Proceedings of the

16th European Lisp Symposium

Amsterdam, Nederlands
April 24 — 25, 2022
In cooperation with ACM

ISBN-13: 978-2-9557474-7-6
ISSN: 2677-3465

Preface

Message from the Program Chair

Welcome to the 16thth European Lisp Symposium!
It is my pleasure to off these proceedings to the community. Herein you will find descriptions of the keynote presentations, as well as the research papers and demo descriptions submitted by researchers. I hold the utmost appreciation to the keynote presenters for agreeing to present their work to this symposium. In additional, I would like to thank everyone who made a submission to this year's symposium.
A special thanks goes out to the chairing committee who had the task of reviewing the submissions, and giving feedback to the authors. This work is mostly done in silence and may not be appreciated by the symposium atendees. So again thank you for your work.
Thank you also to the virtualization team.
Further special thanks to the local chair Breanndán Ó Nualláin for keeping me in the dark of all that was needed to have a venue and to keep us fed.
Finally, thank you to all the attendees. I hope you enjoyed the symposium, and that you find something helpful and inspiring.

Organization

Symposium Organizer

- Didier Verna, EPITA, France

Programme Chair

- Stefan Monnier, DIRO, Université de Montréal, Canada

Local Chair

- Breanndán Ó Nualláin, Machine Learning Programs, Nederlands

Virtualization Team

- Georgiy Tugai
- Michał Herda
- Nicolas Hafner

Programme Committee

Mark Evenson	not.org, Austria
Marco Heisig	Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
Ioanna Dimitriou	Igalia S.L., Germany
Robert Smith	HRL Laboratories
Mattias Engdegård	
Marc Feeley	Université de Montréal, Canada
Marc Battyani	FractalConcept
Alan Ruttenberg	National Center for Ontological Research, USA
Nick Levine	RavenPack, Spain
Ludovic Courtès	Inria, France
Matthew Flatt	University of Utah, USA
Irène Durand	Université Bordeaux 1, France
Jay McCarthy	Brigham Young University, USA
Ambrose Bonnaire-Sergeant	Cisco
Christopher League	Long Island University, NY, USA
Pascal Costanza	Intel, Belgium
Christian Queinnec	

Sponsors

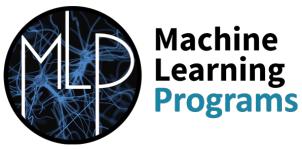
We gratefully acknowledge the support given to the 16thth European Lisp Symposium by the following sponsors:

Franz, Inc.
2201 Broadway, Suite 715
Oakland, CA 94612
USA
`www.franz.com`

SYSCOG
Campo Grande, 378 – 3
1700–097 Lisboa
Portugal
`www.siscog.pt`

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

DIRO
2920, chemin de la Tour
Montréal, QC
Canada
`diro.umontreal.ca`



Machine Learning Programs
Buckholt Drive
Warndon, Worcester, WR4 9SR
United Kingdom
`www.mlprograms.com`

Invited Contributions

Run-Time Verification of Communication Protocols in Clojure

Sung-Shik Jongmans, Open University, Heerlen, the Netherlands

To simplify shared-memory concurrent programming, languages have started to offer core support for high-level communications primitives, in the form of message passing though channels, in addition to lower-level synchronization primitives. Yet, a growing body of evidence suggests that channel-based programming abstractions also have their issues.
The Discourje project aims to help programmers cope with channels and concurrency bugs in Clojure programs, based on dynamic analysis. The idea is that programmers write not only implementations of communication protocols in their Clojure programs, but also specifications. Discourje then offers a run-time verification library to ensure that channel actions in implementations are safe relative to specifications.

Hedy: Gradual, Multi-Lingual, and Teacher-Centric Programming Education

Felienne Hermans, Leiden University, the Netherlands

When kids learn to program they often use either a visual language like Scratch, or a textual language like Python. While visual languages are great for the first steps, children and educators often want to move on to textual languages. However, early on, a textual language and its error messages can be scary. Hedy aims to bridge this gap with a programming language that is gradual, using different language levels.
In level 1, there is hardly any syntax at all; printing is done with: *print hello!*
At every level, new syntax and concepts are added, so learners do not have to master everything at once. Hedy builds up to a subset of Python including conditions, loops, variables, and lists.
To make learning as accessible as possible, Hedy also allows for the use of localized keywords, e.g. in Spanish: *imprimir Hello!* Hedy (`www.hedy.org`) was launched in early 2020 and over 5 million Hedy progams have been created to date, and has been translated into 46 languages.

A Language-Based Approach to Programming with Serialized Data

Michael Vollmer, University of Kent, Canterbury, UK

It is common for software running today to use object representations fixed by the language runtime system; both the Java and Haskell runtimes dictate an object layout, and the compiler must stick to it for all programs. And yet when humans optimize a program, one of their primary levers on performance is changing data representation. For example, an HPC programmer knows how to pack a regular tree into a byte array for more efficient access. Unfortunately, this is error-prone, making it an undesirable way to achieve performance optimization at the expense of safety and readability.

Furthermore, whenever a program receives data from the network or disk, rigid insistence on a particular heap layout causes an impedance mismatch we know as deserialization. Data represented in memory has pointers and arbitrary, sparse layout, while data on disk is packed contiguously, so data must be transformed from one form to another and back.
Programming with serialized data is a technique for unifying the in-memory and on-disk representations of data, where the serialized form is used both on-disk and in-memory. This technique allows data processing programs to skip the deserialization/reserialization steps by operating directly on the data in its serialized form. It also represents a principled approach to optimizing programs by compacting data representations, which increases locality and minimizes indirection.
In this talk, I will present a programming language, LoCal, for programming with serialized data. I will also describe Gibbon, an experimental compiler that automatically transforms functional programs to operate on serialized data.

Artificial Intelligence: a Problem of Plumbing?

Gerald J. Sussman, MIT CSAIL, USA

We have made amazing progress in the construction and deployment of systems that do work originally thought to require human-like intelligence. On the symbolic side we have world-champion Chess-playing and Go-playing systems. We have deductive systems and algebraic manipulation systems that exceed the capabilities of human mathematicians. We are now observing the rise of connectionist mechanisms that appear to see and hear pretty well, and chatbots that appear to have some impressive linguistic ability. But there is a serious problem. The mechanisms that can distinguish pictures of cats from pictures of dogs have no idea what a cat or a dog is. The chatbots have no idea what they are talking about. The algebraic systems do not understand anything about the real physical world. And no deontic logic system has any idea about feelings and morality.
So what is the problem? We generally do not know how to combine systems so that a system that knows how to solve problems of class A and another system that knows how to solve problems of class B can be combined to solve not just problems of class A or class B but can solve problems that require both skills that are needed for problems of class A and skills that are needed for problems of class B.
Perhaps this is partly a problem of plumbing. We do not have linguistic structures that facilitate discovering and building combinations. This is a fundamental challenge for the programming-language community. We need appropriate ideas for abstract plumbing fittings that enable this kind of cooperation among disparate mechanisms. For example, why is the amazingly powerful tree exploration mechanism that is used for games not also available, in the same system, to a deductive engine that is being applied to a social interaction problem?
I will attempt to elucidate this problem and perhaps point at avenues of attack that we may work on together.

Program overview

Monday	Morning	24 April 2023
09:30–09:45		**Registration, Badges, Meet and Greet**
09:45–10:00		Welcome Message
10:00–11:00	Keynote	Sung-Shik Jongmans Run-Time Verification of Communication Protocols in Clojure
11:00–11:30		**Coffee break**
11:30–12:00	Research Paper	Dider Verna A Mop-based Implementation for Method Combinations
12:30–12:30	Research Paper	Marcel Santos A Minimal Run-time Overhead Metaobject Protocol for Julia
12:30–14:00		**Lunch**
14:00–14:30	Research Paper	Jim Newton An Elegant and Fast Algorithm for Partitioning Types
14:30–15:00	Demo	Panicz Maciej Godek GRASP: An Extensible Tactile Interface for Editing S-expressions
15:00–15:30		**Coffee break**
15:30–16:30	Keynote	Felienne Hermans Hedy: Gradual, Multi-Lingual, and Teacher-Centric Programming Education
16:30–17:00		**Enlightening Lightning Talks**

Tuesday	Morning	25 Avril 2023
09:00–09:30		**Meet and Greet**
09:30–10:30	Keynote	Michael Vollmer A Language-Based Approach to Programming with Serialized Data
10:30–11:00		**Coffee Break**
11:00–11:30	Demo *Remote*	Alejandro Zamora Fonseca A stepper for Armed Bear Common Lisp (ABCL)
11:30–12:00	Experience Report	Nicolas Hafner Kandria - A Game in Common Lisp
12:00–12:30	Sponsored	Fábio Almeida SISCOG - 35 years of keeping trains on track
12:30–14:00		**Lunch**
14:00–14:30	Research *Remote*	Hayley Patton Parallel Garbage Collection for SBCL
14:30–15:00	Research Paper	Alexander Wood, Charles Zhang, and Christian Schafmeister Design of an Efficient Lisp Bytecode Machine and Compiler
15:00–15:30		**Coffee Break**
15:30–16:30	Keynote *Remote*	Gerald J. Sussman Artificial Intelligence: a Problem of Plumbing?
16:30–17:00		**Enlightening Lightning Talks**
17:00–17:15		**Closing Ceremony**
17:15		**Conference End**

Monday, 24 April 2023

A Mop-Based Implementation for Method Combinations

Method Combinators Revisited

Didier Verna
EPITA, LRE
Le Kremlin-Bicêtre, France
didier@lrde.epita.fr

ABSTRACT

In traditional object-oriented languages, the dynamic dispatch algorithm is hardwired to select and execute the most specific method in a polymorphic call. In Clos, the Common Lisp Object System, an abstraction known as *method combinations* allows the programmer to define their own dispatch scheme. When Common Lisp was standardized, method combinations were not mature enough to be fully specified.

In 2018, using Sbcl as a research vehicle, we analyzed the unfortunate consequences of this under-specification and proposed a layer on top of method combinations designed to both correct a number of observed behavioral inconsistencies, and propose an extension called "alternative combinators". Following this work, Sbcl underwent a number of internal changes that fixed the reported inconsistencies, although in a way that hindered further experimentation.

In this paper, we analyze Sbcl's new method combinations implementation and we propose an alternative design. Our solution is standard-compliant so any Lisp implementation can potentially use it. It is also based on the Mop, meaning that it is extensible, which restores the opportunity for further experimentation. In particular, we revisit our former "alternative combinators" extension, broken after 2018, and demonstrate that provided with this new infrastructure, it can be re-implemented in a much simpler and non-intrusive way.

CCS CONCEPTS

• **Software and its engineering → Object oriented languages**; **Extensible languages**; **Polymorphism**; *Inheritance*; *Classes and objects*; *Object oriented architectures*; *Abstraction, modeling and modularity*.

KEYWORDS

Object-Oriented Programming, Common Lisp Object System, Meta-Object Protocol, Generic Functions, Dynamic Dispatch, Polymorphism, Multi-Methods, Multiple Dispatch, Method Combinations, Orthogonality

ACM Reference Format:
Didier Verna. 2023. A Mop-Based Implementation for Method Combinations: Method Combinators Revisited. In *Proceedings of the 16th European Lisp Symposium (ELS'23).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.5281/zenodo.7818680

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

ACM ISBN 978-2-9557474-7-6.
https://doi.org/10.5281/zenodo.7818680

1 INTRODUCTION

Common Lisp was the first programming language equipped with an object-oriented (OO) layer to be standardized [14]. Although in the lineage of traditional class-based OO languages such as Smalltalk and later C++ and Java, Clos, the Common Lisp Object System [2, 5, 6, 8], departs from those in several important ways.

