2sd xmm6, ebp
2a53ffd7  addsd xmm7, xmm6
2a53ffdb  movsd [0x41d741d0], xmm7
2a53ffe4  add ebp, +0x01
2a53ffe7  cmp ebp, +0x64
2a53ffea  jle 0x2a53ffd0 ->LOOP
```

The exact same code, again. Where did the branches go? *Dead code elimination* did its trick since LuaJIT could specialize the emitted code on numbers using run-time type information. Within the loop body, code paths for handling strings and *nil* were not emitted at all.

These few examples are intended to show the depths of abstraction that can be collapsed by means of JIT compilation, and to motivate the reader's interest in LuaJIT. In the following sections, we wish to shine a light on LuaJIT's design, and its limitations.

3 ARCHITECTURE AND IMPLICATIONS OF A TRACING JIT COMPILER

At its heart, LuaJIT is a bytecode interpreter. Embedded in this interpreter is a special-purpose run-time profiler. For certain branching bytecodes a table of "hot counts" is maintained. This table is indexed through a hash of the program counter.

Whenever the interpreter encounters one of the bytecodes to be tracked it increments its associated hot count. When incrementing a hot count causes it to overflow beyond a certain value the interpreter will begin recording a trace, starting from the next bytecode instruction.

```
0005  FORI     i=1,n
0006  MODVN    tmp1=i%2
0007  KSHORT   const1=0
0008  ISGE     const1>=tmp1
0009  JMP      if 0008 is true => 0011
0010  ADDVV    A=A+i
0011  FORL     i=i+1, i>n => 0006
```

The exemplary bytecodes above represent a *for* loop that sums odd integers 1..*n*. The `FORL` bytecode controls loop iteration and is tracked in a hot count for the program counter position 0011.

When the `FORL` bytecode becomes hot the interpreter begins recording the following instructions it executes until a trace stop condition is met. In this case, the trace stop condition will be triggered upon encountering the `FORL` bytecode at position 0011 for a second time, or on exiting the loop.

Assuming the loop exit condition is not met, the first bytecode to be executed—and recorded in the trace—will be the `MODVN` bytecode at position 0006, which calculates modulo 2 of *i*. The next bytecodes recorded are then 0007..0008 which load the constant zero, and check if *i* is odd—i.e., whether *i*%2 is greater than zero. If *i* is odd at the time of recording then the following `JMP` bytecode will not be executed, and the remaining bytecodes to be recorded are `ADDVV` and the initial `FORL` that closes the loop, and causes the interpreter to stop recording with a successful trace (0006..0011).

This trace is then handed over to the JIT compiler, which translates the recorded bytecode instructions into a native program for the target instruction set, optimized using the information gathered during recording. [Gal et al. 2009] The interpreter then "patches" the `FORL` bytecode at 0007 by replacing it with a `JFORL` bytecode that causes the emitted code to be executed instead of the original bytecode.

During code generation the recorded bytecodes are translated into a SSA [Cytron et al. 1991] *immediate representation* (IR), and a number of optimizing transformations are performed:

- `FOLD`: A rule-based fold engine dispatches to later optimization stages, but also performs algebraic simplifications.
- `ABC`: Array Bounds Check Elimination.
- `CSE`: Common Sub-expression Elimination.
- `LOOP`: Loop invariant hoisting, and loop unrolling.
- `DCE`: Dead code elimination.
- `AA`: Alias Analysis.
- `FWD`: Load and store forwarding.

- DSE: Dead-Store Elimination.
- NARROW: Narrowing of numbers (doubles to 32-bit integers).
- STRIPOV: Stripping of overflow checks.
- SINK: Allocation Sinking and Store Sinking.

Eventually, execution of a compiled trace will exit and return to the interpreter. A compiled trace can exit for a number of reasons. One way to exit emitted code is by executing it successfuly to completion. Additionally, any deviation from the invariants encountered during trace recording will trigger an exit in the emitted code. Any branches in the recorded trace as well as any checks for invariants of the specialized code are converted to "guards" where each guard represents an invariant assertion and a dedicated exit point.

```
0002 >  tab SLOAD  #1    T
0003    p32 HREF   0002  "sum"
0004 >  p32 EQ     0003  [0x41526458]
```

In the SSA immediate representation of a trace above we can see two dependent guards, marked by > characters. The first guard at 0002 loads an object via SLOAD, and asserts that it is of type *table*. The second guard at 0004 ensures that the *sum* slot of the table contains the object at address 0x41526458[6]. If either of these invariants is violated the compiled trace will abort execution, and exit to the interpreter.

The same is true for branches converted to guards during trace recording. In the IR below we can see a guard that exits the loop body if the index is odd.

```
     ------ LOOP ------------
0010    int BAND   0007  +1
0011 >  int GT     0010  +0
```

Two things should be said about trace exits. First, each guarded trace exit is tracked with a dedicated hot count, and repeatedly taken exits will cause a new trace to be recorded starting from the respective branch. In LuaJIT, traces starting from a trace exit form a distinct class of traces called "side traces", and can not themselves record loops.

Second, exiting a compiled trace represents the end of a compilation unit, and requires consolidation between the interpreter state, and the state mutated by the emitted code. Mutations performed by emitted code are organized by LuaJIT in "snapshots", and these snapshots need to be restored to the interpreter state upon trace exit. Likewise for transitions between compiled traces, side traces cannot return directly into a loop body of their parent, and force repeated execution of invariant guards.

3.1 Mechanical Sympathy

Trace selection in LuaJIT works analogous to a CPU branch predictor. While a modern computer speculatively executes certain branches, a tracing JIT compiler might speculatively compile, and hence bias, certain branches. Pitfalls of speculative execution apply equally to CPU branch predictors, and JIT engines. In LuaJIT specifically, the speculative aspects of the compiler are less mature than their hardware counterparts, and some pitfalls are present in exacerbated variants.

[6]I.e., the object must be the *sum* method from section 2.1

It is easy, to construct a Lua program, even unknowingly, that executes an unbiased branch in a loop which cannot be hoisted before the loop body. In LuaJIT's current implementation, and specifically under the limitations of the interaction between the trace as a compilation unit, trace exits, and exit snapshots, the emitted code can be unfavorable compared to traditional AOT compilers.

Furthermore, adversarial inputs can manipulate the outcomes of speculative execution. [Kocher et al. 2019] This is an inherent aspect of speculative execution in general, and deserves particular attention when designing JIT compilers.

Listed below is the machine code emitted by LuaJIT for our branchy loop from section 3. The first trace recorded covers the loop. The second trace begins at the fifth exit (->5) of trace #1, and covers a single iteration of the loop.

```
---- TRACE 1 mcode 100
2a53ff90  mov dword [0x41991410], 0x1
2a53ff9b  cvttsd2si ebp, [rdx+0x8]
2a53ffa0  test ebp, 0x1
2a53ffa6  jle 0x2a530014 ->1
2a53ffac  cmp dword [rdx+0x4], 0xfffeffff
2a53ffb3  jnb 0x2a530018 ->2
2a53ffb9  xorps xmm7, xmm7
2a53ffbc  cvtsi2sd xmm7, ebp
2a53ffc0  addsd xmm7, [rdx]
2a53ffc4  add ebp, +0x01
2a53ffc7  cmp ebp, +0x64
2a53ffca  jg 0x2a53001c ->3
->LOOP:
2a53ffd0  test ebp, 0x1
2a53ffd6  jle 0x2a530024 ->5
2a53ffdc  xorps xmm6, xmm6
2a53ffdf  cvtsi2sd xmm6, ebp
2a53ffe3  addsd xmm7, xmm6
2a53ffe7  add ebp, +0x01
2a53ffea  cmp ebp, +0x64
2a53ffed  jle 0x2a53ffd0 ->LOOP
2a53ffef  jmp 0x2a530028 ->6
---- TRACE 1 stop -> loop
```

```
---- TRACE 2 mcode 49
2a53ff58  mov dword [0x41991410], 0x2
2a53ff63  add ebp, +0x01
2a53ff66  cmp ebp, +0x64
2a53ff69  jg 0x2a530014 ->1
2a53ff6f  xorps xmm6, xmm6
2a53ff72  cvtsi2sd xmm6, ebp
2a53ff76  movsd [rdx+0x20], xmm6
2a53ff7b  movsd [rdx+0x8], xmm6
2a53ff80  movsd [rdx], xmm7
2a53ff84  jmp 0x2a53ff90
---- TRACE 2 stop -> 1
```

Note how in trace #1 the first loop iteration is unrolled, and invariant checks performed in the first iteration are not repeated in subsequent interations. Trace #2 performs a single iteration of the loop where *ebp* is even, and returns to the beginning of trace #1. Given the unbiased branch, every other iteration of the loop will exit trace #1, and likely cause a rentry at its top re-executing any invariant guards. This compiler behavior effectively cancels out high-impact loop optimizations.

Another important observation is that the emitted code is dependent on which branch taken at the time of recording. Naturally, control flow is exercised by the input to the evaluation of the program. Situations arise in which for a heavily biased branch—more common in practice than unbiased branches—either favourable (as in trace #1) or unfavourable (as in trace #2) code is emitted depending on the input to the evaluation. The quality of generated code and, by extension, execution performance being affected by adversarial input is problematic.[7]

3.2 Virtual Machine Words

LuaJIT uses *NaN tagging* to represent doubles and other built-in types as single tagged 64-bit *virtual machine* (VM) words. [8] This representation allows the most common types of values to be stack-allocated without cooperation from the garbage collector (GC).

We have experience using LuaJIT for systems applications that handle many 64-bit values such as large integers and pointers that do not fit within a tagged VM word. This was made possible because, within emitted machine code, LuaJIT is able to *sink* allocations of objects, which otherwise must generally be heap-allocated, as long as they are held in registers.[9]

However, we found this optimization to be unreliable in situations where 64-bit values spill out of registers onto the stack, and subsequently cause GC pressure.

4 WHERE TO GO NEXT

LuaJIT is yet incomplete. Advancements in JIT compilation techniques, such as in better code generation for loops with unbiased branches could be incorporated in future implementations of JIT compilers. [Gal and Franz 2006]

Double-precision floating point numbers have become a popular base type for numbers in dynamic programming languages. Considering the advanced floating-point support of dominant ISAs, we would like to pose a question: rather than building a machine for their Lisp, should hackers build a Lisp for the best available machine?

If we look at the interpreter as a component of an optimizing compiler, rather than the primary execution engine itself, we might wish to choose nontraditional trade-offs. We might increase the VMs word size to fit the common 64-bit values we are having trouble with. [Soldatov and IPONWEB 2018] After all, the stack overhead of the interpreter is rendered mostly irrelevant in our emitted code.

A general design goal should be to find optimizing transformations with high-generality in order to provide reliable performance. Brittle performance is giving JITs a bad name as it is. To no lesser importance, future JIT compilers must ensure that adversarial program inputs can only control which code paths are to be compiled, but can never affect the quality of the emitted code.

With respect to Lisp, there are implementations such as *Armed Bear Common Lisp* and *Clojure* that have inherited big, mature JIT compilers. However, there is also *Guile* which recently added a new young JIT compiler. [Wingo 2020]

We hope to present JIT compilers as an exciting, young field. And in an ode to *Squeak*, we hope to garner interest in JIT compilation as a technique for iteratively writing small, beautiful, and fast dynamic language implementations. [Ingalls et al. 1997]

[7]Adversarial need not imply malevolent; undeterministic performance depending on the workload handled within the first milliseconds of your application's run-time is frustrating, to say the least.

[8]*NaN tagging* or *NaN boxing* [Wingo 2011]

[9]Sinking here refers to avoiding *boxing* and *unboxing* of the object.

REFERENCES

Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. 2000. Dynamo: A Transparent Dynamic Optimization System.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems* 13, 4 (Oct 1991), 451–490. http://doi.acm.org/10.1145/115372.115320

Andreas Gal, Brendan Eich, Mike Shaver, David Anderson, David M, Mohammad R. Haghighat, Blake Kaplan, Graydon Hoare, Boris Zbarsky, Jason Orendorff, and Mozilla Corporation. 2009. Trace-based Just-in-Time Type Specialization for Dynamic Languages.

Andreas Gal and Michael Franz. 2006. Incremental Dynamic Code Generation with Trace Trees.

Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, Alan Kay, and Walt Disney Imagineering. 1997. Back to the future: The story of Squeak, A practical Smalltalk written in itself. In *In Proceedings OOPSLA '97, ACM SIGPLAN Notices*. ACM Press, 318–326.

Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*.

John Mccarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I.

Mike Pall. 2017. The LuaJIT Project. https://luajit.org/

Anton Soldatov and IPONWEB. 2018. Rewriting LuaJIT: Why and How. https://www.lua.org/wshop18/Soldatov.pdf

Andy Wingo. 2011. value representation in javascript implementations. https://wingolog.org/archives/2011/05/18/value-representation-in-javascript-implementations

Andy Wingo. 2020. lessons learned from guile, the ancient & spry. http://wingolog.org/archives/2020/02/07/lessons-learned-from-guile-the-ancient-spry

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.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27–28 2020, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s).
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

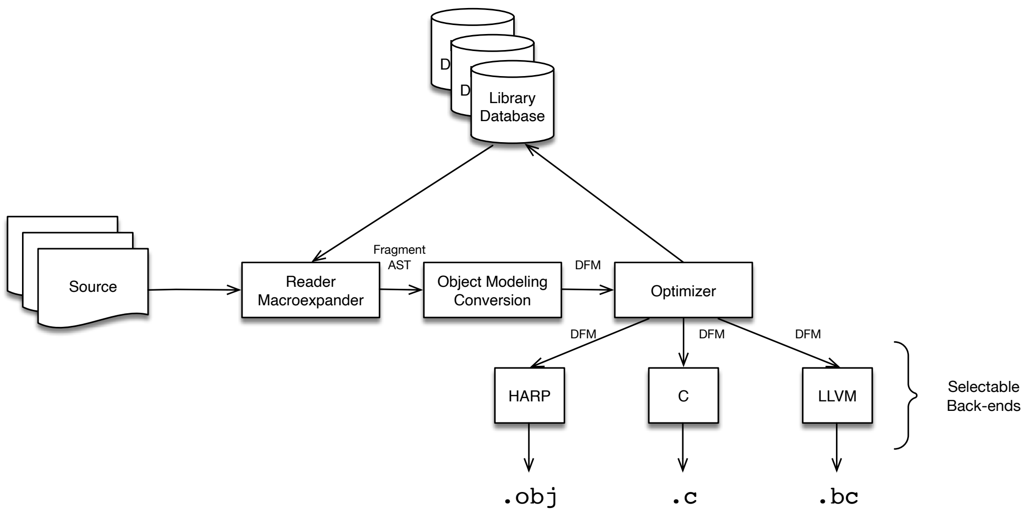


Figure 1: DFMC Compiler Structure

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.

Partial Evaluation Based CPS Transformation: An Implementation Case Study

Rajesh Jayaprakash
TCS Research, India
rajesh.jayaprakash@tcs.com

ABSTRACT

We demonstrate the implementation of a partial evaluation based CPS transformation in the context of pLisp, a Lisp dialect and IDE for beginners. The CPS transformation employs a modular technique that unifies the treatment of the language constructs; we illustrate the transformation by explicating the conversion process for a single construct (viz., `if`). To the best of our knowledge, this framework is also novel in that the partial evaluation and CPS transformation techniques are implemented in the implementation language of the system itself (i.e., C), as opposed to bootstrapping from an existing Lisp dialect.

CCS CONCEPTS

• **Software and its engineering → Compilers**;

KEYWORDS

Lisp, partial evaluation, CPS transformation

ACM Reference Format:
Rajesh Jayaprakash. 2020. Partial Evaluation Based CPS Transformation: An Implementation Case Study. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3740941

1 INTRODUCTION

Partial evaluation [12] is a well-established technique for optimizing programs. A program is partitioned into a static and a dynamic part, with the static part comprising data and values known beforehand, and the dynamic part comprising data and values not known at compile-time. The compilation or translation process 'executes' the static part so that only a residual program is left over for execution later, thereby resulting in a smaller/faster program.

Continuation Passing Style (CPS) [14] is a style of programming in which every function call is augmented with an additional argument known as a continuation; this continuation embodies the rest of the computation, and the function is expected to perform its computation and then invoke the continuation with the result of the computation. There are a number of advantages to programming in CPS, a few of them being a) simplifying the effort needed at the compiler back-end b) making explicit the semantics of the computation (e.g., order/sequencing of execution primitives) and c) enabling the easier implementation of advanced control structures like non-local control transfers. Transformation of a program to CPS is a step in a typical compiler pipeline that includes steps like assignment conversion, renaming, closure conversion, and lift transformation.

A naive CPS transformation [8] results in quite inefficient code, and these inefficiencies are removed using techniques like inlining. Another option for removing these inefficiencies is to leverage partial evaluation based techniques [6]. The 'static' program fragments that would have been generated by the naive CPS transformation are recognized as such and are executed by the so-called metalanguage interpreter [15] during the transformation process itself, thereby preventing the inefficient code from being generated in the first place.

Figure 1 illustrates the CPS transformation of the expression `(+ x 1)` using both a naive approach (Figure 1a) and a partial evaluation based approach (Figure 1d) [1]. The output of the naive transformation is typically optimized by repeated β-reductions (Figure 1b) and *inlining* (constant propagation of a lambda form followed by a β-reduction; Figure 1c). The quasi-syntactic (and semantic) equivalence of the optimized naive CPS transformation and the partial evaluation based CPS transformation is to be noted, although this equivalence is achieved by different routes; in the case of the partial evaluation approach, the inlining optimization is effected by execution of the relevant code fragment (the let forms in Figure 1b) by the metalanguage interpreter (explained in section 3).

pLisp [11] is a Lisp-1 dialect and an integrated development environment modelled on Smalltalk that targets beginners, with the following features:

- Graphical IDE with context-sensitive help, syntax colouring, autocomplete, and auto-indentation
- Native compiler
- User-friendly debugging/tracing
- Image-based development
- Continuations
- Exception handling
- Foreign function interface
- Package/Namespace system

The native compiler in pLisp implements a partial evaluation based CPS transformation step through a modular and flexible framework, embodying an elegant construct-dependent technique for the creation of the metalanguage closures, which is detailed in this paper.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27–28 2020, Zurich, Switzerland
© 2020 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.3740941

[1]This code was generated by the pLisp compiler; the names of the binding variables in the abstractions have been shortened to improve readability

```
(lambda (g8216)
  ((lambda (g8218)
     (g8218 +))
   (lambda (g8217)
     ((lambda (g8220)
        (g8220 x))
      (lambda (g8219)
        ((lambda (g8222)
           (g8222 1))
         (lambda (g8221)
           (g8217 g8219 g8221 g8216)))))))
```

(a) Naive CPS conversion

```
(lambda (g8216)
  (let ((g2818 (lambda (g8217)
                 (let ((g8220 (lambda (g8219)
                                (let ((g8222 (lambda (g8221)
                                               (g8217 g8219 g8221 g8216))))
                                  (g8222 1)))))
                   (g8220 x)))))
    (g8218 +)))
```

(b) After β-reduction

```
(lambda (g8216)
  (+ x 1 g8216))
```

(c) After inlining

```
(lambda (g8223)
  (let ((g8224 (lambda (g8225)
                 (g8223 g8225))))
    (+ x 1 g8224)))
```

(d) Partial evaluation based CPS conversion

Figure 1: Naive and partial evaluation based CPS conversion of '(+ x 1)'

2 COMPILER PIPELINE

The pLisp compiler transforms the Lisp source code to CPS and emits C code, which is then passed to LLVM to produce native code. The compiler does the transformation in the following passes [15]:

- Desugaring/Macro expansion
- Assignment conversion
- Translation
- Renaming
- CPS conversion
- Closure conversion
- Lift transformation
- Conversion to C

These compiler passes produce progressively simpler Lisp dialects, culminating in a version with semantics close enough to C. Figure 2 illustrates this transformation. Since LLVM is used to convert the C code to native code, the pipeline does not include passes like register allocation/spilling.

Desugaring/Macro expansion: This pass performs macro expansion, resulting in a dialect called `plisp_k` ('k' stands for 'kernel') that is shorn of all macro invocations. The backquote construct used in macro-expansion is itself implemented as a macro, so the order of definitions of backquote and its supporting functions is critical. Accordingly, the source file listing the definitions of the core library objects contains the macro-expansion-related infrastructure before the other definitions.

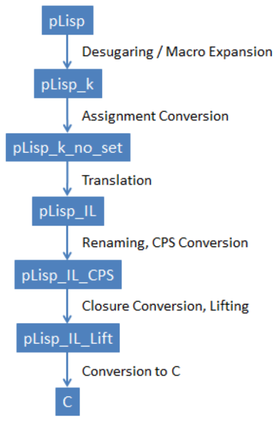


Figure 2: pLisp compiler transformations

Assignment conversion: Assignment conversion replaces all assigned variables with mutable cells, thereby making variable bindings immutable (i.e., the contents of the cell may change over time, but the binding of a cell to a variable will not). In the interests of simplicity and to avoid the introduction of one more object type, mutable cells in pLisp are simulated by CONS objects whose CDR is NIL. Assignment conversion produces code in a dialect called `plisp_k_no_set`, which differs from plisp_k only with respect to the absence of the `set` construct (replaced by the `setcar` primitive).

Translation: This pass produces an intermediate language dialect called `pLisp_IL` (Figure 3) and differs from the previous dialects in that a) it does not have a recursion construct (`letrec`) and b) a new multibinding construct called `let*` is introduced (note: this `let*` is distinct from the similarly-named core library form). This pass also performs syntactic simplifications like removal of empty `let`s, conversion of applications of `lambda` expressions to `let`s, η-reductions and copy propagation.

```
exp  ::=  literal | var
          | (if exp exp exp?)
          | (primop exp+)
          | (lambda (var*) exp)
          | (error exp)
          | (call/cc exp)
          | (exp exp*)
          | (let ((var exp)*) exp)
          | (let* ((var exp)*) exp)
```

Figure 3: pLisp_IL grammar

Renaming: The renaming pass ensures that no two logically distinct variables in the code have the same name, so that variable capture is avoided. Introduction of fresh variable names utilizes the same infrastructure that implements the `gensym` feature in the pLisp core.

CPS Conversion: The CPS conversion pass converts the code to CPS style, and produces the dialect `plisp_IL_CPS` (Figure 4) characterized by restrictions on `let` forms: they can now only have one binding, and the expressions that can be bound to the `let` identifiers (denoted by `le` in the figure) are restricted to literals, lambda expressions, and primitive operations. The CPS conversion pass also injects code to save the generated continuation object: this is useful for implementing the break/resume functionality for the debugger.

```
exp  ::=  (var val*)
          | (if val exp exp)
          | (let ((var le)) exp)
          | (error exp)
          | (call/cc exp)
val  ::=  literal | var
le   ::=  literal
          | (lambda (var*) exp)
          | (primop val+)
```

Figure 4: pLisp_IL _CPS grammar

Closure conversion: In this pass, functions are converted into closures, so that the free variables referred in the lambda expression body are fetched from the environment that is stored along with the code. The lambda expressions are thus implicitly passed the code/environment pair as their first parameter.

Lift transformation: The lift transformation pass converts all procedures to the top level and thereby linearizes the code. Please note that lift transformation is predicated on the procedures having undergone a closure conversion. This pass results in the `plisp_IL_Lift` dialect, where the code linearization is manifested as further restrictions on `let` bindings (only literals and primitive operations are permitted as `let`able expressions).

Conversion to C: The C conversion pass, in addition to handling the Lisp-to-C syntax translation, also performs additional tasks like decorating the variable names so that they do not violate the C syntax rules for variable names and keywords.

Expressions entered at the top level (i.e., the Workspace in pLisp) are compiled into anonymous closures, and are supplied the identity function as their continuation.

3 PARTIAL EVALUATION BASED CPS TRANSFORMATION

3.1 pLisp Object Representation

In this subsection we briefly describe the pLisp object model, inasmuch as is required to set the context for this work. The following object types are supported by pLisp [11]:

- Integers
- Floating point numbers
- Characters
- Strings
- Symbols
- Arrays
- CONS cells
- Closures
- Macros

Objects are internally represented by `OBJECT_PTR`, a `typedef` for `uintptr_t`, the C language data type used for storing pointer values. The four least significant bits of the value are used to tag the object type (e.g., `0011` for character objects, `0110` for CONS cells, and so on), while the remaining *(n-4)* bits (where *n* is the total number of bits) of the value take on different meanings depending on the object type, i.e., whether the object is a boxed object or an immediate object. If the object is a boxed object, the remaining bits store the referenced memory location. The use of the `GC_posix_memalign()` call (from the Boehm Garbage Collector library) for the memory allocation obviates the loss of the four least significant bits and ensures that the four least significant bits of the returned address are zeros.

3.2 Metalanguage Interpreter

The metalanguage interpreter in pLisp is written in C. While the semantics of the metalanguage interpreter are easier and more natural to represent and implement in Lisp, this option is not available in the present situation: we are implementing pLisp from scratch and therefore do not have a core/kernel Lisp implementation from which to bootstrap. S-expressions which are input to the interpreter are pLisp objects, more specifically linked CONS cells, internally represented as `OBJECT_PTR` values. A subset of the pLisp object types, viz., atoms (excluding types like closures, of course) and CONS cells comprising atoms or other CONS cells, is thus accepted by the interpreter.

The workhorse of the CPS transformation process is the function `mcps()` [2]. This function accepts the source language expression, and depending on its type (i.e., abstraction, application, and so on), creates the corresponding closure *M* which, when invoked, would perform the code transformation for that type.

The CPS transformation process is also predicated on a class of secondary closures *m*, the purpose of which is to transform value expressions (identifiers and literals) in the source dialect (`plisp_IL`) into general expressions in the target dialect (`plisp_IL_CPS`). The closure *M* takes an argument of type *m*.

The partial evaluation semantics are captured in these two closures: the code executed by these closures would have, in a naive CPS transformation, formed a part of the generated code, thereby leading to code bloat and the attendant performance hit.

Listing 1 presents the data structures used by the interpreter. The structures `reg_closure_t` and `metacont_closure_t` are the realizations of the closures *m* and *M* respectively. The first three fields of each structure correspond to the implementation of a closure in a language like C, i.e., the function pointer representing the function and the machinery required to store the closed-over values. The function pointer field in effect specializes the closures: by assigning different functions to this field, we are able to handle the various source language constructs (`if`, `let`, and so on) in a modular way. In addition to these three fields, the structure for *m* has a field called `data`; the need for and usage of this field is explained in the next section.

Figure 5 depicts the object model underpinning the interpreter. In the interests of space, not all the elements corresponding to the language constructs are shown. It is to be noted that there isn't a one-to-one mapping between the language construct entities of *M* and *m* (e.g., there is an entity called `lambda_metacont_fn`,

[2] https://github.com/shikantaza/pLisp/blob/master/src/metacont.c

but no entity called lambda_cont_fn). The absence of such an entity is because it is not always the case that the body of M is of the form $(\mathcal{MCPS}[\![E]\!]\ \lambda V.\ \ldots)$, which necessitates the presence of the symmetric entity. Also, the dependency or linkage between the transforming entities and the transformed entities is cleanly captured by the OBJECT_PTR reference; the use of this reference enables flexibility and allows us to switch the object representation easily if desired. Finally, please also note the self-referential loop labelled 'data' for reg_closure_t: the self-reference refers to the data field, which, while generic in intent, is used in practice to store only entities of class m (we have explained this further in subsection 4.2).

4 A DETAILED WALKTHROUGH

In order to explain the internals of the translation process and to bring out the mechanics of the interpreter better, we walk through the translation process for the pLisp if construct in this section.

4.1 An Abstract View

At an abstract level, the part of the implementation function mcps() that translates if constructs (more precisely, the function stored in the field mfn of the structure metacont_closure_t, corresponding to M) can be represented [15] by the function shown in Figure 6.