First of all, Clos offers native support for multiple dispatch [3, 4]. The existence of multi-methods pushes the dynamic dispatch one step further in the direction of separation of concerns: polymorphism and inheritance are clearly separated. Next, when implemented on top of the Mop [9, 11], the very semantics of Clos itself can be extended or modified, hence providing a form of homogeneous behavioral reflection [10, 12, 13].

Yet another improvement over classical OO lies in the concept of *method combination*. In the traditional approach, the dynamic dispatch algorithm is hardwired: every polymorphic call ends up executing the most specific method available (applicable) and using other, less specific ones requires explicit calls to them. In Clos however, a generic function can be programmed to implicitly call several applicable methods, not necessarily by order of specificity, and combine their results in a particular way. Along with multiple dispatch, method combinations constitute one more step towards *orthogonality* [7, chapter 8]: a generic function can now be seen as a 2D concept: 1. a set of methods and 2. a specific way of combining them. As usual with this language, method combinations are also fully programmable, essentially turning the dynamic dispatch algorithm into a user-level facility.

In a private conversation, Richard P. Gabriel reported that at the time Common Lisp was standardized, the committee didn't believe that method combinations were mature enough to make people implement them in one particular way (the only industrial-strength implementation available back then was in Flavors on Lisp Machines). Consequently, they intentionally under-specified them in order to leave room for experimentation. At the time, the Mop was not ready either, and only added later, sometimes with unclear or contradictory protocols.

In 2018 [15], using Sbcl[1] as a research vehicle, we analyzed the unfortunate consequences of this under-specification and exhibited a number of oddities in the design and behavior of method combinations. In particular, it turned out that method combinations weren't required to have a global name-space, meaning that every generic function could end up with its own method combination object,

[1] http://www.sbcl.org

completely disconnected from the original definition, and hence unaffected by subsequent modifications to it. It is worth mentioning that although counter-intuitive, this behavior does *not* contradict the standard. We proposed an extension to method combinations, called "method combinators", designed, amongst other things to establish proper dependencies between global method combination definitions and the associated generic functions. We were able to implement that extension in a non-intrusive, semi-portable way. This means that no modifications to SBCL's internals were needed; only a couple of calls to internal functions here and there. In addition to that, method combinators allowed us to develop an additional feature, *alternative combinators*, namely, the ability to call the same generic function with different method combinations at the same time, and at the minimum performance cost, that is, without the need to reinitialize the function every single time.

After this work, SBCL underwent a number of internal changes that fixed the reported inconsistencies. In particular, in its current state, the dependencies between generic functions and their method combinations are handled in a more intuitive fashion: generic functions are updated if their original method combination is redefined globally. Unfortunately for us, the new dependency management code is buried deep down into SBCL's internals and doesn't go through any of the official or even just suggested protocols. As a consequence, those changes broke our implementation of alternative combinators, and made it impossible to re-implement them as before, in a non-intrusive way.

In this paper, we propose yet another iteration over a possible implementation of method combinations. The paper is organized as follows. Section 2 provides an analysis of the current implementation in SBCL, emphasizing on how the dependencies between method combinations and generic functions are handled. Section 3 proposes an alternative, MOP-based implementation. This implementation conforms to the standard, so it could very well be used not only by SBCL but by all interested Lisp implementations. The design we propose also focuses on extensibility and experimentation, which was in fact the original motivation for leaving the method combinations area under-specified in the standard. Section 4 describes some additional refinements aimed at extensibility, and illustrates the benefits with a couple of examples. In particular, we revisit our former alternative combinators extension, and demonstrate that this time, it can be re-implemented in a much simpler and non-intrusive way. Finally, Section 5 provides some feedback on the performance of the proposed design.

2 METHOD COMBINATIONS IN SBCL

In this section, we analyze how post-2018 SBCL implements method combinations, and how it handles the dependencies between them and the generic functions in the system.

2.1 Method Combinations Hierarchy

The SBCL method combination classes hierarchy is depicted in Figure 1.

2.1.1 Description

The `method-combination` class is the only one mandated by the standard. The existence of a sub-hierarchy is nevertheless also a requirement, as the standard stipulates that method combination

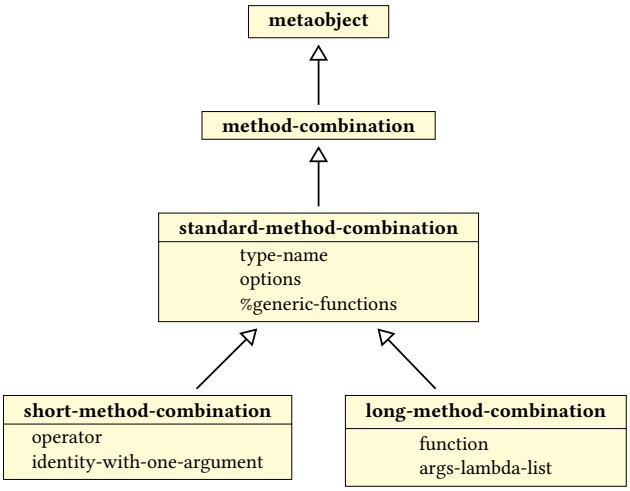


Figure 1: SBCL Method Combination Classes Hierarchy

objects be "indirect instances" of the `method-combination` class[2]; something that the MOP itself confirms by saying that this class should be "abstract".

Although, again, the standard does not require it, there is a `standard-method-combination` class, which is in fact a natural thing to provide. Indeed, it aligns the design of method combinations with the key components of CLOS which do have such a standardized equivalent: `standard-class`, `standard-generic-function`, and `standard-method` notably.

Apart from the standard method combination, every other one (that is, either built-in or user-defined) will be an instance of either the `short-method-combination`, or `long-method-combination` class.

2.1.2 Analysis

Already the case in 2018, a notable aspect of this implementation is the mixture of define-time and use-time attributes to method combinations.

The `type-name`, `operator`, `identity-with-one-argument`, and `args-lambda-list` slots represent information passed to `define-method-combination`. The `options` slot, on the other hand, holds specific sets of options passed to the `:method-combination` option in calls to `defgeneric`. As a consequence, different instances created from the same original method combination will only differ by their `options` slot.

One particular excerpt from the standard may explain this design, which is in fact that of PCL [1] rather than of SBCL itself. The following sentence appears in the description of the `method-combination` class[3].

> *A method combination object contains information about both the type of method combination and the arguments being used with that type.*

In PCL, the ability for a method combination instance to access information related to its original *type* is necessary anyway. Indeed,

[2] http://clhs.lisp.se/Body/t_meth_1.htm
[3] http://clhs.lisp.se/Body/t_meth_1.htm

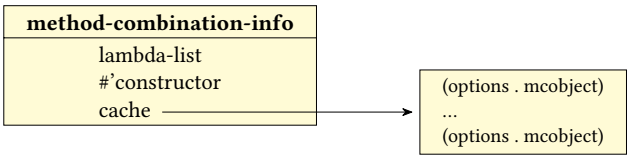


Figure 2: Sbcl Method Combination Info Structure

that information is used by the code computing effective methods when a generic function is called.

Note that even with this particular design, the information related to the method combination *type* is not exhaustive. The method combination's lambda-list is missing (it is in fact stored somewhere else), and so is the potential `:generic-function` option's value for long method combinations.

Finally, let us also mention that the `function` slot of long method combinations exists for historical reasons, but is not used anymore in Sbcl. Instead, Sbcl uses a global hash table mapping method combination names to such functions (stored in the `*long-method-combination-functions*` global variable). The functions in question are each method combination type's specific version of `compute-effective-method`, so there is indeed only one per method combination *type* (they are parameterized by the contents of the `options` slot).

2.1.3 Summary

From this analysis, it turns out that Sbcl's method combinations hierarchy, only slightly divergent from that of Pcl's, contains a mixture of information specific to every instance and information related to a method combination *definition* (in which case that information is duplicated). Two bits of information related to a method combination definition are also stored elsewhere, outside this hierarchy (the method combination's lambda list and function), and the `long-method-combination` class retains one obsolete, unused slot.

2.2 Dependency Management

One notable change in post-2018 Sbcl is a more natural handling of the dependencies between generic functions and method combinations. More specifically, method combinations have regained global name-space semantics, which means that should one of them be redefined, the generic functions using it would be notified. We now explain how this is done.

Each defined method combination is represented by an instance of a structure called `method-combination-info`, which is depicted in Figure 2. A global variable named `**method-combinations**` maintains a hash table mapping method combination names to such instances. The `lambda-list` slot stores the method combination's lambda-list. It is the one that was noted as missing from the hierarchy in Figure 1.

2.2.1 Instantiation

Every method combination info maintains a `cache` of method combination objects. When a generic function is defined to use a specific method combination with a specific set of `options`, the standard Mop function `find-method-combination` is called. If a method combination object associated with those particular options is found in the cache, it is simply returned. Otherwise, a new method combination object is created by calling the `constructor` function, and the cache is populated accordingly. Depending on the context, method combination objects will be instances of either the `short-` or `long-method-combination` classes.

2.2.2 Redefinition

Note, in Figure 1, the existence of a new slot (added post-2018 to Sbcl) named `%generic-functions` in the `standard-method-combination` class. This is how every method combination object keeps track of the generic functions using it.

When a method combination is redefined (by calling `define-method-combination` again), Sbcl updates the concerned info structure, and then traverses its cache, calling `change-class` on every method combination object. Also, for each "client" generic function in each method combination object's `%generic-functions` slot, Sbcl flushes the effective method cache and reinitializes the function by calling `reinitialize-instance`.

Note that this whole redefinition process is done in Sbcl's internals, without going through any standard (there is, in fact, none) or even just public protocols.

2.2.3 Updating

In a similar vein, when a generic function is created or updated, care is taken to add or remove it, to or from the `%generic-functions` slots in the concerned method combination objects. This time, the updating is done through a public protocol, namely, `[re]initialize-instance`.

2.2.4 Summary

Post-2018 Sbcl now handles the dependencies between method combinations and generic functions in a more intuitive way. Unfortunately, half of the dependency management code is buried in the implementation, without going through public protocols.

Note also that with the addition of the `method-combination-info` data structure, the global variable `*long-method-combination-functions*` has become superfluous. Indeed it could be replaced with an additional `function` slot in said structure, although that slot would be unused for short method combinations.

3 METHOD COMBINATIONS REVISITED

The lack of dependency management was our biggest concern in [15]. At the time, we were able to address it in a non-intrusive and extensible way. Although Sbcl's current solution works, it is buried deep down into the internals and doesn't go through any well-defined or public protocols. As a consequence, it is now impossible to continue experimenting with method combinations or providing extensions on top of them, without having to modify the language's implementation.

In this section, we suggest an alternative implementation for method combinations. In addition to proper dependency management, our implementation has the following properties.

- It remains standard-compliant.
- It retains Pcl's method combinations hierarchy (modulo some variations in the classes definitions).

- It clearly separates define-time and use-time method combination properties.
- It is grounded into the MOP. This means that it remains extensible and allows further experimentation, as was the original intent behind their under-specification, and as is, in general, the intent behind any MOP-based implementation of CLOS.

3.1 Overview

In the PCL implementation, and with the exception of the `standard` one, method combination objects are instances of one of the two built-in classes `short-` or `long-method-combination` (Figure 1). Yet, the standard consistently talks of method combination *types*[4][5], which seems to suggest that `define-method-combination` should create new *classes* of method combinations.

In addition to that, recall that `define-method-combination` comes in two forms, which means that there are in fact two *types of types of* method combinations. And so have we naturally entered the world of meta-objects.

The design we propose is thus as follows. We provide a hierarchy of method combination *types*, to distinguish between short and long ones. These are, in fact, meta-classes. `define-method-combination` is made to create a new method combination *class*, which is injected in PCL's method combinations hierarchy, and at the same time implemented as either a short or long method combination *type*. In other words, new method combination classes are sub-classes of either `short-` or `long-method-combination` as before, but are also *instances* of either `short-` or `long-method-combination-type`.

Further details are provided in the following sections. In the new hierarchies presented below, slots beginning with a percent sign contain information that is required for implementation purposes but are considered internal. Other slots are made publicly readable.

3.2 Method Combinations

Figure 3 depicts the updated method combinations hierarchy. It departs from PCL's in a number of ways.

First of all, the `standard-method-combination` class will not represent the built-in "standard method combination" anymore (there is, in fact, an ambiguity in the term). Rather, it exists as an intermediate implementation class similar to `standard-class`, `standard-generic-function`, or `standard-method`.

Also, this updated hierarchy doesn't hold any information related to the method combination *type* in use (information common to all instances). Instead, we only retain two slots: `options` (the options passed to the `:method-combination` option in calls to `defgeneric`), and `%generic-functions` (the cache of functions using this particular method combination object). As a consequence, the `short-` and `long-method-combination` classes are empty, and still exist only for specialization purposes.

3.3 Method Combination Types

Figure 4 depicts the added method combination types hierarchy; in other words, the hierarchy of method combination *meta-classes*. It

[4] http://clhs.lisp.se/Body/m_defi_4.htm
[5] http://clhs.lisp.se/Body/m_defgen.htm

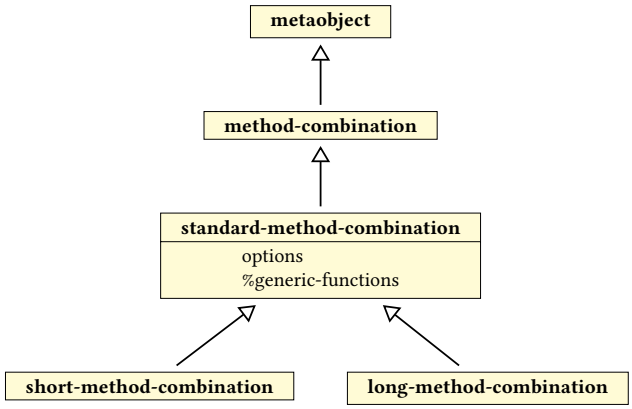


Figure 3: Method Combinations Hierarchy

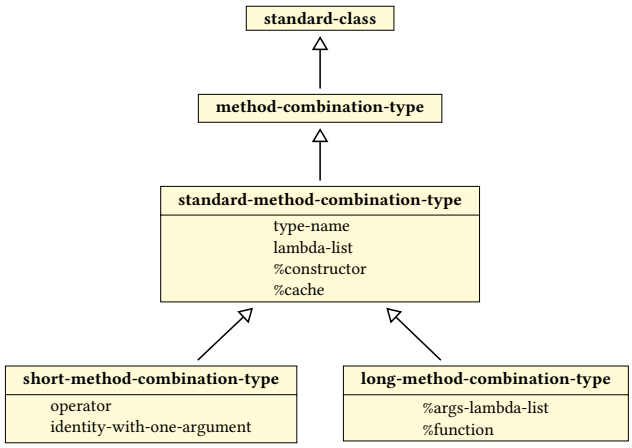


Figure 4: Method Combination Types Hierarchy

essentially serves as a replacement for SBCL's `method-combination-info` structure.

The `standard-method-combination-type` class holds the same information as the former info structure, with the addition of the method combination type's name. The former contents of the `short-` and `long-method-combination` classes, which was indeed common to all instances, is hence moved here, in the `short-` and `long-method-combination-type` classes. Note that in this new implementation, the `%function` slot will actually be used.

3.4 Standard Method Combination

As a first example of how those two hierarchies work together, let us now recreate the standard method combination. This is depicted in Figure 5.

We don't want to treat the standard method combination as an "exception" of any kind, and as mentioned before, we also want to remove any ambiguity around the term "standard" in this particular context. Because of that, the standard method combination *type*

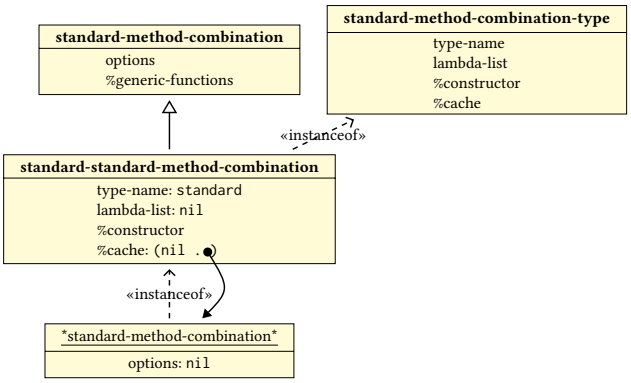


Figure 5: The Standard Method Combination

will be represented by a specific class (like any other method combination type). However, there will only ever be one standard method combination *object*, so the class in question will be a singleton one.

The standard method combination *type* is hence materialized by the singleton class `standard-standard-method-combination`. Because it is neither short nor long, it is a direct subclass of `standard-method-combination`, and it is directly implemented as a `standard-method-combination-type` for which the type name is `standard`, and the lambda-list is `nil`.

The standard method combination *object* (created by the `%constructor` function) is the only instance of that class, for which the options are also `nil`. Sbcl also happens to store that object in the global variable `*standard-method-combination*` for optimization purposes.

Finally, the `%cache` of method combination objects associates the options `nil` with the aforementioned single instance.

3.5 Built-In Method Combinations

Here we demonstrate how the built-in method combinations work as a second example. In Pcl, the built-in method combinations types are created using the short form of `define-method-combination`. Note that the creation of long method combination types works in exactly the same way. Figure 6 illustrates the effect of calling:

```
(define-method-combination progn
  :identity-with-one-argument t)
```

A new subclass of `short-method-combination` is created and implemented as a `short-method-combination-type`. The type name is `progn` (so is the `operator`), the lambda list is that of short method combinations and it falls back to identity with one argument, as specified in the call to `define-method-combination`.

Suppose now that two generic functions are created with:

```
(defgeneric gf1 (...)
  (:method-combination progn))
(defgeneric gf2 (...)
  (:method-combination progn :most-specific-last))
```

The `%constructor` function is called twice, resulting in the creation of two instances of the `progn` method combination type (`mc1` and `mc2`), each with the corresponding options. The method combination's `%cache` is populated accordingly. Finally, each method

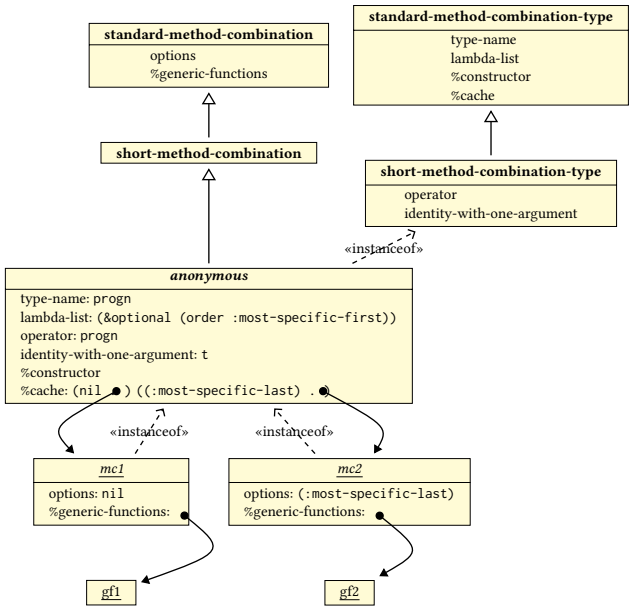


Figure 6: The `progn` Method Combination

combination object registers the concerned generic function as one if its "clients".

The reader may wonder why the `progn` method combination class is *anonymous* (especially since the standard method combination one isn't). The reason is that those classes are created automatically by the system, and as such, are not meant to be visible (even less so manipulable) by the programmer. We don't want them to "pollute" the global class name-space either. The fact that we provide a global name for `standard-standard-method-combination` is merely to facilitate the implementation. Sbcl provides three specializations on `compute-effective-method`; one on `short-method-combination`, one on `long-method-combination`, and one for the standard method combination type. Each user-defined method combination type will thus inherit automatically from one of the first two such methods. In the case of the standard method combination type, naming it explicitly (and statically) allows us to remain in the Mop's first layer (macro layer):

```
(defmethod compute-effective-method
    ((gf generic-function)
     (mc standard-standard-method-combination)
     applicable-methods)
...)
```

3.6 Implementation

We have implemented this approach in Sbcl. The resulting implementation is publicly available on Github, in a specific branch of our own Sbcl fork[6].

The implementation is in fact pretty straightforward, with the exception of one difficulty related to the bootstrapping of Clos. During that phase of the build, the Clos/Mop infrastructure is not

[6]https://github.com/didierverna/sbcl/tree/method-combination-types

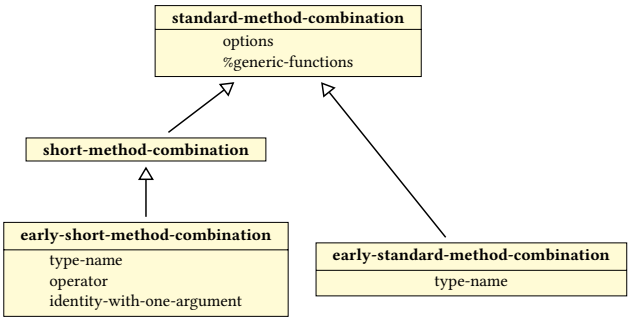


Figure 7: Early Method Combination Classes

fully available, and SBCL creates two "early" method combination objects: the `standard` and the `or` one. The difficulty is that a lot of the early CLOS code relies on those objects having the characteristics of the old infrastructure, which we have modified. In order to be as little intrusive as possible in the bootstrap, we use the following solution.

3.6.1 *Bootstrap*

We defer the creation of the method combination *types* hierarchy until *after* bootstrap. On the other hand, the updated method combinations hierarchy (Figure 3), which *is* available during bootstrap, is extended with two additional classes created specifically for the two initial method combination objects. These are `early-standard-method-combination` and `early-short-method-combination`, as depicted in Figure 7. As you can see, those two classes re-introduce the old slots that we moved around. With the appropriate accessors in place, the bootstrap code doesn't see the difference, and thus the required modifications are minimal: we just need to instantiate those "early" classes instead of the regular ones.

3.6.2 *Injection*

After bootstrap, the method combination types hierarchy is installed and the `standard` and built-in short method combination types are created. At that point, the complete new infrastructure is in place, but it is empty: we still have the two early method combination objects dangling around, and early generic functions using them.