```
$\mathcal{MCPS}[\![$(if $E_{test}$ $E_{then}$ $E_{else}$)$]\!]$
  = ($\lambda m$ . ($\mathcal{MCPS}[\![E_{test}]\!]$
      ($\lambda V_{test}$ . (let (($I_{kif}$ ($mc \rightarrow exp\ \ m$)) ; $I_{kif}$ fresh
                  (if $V_{test}$
                      ($\mathcal{MCPS}[\![E_{then}]\!]$ ($id \rightarrow mc\ \ I_{kif}$))
                      ($\mathcal{MCPS}[\![E_{else}]\!]$ ($id \rightarrow mc\ \ I_{kif}$)))))))
```

Figure 6: Transformation function for *if* construct

We deconstruct the function as follows.

(1) The function $\mathcal{MCPS}$ returns a closure (tagged as M in the previous section). The closed-over values for the returned closure depend on the source language construct in question; in this case, they are E_{test}, E_{then}, and E_{else}.
(2) The argument to this closure is another closure (tagged as m earlier), which handles the conversion of value expressions (literals and identifiers) in the source dialect.
(3) When the closure M is invoked with the argument, it evaluates $\mathcal{MCPS}$ on E_{test}, creates another closure of class m (this is explained below), and applies the former to the latter.
(4) The closure of class m mentioned above builds the target expression in the plisp_IL_CPS dialect. The closed-over values for this closure are E_{then}, E_{else}, and m (the argument to M itself). This is the canonical CPS transformation of if: evaluate the test expression, branch on to the CPS transformation of the consequent or the alternative expression based the truth-value of the test expression.
(5) The body of the closure defined by (λV_{test}.(let.. is an S-expression built up partly with literals like let and fresh symbols, and partly with return values of function applications in the metalanguage.
(6) $id \rightarrow mc$ and $mc \rightarrow exp$ are helper functions; the first converts an identifier to a closure of class m, while the second converts the metalanguage closure m into an abstraction in the target dialect (i.e., the CPS-transformed equivalent expression).

In summary, elements of the source expression are partitioned among multiple closures, and these closures (operating at different stages of the transformation) utilize these elements to recursively build the target expression.

4.2 Implementation

The definitions of the closure functions for transforming if constructs are provided in Listing 2.

The functions if_metacont_fn and if_reg_cont_fn slot into mfn and fn in the respective closure data structures presented in Listing 1. As mentioned earlier, the same data structures are repurposed to handle different constructs by suitably populating these slots as required. Adhering to the convention for closure implementation, the first argument to both these functions are their parent closure data structures themselves.

The computational steps of the abstract function $\mathcal{MCPS}$ are jointly realized in an imperative fashion in these two functions:

if_metacont_fn:

(1) Retrieve the closed-over values from the closure data structure.
(2) Convert the if test expression to a closure M by calling mcps().
(3) Create the regular closure object that would perform the actual transformation.
(4) Invoke the closure created in step (2) on the regular closure object and return the result.

if_reg_cont_fn:

(1) Retrieve the closed-over values from the closure data structure.
(2) Convert the consequent and alternative parts of the if expression to closures by calling mcps().
(3) Build the target expression by splicing together the S-expression from literals like let and fresh symbols, and from results of function calls of the above closures.

A couple of things are to be noted:

(1) The implementation contains calls like gensym() and list(), which are C equivalents of the standard lisp operators.
(2) The slot data in reg_closure_t is also used to store closed-over values (albeit not OBJECT_PTR values). However, such a slot is useful to logically separate such values as originate from the source language expressions (e.g., E_{then}) from values like m for purposes of clarity. There is also an element of type-safety: closed_vals, being of type OBJECT_PTR *, is safer from, e.g., an assignment perspective when compared to data (which is of type void *).

4.3 Another Example

We illustrate the flexibility of the translation framework with a brief look at the machinery for the translation of the let form (Figure 7).

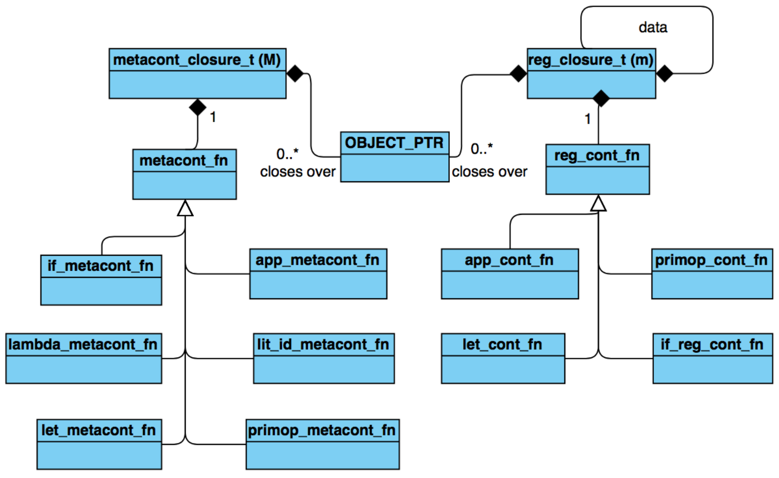


Figure 5: Transformation object model

Listing 1: Data structures used in metalanguage interpreter

```
//forward declarations
struct reg_closure;
struct metacont_closure;

typedef OBJECT_PTR (*reg_cont_fn)(struct reg_closure *, OBJECT_PTR);

typedef struct reg_closure
{
  reg_cont_fn fn;
  unsigned int nof_closed_vals;
  OBJECT_PTR *closed_vals;
  void *data;
} reg_closure_t;

typedef OBJECT_PTR (*metacont_fn)(struct metacont_closure *, struct reg_closure *);

typedef struct metacont_closure
{
  metacont_fn mfn;
  unsigned int nof_closed_vals;
  OBJECT_PTR *closed_vals;
} metacont_closure_t;
```

$$\begin{aligned}
&\mathcal{MCPS}[\![(\texttt{let}\ ((I_i\ E_i)_{i=1}^{n})\ E_{body})]\!]\\
&\quad = (\lambda m\,.\ (\mathcal{MCPS}[\![E_1]\!]\\
&\qquad\quad (\lambda V_1\,.\\
&\qquad\qquad \ldots\\
&\qquad\qquad\quad (\mathcal{MCPS}[\![E_n]\!]\\
&\qquad\qquad\qquad (\lambda V_n\,.\ (\texttt{let*}\ ((I_i\ V_i)_{i=1}^{n})\\
&\qquad\qquad\qquad\qquad\qquad (\mathcal{MCPS}[\![E_{body}]\!]\ m))))\ \ldots\ )))
\end{aligned}$$

Figure 7: Transformation function for *let* construct

This is the canonical transformation of `let`: each of the binding expressions E_i undergoes transformation (sequentially), and invokes its respective continuation. Finally the transformation of the `let` body E_{body}, which would have the references to the bindings I_i populated by the `let*` form, is invoked on m, which is the continuation that would have been provided by the whole `let` expression transformation context. Each of the nested closures (also of type m) closes over a part of the `let` binding components. The

Listing 2: Closure function definitions for *if* construct

```
OBJECT_PTR if_metacont_fn(metacont_closure_t *mcls, reg_closure_t *cls1)
{
  OBJECT_PTR test_exp = mcls->closed_vals[0];
  OBJECT_PTR then_exp = mcls->closed_vals[1];
  OBJECT_PTR else_exp = mcls->closed_vals[2];

  metacont_closure_t *test_mcls = mcps(test_exp);

  reg_closure_t *cls = (reg_closure_t *)GC_MALLOC(sizeof(reg_closure_t));

  cls->fn               = if_reg_cont_fn;
  cls->nof_closed_vals  = 2;
  cls->closed_vals      = (OBJECT_PTR *)GC_MALLOC(cls->nof_closed_vals * sizeof(OBJECT_PTR));

  cls->closed_vals[0]   = then_exp;
  cls->closed_vals[1]   = else_exp;

  cls->data = cls1;

  return test_mcls->mfn(test_mcls, cls);
}

OBJECT_PTR if_reg_cont_fn(reg_closure_t *cls, OBJECT_PTR test_val)
{
  OBJECT_PTR i_kif = gensym();

  reg_closure_t *cls1 = (reg_closure_t *)cls->data;

  OBJECT_PTR then_exp = cls->closed_vals[0];
  OBJECT_PTR else_exp = cls->closed_vals[1];

  metacont_closure_t *then_mcls = mcps(then_exp);
  metacont_closure_t *else_mcls = mcps(else_exp);

  reg_closure_t *kif_cls = id_to_mc(i_kif);

  return list(3,
              LET,
              list(1, list(2, i_kif, mc_to_exp(cls1))),
              list(4,
                   IF,
                   test_val,
                   then_mcls->mfn(then_mcls, kif_cls),
                   else_mcls->mfn(else_mcls, kif_cls)));
}
```

conversion process lends itself naturally to a recursive implementation, with the final closure (which acts on E_{body}) being the only non-recursive component.

These desired behaviours are captured elegantly in our implementation (Listing 3): the recursive behaviour is realized (at closure creation time) by setting the `fn` slot in `reg_closure_t` to a recursive function (`let_cont_fn_recur`), while the 'tail call' behaviour is realized by setting the same slot to a non-recursive function (`let_cont_fn_non_recur`). The bindings and the body of the `let` are closed over both these categories of closure.

The same technique—specializing a closure into recursive and non-recursive variants as needed—is used for applications and primitive operations, which, similar to `let`, involve a variable number of sub-expressions that need to be CPS-transformed.

5 RELATED WORK

The number of Lisp/Scheme compilers is quite large; therefore in the interests of space we cover representative ones, highlighting the implementation techniques utilized by them that are of relevance.

The CHICKEN Scheme-to-C compiler [2, 5] employs a CPS-based compilation strategy, and CPS conversion is one of the steps in its compiler pipeline. However, the CPS conversion is written in Scheme itself, as opposed to the implementation language of the compiler (e.g., C). The CPS conversion algorithm is stated to be based on the relatively naive algorithm outlined in [1], and the optimizations induced by partial evaluation are realized at the later (explicit) stages in the compiler pipeline.

Not all Scheme compilers use CPS transforms to provide support for continuations. For example, Bigloo [3] implements `call/cc` by copying the execution context to the heap, while Guile [10] implements continuations by copying the C stack to the heap. A similar mechanism is employed by Gambit [7]. The Stalin Scheme compiler [13] utilizes a lightweight CPS conversion technique that relies on whole-program analysis, whereas normative CPS transformation is syntax-directed and is concerned with local (expression-level) code units.

Blocks [4] are an extension to the C, C++, and Objective-C implementations of Clang that provides a mechanism to create closures in these languages. In contrast, the current work implements closures in ANSI C using the standard mechanisms available in the language. Nested functions [9] are another available mechanism for realizing closures in C. However, the extent of these nested functions is limited to the containing scope, which falls short with respect to the needs imposed by the CPS transformation.

6 CONCLUSION

We presented an implementation of a partial evaluation based CPS transformation in the context of pLisp, a Lisp dialect and integrated development environment for beginners. The object model underpinning the metalanguage interpreter was presented, and the implementation was illustrated with a detailed walkthrough of the transformation for a single source language construct from both an abstract and an implementation perspective. The framework has been implemented in a modular fashion so that it is easy to swap in and swap out implementations of the transformation functions of the individual constructs, and to add support for new constructs. A further improvement to flexibility would be to set up the structure of target expressions in a declarative manner, and to code the transformation functions in such a way that the functions simply fill in the computed values into a pre-built S-expression template (somewhat along the lines of a context object with holes). This is planned to be taken up as future work.

REFERENCES

[1] Andrew W Appel. *Compiling with continuations.* Cambridge University Press, 2007.
[2] Henry G Baker. Cons should not cons its arguments, part ii: Cheney on the mta. *ACM Sigplan Notices*, 30(9):17–20, 1995.
[3] Bigloo Scheme. URL https://www-sop.inria.fr/indes/fp/Bigloo/.
[4] Blocks. URL https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html.
[5] CHICKEN Scheme. URL https://call-cc.org.
[6] Oliver Danvy and Andrzex Filinski. Representing control: A study of the cps transformation. *Mathematical structures in computer science*, 2(4):361–391, 1992.
[7] M Feeley. Gambit scheme. URL http://gambitscheme.org/wiki/index.php/Main_Page.
[8] Cormac Flanagan, Amr Sabry, Bruce F Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 237–247, 1993.
[9] GCC Nested Functions. URL https://gcc.gnu.org/onlinedocs/gcc-4.0.4/gcc/Nested-Functions.html#Nested-Functions.
[10] Guile. URL https://www.gnu.org/software/guile/.
[11] Rajesh Jayapraksh. plisp: A friendly lisp ide for beginners. In *Proceedings of the 11th European Lisp Symposium*, 2018.
[12] Neil D Jones, Carsten K Gomard, and Peter Sestoft. *Partial evaluation and automatic program generation.* Peter Sestoft, 1993.
[13] Jeffrey Mark Siskind. Flow-directed lightweight closure conversion. Technical report, Technical Report 99-190R, NEC Research Institute, Inc, 1999.
[14] Guy L Steele Jr. Rabbit: A compiler for scheme. Technical report, Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1978.
[15] Franklyn Turbak, David Gifford, and Mark A Sheldon. *Design concepts in programming languages.* MIT press, 2008.

Listing 3: Closure creation for *let*

```
reg_closure_t *create_reg_let_closure(OBJECT_PTR     bindings,
                                      OBJECT_PTR     full_bindings,
                                      OBJECT_PTR     body,
                                      unsigned int   nof_vals,
                                      OBJECT_PTR     *vals,
                                      reg_closure_t  *cls)
{
  reg_closure_t *let_closure = (reg_closure_t *)GC_MALLOC(sizeof(reg_closure_t));

  if(cons_length(bindings) == 0) //last binding
    let_closure->fn = let_cont_fn_non_recur;
  else
    let_closure->fn = let_cont_fn_recur;

  let_closure->nof_closed_vals = nof_vals + 3;
  let_closure->closed_vals     = (OBJECT_PTR *)GC_MALLOC(let_closure->nof_closed_vals
                                                          * sizeof(OBJECT_PTR));

  let_closure->closed_vals[0]  = bindings;
  let_closure->closed_vals[1]  = full_bindings;
  let_closure->closed_vals[2]  = body;

  int i;
  for(i=3; i<let_closure->nof_closed_vals; i++)
    let_closure->closed_vals[i] = vals[i-3];

  let_closure->data = cls;

  return let_closure;
}
```

Representing method combinations

Robert Strandh
robert.strandh@u-bordeaux.fr
LaBRI, University of Bordeaux
Talence, France

ABSTRACT

The Common Lisp standard has few requirements on method combinations, and so does the semi-standard metaobject protocol for Common Lisp. For that reason, there is great variety among different Common Lisp implementations regarding how method combinations are represented and handled. Some implementations allocate a new method-combination instance for each generic function, whereas others attempt to reuse existing instances as much as possible. Most implementations are able to verify the validity of method-combination options for the built-in method-combination types, but no free Common Lisp implementation can verify custom method-combination types using the long form of the macro `define-methodcombination` immediately when a generic function is created, nor when a method-combination type is redefined. Instead, incompatibilities between supplied options and the method-combination type are then verified only when an attempt is made to execute the resulting method-combination procedure in order to create an effective method.

We propose a technique that makes early detection of incompatible method-combination options possible even for custom long-form method-combination types. We augment the lambda list of the method-combination definition with `&aux` entries that verify restrictions, and we construct a function with the augmented lambda list that will fail whenever there is such an incompatibility. With this technique, when an incompatibility is detected, we are also able to signal more relevant errors than most existing free implementations are able to do.

CCS CONCEPTS

• **Software and its engineering** → **Incremental compilers**; **Runtime environments**;

KEYWORDS

Common Lisp, CLOS, Meta-Object Protocol, Method combinations

ACM Reference Format:
Robert Strandh. 2020. Representing method combinations. In *Proceedings of the 13th European Lisp Symposium (ELS'20)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3747553

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27–28 2020, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.3747553

1 INTRODUCTION

The Common Lisp standard [1] contains very little information about method combinations. The dictionary entry in the standard for the system class `method-combination` requires a *method combination object* to be an *indirect instance* of the system class named `method-combination`. The standard further requires such an object to contain information both about the *type* of method combination and the *arguments* used with that type.

The term *indirect instance*, as explained in the glossary, excludes the possibility of such a method combination object to be an immediate instance of the class `method-combination`. We can interpret this requirement as the need to create a subclass, say, `standard-method-combination` to parallel the situation for `method` vs `standard-method` and `generic-function` vs `standard-generic-function`, i.e., so as to allow the programmer to create very different objects from those that the `standard-` version can provide.

Clearly, the text of the dictionary entry means that when the macro `defgeneric` is used with the `:method-combination` option given, such a method combination object is what the generic function will contain. We can confirm this view by examining the description of the MOP generic function `generic-function-method-combination` (as described in [2]) which states that the return value is "a method combination metaobject".

However, the macro `define-method-combination` does *not* define a method combination object. The reason is of course that no method-combination options are supplied to this macro. The dictionary entry for this macro also clearly says that the macro is used to define new *types* of method combinations.

The main issue for the person implementing a Common Lisp system, then, is how to interpret the relation between a *method combination type* and a *method combination object.*

It is easy to draw the conclusion that a call to the macro `define-method-combination` creates a new *class*, as suggested by the use of the word *type* in the standard, and that method combination objects of that type are instances of the new class. However, this view creates several problems. In particular, one must then determine whether each use of the same combination of the type and the arguments in the `:method-combination` option to `defgeneric` creates a new instance of the class, or whether existing instances are somehow kept track of and reused. The first possibility would have the unfortunate consequence that two

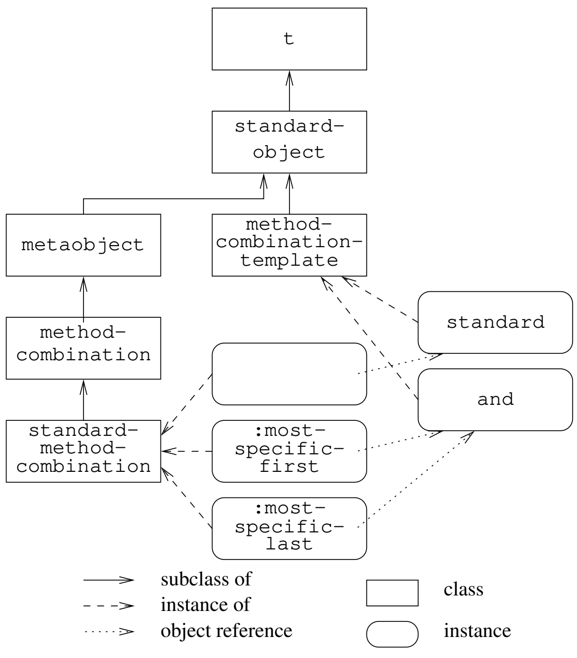


Figure 1: Representation of method combinations.

calls to `generic-function-method-combination` with different generic-function metaobjects would return two method combination objects that are not identical.

In this paper, we argue that a *method combination type* is itself an instance of a completely different class that we shall call `method-combination-template`, and that a *method combination object* is a *variant* of the template in that it contains a reference to the template as well as the values of the *options* that this particular method combination type allows. To conform to the standard, we obviously maintain that method combination objects are instances of `standard-method-combination`. This idea is illustrated in Figure 1, which shows two method combination templates (`standard` and `and`) and two variants of the `and` method-combination template; one variant with the option `:most-specific-first` and another variant with the option `:most-specific-last`.

A call to the macro `define-method-combination` results in a function that can be applied to a list of arguments which include at least a generic function and a list of applicable methods. This function becomes associated with the *name* of the method-combination *type* thus defined. The standard briefly uses the term *procedure* to refer to this resulting function. We adopt that convention in this paper, and refer to the resulting function as the *method-combination procedure*.

When a generic function is defined or redefined, it would be desirable to have the options of the `:method-combination` `defgeneric` option checked for validity immediately when the definition or redefinition occurs. For the built-in method combination types, most implementations also handle this check as a special case. However, all implementations we have investigated fail to check the options to a user-defined method-combination type defined by the long form of the macro `define-method-combination`. Instead, if the options are incompatible with the defined method-combination type, in the best case, an error is signaled when the method-combination procedure is applied to a list of applicable methods by the generic function `compute-effective-method`. Furthermore, the error being signaled can be hard to decipher, as it typically results from invalid arguments to a function with a particular lambda list.

In this paper, we propose a general mechanism for early detection of incompatible options to a particular method-combination type. This mechanism is available to the creator of custom method-combination types, and also used to verify the options to the built-in method-combination types.

The macro `define-method-combination` comes in two versions called the *long form* and the *short form* in the Common Lisp standard. The short form of the macro can be expressed in terms of the long form, but it may not be obvious how the options to the short form should be propagated to the long form.

Furthermore, in the description of the short form of the macro, the standard states that the method-combination procedure resulting from such a definition accepts an optional argument (named `order`) that can have two values, `:most-specific-first` and `:most-specific-last`, with the value `:most-specific-first` being the default. It is not obvious how this restriction can be expressed as a long-form definition of a similar procedure. A common solution to this problem is to define a subclass `short-method-combination` of the class `method-combination`, and to introduce special-purpose code for checking this restriction. The technique presented in this paper does not require such a subclass, as the long-form version of the short-form definition is able to check the restriction.

Throughout this paper, we assume that it is an error to attempt to create a generic function using a method-combination type that is not already created. Recall that the standard states that when a `define-method-combination` form appears at the top level, the compiler must recognize the name of that type as valid in subsequent `defgeneric` forms, but that the resulting method-combination procedure is not executed until the `define-method-combination` form itself is executed. In other words, since the method-combination type is not created at compile time, it may not exist when a `defgeneric` form using the name is encountered by the compiler. However, our assumption is still valid, since the compiler also does not create the generic function when a `defgeneric` form appears at the top level.

For example, assume that some source file contains a `define-method-combination` form, defining a method combination type with a new name, followed by a `defgeneric` form that refers to that method combination type in the `:method-combination` option of the `:defgeneric` form. When the compiler encounters the `define-method-combination` form, it registers its name as being valid for use in subsequent `defgeneric` forms, but the compiler does not *create* the method-combination type. Subsequently, when the compiler encounters the `defgeneric` form, it recognizes a valid name of a method-combination type, but since the compiler also does not create the generic function when the `defgeneric` form is encountered, there is no need for the method-combination type to have been created. When the compiled file is later loaded, the new method-combination type is first created. Subsequently, the generic function is created, referring to an existing method-combination type. In this paper, we do not address the mechanism by which the compile-time behavior required by the standard is implemented.

There are several scenarios that are discussed in this paper:

(1) The user correctly defines a custom method-combination type using `define-method-combination`. Subsequently, the user defines a generic function with that method-combination type, but makes a mistake in the list of options.
(2) The user defines a custom method-combination type using the long form of `define-method-combination`, but makes a mistake in the lambda list supplied to the macro, so that the options of the resulting method-combination procedure are not the ones that were intended. Subsequently, the user defines a generic function with a list of options that were intended to be acceptable.
(3) The user initially correctly defines a custom method combination type using `define-method-combination`, and then also correctly defines one or more generic functions with that method combination type. Then the user decides to make a change to the code of the method-combination type, so the `define-method-combination` form is re-executed, but the new lambda list is incompatible with the options given when the generic functions were created, either as a result of a mistake or of a deliberate decision.

To illustrate these scenarios, we can imagine a restricted form of the `and` method combination that does not admit any `:around` methods. This restriction means that the short form of `define-method-combination` can not be used.

An example of the first scenario would be the following code:

```
(define-method-combination simple-and
    (&optional (order :most-specific-first))
    (primary (and) :order order :required t)
 ...)