These objects are in fact stored in two global variables named `*standard-method-combination*` and `*or-method-combination*`. Thus, it is easy for us to update the system: we transfer the generic function caches from these objects to the new ones, we update the existing generic functions to point to the new method combination objects, and we eventually re-assign the global variables to these new objects as well.

3.6.3 *Additions*

With that infrastructure in place, a number of suggestions already made in [15] can be re-implemented. In particular, we provide the following function which accesses the global method combination type name-space (analog to what `find-class` does).

```
find-method-combination-type
  (name &optional (errorp t))
"Find a NAMEd method combination type.
If ERRORP (the default), throw an error if no such
method combination type is found.
Otherwise, return NIL."
```

Also previously noted ([15, Section 2.3.1]) is the confusing nature of `find-method-combination`, which may be called with a method combination name and options (2^{nd} and 3^{rd} arguments) that do not correspond to the method combination actually in use by the generic function (1^{st} argument). Since the generic function argument to this protocol is in fact unused, it is easy to provide an alternative convenience function as follows.

```
find-method-combination*
  (name &optional options (errorp t))
"Find a method combination object for NAME and OPTIONS.
If ERRORP (the default), throw an error if no NAMEd
method combination type is found.
Otherwise, return NIL.

Note that when a NAMEd method combination type exists,
asking for a new set of (conformant) OPTIONS will
always instantiate the combination again, regardless
of the value of ERRORP."
```

4 EXTENSIBILITY

Establishing a clear distinction between the properties of method combination types and those of method combination objects certainly is a good thing from a software engineering point of view. On the other hand, it may seem overkill to introduce a meta-class hierarchy to do so. Indeed, the existence of duplicated information in the original hierarchy is not a critical problem; a simpler alternative to avoid duplication could have been the use of `:allocation :class` slots, *etc.*

What the proposed design gives back, however, is something quite valuable, especially in the general context of CLOS and the MOP, and that is *extensibility*. Recall that one of the original reasons for the general fuzziness around method combinations in the standard was to leave room for experimentation. The CLOS MOP is notoriously good at that when it provides (meta-)class hierarchies to extend, and protocols to specialize.

With the proposed design, it becomes possible to push experimentation with method combinations further, and in a less intrusive way, by extending the method combination and/or method combination types hierarchies separately.

4.1 Protocol refinements

In addition to the hierarchies proposed in the previous section, a number of refinements can be made to the current implementation to ease experimentation.

4.1.1 *Method Combination Types Redefinition*

When a method combination type is redefined, the current implementation in `sb-pcl` calls `change-class` on every concerned instance, and then reinitializes every (dependent) generic function listed in the instances caches. Here, we provide two small refinements for extensibility.

First, the code dealing with generic function reinitialization is installed in an `:after` method on `update-instance-for-different-class`. This allows potential extensions to get notified if a method combination type has changed.

Next, the code in question is wrapped in a new protocol named `u-g-f-f-r-m-c`[7], a protocol that was already proposed in [15]. This, in turn, allows potential extensions to generic functions to be notified when their method combination is updated.

4.1.2 *Method Combination Types Definition*

Finally, there is a simple way to allow extensions to seamlessly plug sub-classes of method combination (types) into the system. According to the Common Lisp standard (in particular Section 1.6 Language Extensions[8]), it is permissible to add new keyword arguments to functions or macros, provided that "they do not alter the behavior of conforming code and provided they are not explicitly prohibited [...]". Consequently, we can extend `define-method-combination` in the following manner (granted, this is just a macro so it wouldn't be difficult to provide a different one instead).

The short form is made to understand two additional options, the meaning of which should be self-explanatory.

```
:method-combination-class name
:method-combination-type-class name
```

or

```
:method-combination-type-class (name initargs*)
```

Similarly, the long form is made to recognize those as well, provided that they appear in that order, and only *after* the `:arguments` and `:generic-function` options when present.

```
(:method-combination-class name)
(:method-combination-type-class name initargs*)
```

Of course, care is taken to verify that when provided, the alternative classes make sense in the present context, and with each other (*e.g.* only sub-classes of `short-method-combination[-type]` are authorized in the short form, *etc.*).

We now provide two examples making use of this kind of extensibility. The complete code is available on Github[9] and requires the aforementioned fork of Sbcl to work.

4.2 Medium Method Combinations

We want to define "medium" method combinations, that is, method combinations behaving like short ones, but also equipped with `:before` and `:after` methods, and which do not request the qualification of primary methods. Of course, these may be defined as regular long method combinations (*all* method combinations can). However, we may want to keep the short-style `operator` and `identity-with-one-argument` properties around, for information purposes (*e.g.* specializing `print-object` or producing detailed reference manuals with Declt[10]).

We hence provide a new method combination type class, as depicted in Figure 8. A new convenience macro could be used as below:

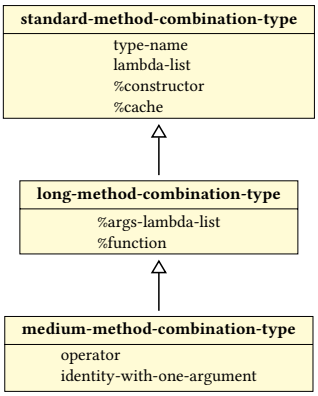


Figure 8: The Medium Method Combination Type Class

```
(define-medium-method-combination-type myprogn
  :operator progn :identity-with-one-argument t)
```

which, in turn, will expand to this:

```
(define-method-combination myprogn
    (&optional (order :most-specific-first))
  ((around (:around))
   (before (:before))
   (primary () :order order :required t)
   (after (:after)))
   (:method-combination-type-class
       medium-method-combination-type
     :operator progn :identity-with-one-argument t)
  ...)
```

Because this new method combination type is fully integrated into the original hierarchy, nothing else is required for it to work. In particular, Sbcl's original specialization on `compute-effective-method` for long method combinations remains applicable here.

4.3 Alternative Method Combinations

Alternative method combinations have been proposed and described in [15, Section 6]. In short, the idea is to be able to call the same generic function with different method combinations efficiently (meaning, without having to reinitialize it at every call), simply by maintaining a cache of discriminating functions.

Just as a quick reminder of a potential use-case, assume that access to alternative calls is provided through a reader-macro such as this one: `#!combination(func arg1 arg2 ...)`. It may be convenient, depending on the context, to vary the calls to a function at minimal cost like this:

```
#!append(func arg1 arg2 ...)
#!nconc(func arg1 arg2 ...)
```

Given the new method combination architecture proposed in Section 3, the implementation of this idea is not only straightforward, but also much simpler than that of 2018. In fact, we don't even need to extend the method combination type hierarchy anymore; only the generic functions one.

[7] `update-generic-function-for-redefined-method-combination`
[8] http://clhs.lisp.se/Body/01_f.htm
[9] https://github.com/didierverna/ELS2023-method-combinations
[10] https://www.lrde.epita.fr/~didier/software/lisp/typesetting.php#Declt

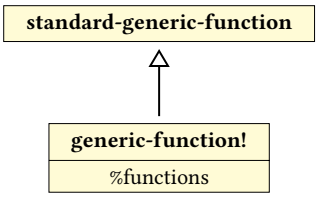


Figure 9: Extended Generic Functions

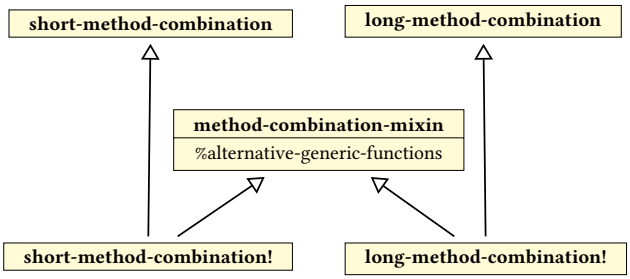


Figure 10: Extended Method Combinations

4.3.1 *Alternative Calls*

We provide a new class of generic functions, as depicted in Figure 9. The `%functions` slot implements the discriminating functions cache. It is a hash table mapping method combination objects to discriminating functions. When a generic function is called with an alternative method combination, it is reinitialized with that method combination, called, and the resulting discriminating function is cached. The function is then switched back to its original method combination.

Note that we also arrange for the two method combination objects (the original and the alternative one) to reference the generic function in their respective cache. This is why we don't need to extend the method combinations hierarchy anymore, but this means that the caches in question contain a mixture of generic functions using the method combination as their "primary" one, and others using it as an alternative one.

Another possible implementation would be to maintain separate caches for primary and alternative generic functions, in which case the method combinations hierarchy (*not* the method combination types one) would need to be extended as depicted in Figure 10.

4.3.2 *Generic Function Modification*

New `:after` methods are installed on `add-method` and `remove-method` to clear the discriminating functions cache in case the generic function is modified.

4.3.3 *Method Combination Change*

An `:around` method on `reinitialize-instance` is installed in order to intercept a method combination change to the generic function. On top of the normal behavior, if the new method combination was previously used as an alternative one for this generic function, both the generic function's discriminating functions cache, and the method combination's generic functions cache are updated.

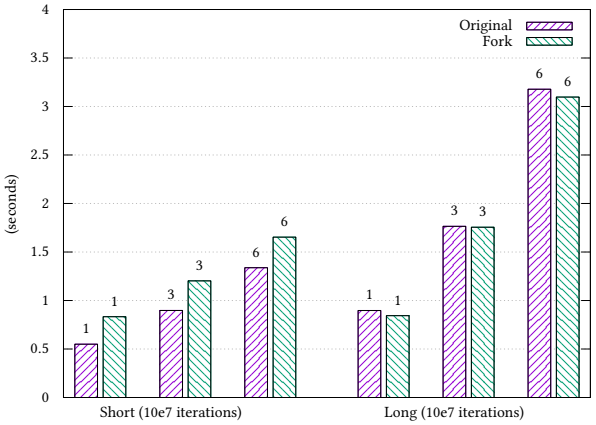


Figure 11: `compute-effective-method` Performance

4.3.4 *Method Combination Redefinition*

Finally, a new method on `u-g-f-f-r-m-c`[11], one of our newly proposed protocols, is installed. This method simply detects the redefinition of a method combination that was used as an alternative one, and invalidates the cached discriminating function associated with it.

5 PERFORMANCE

In this section, we study the impact of our proposed architectural changes on the performance of the system. We are *not* interested in benchmarking the creation or modification of method combinations, since that is bound to happen very rarely. Rather, the impact of the method combinations implementation is likely to be visible where they are *used*, that is, when effective methods are computed. In other words, we are interested in the performance of `compute-effective-method`.

In Sbcl, there are three different cases.

(1) The case of the standard method combination is completely hardwired, so regardless of its implementation, the performance will be exactly the same.
(2) The case of short method combinations is handled by a *single* function, `short-compute-effective-method`, but this function accesses properties specific to the method combination *types* every time it is called (type name, operator, identity with one argument).
(3) Finally, the case of long method combinations is handled by calling a function that is specific to each method combination type.

In order to roughly evaluate the consequences of our proposed architecture in terms of performance, we timed the execution of `compute-effective-method` (ten million calls in a row) on one generic function using a short method combination, and another one using a long method combination, each time with one, three, and six applicable methods. Those tests were run on a regular Sbcl as well as on our forked version. The code is available in the aforementioned Github repository, and the results are presented in

[11] `update-generic-function-for-redefined-method-combination`

Figure 11. The number of applicable methods is indicated on top of each bar.

5.1 Short Method Combinations

In the case of short method combinations, we observe a degradation ranging from 50% to 22% (the degradation decreasing as the number of applicable methods increases). This may be explained as follows.

In the original version of Sbcl, `short-compute-effective-method` retrieves three method combination properties (type name, operator, and identity with one argument) through regular accessors to the slots depicted in Figure 1.

In our new architecture, those properties belong to the method combination *type* rather than to the method combination *object*. As visible in Figure 6, accessing those properties from a method combination object hence requires an additional call to `class-of`. In other words, we are comparing `(accessor mc-object)` with `(accessor (class-of mc-object))`. Also, because the number of such accesses remains constant (exactly three), it is not surprising that the impact on performance decreases when the number of applicable methods increases: it just means that it takes longer to execute the function with more applicable methods.

Even though such a degradation may seem important, we still don't think it matters that much, in the sense that effective methods are not computed very frequently (thanks to caching); only when the set of applicable methods varies. And if it does matter, it is always possible to duplicate that information back into the method combination objects themselves, without sacrificing the new architecture.

5.2 Long Method Combinations

In the case of long method combinations on the other hand, we observe an *improvement* ranging from 7% to 3% (also less important as the number of applicable methods increases), which may or may not be considered significant. Again, this may be explained as follows.

In the original version of Sbcl, `compute-effective-method` retrieves the method combination "function" (in charge of actually computing the effective method in a way specific to that particular method combination type) from a hash table (see Section 2.1.2).

In our new architecture (see Figure 4), that function is stored in the method combination *type* itself, that is, in the *implementation* of the method combination object. So here this time, we are comparing a hash table lookup with `(accessor (class-of mc-object))`.

6 CONCLUSION AND PERSPECTIVES

Method combinations are an extremely powerful, yet somewhat obscure part of Clos. The arguable complexity of `define-method-combination`'s long form is a probable obstacle to a more widespread use, and their under-specification doesn't facilitate experimentation, as there is almost no official protocol that Lisp implementations need to conform to.

In this paper, we have proposed a Mop-based implementation for method combinations. We believe that this implementation presents a number of advantages compared to vendor-specific solutions. First, it reifies the notion of method combination *type*, which, in fact, seems quite a natural thing to do upon careful reading of the Common Lisp standard. It is also standard-compliant, which gives us hope that it would trigger some general interest across the existing Lisp implementations. It remains close to Pcl's original design, notably by maintaining a hierarchy distinguishing short and long method combinations (it merely adds to it). Finally, and this is probably the strongest point, it makes the method combination infrastructure extensible, which really is the philosophy behind the Mop and puts this area of Clos back in line with the rest of it. The proposed architecture as been implemented in an Sbcl fork and is publicly available.

In Section 4, we have presented several simple examples demonstrating how we can benefit from an extensible design for further experimentation. There are still a number of things that we plan to work on. One of them is turning the `:description` option of the long form's method group specifiers into something useful for documentation purposes (Declt would like very much to have that). The Pcl implementation seems to ignore that option completely. Arguably, this would not be done as an extension but rather in the core of the architecture.

Another potential area of research is to extend method combinations to forms that would be neither short, nor long. Provided with an alternative to the `define-method-combination` macro which can't be used anymore, the proposed architecture makes it easy to do so: one simply has to sub-class both `standard-method-combination` and `standard-method-combination-type`, and provide additional methods on `compute-effective-method`.

Granted, every possible method combination type can be created with the long form of `define-method-combination`, since it is always possible to provide a single method group matching *all* applicable methods (using `*` as the pattern), and do everything in the method combination's *form*. So the question is not whether it is possible to do it, but rather how easy it is. We can already think of several investigation routes to ease the creation of complex method combinations: alternative semantics for the long form's method groups, such as the ability to re-use the same method in multiple groups or to specify qualifier patterns with regular expressions.

More generally, every time a method combination type is neither as simple as a short one nor as general as a long one (the "medium" type from Section 4.2 falls into that category), there might be a gain in defining it as an *intermediate* form, with a tailored method combination function, leading eventually to improving the performance of `compute-effective-method`.

REFERENCES

[1] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. *SIGPLAN Notices*, 21(11):17–29, June 1986. ISSN 0362-1340. doi: 10.1145/960112.28700. URL http://doi.acm.org/10.1145/960112.28700.

[2] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *ACM SIGPLAN Notices*, 23(SI):1–142, 1988. ISSN 0362-1340.

[3] Giuseppe Castagna. *Object-Oriented Programming, A Unified Foundation*. Progress in Theoretical Computer Science. Birkhäuser Boston, 2012. ISBN 9781461241386.

[4] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *SIGPLAN Lisp Pointers*, 5(1):182–192, January 1992. ISSN 1045-3563. doi: 10.1145/141478.141537. URL http://doi.acm.org/10.1145/141478.141537.

[5] Linda G. DeMichiel and Richard P. Gabriel. The common lisp object system: An overview. In *European Conference on Object Oriented Programming*, pages 151–170, 1987.

[6] Richard P. Gabriel, Jon L. White, and Daniel G. Bobrow. Clos: integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, 1991. ISSN 0001-0782.

[7] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-61622-X.

[8] Sonja E. Keene. *Object-Oriented Programming in Common Lisp: a Programmer's Guide to CLOS*. Addison-Wesley, 1989. ISBN 0-20117-589-4.

[9] Gregor J. Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[10] Patty Maes. Concepts and experiments in computational reflection. In *OOPSLA*. ACM, December 1987.

[11] Andreas Paepcke. User-level language crafting – introducing the CLOS metaobject protocol. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press, 1993. Downloadable version at http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps.

[12] Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer Berlin / Heidelberg, 2001. ISBN 978-3-540-42558-8.

[13] Brian C. Smith. Reflection and semantics in Lisp. In *Symposium on Principles of Programming Languages*, pages 23–35. ACM, 1984.

[14] ANSI. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[15] Didier Verna. Method combinators. In *11th European Lisp Symposium*, pages 32–41, Marbella, Spain, April 2018. ISBN 9782955747421. doi: 10.5281/zenodo.3247610.

A Minimal Run-time Overhead Metaobject Protocol for Julia

Marcelo Santos
Instituto Superior Técnico
Lisbon, Portugal
marcelocmsantos@tecnico.ulisboa.pt

António Menezes Leitão
INESC-ID/Instituto Superior Técnico
Lisbon, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

ABSTRACT

Metaobject Protocols enable programmers to extend programming languages without the need to understand the lower level details of their implementation. However, designing these protocols presents two challenges: allowing programmers to limit their concerns to higher-level concepts and minimizing performance penalties in programs. In this work, we propose a metaobject protocol for the Julia programming language. Julia does not follow the traditional object-oriented paradigm. However, Julia's compilation approach allows for a considerable degree of code optimization through the exploration of run-time type information. Through the usage of Julia's run-time optimizations, it becomes possible to implement a metaobject protocol that combines user-extensibility with minimal performance penalties in run-time. This paper focuses on the development of a multiple inheritance method dispatch and method combination mechanisms with minimal run-time overhead.

CCS CONCEPTS

• **Software and its engineering → Software notations and tools → General programming languages → Language features**;

KEYWORDS

Julia,Metaobject Protocols,Object-Oriented Programming,Performance

ACM Reference Format:
Marcelo Santos and António Menezes Leitão. 2023. A Minimal Run-time Overhead Metaobject Protocol for Julia. In *Proceedings of the 16th European Lisp Symposium (ELS'23).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.7823097

1 INTRODUCTION

Traditionally, there has been a separation between programmers and language designers. Programmers treat languages as black boxes with pre-defined rules, as established by the language designers. In this model, the semantics of a language is seen as unchangeable, thus imposing limits to its expressiveness.

Metaobject Protocols (MOPs) blur the distinction between programmers and language designers, by providing programmers with an interface to modify the language. By treating the language itself as a mutable object-oriented program, one can alter the semantics and introduce new behaviours in the language through the abstractions made available by the object-oriented (OO) paradigm. With the use of MOPs, one can define a different language semantics specific for only a part of the program and leave the default semantics for the rest of the program. It also becomes fairly straightforward to add new behaviour to existing code without directly modifying it by extending the semantics of the language. This extension is accomplished by the same inheritance mechanisms present in the OO paradigm, but applied to metaclasses. With MOPs, the distinction between object behaviour and language semantics becomes evident, allowing for a better separation of concerns.

A prime example on the implementation of MOPs is the one present in the Common Lisp Object System (CLOS) [8].

1.1 The Julia Programming Language

One of the main challenges faced in the implementation of the CLOS metaobject protocol was guaranteeing good performance alongside the flexibility of the MOP [8]. This is a recurring problem in the implementation of higher-level languages, known as the Paradox of High-Level Languages [8]. The main premise of high-level languages is that they allow programmers to better formulate what their programs do through more expressive methodologies. Consequently, compilers for these languages should be able to exploit this knowledge and output faster programs than their low-level counterparts. Unfortunately, the reality is that there are concepts which are, in fact, not being expressed more clearly, instead requiring programmers to provide sufficient detail to enable efficient execution.

In the most recent years we have seen a fast growth in the popularity of languages featuring object-oriented and dynamic properties [5, 10]. Python is one of these languages, being extensively used in the fields of numerical and scientific computing, which often require large-scale computations. However, one of the major complaints regarding Python is that it is slow [11]. To resolve that problem, a few strategies have been applied:

- Creating libraries that rely on faster languages to process more intensive operations (e.g., Numpy, a numerical computing library for python, is written in C and C++);
- Using a more efficient compiler/interpreter implementation (e.g., Pypy and Numba);
- Prototyping in this high-level language and then translating the code to a more performant and often lower level language like C, C++, or Fortran. This last route is the one yielding the best results performance-wise, but it falls short for relying on a dichotomy of languages, requiring more knowledge and work from the programmer.

The Julia programming language [2], a dynamic language with a focus on performance, tries to solve this last problem. By employing strategies like JIT compilation and code specialization on run-time

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

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.7823097

types, it achieves very good performance when compared to other dynamic languages like Python [1].

Julia adopted some of the distinguishing features of Common Lisp, particularly, generic functions and methods. Multiple dispatch - the dynamic dispatch of methods based on the run-time information of all the arguments - is used extensively in Julia and it allows for extending the language by creating more methods for pre-defined generic functions.

Julia features a very basic object system. There is no inheritance, subtyping can only be accomplished from abstract types and multiple subtyping is not supported.

1.2 Objectives

There are multiple examples of OO languages that resulted from the development of Object Systems atop existing languages. The Lisp community saw the development of CommonLoops [3], which combined Lisp's procedure-oriented paradigm with OO programming. The same evolutionary strategy was applied in Objective-C, which stemmed from the C language and became a prime example of an OO language by just adding a small number of syntactic features taken from the Smalltalk language, while keeping compatibility with the remaining aspects of C [7].

In this work, we focus on the Julia Programming Language and we propose JOS - the Julia Object System - a CLOS-inspired object system, alongside a meta-object protocol designed to have minimal run-time overhead.

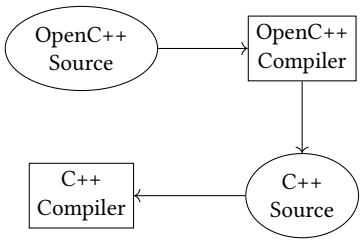
2 RELATED WORK

2.1 Metaobject Protocol Implementations

Although there are languages that leverage MOPs to extend their functionality, we will only focus on the most expressive one, the CLOS MOP, and one of the most efficient ones, the OpenC++ MOP. One of the main problems of the CLOS MOP is the imposed performance hit. The cause for this problem is the existence of metaobjects at run-time, which are needed to change the semantics of the language. Although this allows for greater flexibility, it increases the level of additional run-time logic.