(defgeneric simple-and (...)
  (:method-combination simple-and :msot-specific-first))
```

Here, the user has a typo in the second form. An example of the second scenario would be the following code:

```
(define-method-combination simple-and
    (&optional (order :msot-specific-first))
    (primary (and) :order order :required t)
 ...)

(defgeneric simple-and (...)
  (:method-combination simple-and :most-specific-first))
```

Here, the user has a typo in the first form. Finally, an example of the third form would be the following code:

```
(define-method-combination simple-and
    (&optional (order :most-specific-first))
    (primary (and) :order order :required t)
 ...)

(defgeneric simple-and (...)
  (:method-combination simple-and :most-specific-first))
```

Here, there are no mistakes, but the user later decides to disallow the option , so the first form is altered to become:

```
(define-method-combination simple-and ()
    (primary (and) :order :most-specific-first
                   :required t)
 ...)
```

In the first two scenarios, the ideal consequence would be that a warning is initially signaled, stating that the options supplied to the creation of the generic function are incompatible with the type of the desired method combination. Any subsequent attempt to execute the generic function would result in an appropriate error being signaled. Once the incorrect definition has been corrected and the corresponding form has been re-executed, the generic function should be operational.

In the third scenario, the ideal consequence would be that a warning is signaled, giving a list of generic functions with a list of options that are now incompatible with the redefined method-combination type. Any subsequent attempt to execute one of these generic functions would result in an appropriate error being signaled. If a mistake was made, the re-execution of a corrected `define-method-combination` form should render the existing generic functions operational again. If the change was deliberate, the list of generic functions in the message can be used to determine which definitions to correct and re-execute.

The technique described in this paper handles all these scenarios, but it has been implemented only partially. We are currently working on incorporating the remaining elements of our technique into the SICL[1] code base.

2 PREVIOUS WORK

In this section, we give an overview of how different free Common Lisp implementations represent and handle method combinations. In particular, we compare the technique that each implementation uses with the three scenarios specified in Section 1. We do not include commercial Common Lisp implementations, simply because we can not know in detail how the code is written. Extensive experimentation might have given sufficient clues, but we prefer to limit ourselves to implementation where we can examine the source code.

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

2.1 PCL

Portable Common Loops[2], PCL for short, is a library that implements the functions defined in the book "The Art of the Metaobject Protocol" [2], and is meant as an add-on to pre-standard Common Lisp implementations, i.e., implementations without CLOS.

Most Common Lisp implementations that exist today were initially written before the standard was published, and many of those implementations chose to use PCL to incorporate CLOS functionality, though frequently, the code has since been adapted for each specific implementation. Much of the analysis in this section was also described in [4], although the description in that paper refers to the way SBCL handled method combinations at the time that article was written.

PCL unsurprisingly defines the class `method-combination` and then the class `standard-method-combination` as a subclass of the class named `method-combination`.

More surprisingly, it then defines two subclasses of the class `standard-method-combination`, namely `long-method-combination` and `short-method-combination`, each for use with the different forms (long and short) of the macro `define-method-combination`.

The class `standard-method-combination` contains slots for the method-combination type (i.e., a symbol), and the method-combination options.

The class `short-method-combination` adds two more slots: namely, the operator and a Boolean that indicates whether the operator, when given a single argument, is the identity function.

The short form of `define-method-combination` adds a method to the generic function `find-method-combination`. The second parameter of this method has an `eql` specializer with the name of the method-combination type being defined. The method function of this method first checks that the options given are valid for the short form of `define-method-combination`, and then it creates a fresh instance of the class `short-method-combination`. In other words, a fresh method combination is created whenever `find-method-combination` is called, which is typically whenever a generic function is created. As a result, with a method-combination type defined by the short form, the method combination of a generic function using this type is not updated as a result of redefining that method-combination type, which is undesirable.

Furthermore, `compute-effective-method` has a method specialized to the class `short-method-combination` that handles the case of the short method combination as a special case.

The long form of `define-method-combination` turns the body of the form into a method-combination procedure. This procedure has the same lambda list as `compute-effective-method`. The expansion of the macro stores this procedure in a global hash table, using the method-combination type as a key. There is a slot for this procedure in the class `long-method-combination`, but this slot is not used.

[2]https://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/oop/clos/pcl/0.html

Like the short form, the long form also creates a method on `find-method-combination`, also with an `eql` specializer for the second parameter. This method simply creates an instance of the class `long-method-combination`. The generic function `compute-effective-method` has a method specialized to the class `long-method-combination`. This method consults the hash table to find the method-combination procedure and applies that procedure to the generic function, the method combination, and the applicable methods.

Appendix B of [4] shows some very strange consequences of the use of the global hash table, combined with the fact that the effective-method caches of existing generic functions are not flushed when the method-combination type is redefined by the long form. A generic function may well end up with some effective methods computed *before* the redefinition and some computed *after* it. Needless to say, this behavior is very undesirable.

In summary then, the generic function named `find-method-combination` acts as a container for method-combination types, encoded as `eql`-specialized methods. Furthermore, there is no attempt to reuse existing method combinations. A new one is created whenever `find-method-combination` is called. Finally, while the validity of the options is verified for the built-in method combination types, no such verification is done for custom method-combination types defined by the long form of `define-method-combination`.

2.2 SBCL

The SBCL[3] Common Lisp implementation uses a heavily modified version of PCL (See Section 2.1). Prior to April of 2018, SBCL used the unmodified technique from PCL as described in section 2.1. The technique described in this section is a result of significant modifications to the code for handling method combinations. The article by Didier Verna [4] published at ELS in April of 2018 contained a detailed description of the technique used by SBCL at that time. The improvements to SBCL were likely a result of the descriptions in that article.

One aspect of the SBCL code that remains from the previous version is that the two subclasses of `method-combination` are still present.

An invocation of `define-method-combination` does not create any new class. Instead, an *info* structure is created, and stored in a hash table that uses the name of the method-combination type as a key. This info structure contains a *cache*, which is an association list. The key of an element of the association list is a list of options for the method combination, and the value of an element is the method-combination object. Initially, the cache is empty, except for the info structure associated with the `standard` method combination.

The function `find-method-combination` is given the name of the method combination and the desired options. It looks up the appropriate info structure, and searches the cache for an element corresponding to the options. If such an element is found, the method-combination object is returned. If no

[3]http://www.sbcl.org/

element is found, a new one is constructed, pushed on the cache, and returned. The new element is constructed by consing the list of options and the result of applying a *constructor function* to the list of options. This constructor function is stored in a slot in the info structure. As a result, existing method-combination instances are reused whenever possible.

When a generic function is defined with one of the built-in method combinations, or with a method combination defined using the short form, SBCL will check that the options given to the `:method-combination defgeneric` option are valid. This verification is done by special-purpose code. However, with a user-defined method combination using the long form, no verification is done. It is only when an attempt is made to invoke the generic function that the method-combination procedure is invoked, and the incompatible lambda lists are detected. Furthermore, the error message is very general and can be difficult to decipher by the programmer.

SBCL handles reevaluation of `define-method-combination` forms with the name of an existing info entry in the hash table. Every method-combination instance contains a list of back pointers to generic functions that use this method combination. The cache of the existing info entry is traversed, and for each method combination, the effective methods of its generic functions are invalidated. The problems indicated in Appendix B of [4] therefore no longer exist in recent versions of SBCL. When the method-combination type is redefined with a different form of `define-method-combination`, SBCL correctly changes the class of the method-combinations of the type in question, but it fails to verify that the existing options are compatible with the new definition, even when the redefinition is using the short form of `define-method-combination`. The reason for this failure is that the options are verified only as a result of a call to `find-method-combination`, and this function is not called when a method-combination type is redefined.

2.3 Clozure Common Lisp

The Clozure Common Lisp[4] implementation (CCL for short) defines the class `method-combination` and then three subclasses of that class:

- `standard-method-combination` with a single instance, namely the standard method combination. This class is used as a specializer in a method on the generic function `compute-effective-method` so as to handle the standard method combination as a special case.
- `short-method-combination` which is used for method combinations defined by the short form of the macro `define-method-combination`.
- `long-method-combination` which is used for method combinations defined by the long form of the macro `define-method-combination`.

The class `standard-method-combination` in CCL thus does not play the role of a general instantiable subclass of `method-combination`.

[4]https://ccl.clozure.com/

The generic function `compute-effective-method` has a method specialized to each of these subclasses. The method specialized to `standard-method-combination` uses special-purpose code in order to achieve the effect of the standard method combination. The standard method combination is thus not defined using `define-method-combination`. Similarly, the method specialized to `short-method-combination` uses special purpose code. Only the method specialized to `long-method-combination` invokes the method-combination procedure to achieve the desired effect.

In CCL, the macro `define-method-combination` does not define a method combination class. Instead it defines an *info* vector (disguised as a structure) that acts as a template for creating method combinations later. The info vector contains the following elements:

- The name of the method-combination class to be created which is either `short-method-combination` or `long-method-combination`.
- An element that contains the short-form options if the info vector was created as a result of the short form of `define-method-combination`, and the method-combination procedure (called the *expander function* in CCL) if the info vector was created as a result of the long form.
- A list of *instances*, i.e., method-combination objects that share the same info vector.
- A list of generic functions using method combinations of the type defined by the info vector.

This information is used in order to invalidate effective-method caches when a method-combination type is redefined. Therefore, CCL does not have the problem that PCL does, described in Section 2.1.

When a long method-combination type is redefined using the short form of `define-method-combination`, every generic function having a method combination of that type is accessed, and the method-combination options are checked so that they are valid for the short method combination, i.e., either there are no explicit options or the options consist of a singleton list containing either `:most-specific-first` or `:most-specific-last`. No analogous verification is made when a short method-combination type is redefined using the long form. However, in both cases, the method combination with the redefined type is passed to `change-class`, thereby making the redefinition effective in all generic functions with a method combination of that type.

2.4 ECL

The ECL[5] Common Lisp implementation defines the class `method-combination`, and method-combination metaobjects are direct instances of this class. Thus, in this respect, ECL is not conforming.

Unlike PCL, SBCL, and CCL, ECL does not define any subclasses of the instantiable class. Method-combination types defined by the short form are rewritten to the equivalent long form.

[5]https://common-lisp.net/project/ecl/

The macro `define-method-combination` does not define a new method-combination class. Instead it defines a method-combination procedure. This procedure computes the effective method of a generic function. The lambda list of the method-combination procedure consists of two required parameters: namely, a generic function and a list of applicable methods, followed by the lambda list given to `define-method-combination`. For most built-in method-combination types, that lambda list will contain an optional parameter named `order` with a default value of `:most-specific-first`. The resulting method-combination procedure is stored in a hash table with the name of the method-combination type as a key.

When a generic function is created, a new instance of the `method-combination` class is created. The new instance contains the method-combination procedure and a list of the options given after the method-combination name in the `:method-combination` option to `defgeneric`.

Redefining a method-combination type does not have any effect on existing generic functions having a method combination of that type. The hash table containing method-combination procedures is updated, but this update does not affect existing generic functions.

The `standard` method combination is not defined using the macro `define-method-combination`. Instead, it is defined using special-purpose code.

Incompatibilities between method-combination options given to `find-method-combination` and the lambda list of the method-combination procedure are detected when an effective method needs to be computed. Because there is no specific class for method combinations defined by the short form, this behavior is true also for method-combination types defined by the short form.

Because the short form is rewritten into the long form, and the body of the resulting form contains no verification that the option is either `:most-specific-first` or `:most-specific-last`, it is possible to give any object as an option to `find-method-combination`. Any object different from the keyword `:most-specific-last` will make the resulting method combination behave as if `:most-specific-first` had been given. We argue that this behavior is not conforming, since the description of the short form of `define-method-combination` states that this form "automatically includes error checking".

2.5 Clasp

Clasp [3] is a Common Lisp implementation based on ECL (See Section 2.4), although all the C code in ECL was rewritten in C++.

A large part of the Common Lisp code in Clasp is identical or near-identical to the corresponding code in ECL, and that includes the code for handling method combinations. As a result, Clasp handles method combinations in exactly the same way as ECL.

3 OUR TECHNIQUE

3.1 Representation of method combinations

We introduce a class named `method-combination-template`. An instance of this class represents all method combinations with the same *name*, independent of the options. There is a template for `standard`, a template for `and`, etc. Furthermore, in order to respect the restriction required by the standard, we introduce a class `standard-method-combination` which is a subclass of `method-combination`. All method-combination metaobjects are direct instances of this subclass. There are no subclasses of `standard-method-combination`, neither for specific method-combination types, nor for distinguishing between method combinations defined by the long and the short form of `define-method-combination`. In other words, a method combination is a *variant* of a method-combination template. The template contains a list of all its variants in use.

A method-combination instance contains the following slots:

- A reference to its template.
- The list of method-combination *options* to be given to `find-method-combination`, and that typically appear after the method-combination name of the `:method-combination defgeneric` option.
- The method-combination procedure. This procedure has two parameters, both required. The first parameter is a generic function for which an effective method is to be computed. The second parameter is a list of pairs. Each pair contains an applicable method, and a list of method *qualifiers* for that method. The result of applying the method-combination procedure is a form called the *effective method*. Notice that the method-combination procedure does *not* have the method-combination options in its lambda list.
- A list of generic functions that contain this method combination.

3.2 When `find-method-combination` is called

The expansion of the `defgeneric` macro contains a call to the ordinary function `ensure-generic-function`. If the `:method-combination` option is explicitly supplied to the call to `defgeneric`, then the call to `ensure-generic-function` contains an explicit keyword argument `:method-combination` with the value form being a call to the generic function `find-method-combination` with the generic function, the name of the method-combination type, and the options. If no `:method-combination` option is given in the `defgeneric` form, the `:method-combination` keyword argument to the call to `ensure-generic-function` is not supplied.

The call to `find-method-combination` either returns an existing method-combination instance corresponding to the type and the options given, or it creates and stores a new such instance. If the options are incompatible with the method-combination template, a warning is signaled, and the method-combination procedure is one that signals an error if invoked. The mechanism for detecting this incompatibility is described later in this section.

A call to `ensure-generic-function` results in a call to `ensure-generic-function-using-class` where the first argument is either an existing generic function or `nil` if no generic function with the given name exists. The method on `ensure-generic-function-using-class` specialized to the class `null` supplies the `standard` method-combination as a default value of the `:method-combination` when calling `make-instance` to create a new generic function.

To detect whether a list of method-combination options are invalid for a particular method-combination template, we analyze the *lambda-list* given in the long form of `define-method-combination`. The analysis consists of extracting all parameters that can be referenced in the method-combination procedure. We then construct a lambda expression as follows:

```
(lambda (...)
  (list v1 v2 ... vn))
```

which is then compiled so that a function is obtained. The lambda list of this function is the lambda list that appears in the `define-method-combination` form and `v1`, `v2`, ..., `vn` are the lexical variables resulting from our analysis of the lambda list. Applying this function to the options given to the `find-method-combination` function returns a list of objects. The lambda list typically contains `&aux` lambda list keywords, with forms that check the validity of the options supplied, and signal an error whenever an invalid option combination is detected. Thus, if either the lambda list is incompatible with the options given, or one of these `&aux` forms detects an invalid option combination, an error is signaled. We handle this error, turn it into a warning, and return a method-combination instance with a method-combination procedure that signals an error whenever invoked.

This technique for detecting incompatible or invalid options handles the first scenario described in Section 1. When the user corrects the incorrect form that created or reinitialized the generic function (typically a `defgeneric` form), the validation process is re-invoked and a method-combination with a viable method-combination procedure is assigned to the generic function. This technique also detects the second scenario described in Section 1. The way the user can correct the situation in this scenario is described below.

When the options given to `find-method-combination` are compatible and valid, a viable method-combination procedure is constructed as follows:

```
(lambda (generic-function method-qualifier-pairs)
  (let ((v1 ...) (v2 ...) ... (vn ...))
    <body>))
```

where `v1`, `v2`, ..., `vn` are again the lexical variables resulting from our analysis of the lambda list. The initialization forms for the variables are the values returned in the resulting list of our analysis function.

3.3 Redefining a method-combination type

When a `define-method-combination` form is re-evaluated, we locate the corresponding method-combination template. We then invoke the same analysis as before to every variant, i.e., to every existing method combination having this type name. If an analysis fails, we then signal a warning containing all generic functions using the now invalid method-combination, and we set the method-combination procedure of the invalid method combination to one that will signal an error when invoked. If the analysis succeeds, then the corresponding method combination is assigned a viable method-combination procedure.

3.4 Expanding the short form to the long form

As mentioned in Section 1, it is not obvious how to transform the short form of `define-method-combination` into the long form. Recall that the syntax of the short form is:

`(define-method-combination` *name [[short-form-options]]*`)`
where a *short-form-option* can be:

- `:documentation` *documentation*
- `:identity-with-one-argument` *identity-with-one-argument*
- `:operator` *operator*

Here, *documentation* is a string that is not evaluated. When the short form gets turned into the long form, it becomes an ordinary documentation string, preceding the forms of the body of the long form.

To illustrate where the remaining options end up in the long form, recall the following example from the dictionary entry for `define-method-combination`, where both the short form and the long form are used to define the built-in method-combination `and`. We have changed only the layout of the code so that it will fit on the page.
The short form is:

```
(define-method-combination and
  :identity-with-one-argument t)
```

The long form is;

```
(define-method-combination and
        (&optional (order :most-specific-first))
        ((around (:around))
         (primary (and) :order order :required t))
  (let ((form (if (rest primary)
                  `(and ,@(mapcar
                            #'(lambda (method)
                                `(call-method ,method))
                            primary))
                  `(call-method ,(first primary)))))
    (if around
        `(call-method ,(first around)
                      (,@(rest around)
                       (make-method ,form)))
        form)))
```

The option *identity-with-one-argument* is responsible for the form:

```
(if (rest primary)
    `(and ,@(mapcar
              #'(lambda (method)
                  `(call-method ,method))
              primary))
    `(call-method ,(first primary)))))
```

Had this option been `nil` or not present, the corresponding form would have looked like this instead:

```
`(and ,@(mapcar
          #'(lambda (method)
              `(call-method ,method))
          primary))
```

In order for our technique to work for the short form, when we express the short form in terms of the long form, we modify the lambda list of the long form compared to the example above as follows:

```
(&optional (order :most-specific-first)
 &aux (ignore (unless (member order
                              '(:most-specific-first
                                :most-specific-last))
                (error ....))))
```

Now, any attempt to call a function with this lambda list with a number of arguments other than exactly 1, or with one argument that is neither `:most-specific-first` nor `:most-specific-last` will fail.

4 CONCLUSIONS AND FUTURE WORK

We define a subclass `standard-method-combination` of the specified class `method-combination`. Method combinations created from a method-combination type defined by the macro `define-method-combination` are all instances of this subclass.

Our technique allows for early detection of mismatches between the method-combination options given when a method combination is created as a result of calling `find-method-combination` and the lambda list given to the invocation of `define-method-combination`. We detect such mismatches when a new method combination is created, but also when a method-combination type is redefined with a modified invocation of `define-method-combination` using the name of an existing method-combination type.

Furthermore, while a mismatch exists, our technique results in an error being signaled whenever an attempt is made to use the faulty method combination in order to create an effective method.

Future work includes incorporating our technique into the SICL code base. The technique described in this paper was developed after our initial implementation of method combinations in SICL. (Hence, this technique was not in SICL from the start.) Currently, SICL does not have any data structure allowing weak references, but such references would be desirable for the back pointer from a method combination to the generic functions using it. Otherwise, a memory leak would result from using `fmakunbound` or some other operator that makes the back pointer be the only reference to the generic function. In general, it is impossible to have such operators remove the back pointer, since there could be any number of references to the generic function in question.

5 ACKNOWLEDGMENTS

We would like to thank Yan Gajdoš and Cyrus Harmon for providing valuable feedback on early versions of this paper.

REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp.* American National Standards Institute, 1994.

[2] Gregor Kiczales and Jim Des Rivieres. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991. ISBN 0262111586.

[3] CE Schafmeister. Clasp–A Common Lisp that Interoperates with C++ and Uses the LLVM Backend. In *Proceedings of the 8th European Lisp Symposium*, pages 90–91, New York, NY, USA, 2015. ACM.

[4] Didier Verna. Method combinators. In *11th European Lisp Symposium*, Marbella, Spain, April 2018. ISBN 9782955747421. doi: 10.5281/zenodo.3247610.

Bringing GNU Emacs to Native Code

Andrea Corallo
akrl@sdf.org

Luca Nassi
luknax@sdf.org

Nicola Manca
nicola.manca@spin.cnr.it
CNR-SPIN
Genoa, Italy

ABSTRACT

Emacs Lisp (Elisp) is the Lisp dialect used by the Emacs text editor family. GNU Emacs can currently execute Elisp code either interpreted or byte-interpreted after it has been compiled to byte-code. In this work we discuss the implementation of an optimizing compiler approach for Elisp targeting native code. The native compiler employs the byte-compiler's internal representation as input and exploits *libgccjit* to achieve code generation using the GNU Compiler Collection (GCC) infrastructure. Generated executables are stored as binary files and can be loaded and unloaded dynamically. Most of the functionality of the compiler is written in Elisp itself, including several optimization passes, paired with a C back-end to interface with the GNU Emacs core and *libgccjit*. Though still a work in progress, our implementation is able to bootstrap a functional Emacs and compile all lexically scoped Elisp files, including the whole GNU Emacs Lisp Package Archive (ELPA) [6]. Native-compiled Elisp shows an increase of performance ranging from 2.3x up to 42x with respect to the equivalent byte-code, measured over a set of small benchmarks.

CCS CONCEPTS

• **Software and its engineering** → **Compilers**; *Dynamic compilers*; **Software performance**; *Development frameworks and environments.*

KEYWORDS

GNU Emacs, Elisp, GCC, libgccjit

ACM Reference Format:
Andrea Corallo, Luca Nassi, and Nicola Manca. 2020. Bringing GNU Emacs to Native Code. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3736363

1 INTRODUCTION

GNU Emacs is known as the extensible, customizable, free/libre text editor [19]. This is not only one of the most iconic text editors, GNU Emacs (from now on just "Emacs" for simplicity) represents metaphorically the hearth of the GNU operating system. Emacs can be described as a Lisp implementation (Emacs Lisp) and a very broad set of Lisp programs written on top that, capable of a surprising variety of tasks. Emacs' design makes it one of the most popular Lisp implementations to date. Despite being widely employed, Emacs has maintained a remarkably naïve design for such a long-standing project. Although this makes it didactic, some limitations prevent the current implementation of Emacs Lisp to be appealing for broader use. In this context, performance issues represent the main bottleneck, which can be broken down in three main sub-problems:

- lack of true multi-threading support,
- garbage collection speed,
- code execution speed.

From now on we will focus on the last of these issues, which constitutes the topic of this work.

The current implementation traditionally approaches the problem of code execution speed in two ways:

- Implementing a large number of performance-sensitive primitive functions (also known as *subr*) in C.
- Compiling Lisp programs into a specific assembly representation suitable for targeting the Emacs VM called Lisp Assembly Program (LAP) and assembling it into byte-code. This can be eventually executed by the byte-interpreter [3, Sec.1.2], [15, Sec. 5.1].

As a result, Emacs developers had to implement a progressively increasing amount of functions as C code primarily for performance reasons. As of Emacs 25, 22% of the codebase was written in C [3, Sec. 1.1], with consequences on maintainability and extensibility [24]. The last significant performance increase dates back to around 1990, when an optimizing byte-compiler including both source level and byte-code optimizations was merged from Lucid Emacs [15, Sec. 7.1]. However, despite progressive improvements, the main design of the byte-code machine stands unmodified since then. More recently, the problem of reaching better performance has been approached using Just-In-Time (JIT) compilation techniques, where three such implementations have been attempted or proposed so far [15, Sec. 5.11], [23]. Possibly due to their simplistic approaches none of them proved to introduce sufficient speed-up, in particular if compared to the maintenance and dependency effort to be included in the codebase. In contrast, state-of-the-art high-performance Lisp implementations rely on optimizing compilers targeting native code to achieve higher performance [12]. In this context, C-derived toolchains are already employed by a certain number of Common Lisp implementations derived from KCL [2, 9, 17, 18, 31], where all these, except CLASP, target C code generation.

In this work we present a different approach to tackle this problem, based on the use of a novel intermediate representation (IR) to bridge Elisp code with the GNU Compiler Collection [20]. This intermediate representation allows to effectively implement a number of optimization passes and for Elisp byte-code to be translated

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27–28 2020, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.3736363

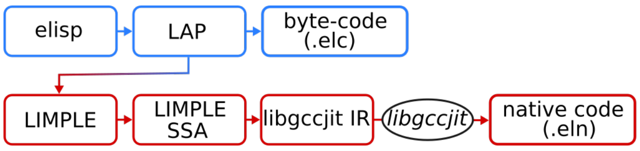


Figure 1: Program representation formats used by byte-compiler (blue) native compiler (red) pipelines.

to a C-like semantic, compatible with the full pipeline of GCC optimization passes. This process relies on *libgccjit* to plug into the GCC infrastructure and achieve code generation without having to target any intermediate programming language [14]. The result of the compilation process for a compilation unit (CU) is a file with `.eln` extension (Emacs Lisp Native). This is a new file extension we have defined to hold the generated code and all the necessary data to have it re-loadable over different Emacs runs. This last characteristic, in contrast with typical JIT-based approaches, saves from having to recompile the same code at each run and allows for more time expensive optimization passes. Also, the classical Lisp image dump feature is supported.

From a more general point of view, here we demonstrate how a Lisp implementation can be hosted on top of *libgccjit*. Although different libraries for code generation, such as *libjit* [13] or LLVM [11], have been successfully employed by various Lisp implementations so far [10, 18], we are not aware of any leveraging *libgccjit*. Moreover, the proposed infrastructure introduces better support for functional programming style in Emacs Lisp with a pass performing tail recursion elimination [30] and the capability to be further extended in order to perform full tail call optimization.

2 IMPLEMENTATION

The proposed compiler pipeline can be divided in three main stages:

- Front-end: Lisp programs are compiled into LAP by the current byte-compiler.
- Middle-end: LAP is converted into "LIMPLE", a new intermediate representation named after GCC GIMPLE [8] which is the very core of the proposed compiler infrastructure. LIMPLE is a sexp-based IR used as static single assignment (SSA) representation [7, 26]. Middle-end passes manipulate LIMPLE by performing a series of transformations on it.
- Back-end: LIMPLE is converted into the *libgccjit* IR to trigger the final compilation through the conventional GCC pipeline.

The sequence of program representation formats is presented in Figure 1. The compiler takes care of type and value propagation through the program control flow graph. We point out that, since Emacs Lisp received in 2012 lexical scope support, two different sub-languages are currently coexisting [15, Sec. 8.1]. The proposed compiler focuses on generating code for the new lexically scoped dialect only, since the dynamic one is considered obsolete and close to deprecation.

2.1 LAP to *libgccjit* IR

Here, we briefly discuss the two endpoints of our compilation pipeline: the Lisp Assembly Program and the *libgccjit* IR, by showing different representations of an illustrative and simple code.

LAP representation is a list of instructions and labels, expressed in terms of S-expressions. Instructions are assembled into byte-code, where each one is associated with an operation and a manipulation of the execution stack, both happening at runtime and defined by the opcode corresponding to the instruction. When present, control flow instructions can cause a change in the execution flow by executing jumps to labels. As an example, the Lisp expression `(if *bar* (+ *bar* 2) 'foo)` is compiled into the following LAP representation:

```
(byte-varref *bar*)
(byte-goto-if-nil TAG 8)
(byte-varref *bar*)
(byte-constant 2)
(byte-plus)
(byte-return)
(TAG 8)
(byte-constant foo)
(byte-return)
```

where `byte-varref` pushes the value of a symbol into the stack, `byte-goto-if-nil` pops the top of the stack and jumps to the given label if it is `nil`, `byte-constant` pushes the value from an immediate into the stack, `byte-plus` pops two elements from the stack, adds them up and pushes the result back into the stack and `byte-return` exits the function using the top of the stack as return value. An extensive description of these instructions is available in Ref. [3, Sec. 1.3].

libgccjit allows for describing code programmatically in terms of `gcc_jit_objects` created through C or C++ API [14, Sec. *Objects*]. The semantic it can express can be described as a subset of the one of the C programming language. This includes `l` and `r` values, arithmetic operators, assignment operators and function calls. The most notable difference with respect to C is that conditional statements such as `if` and `else` are not supported and the code has to be described in terms of basic blocks. Inside GCC, *libgccjit* IR is mapped into GIMPLE when the actual compilation is requested. One key property of Emacs Lisp LAP is that it guarantees that, for any given program counter, the stack depth is fixed and known at compile time. The previous LAP code can be transformed in the following pseudo-code, suitable to be described in the *libgccjit* IR:

```
Lisp_Object local[2];

bb_0:
  local[0] = varref (*bar*);
  if (local[0] == NIL) goto bb_2;
  else goto bb_1;

bb_1:
  local[0] = varref (*bar*);
  local[1] = two;
  local[0] = plus (local[0], local[1]);
  return local[(int)0];

bb_2:
  local[0] = foo;
  return local[0];
```

This transformation accomplishes the following:

- performs opcode decoding during the transformation so that it is not needed anymore at runtime.
- decodes and compiles all the operations within the original stack into assignments.
- splits the initial list of LAP instructions into basic blocks.

These tasks are performed by means of an intermediate translation into LIMPLE, which enables standard code optimization routines present in GCC as well as dedicated optimization passes.

2.2 LIMPLE IR

As previously introduced, in order to implement a series of optimization passes, we defined an intermediate representation, that we called "LIMPLE", whose main requirement is to be SSA. The description of every variable inside the compiler is accomplished with instances of a structure we called `m-var` and reported in Appendix A. This represents the Lisp objects that will be manipulated by the function being compiled. A function in LIMPLE is a collection of basic blocks connected by edges to compose a control flow graph where every basic block is a list of `insn` (instructions). The format of every LIMPLE `insn` is a list `(operator operands)` whose valid operands depend on the operator itself, such as:

- `(set dst src)` Copy the content of the slot represented by the `m-var` `src` into the slot represented by `m-var` `dst`.
- `(setimm dst imm)` Similar to the previous one but `imm` is a Lisp object known at compile time.
- `(jump bb)` Unconditional jump to basic block whose name is represented by the symbol `bb`.
- `(cond-jump a b bb_1 bb_2)` Conditional jump to `bb_1` if `a` and `b` are `eq` or to `bb_2` otherwise.
- `(call f a b ...)` Call a primitive function `f` where `a, b, ...` are `m-vars` used as parameters.
- `(comment str)` Include annotation `str` as comment inside the `.eln` debug symbols (see Sec. 5.3).
- `(return a)` Perform a function return having as return value the `m-var a`.
- `(phi dst src1 ... srcn)` Conventional Φ node used by SSA representation. When all `m-vars` `src1 ... srcn` have the same immediate value this is assigned to `dst`. Similarly it happens for the type (no handling for type hierarchy is implemented up to date).

3 COMPILATION PASSES

The compilation pipeline is composed by a sequence of passes that, starting from the input Elisp source code, apply a succession of transformations to finally produce the executable code in the form of a `.eln` file. The following sections describe each of the compilation passes, all of them are implemented in Lisp with the exception of `final`.

However, before getting into the details of each pass, it is useful to discuss the reason why the data-flow analysis and optimization algorithms already present in the GCC infrastructure are not enough for the Elisp semantic, and dedicated ones had to be developed in Lisp.

Type propagation. Emacs Lisp is a strong dynamically-typed programming language. Typing objects is done through tagging pointers [27]. While GCC has passes to propagate both constants and ranges, it has no visibility of the Lisp type returned by Lisp primitive functions and, as a consequence, on the tag bits set.

Pure functions. Similarly, GCC does not know which Lisp functions can be optimized at compile time having visibility only on the local compilation unit. Optimizable functions are typically pure functions or functions that are pure for a specific set of parameters.

Reference propagation. Another useful property to be propagated is if a certain object will or will not be referenced [29] during function calls. This information is required to generate a more efficient code, as discussed in Sec. 3.8.

Unboxing. GCC does not offer infrastructure for unboxing values. Although not yet implemented, the proposed infrastructure is designed to host further improvements, such as unboxing, requiring data-flow analysis [28].

Compiler hints. The data-flow analysis can be fed with compiler hints about the type of certain expressions, included as high-level annotations in the source code by the programmer.

Warning and errors. A data-flow analysis engine as the one proposed in this work could be used in the future to provide more accurate warnings and errors during the compilation phase.

GCC optimization constraints. GCC optimization passes often adopt conservative strategies not to break the semantic of all the supported programming languages. As an example, the GCC tail call optimization pass does not perform transformations whenever an instruction referencing memory is present in the compiled function. Given the specific semantic of the code generated by the proposed work, conditions as the one mentioned may be too restrictive resulting in missed optimizations.

3.1 *spill-lap*

As already discussed, the main input for the compilation process is the Lisp Assembly Program Intermediate Representation (LAP IR). *spill-lap* runs the byte-compiler infrastructure with the Elisp source as input collecting all top-level forms and spilling the LAP before it is assembled into final byte-code.

3.2 *limplify*

This pass is responsible for translating LAP IR into LIMPLE IR. In general, LAP is a sequence of instructions, labels and jumps-to-label. Since the Emacs byte-interpreter is a stack-based machine, every LAP instruction manipulates the stack [3], [15, Sec. 5.1]. It is important to highlight that at this stage all the stack manipulations performed by LAP instructions are compiled into a series of `m-var` assignments. Spurious moves will eventually be optimized out by GCC. This pass is also responsible for decomposing the function into lists of LIMPLE `insns`, or basic blocks. The code necessary for the translation of most LAP instructions is automatically generated using the original instruction definition specified in the byte-compiler.

3.3 *static single assignment (SSA)*

This pass is responsible for bringing LIMPLE into minimal SSA form, discussed in [7, Sec. 2.2], as follows:

a) Edges connecting the various basic blocks are created.
b) The dominator tree is computed for each basic block [4].
c) Dominator frontiers are computed for each basic block.
d) Φ functions are placed as described in [7, Sec. 3.1].
e) `m-vars` goes through classic SSA renaming.

Once LIMPLE is in SSA form every `m-var` object appears as destination of an instruction only in one place within the SSA lattice. The same object can be referenced multiple times as source though, but each `m-var` can be identified by its unique `id` slot.

3.4 *forward data-flow analysis*

For each `m-var`, this pass propagates the following properties within the control flow graph: value, type and where the `m-var` will be allocated (see Sec. 3.8). Initially, all immediate values set at compile time by `setimm` are propagated to each destination `m-var`. Afterwards, for each `insn` in each basic block the following operations are iteratively performed:

a) If the `insn` is a Φ, the properties of `m-vars` present as source operands are propagated to the destination operand when in agreement.
b) If a function call has a known return type, this is propagated to the result.
c) If a function call to a pure function is performed with all arguments having a known value, the call is optimized out and the resulting value is substituted.
d) Assignments by `set` operators are used to propagate all `m-vars`.

This sequence is repeated until no more changes are performed in the control flow graph.

3.5 *call-optim*

This pass is responsible for identifying all function calls to primitives going through the `funcall` trampoline and substitute them with direct calls. The primitive functions most commonly used in the original LAP definition are assigned dedicated opcodes, as described in [3, page 172]. When a call to one of these functions is performed, the byte-interpreter can thus perform a direct call to the primitive function. All the remaining functions are instead called through the `funcall` trampoline, which carries a considerable overhead. This mechanism is due to the intrinsic limit of the opcode encoding space. On the other hand, native-compiled code has the possibility to call all Emacs primitives without any encoding space limitation. After this pass has run, primitive functions have all equal dignity, being all called directly irrespective of the fact that they were originally assigned a dedicated byte-opcode or not. The same transformation is performed for function calls within the compilation unit when the compiler optimization level is set to its maximum value (see Sec. 5.1). This will improve the effectiveness of inlining and other inter-procedural optimizations in GCC. Finally, recursive functions are also optimized to prevent `funcall` usage.

3.6 *dead-code*

This pass cleans up unnecessary assignments within the function. The algorithm checks for all `m-vars` that are assigned but not used elsewhere, removing the corresponding assignments. This pass is also responsible for removing function calls generated by compiler type hints (see Sec. 5.2) if necessary.

3.7 *tail recursion elimination (TRE)*

This peephole pass [16, Chap. 18] performs a special case of tail call optimization called tail recursion elimination. The pass scans all LIMPLE `insns` in the function searching for a recursive call in tail position. If this is encountered it is replaced with the proper code to restart executing the current function using the new arguments without activating a new function frame into the execution stack. This transformation is described in [16, Chap. 15.1].

3.8 *final (code layout)*

This pass is responsible for converting LIMPLE into *libgccjit* IR and invoking the compilation through GCC. We point out that the code we generate for native-compiled Lisp functions follows the same ABI of Elisp primitive C functions. Also, a minimal example of pseudo C code for a native-compiled Elisp function is listed in Appendix B.

When optimizations are not engaged, `m-vars` associated to each function are arranged as a single array of Lisp objects. This array has the length of the original maximum byte-code stack depth. Depending on the number of their arguments, Elisp primitive functions present one of the following two C signatures [22]:

a) `Lisp_Obj fun (Lisp_Obj arg01, ..., Lisp_Obj argn)`, for regular functions with a number of arguments known and smaller or equal to 8.
b) `Lisp_Obj fun (ptrdiff_t n, Lisp_Obj *args)`, otherwise.

where `ptrdiff_t` is an integral type, `n` is the number of arguments and `args` is a one-dimensional array containing their values. When a call of the second kind is performed, GCC clobbers all the `args` array content, regardless the number of arguments `n` involved in the call. This means that the whole array content after the call is considered potentially modified. For this reason the compiler cannot trust values already loaded in registers and has to emit new load instructions for them. To prevent this, when the optimization we have called "advanced frame layout" is triggered, each `m-var` involved in a call of the second kind is rendered in a stack-allocated array dedicated to that specific call. All other `m-vars` are rendered as simple automatic variables. The advanced frame layout is enabled for every compilation done with a non zero `comp-speed`, as discussed in Sec. 5.1.

This pass is also responsible for substituting the calls to selected primitive functions with an equivalent implementation described in *libgccjit* IR. This happens for small and frequently used functions such as: `car`, `cdr`, `setcar`, `setcdr`, `1+`, `1-`, or `-` (negation). As an example, the signature for function `car` implemented in *libgccjit* IR will be:

```
static Lisp_Object CAR (Lisp_Object c,
                        bool cert_cons)
```

If compared to the original `car` function a further parameter has been added, `cert_cons` which stands for "certainly a `cons`". *final* will emit a call to `CAR` setting `cert_cons` to `true` if the data-flow analysis was able to prove `c` to be a `cons` or setting it to `false` otherwise. This mechanism is used in a similar fashion with most inlinable functions injected by this pass in order to provide the information obtained by the data-flow analysis to the GCC one. Since the GCC implementation has the full definition of these functions, they can be optimized effectively.

4 SYSTEM INTEGRATION

4.1 Compilation unit and file format

The source for a compilation unit can be a Lisp source file or a single function and, as already mentioned, the result of the compilation process for a compilation unit is a file with `.eln` extension. Technically speaking, this is a shared library where Emacs expects to find certain symbols to be used during load. The conventional load machinery is modified such that it can load `.eln` files in addition to conventional `.elc` and `.el` files.

In order to be integrated in the existing infrastructure we define the `Lisp_Native_Comp_Unit` Lisp object. This holds references to all Lisp objects in use by the compilation unit plus a reference to the original `.eln`. Every `.eln` file is expected to contain a number of symbols including:

- `freloc_link_table`: static pointer to a structure of function pointers used to call Emacs primitives from native-compiled code.
- `text_data_reloc`: function returning a string representing all immediate constants in use by the code of the compilation unit. The string is formed using `prin1` so that it is suitable to be read by Lisp reader.
- `d_reloc`: static array containing the Lisp objects used by the compiled functions.
- `top_level_run`: function responsible of performing all the modifications to the environment expected by the load of the compilation unit.

4.2 Load mechanism

Load can be performed conventionally as: `(load "test.eln")`. Loading a new compilation unit translates into the following steps:

a) Load the shared library into the Emacs process address space.
b) Given that `.eln` plugs directly into Emacs primitives, forward and backward version compatibility cannot be ensured. Because of that each `.eln` is signed during compilation with an hash and this is checked during load. In case the hash mismatches the load process is discarded.
c) Lookup the following symbols in the shared library and set their values: `current_thread_reloc`, `freloc_link_table`, `pure_reloc`.
d) Lookup `text_data_reloc` and call it to obtain the serialized string representation of all Lisp objects used by native-compiled functions.
e) Call the reader to deserialize the objects from this string and set the resulting objects in the `d_reloc` array.
f) Lookup and call `top_level_run` to have the environment modifications performed.

We show in Appendix B an example of pseudo C code for a native-compiled function illustrating the use of `freloc_link_table` and `d_reloc` symbols.

When loaded, the native-compiled functions are registered as *subr*s as they share calling convention with primitive C functions. Both native-compiled and primitive functions satisfies `subrp` and are distinguishable using the predicate `subr-native-elisp-p`.

4.3 Unload

The unload of a compilation unit is done automatically when none of the Lisp objects defined in it is referenced anymore. This is achieved by having the `Lisp_Native_Comp_Unit` object been integrated with the garbage collector infrastructure.

4.4 Image dump

Emacs supports dumping the Lisp image during its bootstrap. This technique is used in order to reduce the startup time. Essentially a number of `.elc` files are loaded before dumping the Emacs image that will be invoked during normal use. As of Emacs 27 this is done by default by relying on the portable dumper, which is in charge of serializing all allocated objects into a file, together with the information needed to revive them. The final Emacs image is composed by an executable plus the matching dump file. Image dump capability has been extended to support native-compiled code, the portable dumper has been modified to be able to dump and reload `Lisp_Native_Comp_Unit` Lisp objects.

4.5 Bootstrap

Since the Elisp byte-compiler is itself written in Elisp, a bootstrap phase is performed during the build of the standard Emacs distribution. Conventionally this relies on the Elisp interpreter [15, Sec. 5.2.1]. We modified the Emacs build system to allow for a full bootstrap based on the native compiler. The adopted strategy for this is to follow the conventional steps to produce `.eln` files instead of `.elc` when possible (lexically scoped code) and fall-back to `.elc` otherwise. More than 700 Elisp files are native-compiled in this process.

4.6 Documentation and source integration

The function documentation string, `describe-function`, and "goto definition" mechanism support have been implemented and integrated such that native-compiled code behaves as conventional byte-compiled code.

4.7 Verification

A number of tests have been defined to check and verify the compiler. These include some micro test cases taken from Tom Tromey's JIT [15, Sec. 5.11], [23]. A classical bootstrap compiler test has also been defined, where the interpreted compiler is used to native-compile itself, and then the resulting compiler is used to compile itself. Finally, the two produced binaries are compared. The test is successful if the two objects are bytewise identical.

5 ELISP INTERFACE

5.1 Code optimization levels

Some special variables are introduced to control the compilation process, most notably `comp-speed`, which controls the optimization level and safety of code generation as follows:

(0) No optimization is performed.
(1) No Lisp-specific optimization is performed.
(2) All optimizations that do not modify the original Emacs Lisp semantic and safeness are performed. Type check elision is allowed where safe.
(3) Code is compiled triggering all optimizations. Intra compilation unit inlining and type check elision are allowed. User compiler hints are assumed to be correct and exploited by the compiler.

`comp-speed` also controls the optimization level performed by the GCC infrastructure as indicated by the table below.

`comp-speed`	0	1	2	3
propagate	n	n	y	y
call-optim	n	n	y	y
call-optim (intra CU)	n	n	n	y
dead-code	n	n	y	y
TRE	n	n	n	y
advanced frame layout	n	y	y	y
GCC -Ox	0	1	2	3

5.2 Language extensions

In order to allow the user to feed the data-flow analysis with type suggestions, two entry points have been implemented:

- `comp-hint-fixnum`
- `comp-hint-cons`

These can be used to specify that a certain expression evaluates to the specified type. For example, `(comp-hint-cons x)` ensures that the result of the evaluation of the form itself is a `cons`. Currently, when `comp-speed` is less or equal to 2, type hints are compiled into assertions, while they are trusted for type propagation when using `comp-speed` 3. These low level primitives are meant to be used to implement operators similar to Common Lisp `the` and `declare` [1].

5.3 Debugging facility

libgccjit allows for emitting debug symbols in the generated code and dumping a pseudo C code representation of the *libgccjit* IR. This is triggered for compilations performed with `comp-debug` set to a value greater than zero. Debugging the generated code is achieved using a conventional native debugger such as `gdb` [21]. In this condition, the `final` pass emits additional code annotations, which are visible as comments in the dumped pseudo C code to ease the debugging (see Appendix B).

6 PERFORMANCE IMPROVEMENT

In order to evaluate the performance improvement of the native code, a collection of Elisp benchmarks has been assembled and made available as `elisp-benchmarks` in the Emacs Lisp Package Archive (ELPA) [5]. It includes the following set of programs:

- List processing: traverse a list incrementing all elements or computing the total length.
- Fibonacci number generator: iterative, recursive and tail-recursive implementations.
- Bubble sort: both destructive in-place and non destructive.
- Dhrystone: the famous synthetic benchmark ported from C to Elisp [25].
- N-body simulation: a model of the solar gravitation system, intensive in floating-point arithmetic.

The benchmarking infrastructure executes all programs in sequence, each for a number of iterations selected to have it last around 20 seconds when byte-interpreted. The sequence is then repeated five times and the execution times are averaged per each benchmark. The results reported in Table 1 are obtained from an Intel i5–4200M machine. They compare the execution time of the benchmarks when byte-compiled and run under the vanilla Emacs 28 from master branch against their native-compiled versions at `comp_speed` 3. The native-compiled benchmarks are run under Emacs compiled and bootstrapped at `comp_speed` 2 from the same revision of the codebase.

The optimized native-code allows all the benchmarks to run at least two times faster, with most of them reaching much higher performance boosts. Despite the analysis being still preliminary, the reason behind these improvements can be explained with several considerations. First of all the removal of the byte-interpreter loop which, implementing a stack-machine, fetches opcodes from memory, decodes and executes the corresponding operations and finally pushes the results back to the memory. Instead the native compiler walks the stack at compile time generating a sequence of machine-level instructions (the native code) that works directly with the program data at execution time (see Sec. 3.2). The result of this process is that executing a native-compiled program takes a fraction of machine instructions with respect to byte-interpreting it. Analyzing the instructions mix also reveals a smaller percentage of machine instructions spent doing memory accesses, in favor of data processing ones. This is the fundamental upgrade of native compilation against interpretation and is probably the major source of improvement for benchmarks with smaller speed-ups, where no other optimizations apply.

On the other hand, benchmarks with larger improvements also take advantage of Lisp specific compiler optimizations, in particular from call optimizations (see Sec. 3.5). Function calls avoid the trampoline when targeting subroutines defined in the same compilation unit or when calling pure functions from the C codebase. Moreover, the data-flow analysis step allows to exploit the properties of the structures manipulated by the compiler in order to produce code with less overheads. Function calls can also be completely removed, along with corresponding returns, and replaced by simple jumps for optimized tail recursive functions, avoiding at the same time new allocations on the execution stack.

Finally, the data-flow analysis can be made even more effective when paired with compiler hints. Without these, the only types known at compile time are the ones belonging to constants or values returned by some primitive functions. Type hints greatly increase the chances for the native compiler to optimize out expensive type

benchmark	byte-comp runtime (s)	native-comp runtime (s)	speed-up
inclist	19.54	2.12	9.2*x*
inclist-type-hints	19.71	1.43	13.8*x*
listlen-tc	18.51	0.44	42.1*x*
bubble	21.58	4.03	5.4*x*
bubble-no-cons	20.01	5.02	4.0*x*
fibn	20.04	8.79	2.3*x*
fibn-rec	20.34	7.13	2.9*x*
fibn-tc	21.22	5.67	3.7*x*
dhrystone	18.45	7.22	2.6*x*
nbody	19.79	3.31	6.0*x*

Table 1: Performance comparison of byte-compiled and native-compiled Elisp benchmarks

checks. In our measurements, the same benchmark (*inclist*) annotated with type hints earns a further improvement of 50% in terms of execution speed, while compiled under the same conditions.

7 CONCLUSIONS

In this work we discussed a possible approach to improve execution speed of generic Elisp code, starting from LAP representation and generating native code taking advantage of the optimization infrastructure of the GNU Compiler Collection. Despite its early development stage, the compiler successfully bootstraps a usable Emacs and is able to compile all lexically scoped Elisp present in the Emacs distribution and in ELPA. The promising results concerning stability and compatibility already led this work to be accepted as *feature branch* in the official GNU Emacs repository. Moreover, a set of benchmarks was developed to evaluate the performance gain and preliminary results indicate an improvement of execution speed between 2.3x and 42x, measured over several runs. At last, we point out that most of the optimization possibilities allowed by this infrastructure are still unexplored. Already planned improvements include: supporting `fixnum` unboxing and full tail call optimization, exposing more primitives to the GCC infrastructure by describing them in the *libgccjit* IR during the `final` pass, and allowing to signal warnings and error messages at compile time based on values and types inferred by the data-flow analysis.

ACKNOWLEDGMENTS

Stefan Monnier and Kyrylo Tkachov for reviewing this manuscript. Luca Benso for inspiration, useful discussions and artichokes, and Daniela Ferraro for cooking them.

REFERENCES

[1] American National Standards Institute and Computer and Business Equipment Manufacturers Association. *Draft proposed American National Standard programming language Common LISP: X3.226-199x: Draft 12.24, X3J13/92-102.* pub-CBEMA, pub-CBEMA:adr, July 1992. July 1, 1992.
[2] Giuseppe Attardi. The embeddable common lisp. *Pap. 4th Int. Conf. LISP Users Vendors, LUV 1994*, 8(1):30–41, 1995. doi: 10.1145/224139.1379849.
[3] Rocky Bernstein. *GNU Emacs Lisp Bytecode Reference Manual*, 2018. URL https://rocky.github.io/elisp-bytecode.pdf.
[4] Keith Cooper, Timothy Harvey, and Ken Kennedy. A simple, fast dominance algorithm. *Rice University, CS Technical Report 06-33870*, 01 2006.
[5] Andrea Corallo and Luca Nassi. Elisp benchmarks collection. URL https://elpa.gnu.org/packages/elisp-benchmarks.html.
[6] Emacs Developer Community. Gnu emacs lisp package archive. URL https://elpa.gnu.org/.
[7] National Institute for Research in Digital Science and Technology, editors. *Static Single Assignment Book.* 2018. URL http://ssabook.gforge.inria.fr/latest/book.pdf.
[8] GCC Developer Community. GIMPLE – GNU GCC Internals. URL https://gcc.gnu.org/onlinedocs/gccint/GIMPLE.html.
[9] GNU Common LISP Developer Community. GNU Common LISP. URL https://www.gnu.org/software/gcl.
[10] Jaffer, Aubrey and Lord, Tom and Bader, Miles and GNU Project. Gnu ubiquitous intelligent language for extensions (gnu guile). URL https://www.gnu.org/software/guile/.
[11] Chris Lattner and LLVM Foundation. LLVM. URL https://llvm.org.
[12] Robert A. MacLachlan. The Python compiler for CMU Common Lisp. In *Proc. 1992 ACM Conf. LISP Funct. Program. - LFP '92*, pages 235–246, New York, New York, USA, 1992. ACM Press. ISBN 0897914813. doi: 10.1145/141471.141558. URL http://portal.acm.org/citation.cfm?doid=141471.141558.
[13] David Malcolm and GCC Developer Community. LibJIT, . URL https://www.gnu.org/software/libjit/.
[14] David Malcolm and GCC Developer Community. LibgccJIT, . URL https://gcc.gnu.org/onlinedocs/jit.
[15] Stefan Monnier and Michael Sperber. Evolution of emacs lisp. 2018. URL https://www.iro.umontreal.ca/~monnier/hopl-4-emacs-lisp.pdf.
[16] Steven Stanley Muchnick. *Advanced Compiler Design and Implementation.* 1997. ISBN 1558603204.
[17] 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, 2015. URL https://www.european-lisp-symposium.org/static/proceedings/2015.pdf.
[18] Christian E. Schafmeister and Alex Wood. CLASP Common Lisp Implementation and Optimization. In *Proceedings of the 11th European Lisp Symposium, ELS2018*, pages 59–64, 2018. doi: 10.5555/3323215.3323223. URL https://dl.acm.org/doi/10.5555/3323215.3323223.
[19] Richard M. Stallman and Emacs Developer Community. GNU EMACS text editor. URL https://www.gnu.org/software/emacs/.
[20] Richard M. Stallman and GCC Developer Community. GNU C Compiler. URL https://gcc.gnu.org.
[21] Richard M. Stallman and GDB Developer Community. GNU Debugger. URL https://www.gnu.org/software/gdb/.
[22] Stallman, Richard M. and Elisp Developer Community. GNU Emacs Lisp Reference Manual. URL https://www.gnu.org/software/emacs/manual/html_node/elisp/Writing-Emacs-Primitives.htm.
[23] Tom Tromey. Jit compilation for emacs, 2018. URL https://tromey.com/blog/?p=982.
[24] Tom Tromey. el-compilador, 2018. URL https://github.com/tromey/el-compilador.
[25] Reinhold P. Weicker. Dhrystone: a synthetic systems programming benchmark. *Commun. ACM*, 27(10):1013–1030, oct 1984. ISSN 00010782. doi: 10.1145/358274.358283. URL http://portal.acm.org/citation.cfm?doid=358274.358283.
[26] Wikipedia contributors. Static single assignment form — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/wiki/Static_single_assignment_form.
[27] Wikipedia contributors. Tagged pointer — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/wiki/Tagged_pointer.
[28] Wikipedia contributors. Object type (object-oriented programming)#boxing — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/wiki/Object_type_(object-oriented_programming)#Boxing.
[29] Wikipedia contributors. Reference (computer science) — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/wiki/Reference_(computer_science).
[30] Wikipedia contributors. Tail call — Wikipedia, the free encyclopedia, 2020. URL https://en.wikipedia.org/wiki/Tail_call.
[31] Taiichi Yuasa. Design and implementation of Kyoto Common Lisp. *j-J-INF-PROCESS*, 13(3):284–295, 1990. ISSN 0387-6101.

A DEFINITION OF M-VAR