2.1.1 CLOS. The motivation for the development of the CLOS MOP came from the need to give developers the ability to modify the language from a high-level perspective, i.e., without low-level knowledge of the inner workings of the language. CLOS provides ways to incrementally change the behaviour of fundamental language constructs. This is possible through the reification of elements such as classes, generic functions, and methods. These elements are said to be metaobjects and their exposure and consequent modification is what allows for the manifestation of changes to the language semantics. As is often the case, applying global changes to an already existing system without proper testing can lead to unforeseen results. Furthermore, a programmer's intent might be to just change a portion of the language for a specific section of the program and not its entirety. This problem is solved by CLOS by exposing metaobjects as part of an object-oriented hierarchy capable of being extended. As such, one can look at the semantics of the language as an object-oriented program as well.

2.1.2 OpenC++.
OpenC++[6] is a metaobject protocol extension to C++[12] similar to CLOS but with a very different approach. It moves all the logic of the MOP to compile-time. The purpose of this move is to incur zero run-time speed or space overhead, while still allowing the compiler to perform optimizations. The basic system architecture is as follows:



The OpenC++ compiler generates the metaobjects responsible for executing the protocol when compiling OpenC++ to C++. These metaobjects intercept the parsing of regular C++ entities (e.g., class definitions, member access, object creation, virtual function invocation) and take control of their compilation. In essence, these metaobjects modify the program's semantics through the manipulation of parse trees. The resulting modifications are then passed as a regular C++ source to a C++ compiler which has no knowledge of the MOP. With this strategy, no metaobjects exist at run-time, thus saving time and memory. One big disadvantage of this approach is that, since the protocol only acts at compile-time, it becomes impossible to apply the same changes at run-time. Another minor problem is the increase of time and complexity of compilation.

The solution of OpenC++, which tries to solve the performance issues from metaobject protocols, seems to hint the existence of a tradeoff between execution speed and flexibility, similar to the aforementioned Paradox of High-Level Languages.

2.2 Julia

In this section, we present features of Julia that we consider relevant for our proposal. These include patterns that allow us to build an object system and performance optimizations capable of removing most of the overhead of our implementation.

Julia provides the concepts of generic functions and methods just like CLOS[4]. Generic functions are implicitly declared through the definition of methods.

We now concern ourselves with method specialization, the mechanism employed by Julia which, combined with JIT compilation, generates compiled code and caches it based on the types of the arguments.

We can observe in Listing 1 the definition of a method `g` taking an argument of any type. Afterwards we employ the `@code_llvm` macro to expose the generated LLVM [9] low-level code for two different calls to the method: one passing an argument of an `Int64` type and another passing a `Float64`. Note that Julia applies a JIT strategy to compile, at run-time, type-specific versions of a method according to its arguments.

Listing 1: Specialization of generic function

```
julia> g(x) = x + 1

julia> @code_llvm g(1)
define i64 @julia_g_167(i64 signext %0) {
  %1 = add i64 %0, 1
  ret i64 %1
}

julia> @code_llvm g(1.0)
define double @julia_g_186(double %0) {
  %1 = fadd double %0, 1.000000e+00
  ret double %1
}
```

2.2.1 *Method redefinition.*

Immutability, which Julia takes advantage of, is an important factor when it comes to compiler optimization. The more structures that are guaranteed to stay constant, the more optimizations the compiler can perform. A function accessing variables outside of its scope will generate completely different assemblies depending on their mutability. The immutability of our system would guarantee better performance, but given the nature of run-time metaobject systems, the opposite is expected. The ability to change program structure at run-time is one of the main benefits of using MOPs. We can approach this by exploiting another of Julia's features: method redefinition.

The redefinition of a method changes its behaviour and triggers the recompilation of methods which depend on it. Redefinition allows for behaviour changes without relying on data structures to hold mutable data. This also provides fast access to data without compromising flexibility.

2.3 Problem

This section has shown two Metaobject Protocol systems, CLOS and OpenC++, and exposed a dilemma between flexibility and performance. The nature of these two implementations focuses on the time-frame in which metaobjects exist: run-time in the case of CLOS or compile-time in the case of OpenC++. Our goal is to bridge the gap between these two approaches and create a performant run-time metaobject protocol on top of the Julia language.

3 SOLUTION

In this section we will describe our proposal for an object system for the Julia language that supports multiple-inheritance, method dispatch, and method combination with minimal run-time overhead, without changing the language implementation or semantics.

3.1 Class Definition

We begin by describing our class definition mechanism. One can define a new class by calling the `@defclass` macro. It takes the name of the class to be defined and optionally a list of superclasses. In Listing 2, we define four classes: `A`, which has no explicit superclasses, `B` and `C`, both inheriting from `A`, and `D` which inherits from `B` and `C`. For this example, the macro is responsible for ensuring the following:

Listing 2: Class definition

```
1 @defclass A
2 @defclass B (A,)
3 @defclass C (A,)
4 @defclass D (B, C)
```

- Create a Julia structure and a Julia abstract type for the class.
- Define the default constructor for the class.
- Define the method `superclasses` which takes the Julia abstract type for the class and returns the types corresponding to the direct superclasses. If the class is defined without explicit superclasses, its only superclass is `Any`, the type in Julia of which every type is a subtype.
- Define the method `preclist` which takes the Julia type for the class and returns the precedence list for the class. A class precedence list is an ordered list of classes which determines specificity. By default this order is determined by applying a topological sort to the class hierarchy, similar to CLOS.
- Define the method `classof` that, from an instance of the class, returns the class itself. This method is analogous to Julia's `typeof`.

We present in Listing 3 some examples of the application of these methods in the classes defined in Listing 2:

- In line 1, we show how to call the default constructor. We employ a calling convention similar to the one used in Julia for instantiating structs.
- In line 2, we retrieve the class of the previously instantiated object.
- In line 5, the similarity between our system's classes and the language's types is exposed. Classes are also types in the eyes of Julia.
- In line 8, even though we did not declare any superclasses for `A`, we observe that `Any` is implicitly declared as a superclass. This is consistent with Julia, where a type declared without a supertype has `Any` as its supertype.
- In line 12, we show the explicitly declared direct superclasses of `D`.
- Finally, in line 15, we see the precedence list of `D`, which includes itself up to `Any`.

Note that these mechanisms for retrieving class information do not rely on mutable data structures or constants because:

- Although constants allow for the optimization of generated code, they forbid future changes, which goes against one of the purposes of metaobject protocols.
- Even though mutable data structures would allow for changes in the class system, they prevent the same optimizations permitted by the use of constants.

Methods whose only purpose is to return data give us the best of both worlds: mutability and optimizations. The only cost for mutability is recompilation time.

Listing 3: Class information methods

```
> a = A()
> classof(a)
A

> typeof(classof(a))
DataType

> superclasses(A)
(Any,)

> d = D()
> superclasses(classof(d))
(B, C)

> preclist(classof(d))
(D, B, C, A, Any)
```

Listing 4: Slots definition

```
> @defclass A () (a, b, c)

> obj = A(a=1, b=2, c=3)

> obj.a + obj.b + obj.c
6
```

3.1.1 Slots. In the beginning of this section, we presented class definition. We provided simple examples without concerning ourselves with the allocation of data. As such, in this subsection, we will describe the definition of slots, which hold data for objects. In Listing 4, we define a class `A` with three slots, `a`, `b`, and `c`. We observe that we now have a constructor capable of taking three arguments, one for each slot. Upon creating the instance of `A`, we can access each slot by using the familiar syntax in Julia for `struct` field access.

One limitation of Julia is that `structs`, once defined, cannot be redefined to have a different layout. However, in our object system, just like in CLOS, we want to support the redefinition of classes. In Listing 5, we define a class `X` with one slot, `a`. We also create an instance of `X`, `x1`. We then redefine `X` to add a new slot `b`. We are then able to create a new instance of `X`, `x2`, containing slots `a` and `b`. We verify that both `x1` and `x2` are instances of the same class, although with a slight difference. `x1` is an instance of an `old` version of `X`, meaning that it does not have the new slot, as we can see in the error message (the message will become clearer in subsubsection 3.1.3). Although the underlying `struct` of the instances is different, both are accepted in methods taking `X` as an argument. It is the user's responsibility to ensure the compatibility between both versions of `X`. This is the default behaviour for class redefinition.

3.1.2 Inheritance. We now proceed to show how to define classes using inheritance. For the next example in Listing 6, we define three classes, each with one slot: `A` with slot `a`, `B` with slot `b`, and `C` with slot `c`. Note that, due to inheritance, each class also contains the slots defined in its superclasses: `B` has slots `a` and `b`, and `C` has slots `a`, `b`, and `c`.

Listing 5: Class redefinition

```
> @defclass X () (a)
> x1 = X(a=1)
> @defclass X () (a, b)
> x2 = X(a=1, b=2)
> classof(x1)
X
> classof(x2)
X
> x2.b
2
> x1.b
ERROR: type X__v1 has no field b
```

Listing 6: Simple slot inheritance

```
@defclass A () (a)
@defclass B (A,) (b)
@defclass C (B,) (c)

> objA = A(a=1)

> objB = B(a=1, b=2)

> objC = C(a=1, b=2, c=3)
```

Listing 7: Multiple inheritance

```
@defclass A () (a=1)
@defclass B () (a=2)
@defclass C (B,A) (c=3)

> preclist(C)
(C, B, A, Any)

> objC = C()

> objC.a
2
```

Our proposal also addresses multiple inheritance, using the precedence list to resolve ambiguities.

In Listing 7, we observe a class `C` with direct superclasses `B` and `A`. Because `B` is declared as a superclass before `A` it shows up first in `C`'s precedence list. We notice that both `A` and `B` define `a` as a slot with a distinct default value for each. Our disambiguation rule picks `B` as the defining class for the default value of `a` when instantiating `C`.

3.1.3 Inside `@defclass`. We now explain what the `@defclass` macro and the declared constructors do at a lower level. The approach that we chose to pursue tries to take as much advantage as possible of the language's type system and native direct data storage facilities.

Listing 8: Struct versions

```
@defclass A () (a::Int64, b::String)
@defclass A () (a::Int64, b::Int64)

# Defined types:
abstract type A end

struct A__v1 <: A
  a::Int64
  b::String
end

struct A__v2 <: A
  a::Int64
  b::Int64
end
```

Listing 9: `classof` and `typeof` MOP class

```
> @defclass A
A
> f(::Type{Type{A}}) = 1
> f(classof(A))
1
> f(typeof(A))
1
```

Listing 10: Defining and calling methods (following Listing 2)

```
> @defmethod bar(a::A) = 0
> bar(D())
0
> @defmethod bar(b::B) = 1
> @defmethod bar(c::C) = 2
> bar(D())
1
```

Julia allows the definition of `structs`, in a similar way to what class definitions allow in object-oriented languages. Although structs allow for fast data access, they have several drawbacks as we have seen previously (e.g., definition immutability and lack of inheritance). A naive implementation might maintain the use of `structs` to hold data, but adding a layer above them to implement the desired functionalities. One example would be storing all slot data in a hashtable held by a `struct`. This would easily support all features, but would also remove the speed advantages of the direct memory access provided by `structs`.