```
(cl-defstruct (comp-mvar (:constructor make--comp-mvar))
    "A meta-variable being a slot in the virtual-stack."
    (id nil :type (or null number)
        :documentation "SSA unique id number when in SSA form.")
    (const-vld nil :type boolean
        :documentation "Validity signal for the following slot.")
    (constant nil
        :documentation "When const-vld is non-nil this is used for holding a known value.")
    (type nil
        :documentation "When non-nil indicates the type known at compile time."))
```

B EXAMPLE OF A COMPILATION UNIT

Below we show an example of an elementary compilation unit followed by the pseudo C code generated dumping the *libgccjit* IR. The compilation process is performed using comp-speed = 3 and comp-debug = 1.

```
;;;  -*- lexical-binding: t -*-
(defun foo ()
  (if *bar*
      (+ *bar* 2)
      'foo))
```

```
extern union comp_Lisp_Object
F666f6f_foo ()
{
  union comp_Lisp_Object[2] arr_1;
  union comp_Lisp_Object local0;
  union cast_union union_cast_28;
entry:
  /* Lisp function: foo */
  goto bb_0;
bb_0:
  /* const lisp obj: *bar* */
  /* calling subr: symbol-value */
  local0 = freloc_link_table->R73796d626f6c2d76616c7565_symbol_value (d_reloc[0]);
  /* const lisp obj: nil */
  union_cast_28.v_p = (void *)NULL;
  /* EQ */
  if (local0.num == union_cast_28.lisp_obj.num) goto bb_2; else goto bb_1;
bb_2:
  /* foo */
  local0 = d_reloc[2];
  /* const lisp obj: foo */
  return d_reloc[2];
bb_1:
  /* const lisp obj: *bar* */
  /* calling subr: symbol-value */
  arr_1[0] = freloc_link_table->R73796d626f6c2d76616c7565_symbol_value (d_reloc[0]);
  /* const lisp obj: 2 */
  arr_1[1] = d_reloc[3];
  /* calling subr: + */
  local0 = freloc_link_table->R2b_ (2, (&arr_1[0]));
  return local0;
}
```

Why You Cannot (Yet) Write an "Interval Arithmetic" Library in Common Lisp

...or: Hammering Some Sense into `:ieee-floating-point`

Marco Antoniotti
Dipartimento di Informatica, Sistemistica e Comunicazione
Università degli Studi di Milano Bicocca
marco.antoniotti@unimib.it

ABSTRACT

"Interval Arithmetic" (IA) appears to be a useful numerical tool to have at hand in several applications. Alas, the current IA descriptions and proposed standards are always formulated in terms of the IEEE-754 standard, and the status of IEEE-754 compliance of most Common Lisp implementations is not up to par.

A solution would be for Common Lisp implementations to adhere to the *Language Independent Arithmetic* (LIA) IEC standard, which includes IEEE 754.

While the LIA standard provides a set of proposed bindings for Common Lisp, the format and depth of the specification documents is not readily usable by a Common Lisp programmer, should an implementation decide to comply with the provisions. Moreover, much latitude is left to each implementation to provide the LIA "environmental" setup.

It would be most beneficial if more precision were agreed upon by the Common Lisp community about how to provide LIA compliance in the implementations. In that case, a new set of documentation or manuals in the style of the HyperSpec could be provided, for the benefit of the Common Lisp programmer.

The goal of this paper is to foster a discussion within the Common Lisp community to converge on a complete specification for LIA compliance. The paper discusses some of the issues that must be resolved to reach that goal, e.g., error handling and full specification of mathematical functions behavior.

KEYWORDS

Programming Languages, Common Lisp, Library Specification, Floating Point Arithmetic, Language Independent Arithmetic

ACM Reference Format:
Marco Antoniotti. 2020. Why You Cannot (Yet) Write an "Interval Arithmetic" Library in Common Lisp: ...or: Hammering Some Sense into `:ieee-floating-point`. In *Proceedings of the 13th European Lisp Symposium (ELS 2020).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3759522

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS 2020, April 27–28 2020, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.3759522

1 INTRODUCTION

An interesting exercise (academic or not) that a programmer (Common Lisp or not) may find intriguing is to implement an *Interval Arithmetic* (IA) library[1]. Programmers of all stripes would learn a lot if they tried to really implement an IA library. But since most probably won't, this paper may serve as sufficient summary to get you through a cocktail party conversation on the matter.

The usefulness of IA is rather established; many numerical issues can be naturally dealt with by using an IA library, albeit at a slight increase in computation times. There is a nice body of literature and proposed standards to ensure availability of IA in a computing environment, and many of these eventually provide one or two different *interval* representations (*endpoint* and *midpoint*), the operations on them, and nitpicking treatment of corner cases; e.g., intervals with infinite endpoints and interval division by an interval containing 0.

As we shall see, the "nitpicking" boils down to using the IEEE-754/IEC-60559 standards [11]. Eventually, in this work, the use of the IEC *Language Independent Arithmetic* (LIA) standards [6–8] will be advocated. The LIA standard is a comprehensive collection of concepts and carefully thought out behaviors a basic library of *integer*, *floating point* and *complex* numbers mathematical functions and ancillary environment functionalities should look like. One of the, in the opinion of the writer, unstated goals of LIA is that a programmer should be able to relatively easily understand mathematical software writeen in any language ecosystem that abided the specification.

A word of caution. The present paper is neither a full blown Common Lisp LIA specification, nor a description of an implementation of the functionalities depicted herein. It rather is a *leaflet* that intend to present the community a project which, in the modest opinion of the writer, should be completed after careful debate and careful consideration of all the details.

1.1 An IA Library... Hitting the Wall

An IA library in Common Lisp implementing what is known as an *endpoint representation* can be easily started as follows. For brevity, since it is a valid Common Lisp identifier, we use

[1]See, for example [4], or [5] for a rather complete summary with references to the seminal works in the area. IEEE has also published a preliminary standard for IA [10].

the name [] here for what other languages might call an interval.

```
(defstruct ([] (:constructor [] (low high)))
  (low 0.0 :type real)
  (high 0.0 :type real))

(defun radius (i) (- ([]-high i) ([]-low i)))

(defun pointp (i) (= ([]-high i) ([]-low i)))

(defmethod add ((i1 []) (i2 []))
  ([] (+ ([]-low i1) ([]-low i2))
      (+ ([]-high i1) ([]-high i2))))

(defmethod sub ((i1 []) (i2 []))
  ([] (- ([]-low i1) ([]-high i2))
      (- ([]-high i1) ([]-low i2))))
```

After starting in earnest, a Common Lisp (or JAVA, C, R, PYTHON) programmer is soon faced with a number of numerical issues, should she be willing to achieve the best possible behavior out of the IA library.

The problem is that, as mentioned before, IA specifications are usually formulated in terms of the IEEE-754 standard, which, at this point is readily available only to C and C++ programmers[2] In particular, the IA specifications exploit *rounding modes* and *infinities*, which are unevenly available in Common Lisp implementations; another, related issue is the treatment of *floating point exceptions*.

Rounding Modes. If we had available some ways of handling infinities and rounding modes, we could write the IA library operations as follows:

```
(defmethod add ((i1 []) (i2 []))
  ([] (rounding-down (+ ...))
      (rounding-up (+ ...))))
```

where `rounding-down` and `rounding-up` are macros with an intuitive semantics. Unfortunately, at this point in time, it is not possible to provide the `rounding-down` and `rounding-up` macros without delving deeply in an implementation.

Infinities and NaNs. Another issue is the handling of *special values*: essentially *infinities* and *NaN*s (not-a-number). Both items are handled unevenly in Common Lisp implementations; infinities and *quiet* NaNs (*cfr.* the IEEE-754 standard) are somewhat supported; *signaling* NaNs not so much so.

Handling of Floating Point Exceptions. Apart from our doomed IA library, another issue that is not always very well clarified in Common Lisp implementations (and especially *across* them) is how floating points exceptions are handled. The Common Lisp standard defines the conditions:

```
floating-point-overflow
floating-point-underflow
floating-point-inexact
floating-point-invalid-operation
division-by-zero
```

Alas, their use is inconsistent across implementations (apart from the mostly clear cut case of `division-by-zero`). Two implementations may choose to signal `floating-point-invalid-operation` or `floating-point-inexact` on the same operation.

This is not the only issue vis-a-vis Common Lisp and the IEEE-754. A deeper issue pertains the *notification* machinery that is invoked when one of the aforementioned conditions is to be signaled by an operation. Should an implementation actually *signal* a (floating point) condition using `error`, or should it go the C way [3] and *record* somewhere an *indication* that a condition "happened", for the programmer to check directly?

1.2 Common Lisp Implementations and the `:ieee-floating-point` Feature

The ANSI Common Lisp Standard [1] makes provisions for a Common Lisp implementation to "declare" that it *purports to conform* to the requirements of the *IEEE Standard for Binary Floating-Point Arithmetic* (no reference given). There are a few problems with this statement[3].

The presence of the `:ieee-floating-point` feature in a Common Lisp implementation is a (very) partial indication that *some* support for the IEEE-754 is available. Table 1 shows a summary of the current state of compliance for a number of implementations[4] with respect to the notions just described.

Infinities and NaNs. Many implementations provide infinities and NaNs, but with obviously different lineages. E.g., the following syntax is used by LW and CCL, which is then declined in various interesting ways:

∞	1F++0, 1D-+0
NaN	1F+-0, 1S-0

but ACL chooses to provide "variables" with read-time syntax.

```
ACL prompt> *infinity-single*
#.*INFINITY-SINGLE*

ACL prompt> (+ 42 *nan-single*)
#.*NAN-SINGLE*
```

While this may be perfectly sensible it has the drawback of not playing so nicely with `*read-eval*`.

The only implementations that allow a programmer to get a handle on a *signaling* NaN are CMUCL and SBCL. There appear to be no easy way to create such a value in the other implementations.

[2]There are, e.g., PYTHON bindings to IEEE-754, but they rely on the underlying C library implementation.
Cfr., `https://www.python.org/dev/peps/pep-0754/`.

[3]A form of "left to the implementation", which, as usual, does not bode well for the programmer.
[4]The table is incomplete because not all implementations were checked and because the notion of "compliance" is rather complicated to assess in this case.

	CMUCL/SBCL	LW	ACL	ABCL	ECL	CCL
Infinities	Y	Y	Y	U	U	Y
Quiet NaNs	Y	Y	Y	U	U	Y
Signaling NaNs	Y	N	N	U	U	U
Rounding	Y	N	N	U	U	N
Exceptions NACF	P	P	P	P	P	P
Exceptions NRI	Y	N	N	U	U	N

Table 1: Common Lisp implementations "compliance" status w.r.t. the IEEE-754. The acronym NACF stands for *Notification by Alteration of Control Flow*, while the acronym NRI stands for *Notification by Recording of Indicators* (cfr., [6–8]); they will be discussed later on. The entries are Yes, No, Unknown, and Partial.

Rounding. Only CMUCL and SBCL allow the setting of the rounding mode by accessing directly the equivalent of the IEEE-754 *floating point environment.* However, the facility – which resembles the C library `fenv.h` setup – is very low level.

1.3 A Proposal

Alas, "just adding" infinities and rounding modes to a Common Lisp implementation may not not be quite sufficient, as their semantics is deeply intertwined with the other parts of the IEEE-754 standard. A better course of action would be to nudge the implementors to comply with the current standards. The definition of a new Common Lisp "arithmetic" specification may be a better way to achieve the goal of providing Common Lisp programmers with a layered set of documented functionalities.

The observation is that by now, the IEEE-754 standards (and to a lesser extent the LIA standards) appear to be quite accepted and common place in many programming language ecosystems. It is the opinion of rhe writer that for the Common Lisp community, "to go with the flow" would be the pragmatic thing to do.

2 GOALS AND ISSUES

The goal of this paper is to urge the various Common Lisp implementations to provide better support for floating point (hence complex) arithmetic, in order to make it possible to directly write a IA library (and other numerical routines) in an easier and and more *correct* way. The point of view is that of a Common Lisp programmer and user. The main source of this proposal are the *Language Independent Arithmetic* (LIA) specifications [6–8], which incorporate IEEE-754/IEC-60559 [11].

2.1 The LIA Specifications and Common Lisp

The LIA specifications are three documents covering the following topics.

LIA Part 1: integer and floating point arithmetic
LIA Part 2: elementary numerical functions
LIA Part 3: complex integer and floating point arithmetic

The LIA specifications take great care not to be overly constraining (they relax a few requirements of IEEE-754/IEEE-754) while being very precise about the behavior of each item they define. They also contain appendices describing suggested bindings for various languages, C/C++ and FORTRAN being prominent, including Common Lisp.

A Common Lisp implementor could, in principle, just read through the LIA specifications and provide all the necessary bits and pieces while building the arithmetic facilities of the language. Yet, it is the writer's opinion that this course of action would still fall a bit short of providing a programmers' computing environment with an "implementation independent" firm ground. The reasons lie in the LIA specifications themselves, as they understandably cannot provide more than a suggestion about how a language binding should cover and look like. There are some issues that a more Common Lisp centric specification would and should clarify: *naming conventions*, *layering and packaging*, *programming environment setup*, *rounding-modes* and, above all, *error handling* and the *floating point environment.* In the following each of these issues will be discussed. Eventually, the result should be an in-depth specification formatted in the style of the Common Lisp standard [1, 2].

2.1.1 Naming Conventions. The LIA specifications suggest a naming convention for its functionalities that reuses much of Common Lisp names. some of the choices are not particularly in line with Common Lisp style. Two examples are the functions `sqrtUp` and `sqrtDwn`, which compute square roots with "up" or "down" rounding modes; Common Lisp style would have avoided the "camel case", given that Common Lisp implementations are uppercasing out-of-the box, while preferring an hyphenated naming.

Another issue with the LIA suggested naming is that it essentially requires an implementation to provide a set of very basic LIA-compliant functions – e.g., `+`, `*`, `1-`, `sin`, etc. – which implies a reworking of an implementation core.

2.1.2 Layering and Packaging. In order to provide a Common Lisp centric LIA specification and adoption path, it would be better to ensure that the new functionality were provided as a library. This means, at a minimum, to provide a package that contained all the "new" names introduced. Given the partition of the LIA specifications, further sub-packaging could be provided.

A first cut proposal would be to have a package named (or nicknamed) `CL-LIA` that exported all the names that are necessary to implement a form of the LIA specifications.

As it will be discussed later on, it will be useful to have a `cl-lia:floating-point-invalid-operation` condition, despite the presence of the standard Common Lisp one.

2.1.3 Programming Environment Setup. The IEEE-754/IEC-60559 and LIA specifications define a number of environment checks that a compiler or a program may check to produce code that complies and/or exploits their semantics. These are akin to the `:ieee-floating-point` feature (which, as we have seen, is only partially informative). At least two sets of "checks", both in functional and "feature" form could be provided by a Common Lisp LIA implementation.

Library Compliance Checks. The LIA specifications require a boolean variable $\texttt{iec60559}_F$ that reports whether or not an implementation complies with the IEEE-754/IEC-60559 implementation of the floating point type F. This is more stringent than the "bulk" statement implied by the `:ieee-floating-point` feature, although LIA1 (cfr, [6] Section 5.2) explicitly states that no exact floating point representation is required. A Common Lisp LIA implementation should define similar constants, boolean functions and, possibly, features.

Layered Library Checks. Another set of variables, boolean functions and features should be made available to indicate the level of compliance with the LIA specifications. A suggestion is to provide the functions (and therefore constants and features) `LIA1-compliance`, `LIA2-compliance`, and `LIA3-compliance`. Finer statements may pertain parts of each specification; one example is the `provides-infinities-p` and `provides-nans-p`. Other such checks can be described for other parts of the Common Lisp LIA implementation, as seen below for *exception handling*. Finally, one important check that could be provided is whether the Common Lisp implementation carries over the LIA semantics to the functions in the `COMMON-LISP` package: the check `is-cl-using-lia` would state that a function like, for example, `cl:sin` implements the LIA semantics w.r.t. infinities and NaNs, rounding modes and exception notifications (see below). Of course, whether or not provide such "history rewriting feature" is up for debate. One argument in favor is that old code would still work without having to be tweaked to use the new functions provided in a `CL-LIA` package.

2.1.4 Rounding Modes. Floating points numbers being approximation of real numbers carry with them notions of *rounding*. The LIA specifications define how rounding modes affect elementary and library operations.

Following in this case the C/C++ example, it could be possible to define a set of constants with the meaning showed in Table 2. The type `rounding-mode` can then be defined as:

Common Lisp constants	Value	LIA meaning
`indeterminate`	-1	
`to-zero`	0	truncate
`to-nearest`	1	nearest
`to-positive-infinity`	2	other
`to-negative-infinity`	3	other
`to-nearest-even`	4	nearesttiestoeven

Table 2: Proposed Common Lisp constants representing rounding modes.

```
(deftype rounding-mode ()
  `(member ,indeterminate
           ,to-zero
           ,to-positive-infinity
           ,to-negative-infinity
           ,to-nearest-even))
```

The rounding mode can then be tracked using a special variable `*rounding-mode*`. A macro `with-rounding-mode` is an obvious extension as well as macros wrapping one expression: `round-upward`, `round-downward`, `round-nearest` etc.

Moreover, the LIA specifications define some functions that guarantee a given rounding result; e.g., there are three versions of $\sqrt{\cdot}$, which compute square roots guaranteeing rounding upward, downward and to nearest. For Common Lisp it will probably be better to provide *four* such "names" with the following, LIA-inspired, suffixes: `sqrt` (no suffix), `sqrt.<`, `sqrt.<>`, and `sqrt.>`. The first version depends on the current rounding mode, the other ones round down, near and up (suffixes `.<`, `.<>` and `.>`).

2.1.5 Error Notification/Handling and the Floating Point "Environment". The LIA specifications must address the differences (and . . . "traditions") that different communities have developed over the years. The exegesis of the LIA specifications seems to point out that the major concern was to disentangle concerns that earlier language specifications (especially the C/C++ ones) addressed in a idiosyncratic way. Within the Common Lisp community, the Condition System - as prefigured and hashed out in [9] – offers all the bells and whitles to implement the programmer's side of error handling, but some issues must be dealt with at the implementation level.

One of the main tangled issues regards the *handling of "errors"*, that is, after what an "error" is agreed upon. This is a notoriously complicated issue which the LIA specifications appear to break down into two parts.

- How errors are *notified.*
- What happens depending on the *notification style.*

The LIA specifications assume that a language "environment" establishes some forms of *notification* machinery. Three major modalities are singled out.

(1) *Notification by recording in indicators* (**NRI** – LIA1, Section 6.2.1).

(2) *Notification by alteration of control flow* (**NACF** – LIA1, Section 6.2.2).
(3) *Notification by termination with message* (**NTM** – LIA1, Section 6.2.3).

The LIA1 specification, Annex D, proposes that Common Lisp defined the arithmetic exception handling using the Condition System, i.e., using the NACF notification approach. However, SBCL/CMUCL provide – de facto – a NRI setup modeled on the C NRI interface provided by `<fenv.h>`. It would be better to accommodate **both** alternatives NRI and NACF, and make them available to the programmer for finer control.

LIA1 provides an example about how FORTRAN may provide some compiler directives to choose between NRI and NTM (cfr., LIA1, Annex E).