To preserve performance, a class definition should be equivalent to a `struct` definition. Our implementation does exactly that: a `@defclass` form expands into a `struct` with the fields as the slots. However, to allow future redefinitions of the class, each expansion generated a different version of the `struct`.

Following the example in Listing 8, we demonstrate that for each time we call the `@defclass` macro, we define a new struct associated to the class with the version number appended to its name. In order to maintain some degree of compatibility between each struct version and the actual class `A`, we define an abstract type with the same name as the class stated in the macro and declare each struct version as its subtype.

Each time the user tries to instantiate `A`, the constructor returns an instance of the `struct` with the highest version number. Consequently, previously defined instances might be incompatible with newer instances. With the goal of eliminating indirection in favor of performance, we leave the responsibility of updating older instances to the user.

3.1.4 Metaclasses. For the implementation of this metaobject protocol, we follow a strategy similar to that chosen in CLOS when it comes to metaclasses.

3.2 Method Dispatch

We now describe how we use Julia's language features to build a multiple-inheritance method dispatch mechanism. We provide the `@defmethod` macro, that, just like regular Julia method definitions, takes a method name, the list of parameters and optionally their types, and the method body, as we can see in Listing 10.

This macro is responsible for:

- Storing an anonymous function that takes generic arguments with the same method body as the one passed to `@defmethod` macro. This is stored in a key-value manner, in which the key is the list of the types of the parameters and the value is the anonymous function.
- Creating a julia method, the method computer, accepting the same number of arguments as the one specified with `@defmethod` whose purpose is to select the appropriate method to apply given the types of the arguments. If the method computer already exists, it is redefined to include the update of the available anonymous functions.

3.2.1 The Method Computer. The method computer is where most of the dispatch work takes place. It executes the following steps:

- Gather a list of the types of the arguments supplied to the method.
- Get the list of methods whose parameters are compatible with the arguments.
- Sort this list of methods by specificity.
- Call the most specific method, passing it the arguments received and returning its return value.

We say that a list of parameters P is compatible with a list of arguments A if:

- The length of P is equal to the length of A.
- For each pair (P_k, A_k) where P_k is the k^{th} parameter and A_k is the k^{th} argument, the class of P_k is in the precedence list of the class of A_k.

A method M is more specific than some method N, with respect to a list of arguments A if M has the smallest k for which M_{P_k} comes before N_{P_k} in the precedence list of A_k, where M_{P_k} is the k^{th} element of the parameters of M, N_{P_k} is the k^{th} element of the parameters of N, and A_k is the k^{th} argument.

Listing 11: Method combination

```
> @defmethod foo(d::D) =
    println("Primary method")
> foo(D())
Primary method
> @defmethod foo::before(d::D) =
    println("Calling before")
> @defmethod foo::after(d::D) =
    println("Calling after")
> foo(D())
Calling before
Primary method
Calling after
```

Listing 12: Joining anonymous functions with join_lambdas.

```
function join_lambdas(around::Tuple, b::Tuple,
                      p::Tuple, a::Tuple)
  (x...) -> begin
    next = join_lambdas(around[2:end], b, p, a)
    callnextmethod() = next(x...)
    callnextmethod(y...) = next(y...)
    hasnextmethod() = true
    around[1](x...,
              callnextmethod,
                   hasnextmethod)
  end
end
```

3.3 Method Combination

Currently we support the same method combination implemented by the CLOS Standard Method Combination, i.e., before, after and around methods. See Listing 11 for an example.

Besides selecting the most specific method, the method computer must also take into account method combination. To do so, it must:

- Separate the method lists into four groups: before, after, around and primary. Primary methods are those defined without any method combination specifier.
- Apply the same filtering and ordering logic from the arguments for each group.
- Generate and call an effective method, which results from applying all valid methods from the combination.

3.3.1 Computing the Effective Method. The effective method is an anonymous function returned by the recursive method `join_lambdas`. `join_lambdas` receives four arguments, one for each method group. Each call to `join_lambdas` attaches one method from one group, while processing one group at a time. The order for processing groups is the same as the one specified by CLOS' method combination semantics: around, before, primary, and after.

The method `join_lambdas` has a different way of joining methods depending on the group it is currently processing:

- While processing a method from the around group, it returns an anonymous function which calls the method being processed. Besides receiving the arguments passed to the effective method, the method being processed also receives two functions as arguments: `callnextmethod` and `hasnextmethod`. The former encodes the next recursive call to `join_lambdas` and passes the same arguments to the next call if called without arguments or override them if called with arguments. The latter returns true if there are more methods following in the combination and false otherwise. The `join_lambdas` method for processing the around group is shown in Listing 12.
- Processing the before group is much simpler. The only concern of the returned anonymous function is calling the current method and recurring into the `join_lambdas` call.
- Handling the primary group is very similar to the around group, given that it must allow calling `hasnextmethod` and `callnextmethod`. The biggest difference from the around group processing is storing the return value from the call and returning it only after the `after` group.
- Processing the after group is identical to what is done with the before group.

We follow these steps instead of an iterative method-calling approach because the JIT compilation approach can better optimize method call chains. The resulting effective method generated for the `foo` call in Listing 11 can be seen in Listing 13.

Listing 13: Resulting effective method of `foo` from Listing 11 (simplified)

```
(x1...) -> begin
  ((d) -> println("Calling before"))(x1...)

  ((x2...) -> begin
     res = ((d) ->
       println("Primary method"))(x2...)

     ((x3...) -> begin
        ((d) ->
           println("Calling after"))(x3...)
      end)(x2...)

     return res
   end)(x1...)
end
```

3.4 Metaobject Protocols

With an object system and method combination mechanism in place, we are now able to implement new metaobject protocols.

In Listing 14, we create a new class `SpecialClass` subclass of `StandardClass`. By default every class metaobject in our system is an instance of `StandardClass`. Afterwards, we define class `A` as an instance of `SpecialClass`. A call to a class constructor results in a call to the `makeinstance` method, passing the class and its arguments. For this example we define a method combination for `makeinstance`, specifically for the metaclass `SpecialClass`, to print a message before instantiation. The default behaviour for instantiation is still preserved, as we can see in the last line, and the message is displayed before the object is returned.

Listing 14: Makeinstance protocol

```
> @defclass SpecialClass (StandardClass,)
> @defclass A () () SpecialClass
> @defmethod makeinstance::before(
            class::SpecialClass, args) begin
    println("Creating an instance of ", class)
end
> A()
Creating an instance of A
A
```

Listing 15: Setslot protocol

```
> @defclass SpecialClass (StandardClass,)
> @defclass A () (a=0) SpecialClass
> a1 = A()
> @defmethod setslotwithclass!::before(
    t::SpecialClass, instance, symbol, value) =
    println("Setting slot `$(symbol)`
  of instance of $(t) with value `$(value)`")
> a1.a = 1
Setting slot a of instance of A with value 1
```

Another protocol is slot data setting. For slots, we have set the default Julia methods to call the `getslot` and `setslot!` methods for data access. These two methods then call `getslotwithclass` or `setslotwithclass!`, passing the same arguments and the class metaobject of the instance whose data we are trying to access. In Listing 15, we create a new metaclass like in the previous example and set it as `A`'s metaclass. Defining a method combination for `setslotwithclass!` gives us the ability to change the slot setting behaviour.

4 EVALUATION

We now proceed to analyse the performance of our solution. We will take two approaches. First, we look at the LLVM Intermediate Representation (IR) language [9], a higher-level assembly, generated by the JIT compiler. Second, we compare execution times with CLOS. The following tests were executed on an Intel Core i5-8250U 3.4GHz PC with 16GiB of RAM running the Linux operating system.

4.1 Generated IR

We can analyse the performance of our method dispatch mechanism by taking the code from Listing 10 and reading its IR in Listing 16. As we can see, no computation from method dispatch is present in the end result and only the relevant method body remains. This shows why our multiple-inheritance method dispatch is just as performant as Julia's single-inheritance method dispatch.

The overhead resulting from computing the effective method in Listing 11 can be seen in Listing 17. Just like in the previous example, no overhead is present. Not only that, but there are also no intermediate method calls being made from the chain produced by `join_lambdas`. The bodies from each method were joined into a single body and only the calls to `println` are visible.

Listing 16: IR code from calling `bar(D())`

```
define i64 @julia_foo_1266() #0 {
top:
  ret i64 1
}
```

Listing 17: IR code from calling the method combination of `foo(D())` (simplified)

```
define void @julia_foo_1172() #0 {
top:
  %1 = call nonnull {}* @j1_println_1178("...")
  %2 = call nonnull {}* @j1_println_1179("...")
  %3 = call nonnull {}* @j1_println_1180("...")
  ret void
}
```

Listing 18: Regular Julia method equivalent to method combination in Listing 11

```
function foo_effective(d::D)
    println("Calling before")
    println("Primary method")
    println("Calling after")
end
```

This makes the use of our method combination mechanism equivalent to creating a regular Julia method with a body as the concatenation of the bodies of the methods defined for the combination. This regular Julia method, defined in Listing 18 yields the same IR as the method combination in Listing 11, shown in Listing 17, thus having the same execution times.

4.2 Execution time against CLOS

In this section we present preliminary results regarding the performance of our proposal. For comparison, we use Steel Bank Common Lisp (SBCL), a high-performance Common Lisp implementation. More specifically, we compare an ad-doc example of method dispatch between SBCL and the proposed JOS.

For this example, we iterate a list of objects to force method dispatch. We have the Julia version in Listing 19 and the SBCL version in Listing 20. Both versions assume a class hierarchy like the one specified in Listing 2. Each iteration example ran twice and we registered the results of the second run. Calling `foo` with `arr` as an argument takes `0.016` seconds in Julia and `0.156` seconds in SBCL. However, the comparison is not entirely fair because Julia uses modular arithmetic and it correctly infers the element type of the array. To do a better comparison we included type declarations in the SBCL code, namely, to ensure `fixnum` arithmetic and iteration over a simple array of `D` element-type. Additionally we included a `(declare (optimize (speed 3) (safety 0)))` in all functions and methods. With these changes, SBCL executes the example in `0.092` seconds, still less performant than the Julia implementation. Although the example provided is quite simple and does not expose

Listing 19: Julia example

```
@defmethod baz(b::B, n) = n+1

@defmethod baz(c::C, n) = n*2

arr = fill(D(), 10000000)

foo(a) =
    let y = 0
        for e in a
            y += baz(e, 10)
        end
        y
    end
```

Listing 20: CLOS example

```
(defclass B (A) ())
(defclass C (A) ())
(defclass D (B C) ())

(defmethod baz ((b B) n)
  (+ n 1))
(defmethod baz ((c C) n)
  (* n 2))

(defparameter arr
  (make-array '(10000000)
    :initial-element (make-instance 'D)))

(defun foo (a)
  (let ((y 0))
    (loop for e across a do
          (incf y (baz e 10)))
    y))
```

a real-world scenario, it serves as a good starting point in the development of our proposed metaobject protocol for Julia.

5 FUTURE WORK

The work presented in this paper is an initial attempt at the implementation of a metaobject protocol for Julia. It is far from being a replacement of the CLOS MOP. There are several topics left to be developed in the future, for example: mechanisms for introspecting and analysing generic functions and methods; the possibility of changing the slot inheritance mechanism; and a more complex metaclass hierarchy. Even though our implementation lacks many features, it sets a foundation for the further development of the system, opening a way to get closer to the levels of expressiveness of CLOS.