```
!LIA$ NOTIFICATION=RECORDING
!LIA$ NOTIFICATION=TERMINATE
```

In order to select and introspect what kind of exception handling regime is in place in a given computation[5] an appropriate Common Lisp API will have to be defined.
To complete the discussion, we must consider the *floating point environment*, *conditions and continuation values*, and *underflow/overflow*.

Floating Point Environment. Having control over the kind of notification style is nice, but it requires a better handling of the *floating point environment*, which C handles through `<fenv.h>`, and that CMUCL/SBCL manipulate using a few functions and what looks like is an a-list.

The floating point environment is used as a kitchen sink to keep track of rounding modes, exceptional situation notifications and other information. A unified item representing these concerns still seems the best way to give access to them in a dynamic way.

Conditions and "Continuation" Values. The operations that operate on the "borderline" case in LIA (e.g., operations on *NaN*s, or that generate *underflow*s and *overflow*s, are specified alongside a *continuation value.* This is most important for the NRI notification style, where an operation "continues", while recording the indication of an exceptional situation. To facilitate the implementation of a LIA-compliant package for Common Lisp, it would be useful to mirror the `arithmetic-error` sub-hierarchy and to equip the classes with a `continuation-value` slot (alongside appropriate initargs and readers).

3 CONSIDERATIONS FOR A NEW COMMON LISP ARITHMETIC SPECIFICATION

Having discussed some of the issues about providing support for the LIA specification in Common Lisp, we here offer a detailed opening bid in a hoped-for public discussion on the creation of a Common Lisp binding, or otherwise integrating its ideas into the language.

[5]The LIA specifications make no mention of any *threading model*; however, it is assumed that an implementation can make all the dynamical behavior of numerical computations "thread safe".

A full-blown document containing the full description of each item, in a style reminiscent of the Common Lisp[] ANSI Standard [1], is in the works. The LIA specification describes each item and, especially, operation, in a terse and abstract way, which requires quite a bit of effort to map onto a typical Common Lisp description, especially for functions results. The full blown description is intended to be read by a Common Lisp programmer.

Package. All the names that will be introduced or shadowed from the `"COMMON-LISP"` package will be exported from a new package. The proposed nickname is `"CL-LIA-MATH"`[6].

Environmental Features and Switches. An implementation will state what parts of the specification will be available. The following (semi-hierarchical) set of environmental queries will be available as *functions*, *special variables*, and/or *features.*

```
lia-subset-available
  lia1-subset-available
  lia2-subset-available
  lia3-subset-available
lia-compliance
  lia1-compliance
    provides-infinities
    provides-nans
    provides-rounding-modes
    provides-floating-point-environment
    provides-nacf
    provides-nri
    provides-ntm
  lia2-compliance
    cl-package-uses-lia
  lia3-compliance
```

The above list represents Boolean functions. The `provides-...` functions return *true* when the specific functionality is *fully provided.* In the above example `lia1-compliance` returns *true* when *all* the `provides-...` functions return *true.*

Note that, while the list of introspective facilities listed covers most of the dimensions in LIA compliance, certain combinations are ruled out by the detailed specification and by the way it is presented in the standards. E.g., it is not very sensible to have an implementation for which `provides-floating-point-environment` returned *false*, while `provides-nacf` returned *true.*

Infinities, NaNs and Rounding. An implementation of the specification will offer all the values pertaining *infinities*, *NaNs* and *rounding.* In particular, a fully LIA compliant will provide the environment introspection functions and variable mandated by LIA1 and "typed" constants like `double-float-positive-infinity` and `infD`. Moreover, the full specification will clarify the behavior of each function and operation when presented with *NaNs*, both *quiet* and *signaling*, especially regarding the interplay with the *error notification style* (see below).

[6]Or `"CL.MATH"`, should a more ambitious naming be adopted.

The rounding versions of all the LIA mandated operations will be marked by the postfixes `.<`, `.<>`, and `.>`, signifying rounding towards negative infinity, nearest, and positive infinity. The macro `rounding` has effect on the "unqualified" versions of the arithmetic operations. E.g.

(`+.< pi pi`) $\Rightarrow$ 2π rounded toward $-\infty$.
(`+.> pi pi`) $\Rightarrow$ 2π rounded toward ∞.

..., while using the `rounding` macro

```
(rounding :positive-infinity (+ pi pi))
```
$\Rightarrow$ 2π rounded toward ∞.

..., but

```
(rounding :positive-infinity (+.< pi pi))
```
$\Rightarrow$ 2π rounded toward $-\infty$.

I.e., the `rounding` macro establishes a dynamic environment with a specific rounding mode set up, which can be ignored by the "hard rounding" versions of the operations in the body.

Floating Point Environment and Error Handling. An implementation of the specification shall assume the presence of an *opaque* data type called `floating-point-environment`. The access functions for this data structure are patterned after the C/C++ standards in order to offer familiarity and, possibly, ease of implementation.

An implementation of the specification will always offer *both* NACF and NRI notification styles, with full control offered to the programmer about when and where they turn on and off each style. The NTM notification style will used only for catastrophic events, which will be documented accordingly.

The type `arithmetic-notification-style` can be defined as:

```
(deftype arithmetic-notification-style ()
  '(member :recording   ; I.e., NRI.
           :error       ; I.e., NACF.
           :terminating ; I.e., NTM.
           ))
```

The main functions and macros that allow full control of the notification style and *continuation values* possibly returned by an operation are the following.

```
current-notification-style
set-notification-style
with-notification-style
trap-math
```

The `trap-math` macro is intended as a wrapper around the Common Lisp error handling machinery (`handler-case`, `unwind-protect`, etc...) that automated some of the setup and teardown operations on the floating point environment, alongside the handling of continuation values. A possible syntax is the following

```
(trap-math (<options>)
    <expr>
    <handler>* )
```

The *`options`* parameter is a list that may contain the keywords `:notify-by`, `:before`, and `:after`. The `:notify-by` is the *notification style* defaulting to `:error`, `:before` and `:after` instead demarcate lists of *actions* that intend to simplify the setting up and the teardown of indicators in the floating point environment: `:save` saves the current floating point environment, `:clear` creates a fresh floating point environment with no indicators recorded, and `:merge` (in an `:after` position) merges the current floating point environment with the possibly saved one. Of course, a different syntax is possible, and it is unclear to the writer which would be the best; consider the following alternative.

```
(trap-math (&key notify-by before after)
    <expr>
    <handler>* )
```

The *`handler`* is a simplified list that has the following syntax.

```
(<aec> (&optional <varname>)
     &rest <actions>)
```

where *`aec`* is an `arithmetic-condition` carrying a continuation value, *`varname`* is a symbol that can be bound to the condition instance and *`actions`* is a list that may contain the following items.

- `:default` – the behavior is the standard one for *`aec`*.
- `:clear` – when combined with the `:continue` forms and the complex `:error` form, it clears the indicator corresponding to `aec` from the floating point environment.
- `:raise` – re-signals the `aec`
- `(:raise` *`c`* `&rest` *`args`*`)` – signals a new contition *c*.
- `:continue`, `(:continue` *`expr`*`)` – continues the computation by yielding the standard continuation value of the result of evaluating *`expr`*; the `:continue` actions can be rendered by means of `cl:use-value` and/or `cl:continue` restarts.

An example of the use of `trap-math` is the following, which is also a rendering of [6] Appendix A.6.

```
(trap-math (:before :save :clear
            :after :merge)
    (fast-solution input)
    (cl:floating-point-overflow ()
       :clear
       (:continue (reliable-solution input))))
```

or, with a different syntax

```
(trap-math (:before (:save :clear)
            :after :merge)
    (fast-solution input)
    (cl:floating-point-overflow ()
       :clear
       (:continue (reliable-solution input))))
```

3.1 Providing the Specification – A Descriptive Example

Eventually, the considerations put forth in this paper should be crystallized in a specification that clarified all the many thorny issues that will crop up when considering as

Function `=`, `/=`

Syntax:
`=` a, b $\Rightarrow$*boolean*
`=` a `&rest` bs $\Rightarrow$*boolean*
`/=` a, b $\Rightarrow$*boolean*
`/=` a `&rest` bs $\Rightarrow$*boolean*

Arguments and Values:
a b – Numbers.
bs – A list of numbers.
boolean – a *generalized boolean.*

Description:
The dyadic version of `=` (and `/=`) performs an arithmetic equality (inequality) test between a and b. The monadic and n-adic versions are built upon the dyadic one as per the regular Common Lisp description in [2].
It is assumed that a and b are converted (as per the *contagion rules* of Common Lisp) to be of the same type. Therefore the following cases can be be considered as per the LIA specifications.

(a) If a and b are either finite integers, finite floating point numbers, or finite complex numbers then the result is *true* (respectively, *false*) if the two numbers are equal (respectively, different) in the mathematical sense. In the LIA spec this is the result of $eq_T a, b \equiv a = b$ or $neq_T a, b \equiv a \neq b$ for an appropriate T. This is the standard Common Lisp case.
(b) If a and b are *infinities* then `=` returns *true* (respectively *false*) if they are both positive or both negative; otherwise it returns *false* (respectively *true*).
(c) If either a or b is a *quiet NaN*, and, respectively, b and a is not a *signaling NaN*, then the result is *false*.
(d) Complex numbers are checked recursively on the real and imaginary parts.

Exceptional Situations:
If either a or b is a *signaling NaN*, then, under the notification NACF regime, the indicator `:invalid` is recorded and the `floating-point-invalid-operation` is signaled (with *continuation value* `NIL` recorded); otherwise, under the NRI notification regime, the indicator `invalid` is recorded and `NIL` (*false*) is returned as *continuation value.*
For complex numbers, the recording and signaling operations (under NRI and NACF) happens if the condition above applied to either of the real or the imaginary parts of a and b.

Figure 1: An example entry that should appear in the full specification for the Common Lisp LIA-compliance documentation.

many details as possible. The goal will be to provide a specification *á la* Common Lisp HyperSpec [2], where each item (function, variable, class, etc.) has a mostly self-contained description with, by now, a conventional structure. This is different from the presentation style adopted by the LIA specifications, which heavily rely on quite formal yet "generic" description of each operation's behavior.

The most important aspects a Common Lisp LIA specification will be to describe, for each function (or other item), the following behaviors.

(a) The *corner cases*: infinities and NaNs.
(b) The interplay between the *notification style*, the handling of errors, the floating point environment and the *continuation values* that are specified according to the LIA documents.

As an example of what an entry in the envisioned full Common Lisp LIA specification would look like, Figure 1 shows a description of the (dyadic) `=` and `/=` functions. Hopefully, all details and corner cases listed above have been taken into consideration. The reader can compare the *equality* specification in the LIA1 document with the one in Figure 1.

4 CONCLUSIONS AND FINAL DISCALIMERS

In order to write a fully functional (according to the available literature and proposed standards) IA library in Common Lisp, a programmer needs a finer control over the *floating point* environment and access to functionalities such as rounding modes.

This paper puts forth a proposal to complete a Common Lisp HyperSpec styled, LIA-based specification that would provide a more accessible documentation for a programmer and a clear guideline about how certain functionalities should be provided by an implementation.

Given an implementation of the proposed "New Arithmetic Specification" a programmer could at least start to write a proper IA library. As an example, the add method would look like the following.

```
(defmethod add ((i1 []) (i2 []))
  ([] (+.< ([]-low i1) ([]-low i2))
      (+.> ([]-high i1) ([]-high i2))))
```

where +.< and +.> are the addition operations on floating point numbers that round, respectively, *downward* and *upward*.

Again, the writer wants to insist and repeat that the present paper is a *leaflet* that intends to present the Common Lisp community a project which, in his modest opinion, should be completed after careful debate and careful consideration of all the details within the Common Lisp community.

A full blown specification covering LIA-1, LIA-2 and LIA-3 will run close to two hundred pages if written and formatted according to the [2] style (a worthy goal in itself). Many of the examples contained in this paper are *suggestions* about how they could look. Agreement withing the Common Lisp community will help settle down several issues this paper puts forth.

What this paper wants to point out though, is that many researchers and practitioners did lay down a sensible set of standards, the LIA standards, which *did* take into account Common Lisp. Following them appears to be one good way to ensure that Common Lisp will keep its place among the most important language ecosystems around, and welcome programmers from other communities by offering them a familiar playpen and more.

REFERENCES

[1] ANSICL. Information Technology — Programming Language -– Common Lisp. ANSI Standard X3.226-1994, 1994.

[2] ANSIHyperSpec. The Common Lisp Hyperspec. published online at http://www.lispworks.com/documentation/HyperSpec/Front/index.htm, 1996.

[3] C18. Programming Languages – C. ISO/IEC Standard 9899-2018, 2018.

[4] T. Hickey, Q. Ju, and M. H. van Emden. Interval Arithmetic: From Principles to Implementation. *Journal of the ACM*, 48(5): 1038–1068, 2001.

[5] Ulrich W. Kulisch. Complete Interval Arithmetic and Its Implementation on the Computer. In A. Cuyt, W. Krämer, W. Luther, and P. Markstein, editors, *Numerical Validation in Current Hardware Architectures*, volume 5492 of *Lecture Notes in Computer Science*. Springer, 2009.

[6] LIA1. Information technology – Language independent arithmetic – Part 1: Integer and floating point arithmetic (LIA-1), Second edition. ISO/IEC Standard 10967-1, July 2012.

[7] LIA2. Information technology – Language independent arithmetic – Part 2: Elementary numerical functions (LIA-2), First edition. ISO/IEC Standard 10967-2, August 2001.

[8] LIA3. Information technology – Language independent arithmetic – Part 3: Complex integer and floating point arithmetic and complex elementary numerical functions (LIA-3), First edition draft. ISO/IEC Standard 10967-3, June 2004.

[9] Kent Pitman. Exceptional Situations in Lisp. In *Proceedings of First European Conference on the Practical Application of Lisp (EUROPAL'90)*, March 1990.

[10] Nathalie Revol. Introduction to the IEEE 1788-2015 Standard for Interval Arithmetic. In Alessandro Abate and Sylvie Boldo, editors, *10th International Workshop on Numerical Software Verification (NSV) 2017; Workshop of CAV 2017*, Lecture Notes in Computer Science. Springer, 2017.

[11] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. IEEE Standard 754$^{\text{tm}}$-2008, 2008.

A portable, annotation-based, visual stepper for Common Lisp

João Távora
Ravenpack
joaotavora@gmail.com

ABSTRACT

Many programming systems feature a stepping debugger, a tool that lets users execute code, section by section, in steps of their own choosing. Despite many attempts throughout the decades, the Common Lisp language is still lacking in this regard. We propose and describe the workings of a new, portable, visual stepping facility for Common Lisp, realized as an extension to SLY, a cross-implementation Common Lisp IDE for the Emacs editor. This facility is realized as an increment to an existing source code annotation system known as "stickers", whose working principles we also describe in this work. As part of the solution arrived at for the main objective, we also present two reusable software components: (1) a simple, near portable technique for constructing a source-tracking Common Lisp expression reader in terms of a preexisting compliant expression reader and (2) a technique to carry over source-tracking information to the expansion of macro expressions.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools**.

KEYWORDS

Debugging, Stepping, Common Lisp

ACM Reference Format:
João Távora. 2020. A portable, annotation-based, visual stepper for Common Lisp. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.3742759

1 INTRODUCTION

1.1 What is stepping

A program stepper lets users execute code, section by section, in steps of their own choosing. A number of hidden control points are inserted along the execution paths of the program. At each point, the system may interrupt the program and wait for user instructions before and/or after executing the next section. Steppers often fall within the category of *program execution monitors*, along with profilers, tracers and other debugging tools. Stepping is the one of the most popular forms of debugging since it allows users to study the evolution of the state of a program by direct inspection, as opposed to combining conjecture and experimentation. A stepper gives users full control over the speed of the actual program and often even allows manipulations of that state. This frequently presents a debugging advantage over guessing a programs' state or monitoring its outputs.

Many programming systems, indeed all of the most popular programming systems provide stepper tools as a part of a debugging tool-chain, i.e. a set of programs often developed and distributed alongside language compilers and interpreters. In languages such as C, the prevalence of a stepping system (such as the popular `gdb` program) is so strong that it becomes synonymous with the term "debugger". This sometimes leads to users migrating to the Lisp family of languages being surprised to find debugging systems that aren't about stepping at all.

Stepper systems usually function through text-based interfaces that are capable of displaying the source code of the relevant code sections. Nevertheless, many users of stepper tools prefer to use them through the more sophisticated user interfaces of editor programs and integrated development environments (IDEs). We shall call these tools *visual* steppers[1]: special-purpose programs running inside the IDE communicate with the stepper tool by means of a protocol and visually annotate the source text of a program with the results of the stepping session. The expression about to be executed may e.g. be marked with a red dot or highlighted in a special color. Furthermore, a modern user's expectations of a stepper system might include the ability to add break points; to "step over" an expression; to "step into" a call to a function defined elsewhere in the source code; to "step out of" the current call; to "continue" to a certain point, and to inspect the values of local and global variables by mouse-clicking on their manifestations in the source code.

1.2 Common Lisp stepping

Common Lisp programmers are so detached from the practice of stepping that some will simply declare that's proof that they don't need one at all. There is a hint of truth to this declaration, as Common Lisp systems have traditionally directed efforts to other types of debugging facilities such as powerful interactive program restarts and function traces. At any rate, it seems undeniable that, stepper or no stepper, Common Lisp programs will be debugged.

Nevertheless, an abundance of Common Lisp steppers have been proposed throughout the decades, even if few have actually enjoyed any adherence. Anecdotal evidence suggests potential newcomers shy away from Common Lisp because of missing stepper functionality. Indeed, this reality appears not to be lost even on the authors of the Common Lisp specification, which have included in their work a provision for a special implementation-defined `CL:STEP` macro.

Some Lisp implementations can and do implement stepping to various levels of ability. SBCL combines its implementation of `CL:STEP` with the restarts system to provide a text-based stepper,

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27–28 2020, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.3742759

[1]After the nomenclature used in [9]

while the LispWorks[2] and Allegro CL systems have sophisticated graphical stepping dialogues with views to the source code and program state. Unfortunately, these systems are unavailable to whomever wishes to try portable programs across different Common Lisp implementations. Even if one switches between two implementations that do have steppers, the difference in capability and user interface is often enough to discourage the use of either system.

The problem of interface inconsistency across implementations is not exclusive to the stepper feature: other debugging tools such as the inspector, the debugger or the REPL suffer from it. Thus, many Common Lisp programmers will use a generic text editor such as GNU Emacs[1] together with the SLIME[5] extension. This combination forms a capable, implementation-agnostic Common Lisp IDE that suppresses the problems described above and provides a consistent user interface to many debugging features.

Regrettably, even though Emacs provides visual stepper interfaces for many programming languages, SLIME doesn't provide a portable stepper interface. We believe this to be due to the fact that the technical challenges to be surmounted are greater than for other debugging tools. Among other problems, the portability mandate of SLIME implies it is ultimately only allowed to invoke functionality present in all the Common Lisp systems it connects to. This mandate implies that even if all implementations where to implement some form of `CL:STEP`, that alone wouldn't be powerful enough to, say, communicate source location information to and from Emacs, annotating the program source.

1.3 A portable, visual stepper for Common Lisp

We should note that none of the obstacles listed above are conceptual in nature, so there must still be hope for a portable, visual stepper for Common Lisp. The SLIME/Emacs combination makes it a particularly attractive target for such a tool, given its relevance among Common Lisp users and the flexibility of Emacs's Elisp language.

In fact, the SLY Common Lisp IDE[11], a derivative program of SLIME, has redesigned some of the underpinnings of its predecessor to make the development of new extensions easier. We shall describe how "stickers", a feature that SLY has recently acquired, lets users manually instrument selected Common Lisp forms whose results they are interested in. Stickers are already a "poor man's stepper", in the sense that they have some fundamental semantics of stepping but still encumber the user with work than could be performed automatically. To fill this gap, we shall explore methods of combining stickers with automatic code analysis. We shall then be able to present an innovative stepper tool for the Common Lisp language based on the SLY extension to the Emacs editor, hereafter designated the *SLY/Emacs stepper*.

2 RELATED WORK

"Stepping is an old idea." So go the opening words of this section's namesake in the article "Annotation-Based Program Stepping", written by MIT's G. Parker in 1987[9]. In this article, the author surveys the efforts of the 1970 decade to develop various kinds of stepping tools in the MACLISP environment. Likewise, we shall proceed to evaluate a small sample of Common Lisp stepper systems, focusing on the ones that are portable, visual and intersect our methodology.

In the remainder of Parker's paper[9], a visual stepper system, ViSiStep, is described. Its distinguishing characteristics are the integration with the MACLISP system and an *annotation-based* approach, a key difference to other *evaluator-based* techniques.

Annotation-based program stepping is a form of code instrumentation. It comprises the addition of statements to the program shortly before its compilation. By way of a so-called *wrapper* macro, these statements are added before and after each section to be stepped. The addition is transient and invisible: it does not modify the source file. Furthermore, the program cannot itself discern the presence or absence of these additional statements, so its outputs are unmodified. Parker[9] points out that this approach can work with an unmodified evaluator, since the compiler is simply given more instructions to compile. He also asserts this approach to be more efficient, more portable and more selective, the latter meaning that it allows the user to select only those sections of the code that he wishes to step through. However, the author acknowledges the annotation-based stepper's difficulty in handling some macro's non-evaluated syntactic elements (such as the arguments to `COND`), and how it must rely on "code-walking knowledge"[9, I-4.8] to determine the forms where the wrapping may take place.

By contrast, an evaluator-based or *interpretative* approach involves writing a Lisp evaluator or instrumenting the Lisp interpreter. The evaluator itself then becomes responsible for issuing the stepper-enabling statements before and after each evaluation. "UniCStep - a Visual Stepper for COMMON LISP", written by I. Haulsen and A. Sodan in 1989[6], presents such a stepper system, written for an early version of the GNU Emacs editor. The authors reply directly to Parker's contention's of the superiority of the annotation-based approach, asserting the evaluator-based approach to be more comfortable and flexible because the user does not have to specify in advance what to step and where to stop. They assent to one technical disadvantage such as the fact that evaluator-based alternative need a loader to be emulated and more sophisticated ways of remembering the source of the loaded code.

We should note a more recent attempt at a Common Lisp stepper, such as Pascal Bourguignon's work[4]. This consists of a portable, evaluator-based approach that replicates the implementation-defined behavior of Common Lisp's `STEP` macro. Bourguignon's stepper provides a replacement package for the standard `COMMON-LISP` package, through which the user must re-load the code whose forms can then be passed to the `STEP` macro. This stepper has no editor or source-tracking integration as of yet, but it seems to have been in the plans at some point during its development.

Finally, a word should be spared for Emacs's `edebug.el` authored ca. 1988 by Daniel LaLiberte[7]. Edebug is designed to step through Emacs Lisp programs within Emacs itself. Since it executes inside the Lisp machine that is also the editor, the source-tracking integration is very good. `edebug.el` is an annotation-based stepper that deals with the problem of a macro's un-evaluated syntactic elements by skipping macros it knows nothing about. The macros whose expansions the user is interested in can be annotated separately with `edebug.el`-specific declarations.

3 METHODS

Our proposed portable stepper system for SLY/Emacs can be broken down into three main components:

(1) A non-intrusive source code annotation system, called "stickers". This system primarily allows "interesting" Common Lisp forms to be designated by the user. On compilation, the annotated code is transmitted to the Common Lisp compiler, and executes equivalently to non-annotated code;
(2) A *source-tracking reader*, i.e. a process by which a stream of characters containing source code forms is read into a symbolic expression representing the form, whilst recording the positions of the start and end characters of each sub-form;
(3) A *specialized code walker*, a process by which an arbitrary Common Lisp form can be traversed at compilation-time to determine the syntactic value of each of its sub-forms as processed by the compiler after the macro-expanding phase.

It should become apparent that the application of 3. to the results of 2. relieve users in 1. of the need to manually designate forms of interest. They sole job becomes requesting the annotated compilation of arbitrary lengths of source code, leaving the stepper system to automatically annotate all possible forms of interest.

The following subsections detail the workings of each component in this arrangement.

3.1 Stickers

Stickers are a form of code annotation in use in SLY/Emacs. Initially conceived as an alternative to the PRINT or FORMAT statements introduced by users when debugging programs, this system lets users visually mark individual symbolic or compound forms in whose future execution they're interested in. Crucially, the visual markings exist only in Emacs's memory for as long as the user wishes. They aren't saved in the source code itself. When the compilation of the containing top-level form happens from within SLY/Emacs, SLY will collect those visual markings, enumerate them, and emit for compilation a modified version of the form. This process is called *arming the stickers.*

The modified version is functionally equivalent to the original in the sense that i.e. user programs have no way to detect which one they are executing. The modifications consist of multiple invocations of a special RECORD wrapper macro[2], whose definition is presented below in much simplified fashion:

```
(defmacro record (id &body body)
  `(let ((%retval :exited-non-locally)
         (%condition)
         (%sticker (find-sticker ,id)))
     (handler-bind ((condition (lambda (c)
                                 (setq %condition c))))
       (before-sticker %sticker)
       (unwind-protect
            (values-list
             (setq %retval (multiple-value-list
                            (progn ,@body))))
         (after-sticker %sticker %retval %condition)))))
```

[2]This macro is very similar to the WRAP macro presented in [9], which the exception that for some technical reason that other version was realized as a special form.

E.g., if the user marks the forms (foo (bar)) and (bar) inside the following expression:

```
(let ((baz 42)) (+ (foo (bar)) baz))
```

The sticker system will collect the two markings, label them with the numbers 1 and 2 and emit the following equivalent form for compilation:

```
(let ((baz 42)) (+ (record 1 (foo (record 2 (bar)))) baz))
```

As can be seen, every time the expression above is executed, expansion of the wrapper macro causes the functions before-sticker and after-sticker to be called with the appropriate %sticker object. Depending on the user's preference, these functions may decide to stop execution of the program (by way of invoke-debugger), or to simply record the fact that the sticker was traversed. The list of recordings can later be retrieved and replayed later inside SLY/Emacs.

In our simple example, the benevolent user placed stickers on two expressions that are indeed executed, i.e. they exist in places of evaluation as defined by the syntax of the (let ...) special form and the (+ ...) function form. If a sticker had instead been placed on the expressions ((baz 42)) or + – two examples of non-evaluated forms – that would have created a difficulty, since the code would become syntactically invalid and fail compilation. In a worse situation, the code would still be syntactically valid but semantically absurd.

SLY/Emacs's sticker system doesn't have a way to reject these individual stickers, so it proceeds heuristically: it rejects the compilation of the whole top-level form if it determines that arming stickers results in an *increase* to the number of compilation warnings. In that case, the original form is compiled but the stickers *fail to arm.* This strategy works for a vast majority of cases, but it doesn't seem impossible to construct a pathological case where the principle of functional equivalence stated above is violated.

3.2 Source-tracking form reader

At its simplest definition, a source-tracking form reader is a variation of the Common Lisp CL:READ function that invokes a hook every time a sub-form is read, and proceeds to pass to this hook a measure of the character distance traveled so far to read it. This enables programs that need both the usual results of CL:READ and a table of form-to-source-code pairings.

Since the Common Lisp standard doesn't specify any form of source-tracking reader, some preexisting alternatives were evaluated:

(1) Eclector[8], a self-described "portable Common Lisp reader that is highly customizable and can return concrete syntax trees", is a full realization of a Common Lisp code reader that doesn't rely on any preexisting reader. Its distinguishing characteristic are "concrete syntax trees" that aren't represented by CONS cells, rather by CLOS objects that mimic the properties of such cells while also keeping "concrete" source file information;
(2) hu.DWIM.reader[3], by Pascal Bourguignon, is another full realization of a compliant, portable and programmable Common Lisp reader. Though not a source-tracking reader *per se*,

it could be used in conjunction with a mechanism to track character counts in streams;

Any alternative could have been used for our purposes (the second one with minor changes). However, since we wish to minimize our program's dependency chain we also searched for simpler alternatives.

We reasoned that our stepper tool already expects a compliant Common Lisp implementation. Therefore it may, by definition, also expect a compliant `CL:READ`. So we set out to design a portable source-tracking reader that completely reuses the implementation's reader instead of replacing its implementation entirely. To achieve this, we settled on an arrangement of two separate techniques:

- A *character counting stream*. This wraps the input `CL:STREAM` object (from which we intend to `CL:READ` from) inside a so-called "gray stream" object. Such objects are not in the standard but are still widely available and used extensions to the Common Lisp standard[3]. Gray streams allow the individual character reading operations to be intercepted and controlled by the user. In this case, our character counting stream is equivalent to the wrapped input stream except for the fact that it keeps count of the number of characters read so far.
- A *substitute read-table*, achieved by rebinding the variable `*READTABLE*`. This table *shadows* each of the entries of the current read-table (using `GET/SET-MACRO-CHARACTER`) with a function that fully controls the influence of each character over the returned symbolic expression. By setting up this function in a particular manner, the resulting table remains functionally indistinguishable from the original one, while gaining the ability to invoke a hook that records source positions.

The inter-operation of the two techniques is summarized by the `READ-TRACKING-SOURCE` function:

```
(defun read-tracking-source
    (&optional (stream *standard-input*)
       (eof-error-p t) eof-value
       recursive-p (observer #'ignore))
  (let* ((ccs (char-counting-stream stream))
         (*readtable*
           (substitution-table
            *readtable*
            (lambda (shadowed-entry)
              (let (;; correct for the fact that
                    ;; one character of the form
                    ;; has already been read.
                    (start (1- (char-count ccs)))
                    (results (multiple-value-list
                              (funcall shadowed-entry)))
                    (end (char-count ccs)))
                (multiple-value-prog1 (apply #'values results)
                  (when results
                    (funcall observer (car results)
                             start end))))))))
    (read ccs eof-error-p eof-value recursive-p)))
```

[3]They are already heavily used in SLY/Emacs, for example

This mechanism provides a simple, reliable[4] and portable[5] way to read source-code. We can demonstrate its use on the simple `(let ...)` form already presented above:

```
(with-input-from-string
    (s "(let ((baz 42)) (+ (foo (bar)) baz))")
  (read-tracking-source
   s t nil nil
   (lambda (form start end)
     (format t "~&(~2,a ~2,a) <=> ~a"
             start end form))))
```

This expression returns the intended symbolic expression representing the form, `(LET ((BAZ 42)) (+ (FOO (BAR)) BAZ))`, while also producing the desired table of source positions:

```
(1  4 ) <=> LET
(7  10) <=> BAZ
(11 13) <=> 42
(6  14) <=> (BAZ 42)
(5  15) <=> ((BAZ 42))
(17 18) <=> +
(20 23) <=> FOO
(25 28) <=> BAR
(24 29) <=> (BAR)
(19 30) <=> (FOO (BAR))
(31 34) <=> BAZ
(16 35) <=> (+ (FOO (BAR)) BAZ)
(0  36) <=> (LET ((BAZ 42))
              (+ (FOO (BAR)) BAZ))
```

3.3 Specialized code walker

Equipped with a correspondence between forms and source code, the Common Lisp side of our stepper system nears a state where it may inform SLY/Emacs of where to place sticker annotations. A final obstacle remains: as we have seen in 3.1, only a subset of these forms may be annotated with the `RECORD` macro, i.e. we are only interested in the ones that are executable.

Clearly, we need an agent that understands the semantics of each Common Lisp special form[6], determines sub-expressions of interest in our source-mapped tree and discards all the others. As was already noted in [9], this seemingly simple task is severely complicated by macros and by the specific constraints of a stepper system.

3.3.1 Mnesic macroexpansion. To reach a state where nothing but special and function forms exist, a macroexpander must remove macro calls by expanding them. However, in doing so, our program *must also remember* whence each macro's expansion came, specifically the source position of the form in its pre-expansion state. This behavior is what we refer to as *mnesic macroexpansion*, the opposite of *amnesic macroexpansion*.

Take the form:

[4]This was tested in SBCL, Allegro CL, CCL and ECL. There are differences to the way that some implementations will construct the standard read-table (for example, SBCL and Allegro CL represent "constituent" characters differently) and `SUBSTITUTION-TABLE` has special provisions for that. The same technique should in theory work with non-standard read-tables, but this has not been tested.
[5]As noted, except for the use of gray streams.
[6]The human programmer *is* such an agent, but he is precisely the one we are trying to relieve of these tasks.

```
(LET ((BAZ 42))
   (COND ((PLUSP BAZ) (FOO)) (T (BAR))))
```

It may be expanded to something like[7]:

```
(LET ((BAZ 42))
  (IF (PLUSP BAZ)
      (FOO)
      (THE T (BAR))))
```

By this point, the system may finally come to the realization that only the forms 42, baz, (plusp baz), (foo), (bar), (if ...) and (let ...) are in positions of evaluation. Regrettably, it may now have lost track of *where* each form lives in the source.

To recover this information, it is not enough to naively consult the hash-table produced in 3.2, since some forms didn't exist in our original source-tracked version. Even if they did, the macroexpanding facilities are not generally obliged to return the same CONS objects for the forms, regardless of whether they expand them or not.

In the Eclector library discussed in section 3.2, a RECONSTRUCT function attempts to solve this very problem by correlating the fully macro-expanded "raw" tree with the original "concrete syntax tree", returning a mirroring of the former that keeps as much from the latter as possible. After some experimentation with this approach, we noticed it missed many forms in executable positions and so decided it wasn't producing the results we had hoped for. Moreover, the quality of results tended to vary across implementations, possibly due to the aforementioned CONS-related problems.

To overcome this obstacle we need a different approach. Instead of trying to recover from a fully macroexpanded tree, we must hook into macroexpansion as soon as it happens. This shall allow us to lose as little source-tracking information as possible. Thus, we conclude that a programmable, portable code-walker is necessary. Such a system shall let us execute a hook at each macroexpansion step shortly before and shortly after each expansion.

After surveying open-source alternatives for code-walkers, we settled on a program called AGNOSTIC-LIZARD[10], which fits exactly these requirements[8]. AGNOSTIC-LIZARD:WALK-FORM, its main primitive, produces the desired full macroexpansion and can be given a set of callback functions as hooks.

Here's the snippet that illustrates our use of this walker:

```
(defun mnesic-macroexpand-all (form subform-positions)
  (let (stack (expansion-positions (make-hash-table)))
    (values
     (agnostic-lizard:walk-form
      form nil
      :on-every-form-pre
      (lambda (subform env)
        (push (list
               :from subform
              :at (gethash subform subform-positions))
         stack)
        subform)
      :on-every-form
      (lambda (expansion env)
        (push (pop stack)
              (gethash expansion expansion-positions))
        expansion))
     expansion-positions)))
```

As we can see, our MNESIC-MACROEXPAND-ALL function uses a stack to take advantage of the manner in which macroexpansion traverses the tree: there may be more than one consecutive call to each of the callbacks :ON-EVERY-FORM-PRE and :ON-EVERY-FORM. However, in the end, the calls to one and the other perfectly mirror each other. Each element of the stack holds the result looked up in the SUBFORM-POSITIONS hash-table for non-expanded forms. That source information is later saved on the output hash-table EXPANSION-POSITIONS, whose keys are of expanded forms.

If we give this function the form:

```
(COND ((FOO) (BAR)) ((BAZ) (QUUX)) (T 42))
```

We may obtain something[9] like:

```
(IF (FOO) (PROGN (BAR))
    (IF (BAZ) (PROGN (QUUX))
        (IF T (PROGN 42) NIL)))
```

The resulting hash-table EXPANSION-POSITIONS, returned as a second value, has these mappings:

```
(IF (FOO) ..) => (:from (COND ((FOO)..)) :at (0 . 42))
42            => (:from 42               :at (38 . 40))
(BAR)         => (:from (BAR)            :at (13 . 18))
(PROGN 42)     => (:from (PROGN 42)       :at NIL)
(PROGN (BAR)) => (:from (PROGN (BAR))    :at NIL)
T             => (:from T                :at (36 . 37))
(IF T ...)     => (:from (COND (T 42))    :at NIL)
(QUUX)        => (:from (QUUX)           :at (27 . 33))
NIL            => (:from NIL              :at NIL)
(IF (BAZ) ..) => (:from (COND ((BAZ) ..)):at NIL)
(BAZ)         => (:from (BAZ)           :at (21 . 26)))
(FOO)         => (:from (FOO)            :at (7 . 12)))
(PROGN (QUUX))=> (:from (PROGN (QUUX))   :at NIL)
```

As can be seen, numerous new forms appeared in the expansion, but MNESIC-MACROEXPAND-ALL succeeded in keeping the source information information for all of the relevant ones.

3.3.2 *Annotating interesting forms and putting it all together.*

A final piece of the puzzle is needed. A function named FORMS-OF-INTEREST is to be given the fully macroexpanded tree and along with the source-tracking information for that tree. Its task is to traverse the tree while looking for each of the 25 Common Lisp special compound forms[10], considering the evaluation rules of each. Unknown forms are assumed to be function calls, whose evaluation rules are equally well known. For each sub-expression in a position of execution, the source location is looked up and the form is collected, so that it can later be reported to SLY/Emacs's sticker system for annotation. Though its listing is too large to include here, its implementation is straightforward but for one detail described in section 4.1.

[7]This expansion is SBCL's.

[8]Furthermore, AGNOSTIC-LIZARD contains useful provisions to shield user code from certain nonconformities in the macroexpansions of certain built-in macros, such as DEFUN.

[9]This is Allegro CL's expansion. SBCL's is much simpler, and thus not so good for illustrative purposes.

[10]In reality, a few macros like COND and DEFUN are left unexpanded by AGNOSTIC-LIZARD so they are analysed separately as well

As we are nearing the end of our journey, we can now start putting all the pieces together. The following snippet is the final form of our Common Lisp function. Its results can be handed to SLY/Emacs for instrumentation through stickers as described in section 3.1. After compilation, the instrumented code is now steppable.

```
(defun stepper-sticker-locations (string)
  (with-input-from-string (stream string)
    (let* ((form-positions (make-hash-table))
           (form-tree
             (read-tracking-source
              stream nil nil nil
              (lambda (form start end)
                (setf (gethash form form-positions)
                      (cons start end))))))
      (multiple-value-bind (expanded-tree
                            expansion-positions)
          (mnesic-macroexpand-all form-tree
                                  form-positions)
        (forms-of-interest
         expanded-tree expansion-positions)))))
```

4 RESULTS AND FURTHER WORK

We have released the result of our work on the GitHub platform[11].

From an end user's perspective, to put the new SLY/Emacs stepper to work means pressing the key chord `C-c C-s P` (*control-c, control-s, capital P*) while the cursor is on a top-level form. This causes the interesting sub-forms of that top-level form to be automatically decorated with sticker *overlays*, which by default uses different shades of the color gray. As described in 3.1, a posterior compilation of that same top-level form shall *arm* the stickers and convert the overlays' color to shades of blue. From this point on, the stickers are executed as soon as the user arranges for the instrumented code to be run as usual.

Note that the default behavior of the sticker system doesn't equate the execution of an instrumented form to a break point, i.e. the invocation of the Lisp debugger. This is by design. As was explained in section 3.1, the default behavior is to have sticker executions merely record the return values (or non-local exits) for later replay. This which can be achieved with the key chord `C-c C-s C-r` or via `M-x sly-stickers-replay`. Alternatively, the key chord `C-c C-s S` (or `M-x sly-stickers-fetch`) can be used to fetch the most recent recordings for each sticker and visually decorate the source code, indicating (1) stickers that have been executed; (2) those that haven't yet, and (3) those that have exited non-locally.

Finally, to enable the classic stepper functionality, the user must explicitly select "breaking stickers" by affecting the value of the `SLYNK-STICKERS:*BREAK-ON-STICKERS*` variable.

We shall see, as we discuss its limitations, that the resulting SLY/Emacs stepper tool is still in its infancy. It can nevertheless be said to work reasonably well for a majority of normal circumstances, succeeding in instrumenting forms effectively and efficiently, while providing satisfactory methods of navigation among stickers.

[11]See `https://github.com/joaotavora/sly-slepper.git`.

4.1 Limitations concerning atoms

In section 3.3.2, we sidestepped a notorious difficulty with atomic forms, i.e. one-symbol symbolic expressions. This class of difficulties is already alluded to in [9, I-4.8]. The problem with atoms can be observed with the simplest of forms:

```
(lambda (x) x)
```

In this example, we note that the atom `X` has two different *manifestations* in the encompassing form. Naturally one wants only the latter to be annotated, and not the first. However, that is hard to determine reliably since both are represented by the very same object. This is in stark contrast to compound forms represented by different `CONS` cells.

We can enhance the form/position pairings table used above to record the fact that there is more than one manifestation of an atom, but that's not enough to know in `MNESIC-MACROEXPAND-ALL` which of those is a in a position of execution. The reason is that the `AGNOSTIC-LIZARD` macroexpander will only traverse sub-forms of forms actually returned by a macro's expansion. In this example, our hook is only called on the `(lambda (x) x)` and `x` forms, *not* on the `(x)` form. The latter form is merely an argument to the macro itself where we have no power of intervention, and thus there is no easy way to invalidate the first manifestation.

However, since we *do* know that `x` is manifested at `(9 . 10)` and `(12 . 13)` it is possible to devise heuristics to trace back to the knowledge gathered when first reading the form and traverse the atom's parent forms, given only the atom. A very simple heuristic can proceed like this: if the atom exists inside a compound form that does not occur in the final expansion, then that atom isn't interesting, otherwise, it is. This appears to solve the above situation but fails miserably in the presence of the `LOOP` macro since this macro has all the variable definition "unprotected" by parenthesis.

Hard-wiring exceptions to `LOOP` and other macros could ameliorate the situation, but overall this strategy feels murky and insufficient. On the other hand, if more aggressive strategies of atomic annotation are attempted, the SLY/Emacs sticker system has already shown to be reliable in the sense that if it needs to fail (because of an incorrect form being annotated), it will mostly do so early. Thus the potential to mislead users to wrong debugging conclusions is minimized.

A different approach to solve this problem could revisit `Eclector`'s "concrete syntax trees" or a variation thereof and use a portable, programmable macroexpander that also understands these types of trees, where different manifestations of the same atomic form are represented by different objects.

For now, the proposed SLY/Emacs's stepper works around this limitation by behaving conservatively and only annotating atoms in positions that are guaranteed to be safe, such as inside function call forms. This makes for the majority of situations in practice. Furthermore, users can always manually add stickers to other atom manifestations they are interested in and know to also be safe.

4.2 Interface limitations

As seen in section 3.1, SLY/Emacs's sticker system has no notion of a stack: all the armed stickers are enumerated serially and thus hierarchically equivalent. Therefore, the common "step in/step out/finish" functionality of common steppers is unavailable as such. Once stepping has been initiated, it is currently not possible to "step over" arbitrarily large sections of uninteresting code, nor is it yet possible to designate a sticker to continue to. The nearest thing available is the possibility to ignore a particular sticker number. Furthermore, there is as of yet no notion of a stepping "session": once armed, stickers take effect immediately and stay armed (even if the corresponding source code is deleted) until the definition they pertain to is compiled again without stickers.

These features don't seem hard to realize. E.g., to enable more sophisticated navigation behavior behavior the `RECORD` macro could use a special variable to be made aware of recursive invocations to itself or to the currently executing stack frame. Thus we could keep track of stickers being executed *inside* each other, i.e. within the dynamic scope of a previously active sticker annotation.

4.3 Portability

The system as been described as "portable" or "near-portable". Indeed, if a completely new conforming implementation of Common Lisp were to spring into being, support for our stepper system would have to deal with three potential sources of non-portability: (1) support of the `AGNOSTIC-LIZARD` code-walking program, (2) differing representations of constituent characters in readtables (for the source-tracking reader described in section 3.2), and (3) support for "gray streams". We defer the discussion of (1) to [10], noting that that system is built with portability as its foremost requisite. The "shielding" it offers is useful e.g. in dealing with SBCL's implementation of `CL:DEFMETHOD`, which itself produces a complicated transformation of its body. In (2), we note that the adjustments to the read-table were needed only for SBCL's implementation, which doesn't use a macro-character function for constituents. If the hypothetical new implementation *also* did so, it is plausible that the current code would support it. Otherwise, it would behave as the remaining implementations, also requiring no extra work. Lastly, for (3), we think it reasonable to expect support for "gray streams" in new implementations, since it is a widely adopted extension, and required by SLY/Emacs to begin with.

5 DISCUSSION

Common Lisp users are concerned about features that facilitate day-to-day development. `TRACE`, REPLs, `PRINT` forms, the interactive debuggers, profilers and stickers are all ways to solve certain debugging problems: some are more suitable to some situations than others. Therefore, it's important to note that program stepping is just another tool in the toolbox, not a panacea.

By the same token, if one does implement such a tool, it should be done in a manner that is of actual, practical use. This was the reasoning behind the SLY/Emacs stepper. We think it especially fortunate that the annotation-based approach and the manner of source-code correlation in SLY/Emacs's stickers don't make use of direct references to source files or file positions. As was shown in 3.1, we merely keep an enumerated list of stickers identifiers synchronized between Common Lisp and SLY/Emacs, a mechanism that is simple but effective. It adequately resists some modifications to the source of the instrumented form or its whereabouts, such as moving it around in the source file, or even adding white-space and comments inside it. This detail is crucial in making stickers and stepping usable for day-to-day programming, since the user isn't dragged away from his editor or forced to a special confinement while stepping. We note that these advantages of annotation-based steppers were already hinted at back in 1989 by the proponents of evaluation-based steppers[6, l.41, l.42].

By contrast, a hypothetical evaluation-based visual stepper (such an as enhanced version of [4]) would find it difficult to maintain this advantage since the source-code correlation is achieved at evaluation-time and is harder to mutate effectively afterwards. Perhaps this fact can explain why the non-portable visual steppers of the Allegro CL and LispWorks implementations don't work directly with the source-editing facilities present in these IDEs. A possible solution would be to represent source-code correlation in terms of sub-expression paths in the form tree instead of character positions, but it would still be hard to resist deletions and insertions at top level. A more heavy-handed solution would lock the source file read-only for the duration of the stepping session. However, users are normally adverse to such confinements.

We also think it fortunate that stepping is implemented as an increment to the existing stickers functionality. As in [9], we consider it an advantage of the annotation-based systems that users are allowed to instrument only the definitions they are interested in. Indeed, it is often the case that steppers become tedious to operate because they step on *too much.* Yet, in our system, users can manually adjust the automatically placed stickers, removing the ones they are not interested in, or adding others.

It may also be noted that the usual stepping paradigm where the program is stopped at each point is only one of the possibilities afforded by stickers. As we explained in sections 3.1 and 4, two further ones constitute innovative means of debugging: the *post-mortem* replay of sticker recordings and the visual decoration of source code with colors indicating the state of the most recent execution of each sticker.

In formulating the development of the SLY/Emacs stepper, we have also described (1) a simple, portable technique for constructing a source-tracking reader in terms of a compliant reader implementation and (2) a reliable technique to carry over source-tracking information to macroexpansion. It is conceivable that other debugging tools could be constructed from either of these elements.

The SLY/Emacs stepper described in this essay is an effective example of a portable, visual stepping facility for the Common Lisp ecosystem. To the best of our knowledge, it is indeed the *only* system combining these characteristics. As such, there is little to compare it against. The closest match could well lie outside of Common Lisp, in the aforementioned `edebug.el`[7] stepper, an annotation-based approach that is equally well integrated with the source code editor. That stepper has a more developed interface, but requires custom declarations for stepping into macro expansions, something the SLY/Emacs stepper handles automatically. Its current limitations notwithstanding, we believe that the underpinnings of the SLY/Emacs stepper – *stickers,* a simple source-tracking reader, and mnesic macroexpansion – are solid. We envision enhancements to its interface, perhaps by incorporating ideas of non-portable visual steppers, or steppers for other programming languages.

6 ACKNOWLEDGEMENTS

We would like to thank Luís Oliveira and Michael Kirkland for early comments and advice when discussing the stepper feature, as well as Andrew Lawson for the encouragement to produce this work. We also offer thanks to the reviewers who provided very insightful comments.

REFERENCES

[1] Free Software Foundation 1984. *Emacs GNU Emacs: An extensible, customizable, free/libre text editor — and more.* Free Software Foundation. https://www.gnu.org/software/emacs/

[2] LispWorks 2011. *LispWorks IDE User Guide: The Stepper.* LispWorks. http://www.lispworks.com/documentation/lw61/IDE-W/html/ide-w-496.htm

[3] Pascal Bourguignon. 2007. *Implements the Common Lisp Reader.* https://hub.darcs.net/hu.dwim/hu.dwim.reader/browse/source/reader.lisp

[4] Pascal Bourguignon. 2012. *Implements a Common Lisp stepper.* https://gitlab.com/com-informatimago/com-informatimago/-/blob/master/common-lisp/lisp/stepper.lisp

[5] Helmut Eller, Luke Gorrie, Eric Marsden, et al. 2003. *SLIME: The Superior Lisp Interaction Mode for Emacs.* Common-Lisp.net. https://common-lisp.net/project/slime/

[6] Ivo Haulsen and Angela Sodan. 1989. UnicStep-a visual stepper for COMMON LISP: portability and language aspects. *ACM Sigplan Lisp Pointers* III (07 1989). https://doi.org/10.1145/121999.122003

[7] Daniel LaLiberte. 1988. *Edebug: a source-level debugger for Emacs Lisp.* FSF. https://github.com/emacs-mirror/emacs/blob/master/lisp/emacs-lisp/edebug.el

[8] Jan Moringen and Robert Strandh. 2018. *A Portable, Source-tracking Reader for Common Lisp.* LaBRI, University of Bordeaux. https://github.com/s-expressionists/Eclector/blob/master/papers/to-submit-1/eclector.tex

[9] Glen Randolph Parker. 1987. Annotation-Based Program Stepping. *SIGPLAN Lisp Pointers* 1, 4 (Oct. 1987), 3–11. https://doi.org/10.1145/1317216.1317217

[10] Michael Raskin. 2017. Writing a best-effort portable code walker in Common Lisp., In Proceedings of 10th European Lisp Simposium. *10th European Lisp Simposium* I, 1, 11. https://doi.org/10.5281/zenodo.3254669

[11] João Távora et al. 2014. *SLY: Sylvester the Cat's Common Lisp IDE.* GitHub. https://github.com/joaotavora/sly.git

Demonstrations

Peer-reviewed demonstrations, that is, short papers highlighting features and usage.
The papers appear in the order they appear in the programme.

Indexing Common Lisp with Kythe

A Demonstration

Jonathan Godbout
jgodbout@google.com

ABSTRACT

For decades Lispers have had the power of code cross-references (jump to definition, list callers, etc.) for any code they've loaded into their Lisp image. But what about cross referencing code that isn't (or can't be) loaded into the image? Wouldn't it be great if we could ask "who, in the global Lisp community, calls this function?" The only option currently available is to download all Lisp code and use "grep" or similar text-based tools. At Google we use Kythe [4] as a cross-reference database for all Lisp code, whether loaded into our local Lisp image or not. We will show how Lisp is cross-referenced on a static web-page with hyperlinks between definitions. With this we can also get call graphs and call hierarchies [1].

ACM Reference Format:
Jonathan Godbout. 2020. Indexing Common Lisp with Kythe: A Demonstration. In *Proceedings of the 13th European Lisp Symposium (ELS'20).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.5281/zenodo.3765987

1 INTRODUCTION

Almost every software project will have a large number of files and functions. As soon as the number of files goes above 1, or the number of possible on-screen pages goes above 1, users will get confused about what definitions are used where. SLIME [5] has jump-to-definition using "`M-.`", so when the code has been loaded into the Lisp image we can jump to function definitions and call sites. On websites with static code, such as https://www.github.com, where the code is viewed statically on screen, it would be nice to get hyperlinks between the definitions and their usage. Kythe https://kythe.io/ is a service that allows users to implement language-specific indexers and then to upload graphs describing the structure of the code. This allows for code display and editing engines to provide services like jump-to-definition. At Google we have implemented a Lisp plugin for the Kythe indexer to produce cross reference data for Google's Common Lisp code base. We will start with a brief overview of Kythe, and then discuss indexing Lisp.

2 KYTHE OVERVIEW

Kythe is a database for storing code graphs for large code bases across multiple languages. Its schema is designed to accommodate facets of different languages. Part of its schema are `nodes` which name functions and variables, define exact locations in a file, or give other information about the indexed code. It has `VNames` which uniquely identify a node in a code base. It has `Edges` which annotate how two nodes relate to each other.

For example, take the variable object from `threadp` in Bordeaux-threads [7]:

```
(defun threadp (object)
  (typep object 'sb-thread:thread))
```

The variable `object` next to `threadp` would have a `node`:

```
{
  ticket: "kythe://corpus??lang=lisp?path=PATH
#BORDEAUX-THREADS%3A%3AOBJECT%20%3AVARIABLE
%20loc%3D%2825%3A16-25%3A22%29",
  kind: "variable",
  language: "lisp",
  name: "object",
  qualified_name: "object",
  location: {
    corpus: "corpus",
    path: "PATH/TO/bordeaux-threads
/src/impl-sbcl.lisp",
    line_number: 25,
    line_number_end: 25,
    column_number: 16,
    column_number_end: 22
  },
  v_name: {
    signature:
"BORDEAUX-THREADS::OBJECT :VARIABLE loc=(25:16-25:22)",
    corpus: "corpus",
    path:
"PATH/TO/bordeaux-threads/src/impl-sbcl.lisp",
    language: "lisp"
  }
}
```

The `VName` uniquely identifies the node. The slot `kind` tells which kind of node this is, so "`variable`" tells us this is a variable. The slot `location` tells us where the source location of the referenced code. The slot `ticket` is just a URI encoding of the `VName`. By location reference we mean a `node` containing the `location` of a form in the code.

There would be a second `node` for the instance of the variable which is the first argument to `typep`. Finally there would be an `edge`

```
{
  source: node1,
  target: node2,
  edge_kind: ref
}
```

[1]some limitations apply

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'20, April 27–28 2020, Zürich, Switzerland
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.3765987

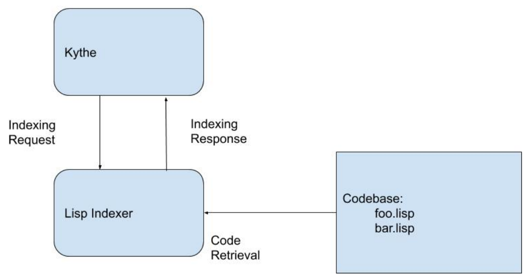


Figure 1: Kythe Calling the Lisp Indexer

where `node1` and `node2` are the first and second nodes discussed above.

For full details on Kythe's schema please reference https://kythe.io/docs/schema/.

3 STRATEGY

In an out-of-band process, we start up a Lisp indexing service, and have it load all the code required to populate the who-calls database with the requisite information. This is essentially how SLIME determines jump-to-definition targets (along with some heuristics needed for problems discussed later).

You may have:

```
foo.lisp uses bar.lisp
```

The Lisp indexing plugin loads bar.lisp and foo.lisp into the Lisp image and the Lisp implementation determines the cross-reference information locally. If you are trying to create all cross-references for foo.lisp and bar is a function defined in bar.lisp we can inspect the who-calls database to get this cross-reference.

In SBCL [3] you get all of the top level `defun` and `defvar` forms, but none of the top level forms that don't define a data structure that are needed later. For example, code that is run at start-up time, such as `(setf *foo* 'foo)`, at the top level may not have a cross reference in the who-calls database because the compiler can compile the call away. We will go through some examples:.

Local variable bindings aren't stored in the who-calls database. If you have a function

```
(defun print-a (a)
  (print a))
```

you would like to have a cross-reference from the a in print-a's lambda list to its use in the function's body. This is not stored in the who-calls database. To solve cases such as this we have a number of parsers (e.g. "`defun`" parser) that will get the symbols to be bound and store their location. Iterating through all of the code, with the correct set of parsers, will give us all of the local definitions. Currently our parser is only a decent heuristic, and our method parser does not correctly cross-reference types.

Next we have hidden parameters that don't show up in the code or the who-calls database. Take for example:

```
(defstruct bear cat)

(defun set-bear-cat-friendly (my-bear-cat)
  ... lots of code ...
  (setf (bear-cat my-bear-cat) 'friendly))
```

We would like a reference from the `bear-cat` setter to the cat slot in the `bear` structure. In (most) Lisps, this would be fine, we would just add a call to who-calls for `(setf bear-cat)`, but the Lisp language specification does not require such a function to exist. In fact SBCL does not create `setf` functions for `structure-objects`, so we must start by going through the code and creating location references for all `structure-object` accessors.

4 INTER-LANGUAGE REFERENCES

We often make calls from one language into another language, for example Lisp's foreign functions calls into C. At Google, the most common format for data interchange between systems is called Protocol Buffers [2], or protobuf for short. A protobuf is a data interchange format that a language can implement.

To implement support for protobuf messages languages can use their native structures but they must serialize the messages into a standard format before sending them out. Then any other language that implements the protobuf standard can deserialize and read the messages. The content of the messages can be deserialized without knowledge of the protobuf schema used, but a protobuf schema detailing types and names are required for human readable output.

Here is an example protobuf schema defining one "message" (a structure) that contains a string:

```
syntax = "proto2";

package example;

message HelloWorld {
  optional string hello_world_string = 1;
}
```

Below we have lisp code that creates the Lisp `standard-object` corresponding to the structure.

```
(let ((my-proto
        (make-instance 'example:hello-world
          :hello-world-string ``hello-world'')))
  (print (hello-world-string my-proto)))
```

We would like a reference from "`hello-world-string`" in the Lisp code to the "`hello_world_string`" in the protobuf schema. As Kythe is just a database service that stores a graph of the code for contextualization in a language agnostic form, so long as you know the signature for the "`hello_world_string`" you can just create a cross-reference in Kythe.

5 MACROS

The use of a small number of parsers to understand local bindings is not ideal but it is doable for the built in commands. In contrast Common Lisp is known for its powerful syntax-extending ability, namely macros. For a detailed look at macros please consut *Let Over Lambda* [6], we will go over a basic examples below.

```
(defvar *process-data-mutex* (make-mutex))

(defmacro with-data-mutex ((mutex) &body body)
  `(let ((,mutex *process-data-mutex*))
    (sb-thread:get-mutex ,mutex)
```

```
        ,@body
        (sb-thread:release-mutex ,mutex)))

  (defun process-data (data)
    (with-data-mutex (data-mutex)
      (format t "I have mutex ~a"  data-mutex)
      (print a)))
```

The variable "`data-mutex`" is bound in the macro with-data-mutex but we would need to create a new parser for `with-data-mutex`! This technique is inherently non-scalable; sadly we do not yet have a solution.

There have been two possible ways brought up to extend our indexers support for macros. The first is updating the brief support for SBCL in the who-calls database, or in a contrib. This would necessarily be tightly bound to SBCL and any language which wants decent macro cross-references would have to do the same.

The other idea is to implement a code walker that expands macros and determines what variables are being bound during expansion. This would be less robust, but it would be compiler-independent.

6 DOCUMENTATION

Kythe creates a code graph with nodes representing objects such as functions and variables, edges connecting those object, and properties of the objects themselves. For functions this can include their docstrings and their variables. For globals this also includes their docstrings. Lisp makes this easy by having the docstrings of a function or global reference as a slot on the function description during run time. That way, when you parse a function, you can just send Kythe its comment as a Kythe graph node.

7 SO WHAT IS IN THIS FOR ME?

The beauty of Kythe is you get a graph of your code sitting in a database you can use to for code hyperlinking (as with Slime) or any other kind of code introspection. You can make code graphs over large projects, or multiple projects, without needing anything loaded in a REPL; the indexing is completely out-of-band. You could power Emacs without having to load Slime, though that seems far-fetched as we already have Slime. You can create a Kythe plugin for your own favorite inter-operating language, and have cross references between them. For example a Java-based Lisp (ABCL [1]) index is well within reason!

8 FUTURE WORK

Sadly, we do not have a great answer to local bindings with macros. Macros are hard, and syntax is always changing. My current working idea is to use a code walker and inspect the environment as we go.

The Kythe Lisp plugin only works for SBCL. It would be nice to get it to work for every Common Lisp, or at least the major versions of Common Lisp. Since the code stopped trying to be generic a while back, this would take a little bit of effort.

Kythe itself is an open source system, as well as several language plugins such as C++ and Java. We plan to open source the Common Lisp plugin.

REFERENCES

[1] Armed bear common lisp. https://abcl.org/.
[2] Protocol buffers. https://developers.google.com/protocol-buffers. Accessed: 2020-02-10.
[3] Steel bank common lisp. http://www.sbcl.org/.
[4] Kythe: A pluggable, (mostly) language-agnostic ecosystem for building tools that work with code. https://kythe.io/, 2019. Accessed: 2020-02-10.
[5] Slime: The superior lisp interaction mode for emacs. https://www.economics.utoronto.ca/osborne/latex/BIBTEX.HTM, 2019. Accessed: 2020-02-10.
[6] Doug Hoyte. *Let Over Lambda*. Lulu.com, 2008.
[7] Stelian Ionescu. Bordeaux threads. https://github.com/sionescu/bordeaux-threads.

JACL: A Common Lisp for Developing Single-Page Web Applications

Alan Dipert
alan@dipert.org

ABSTRACT

This paper demonstrates JavaScript-Assisted Common Lisp (JACL), an experimental Web-browser based implementation of an extended subset of Common Lisp. JACL, which is in the early stages of development, is an effort to explore new techniques for large-scale Single-page Web Application (SPA) development in Lisp. JACL includes an optimizing Lisp-to-JavaScript compiler and interoperates with JavaScript. JACL promotes interactive, *residential* development in the Web browser environment with its *asynchronous reader* and Chrome DevTools-based REPL client.

CCS CONCEPTS

• **Software and its engineering → Dynamic compilers**; **Runtime environments**.

KEYWORDS

Common Lisp, JavaScript, web applications

ACM Reference Format:
Alan Dipert. 2020. JACL: A Common Lisp for Developing Single-Page Web Applications. In *Proceedings of ELS '20: European Lisp Symposium (ELS '20).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.3764494

1 INTRODUCTION

The demand for SPAs in the past decade has only grown, and users and stakeholders continually expect larger and more sophisticated applications. Unfortunately, large-scale development on the Web browser platform presents a particular set of challenges that are not easily overcome. Developers have responded to these challenges by creating a widening variety of special-purpose programming languages that compile to JavaScript [12, 23, 24]. Each new language promotes one or more paradigms, application architectures, or development workflows, and claims some advantage relative to the status quo.

This paper demonstrates one new such language, JavaScript-Assisted Common Lisp (JACL), an experimental implementation of an extended subset of Common Lisp. JACL was created to explore new techniques for applying Common Lisp — a proven[6, 13, 14] substrate for UI innovation — to SPA development.

Many projects involving compilation of Lisp to JavaScript precede JACL. Lisps that have either demonstrated industrial utility or that implement a significant subset of Common Lisp are surveyed in appendix A. Like many of these related efforts, JACL includes an online, optimizing compiler and supports interoperation with JavaScript. JACL distinguishes itself from these efforts by placing special emphasis on the value of *residential* development style, where both applications and the tools used to create them co-evolve in a shared environment. JACL provides fundamental support for residential development with its *asynchronous reader.*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ELS '20, April 27–28, 2020, Zürich, Switzerland
© 2020 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.5281/zenodo.3764494

2 INTEROPERATION WITH JAVASCRIPT

JACL integrates tightly with JavaScript and depends heavily on the JavaScript runtime. As a result, JACL enjoys roughly the same applicability and performance characteristics as the JavaScript platform. However, this high degree of integration is at odds with comformance to the Common Lisp specification, and so JACL will never strictly conform.

2.1 Object Types

JACL introduces several of its own object types, currently implemented in JavaScript, including `Cons`, `LispSymbol`, and `LispString`. `Cons` and `LispSymbol` are introduced because JavaScript does not include direct equivalents. `LispString` is introduced because the native JavaScript `String` is immutable, whereas Lisp strings are mutable.

JACL includes support for only one numeric type, the JavaScript `Number` object. The JavaScript `Number` is a double-precision 64-bit IEEE 754 value. The JACL reader interprets integers as `Number` objects. In the future, JACL will also interpret floating-point numbers as `Number`. This decision trades ANSI conformance for performance. If either type were boxed, arithmetic performance would suffer intolerably. JSCL[20] and Valtan[11] make the same tradeoff.

JACL functions are JavaScript functions, and may be invoked by JavaScript callbacks without a special calling convention. JavaScript functions named as Lisp values may be invoked with `FUNCALL` or `APPLY`. Neither arguments nor return values are automatically coerced to or from any particular set of object types.

2.2 Operators

The JACL compiler supports a special operator for constructing fragments of JavaScript code, verbatim, from Lisp. The semantics of this operator, `JACL:%JS`, are inspired by a similar feature of ClojureScript[9], `js*`. For example, the following JACL code displays the number 3 in an alert box:

```
(JACL:%JS "window.alert(~{})" 3)
```

The character sequence `~{}` is distinct from any plausible JavaScript syntax and so is used as placeholder syntax. There must be as many placeholders as there are arguments to `JACL:%JS`.

In addition to JACL:%JS, the JACL compiler currently supports three more special operators for interacting with the host platform: JACL:%NEW, JACL:%DOT and JACL:%CALL. These operators perform JavaScript object instantiation, field access, and function calls, respectively. Since JACL functions are JavaScript functions, JACL:%CALL is the basis for FUNCALL in JACL, and for function calls generally.

JACL also supplies a convenience macro, JACL:\. or "the dot macro" for performing a series of field accesses and method calls[1] concisely. The dot macro takes direct inspiration from the .. macro of Clojure[8]. JACL:\. expands to zero or more nested JACL:%DOT or JACL:%CALL forms. Here is an example of a JACL:\. form — equivalent to the JavaScript expression (123).toString().length — and its corresponding expansion:

```
(\. 123 (|toString|) |length|)
(%DOT (%CALL 123 |toString|) |length|)
```

Note that JavaScript identifiers are case sensitive, and so case-preserving, pipe-delimited Lisp symbols must be used to refer to JavaScript object field and method names. The *readtable case* of the JACL reader cannot currently be modified. The dot macro also recognizes Lisp or JavaScript strings as JavaScript identifiers.

2.3 Reader Macros

JACL includes two reader macros to support interoperation with JavaScript. These macros may be added to the *READTABLE* by calling the function (JACL:ENABLE-JS-SYNTAX). @" denotes JavaScript String objects and @| denotes JavaScript identifiers.

For example, the following two forms, which both evaluate to a JavaScript String, are equivalent:

```
@"Hello"
(\. "Hello" (|toString|))
```

@| may generally be used in place of the JACL:%JS special form to refer to JavaScript identifiers. (JACL:%JS "alert") and @|alert| are equivalent.

3 RUNNING JACL PROGRAMS

Currently, JACL programs may be evaluated in the Web browser in two ways: by adding Lisp <script> tags to the <head> of a Web page that also includes jacl.js, or by using the jacl tool included in the JACL distribution[1] to connect to a running Web browser.

3.1 Lisp Scripts

Development of JACL itself is currently driven primarily by modifying jacl.js and the boot.lisp and jacl-tests.lisp Lisp scripts. The Lisp scripts are included in the jacl.html file in the JACL distribution[1]. After each modification, the Web browser is reloaded, and test results are displayed.

This Lisp script-based workflow is similar to the traditional JavaScript development workflow and has served JACL development so far. However, Lisp scripts require runtime parsing and compilation of JACL source code, among other inefficiencies. Reloading the Web browser also destroys the entire runtime environment.

[1]Strictly speaking, JavaScript "method calls" are normal function calls but with a particular value of this.

The easiest way to create JACL programs in this manner is to start with the jacl.html Web page provided by JACL and then modify it by removing or adding new Lisp scripts.

It is imagined that ultimately, Lisp sources will be incorporated into the Lisp *image* exclusively by the REPL client tool. An arrangement such as this decouples source code loading from the Web browser lifecycle. Production executables may then be produced at any time from the Lisp image using a Lisp function in a manner similar to the SAVE-LISP-AND-DIE[22] function in SBCL or the DELIVER[16] function in LispWorks.

3.2 REPL

JACL includes a REPL client program, jacl, that may be used to execute JACL programs in a Web browser from a terminal on the host. This process is described in detail in the RUN.md document included in the JACL distribution[1], but is summarized here.

In order to use the REPL, the user must first start either the Google Chrome or Chromium browser with the remote debugging feature enabled. With remote debugging enabled, the Web browser may be controlled using a client program over a WebSocket connection. Then, the user must navigate to a Web page that includes at least jacl.js and boot.lisp.

Finally, the user must start the jacl REPL client in a terminal. jacl leverages the remote debugging feature as a REPL transport, using it to send and receive characters between the host and the remote JACL runtime. The jacl tool is currently written in R[21] and uses the chromote[7] package for interacting with the remote Chrome or Chromium browser.

The jacl program has no knowledge of JACL syntax or semantics; it merely sends and receives characters. The intentional simplicity of jacl is part of the larger project goal of promoting residential-style tool and program development in the target environment. The simplicity of jacl is possible because of the asynchronous nature of the JACL reader. Incoming characters delivered over the WebSocket debugging connection are received by callback functions in the Web browser. The received characters are asynchronously and incrementally parsed into Lisp data. When a complete datum is formed, the compiler is called, and the resulting JavaScript is evaluated. Finally, any output is sent back over the debugger connection and received and printed by the jacl program.

4 CONCLUSION

We introduced JACL, a new and experimental Common Lisp created to explore techniques for building sophisticated SPAs. JACL integrates tightly with the Web browser platform and interoperates directly with JavaScript. Compared to other browser-based Lisps, JACL promotes residential development, and introduces a new technique for integrating the REPL into the development workflow.

5 FUTURE WORK

JACL currently lacks many basic Common Lisp data types, functions, and operators. Ultimately, JACL should support as much of Common Lisp as is possible, accounting for the severe limitations imposed by JavaScript and the Web platform. Fortunately, the many other existing Common Lisps that compile to JavaScript demonstrate that a compelling implementation is achievable.

An in-browser REPL and other tools for interacting with the JACL runtime in the Web browser would be desirable. Such tools could optionally remain as parts of deployed applications and provide a degree of introspection and extension capability even after the application has been deployed.

Other than work related to missing features such as multiple values, CLOS, and the conditions system, much design work remains with regard to the specific affordances of the `jacl` tool. For example, it's unclear how a large JACL project involving library dependencies and multiple source files should be managed and loaded.

6 ACKNOWLEDGMENTS

The author wishes to thank Micha Niskin, Bart Botta, Kevin Lynagh, Lionel Henry, and Andy Keep for invaluable feedback on early versions of this paper. The author wishes to express particular thanks to Robert Strandh not only for his feedback, but also for his guidance on the writing process.

The author also wishes to express special gratitude to his beautiful wife, Sandra Dipert, for her encouragement and support.

Finally, the author wishes to express his deepest thanks to his father, the late Randall R. Dipert, for first telling him about Lisp and many other things besides.

REFERENCES

[1] Alan Dipert. 2020. *JACL*. Retrieved April 16, 2020 from https://tailrecursion.com/JACL/
[2] Marco Baringer. 2005. *Parenscript*. Retrieved February 12, 2020 from https://web.archive.org/web/20051122141019/http://blogs.bl0rg.net/netzstaub/archives/000525.html
[3] Mihai Bazon. 2012-2018. *Implementation notes*. Retrieved February 12, 2020 from http://lisperator.net/slip/impl
[4] Mihai Bazon. 2012-2018. *SLip âĂŤ a Lisp system in JavaScript*. Retrieved February 12, 2020 from http://lisperator.net/slip/
[5] Mihai Bazon. 2012-2018. *Versus Common Lisp*. Retrieved February 12, 2020 from http://lisperator.net/slip/vscl
[6] Howard I. Cannon. 2007. *Flavors: A non-hierarchical approach to object-oriented programming*. Retrieved February 12, 2020 from http://www.softwarepreservation.org/projects/LISP/MIT/nnnfla1-20040122.pdf
[7] Winston Chang. [n.d.]. *chromote: Headless Chrome Web Browser Interface*. https://github.com/rstudio/chromote
[8] Cognitect, Inc. 2020. *Clojure*. Retrieved February 12, 2020 from https://clojure.org/
[9] Cognitect, Inc. 2020. *ClojureScript*. Retrieved February 12, 2020 from https://clojurescript.org/
[10] Cognitect, Inc. 2020. *Companies*. Retrieved https://clojure.org/community/companies from https://clojurescript.org/community/companies
[11] cxxxr. [n.d.]. *cxxxr/valtan*. Retrieved April 4, 2020 from https://github.com/cxxxr/valtan
[12] Evan Czaplicki. 2012. *Elm: Concurrent FRP for Functional GUIs*. Retrieved February 12, 2020 from https://elm-lang.org/assets/papers/concurrent-frp.pdf
[13] B. A. Myers et al. 1990. Comprehensive Support for Graphical, Highly-Interactive User Interfaces: The Garnet User Interface Development Environment. *IEEE Computer* 23, 11 (Nov. 1990), 71–85. https://doi.org/10.1109/2.60882
[14] Paul Hammant. 2013. *Interface Builder's Alternative Lisp timeline*. Retrieved February 20, 2020 from https://paulhammant.com/2013/03/28/interface-builders-alternative-lisp-timeline/
[15] Rich Hickey. 2012. ClojureScript Release. Retrieved February 12, 2020 from https://www.youtube.com/watch?v=tVooR-dF_Ag
[16] LispWorks Ltd. 2017. *deliver*. Retrieved February 21, 2020 from http://www.lispworks.com/documentation/lw71/DV/html/delivery-220.htm
[17] Vladimir Sedach Marco Baringer, Henrik Hjelte. 2005-2019. *Parenscript Reference Manual*. Retrieved February 12, 2020 from https://common-lisp.net/project/parenscript/reference.html
[18] Peter Norvig. 1992. *Paradigms of Artificial Intelligence Programming: Case studies in Common Lisp*. Morgan Kaufmann Publishers, San Francisco, CA, USA.
[19] David Vázquez Púa. 2018. Growing a Lisp compiler. Retrieved February 12, 2020 from https://www.youtube.com/watch?v=XT7JYPtWMd8
[20] David Vázquez Púa and contributors. [n.d.]. *jscl-project/jscl*. Retrieved February 12, 2020 from https://github.com/jscl-project/jscl/
[21] R Core Team. [n.d.]. *R: A Language and Environment for Statistical Computing*. http://www.R-project.org/
[22] SBCL Project Contributors. 2020. *SBCL 2.0.1 User Manual*. Retrieved February 21, 2020 from http://www.sbcl.org/manual/
[23] Soma Somasegar. 2012. *TypeScript: JavaScript Development at Application Scale*. Retrieved February 4, 2020 from https://web.archive.org/web/20121003001910/http://blogs.msdn.com/b/somasegar/archive/2012/10/01/typescript-javascript-development-at-application-scale.aspx
[24] Wikipedia contributors. 2020. Reason (syntax extension for OCaml) — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Reason_(syntax_extension_for_OCaml)&oldid=940051580 [Online; accessed February 12, 2020].

A SURVEY OF RELATED LISPS

A.1 Parenscript

Released in 2005[2], Parenscript[17] was the first Common Lisp compiler to target JavaScript. Parenscript is not bootstrapped and its compiler is not written in JavaScript, and so it relies on a hosting Common Lisp system for compilation. Only JavaScript types are available to Parenscript programs at runtime, and so Parenscript is more of a syntax frontend for JavaScript than it is an interactive Lisp system. While Parenscript is not positioned to facilitate large-scale SPA development, it remains a popular way to add dynamic JavaScript-based behaviors to static Web sites.

A.2 SLip

SLip[3, 4] is arguably the most ambitious Common Lisp-on-JavaScript system created to date, even though it intentionally diverges[5] from Common Lisp in certain ways. It offers a stunning array of powerful features including a self-hosting compiler, a full set of control operators, JavaScript Foreign-Function Interface (FFI), tail-call optimization, green threads, and perhaps most impressively, a resident Emacs clone, *Ymacs*. SLip is based originally on the compiler and bytecode interpreter presented in Chapter 23 of *Paradigms of Artificial Intelligence Programming: Case studies in Common Lisp*[18].

A.3 JSCL

JSCL[19, 20] compiles directly to JavaScript and is self-hosting, includes the major control operators, and integrates tightly with JavaScript. JSCL includes a reader, compiler, and printer, and evaluation is performed by the JavaScript `eval()` function. Between these, a Read Eval Print Loop (REPL) is possible, and the JSCL distribution includes an implementation of one.

A.4 ClojureScript

ClojureScript [9, 15] is probably the most successful Lisp dialect for building SPAs by number of commercial users [10]. ClojureScript is a dialect of an earlier language, Clojure[8], which targets Java Virtual Machine (JVM) bytecode. The ClojureScript reader and macro systems were both originally hosted in Clojure, in a manner similar to Parenscript. ClojureScript prioritizes the ability to produce high-performance deliverables.

A.5 Valtan

Valtan[11] compiles to JavaScript and includes a suite of FFI operators for interoperating with JavaScript. It is self-hosting and features a sophisticated, CLOS-based compiler architecture. It also includes a REPL and several example applications.