6 CONCLUSION

In this paper, we discussed metaobject protocols. We analysed two different implementations, CLOS MOP and OpenC++, choosing trade-offs in terms of expressiveness and performance, respectively. We analysed the fundamental differences between them, namely the time in which the logic of the metaobject protocols was evaluated: run-time or compile-time. We also introduced Julia, a dynamic language with JIT compilation features and optimizations. Although it is a fast language, it lacks an object system, which we implemented on top of the language. We explored the language's optimizations and proposed a minimal run-time overhead solution to bridge the gap between the two aforementioned implementations. We focused on two main pieces of metaobjects protocols, multiple-inheritance method dispatch and method combinations, and performed a comparison with SBCL, a high performance Common Lisp compiler.

7 ACKNOWLEDGEMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under the reference UIDB/50021/2020.

REFERENCES

[1] S Borağan Aruoba and Jesús Fernández-Villaverde. A comparison of programming languages in macroeconomics. *Journal of Economic Dynamics and Control*, 58: 265–273, 2015.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[3] Daniel G Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. *ACM Sigplan Notices*, 21(11):17–29, 1986.

[4] Daniel G Bobrow, Linda G DeMichiel, Richard P Gabriel, Sonya E Keene, Gregor Kiczales, and David A Moon. Common lisp object system specification. *ACM Sigplan Notices*, 23(SI):1–142, 1988.

[5] Stephen Cass. The 2015 top ten programming languages.

[6] Shigeru Chiba. A metaobject protocol for c++. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, pages 285–299, 1995.

[7] Brad J Cox. *Object oriented programming: an evolutionary approach*. Addison-Wesley Longman Publishing Co., Inc., 1986.

[8] Gregor Kiczales, Jim Des Rivieres, and Daniel G Bobrow. *The art of the metaobject protocol*. MIT press, 1991.

[9] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[10] Linda Dailey Paulson. Developers shift to dynamic programming languages. *Computer*, 40(2):12–15, 2007.

[11] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, Jácome Cunha, João Paulo Fernandes, and João Saraiva. Energy efficiency across programming languages: How do energy, time, and memory relate? In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering*, SLE 2017, page 256–267, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355254. doi: 10.1145/3136014.3136031. URL https://doi.org/10.1145/3136014.3136031.

[12] Bjarne Stroustrup. *The C++ programming language*. Pearson Education India, 2000.

An Elegant and Fast Algorithm for Partitioning Types

Jim E. Newton
jnewton@lrde.epita.fr
EPITA/LRE
Le Kremlin-Bicêtre, France

ABSTRACT

We present an improvement on the Maximal Disjoint Type Decomposition algorithm, published previously. The new algorithm is shorter than the previously best known algorithm in terms of lines of code, and performs better in many, but not all, benchmarks. Additionally the algorithm computes metadata which makes the Brzozowski derivative easier to compute–both easier in terms of accuracy and computation time. Another advantage of this new algorithm is its resilience limited `subtypep` implementations.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; *Type theory*.

ACM Reference Format:
Jim E. Newton. 2023. An Elegant and Fast Algorithm for Partitioning Types. In *Proceedings of the 16th European Lisp Symposium (ELS'23).* ACM, New York, NY, USA, 9 pages. https://doi.org/10.5281/zenodo.7813576

1 INTRODUCTION

This paper discusses recent developments in a procedure introduced in a previous European Lisp Symposium paper from 2016, where Newton *et al.* [18] introduced *regular type expressions* (RTEs), and suggested a technique to compute membership of the corresponding regular languages. The paper mentioned several limitations which needed further study.

Newton *et al.* [18] developed the theory further applied only to Common Lisp [3], and generalized to other programming languages [17]: Clojure [10, 11], Scala [20, 21], and Python [24].

In Section 2.1, we briefly summarized the theory in order to set the stage for the contributions of this article.

1.1 Contributions

We introduce a new procedure for computing the MDTD (maximal disjoint type decomposition), Definition 2.1. Our new procedure has several advantages over previously known techniques.

(1) It is elegant, Section 3.2.
(2) It is provably correct, Section 3.4.
(3) It eases computation of Brzozowski derivative, Section 4.
(4) It is fast, Section 5.

ELS'23, April 24–25 2023, Amsterdam

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.7813576

1.2 Overview

Given an RTE (Section 2.1), we construct a deterministic finite automaton (DFA) whose language of acceptance (set of all accepted sequences) is the same as the accepting language of the RTE. A DFA consists of states and transitions. The transitions are labeled with Common Lisp type specifiers; see Figures 1 and 2. The construction process simply needs to be able to (1) construct a state with its transitions, and must (2) be repeated until all states have been created. There are two questions to be answered:

- Given a state, what are its transitions?
- What are all the states?

To determine the states and transitions, the Brzozowski derivative of an RTE, r with respect to type v, (Section 2.3) is employed. We compute a value denoted, $\partial_v r$, once for each transition, where r represents an RTE, and v represents a type. The recursive procedure to compute $\partial_v r$ (Section 4) relies heavily on knowledge of disjoint and subtype relations between types (represented in Common Lisp as so-called *type specifiers*). Each time we compute $\partial_v r$ for given values of r and v, we produce yet another RTE, and add it to the working list of RTEs for which we must again compute $\partial_v r$ (for other values of v). Every time we encounter an RTE which we have not seen before, we create a new state, and associate the RTE with that state. We label the transitions with the type used in computing $\partial_v r$. *E.g.*, if states 1 and 2 correspond to RTEs r_1 and r_2 respectively, and $r_2 = \partial_v r_1$ for some value of v, then we construct a transition from state 1 to state 2 labeled with the type v. The process eventually terminates.

The above flow description is not new. What is new and is the novel contribution we present in this article is how to tackle two computational challenges:

- For a given RTE, r, what is the set of types, v, for which we must compute $\partial_v r$? The MDTD Algorithm in Section 3, efficiently computes this set of types.
- Computing $\partial_v r$ relies on knowledge of disjoint and subtype relations between types. Often, programs rely on `subtypep` to decide such relations, but calls to `subtypep` can be compute intensive and may return inconclusive results, which we refer to an *inaccurate*.[1] Our proposed MDTD procedure circumvents reliance on inaccurate results of `subtypep` for these problematic cases.

2 RTE TO DFA FLOW

2.1 Regular Type Expressions

Traditional regular expressions are a DSL (domain specific language) for specifying sets of strings according to which sequences of characters appear in the string. The DSL provides mechanisms

[1] The Common Lisp specification refers to the second return value of `subtypep` as an *accurate* [indicator].

for specifying optionality, alternation, and repetition. Hazel [8, 25] has provided a regular expression library for Common Lisp, but such libraries are standard in most modern programming languages. We assume the reader understands traditional regular expressions. For a theoretical treatment, see Hopcroft [12].

The regular type expression (RTE) is an expression akin to the tradition regular expression, but which the DSL is written in terms of types (rather than characters), and algebraic combinations of these types to specify optionality, repetition, and alternation. Examples of RTEs are:

$$r_1 = \left(symbol \cdot \left(number^+ \cup string^+\right)\right)^+ \quad (1)$$

$$r_2 = (integer \cdot number) \cup (number \cdot integer) \quad (2)$$

Expression (1) means the set of all non-empty sequences of one or more occurrences of a symbol followed by either a number or a string. Sequences such as `(x 3 a "hello")` and `(y 3.1 b "hello" c "world")` are accepted, while sequences such as `(x a "hello")` and `(x 3 a "hello" c)` are rejected. Expression (2) represents the set of sequences of length two, which consist of two numbers, at least one of which is an integer.

The idea of RTE is reminiscent of Clojure Spec [14] and `Malli` [9]. Although Hickey [11] mentions Spec's existence, but we have found no other peer reviewed articles on either. Thus far, conversations between experts on public forums have lead us to contradictory conclusions that Spec is not based on finite automata theory at all, and other claims that it is based on NFA (non-deterministic finite automata) work by Might *et al.* [2, 13]. An NFA-based procedure (presumably using backtracking) would have at least polynomial complexity—our approach offers linear complexity. Grande [1], released a regular pattern matching library in Clojure called `seqexp`. According to an interview with Grande, the `seqexp` does not use a finite automata approach because of JVM limitations.

2.2 DSL for Regular Type Expressions

Expressions (1) and (2) (from Section 2.1) are represented as RTEs respectively as follows

The RTE, $r_1 = (symbol \cdot (number^+ \cup string^+))^+$, is represented in Common Lisp as:

```
(:cat symbol (:+ (:or (:+ number) (:+ string))))
```

The RTE, $r_2 = (integer \cdot number) \cup (number \cdot integer)$, is represented in Common Lisp as:

```
(:or (:cat integer number) (:cat number integer))
```

Keyword symbols such as, `:*`, `:+`, `:and`, `:or`, `:not`, represent the traditional regular expression operators, while leaf level objects represent Common Lisp type specifiers: `number`, `string`, `integer`.

2.3 Constructing a DFA from an RTE

The Common Lisp library, `rte`, is available on `quicklisp`, or directly from GitLab at https://gitlab.lrde.epita.fr/jnewton/regular-type-expression. Analogous to the classical case, an RTE can also be represented by a finite automaton. Whereas in the classical case, transitions are labeled by so-called letters from a fixed, finite alphabet; in our case, we label transitions with type specifiers. Each type specifier denotes the possibly infinite set of possible Common Lisp objects. Newton *et al.* [18] outlined a procedure for converting an RTE to a finite automaton using the Brzozowski derivative [5], in particular Newton follows closely the procedure outlined by Owens *et al.* [22]. The Brzozowski derivative, denoted $\partial_v r$, (read: derivative of r with respect to v) is a function which accepts an RTE, r, and a type specifier, v, and returns an RTE.

As explained in [18] and restated in Algorithm 1, a finite automaton can be constructed by computing the derivative of the given RTE, with respect to each element of a set, $\mathcal{A}$, of types, producing a new set of RTEs. Each of these RTEs represents an additional state in the finite automaton, and the transitions to the states are labeled by the (with-respect-to) type in the derivative computation. The procedure is repeated on the new RTEs, producing more states and transitions, including transitions to pre-existing states, *i.e.* loops to states we have seen already. Brzozowski [5] argues that this process terminates.

Algorithm 1: Construct DFA by Brzozowski derivative

Input: r : an RTE
Output: σDFA

1.1 **begin**
1.2 $q_0 \leftarrow$ new $State(r)$, $T \leftarrow ()$, $Q \leftarrow \{q_0\}$, $W \leftarrow \{q_0\}$
1.3 **while** $W \neq \emptyset$ **do**
1.4 $q_1 \leftarrow$ any element from W
1.5 $r \leftarrow q_1.rte$
1.6 $W \leftarrow W \setminus \{q_1\}$
1.7 **for** $v \in MDTD(1^{st}(r))$ **do**
1.8 $d \leftarrow \partial_v r$ `// canonicalize`
1.9 **if** $d = \emptyset$ **then**
`// avoid unsatisfiable transition`
1.10 continue
1.11 **else if** $\exists\, q_2 \in Q$ *such that* $q_2.rte = d$ **then**
`// transition to pre-existing state`
1.12 $T \leftarrow (q_1, v, q_2) :: T$
1.13 **else**
`// transition to a new state`
1.14 $q_2 \leftarrow$ new $State(d)$
1.15 $T \leftarrow (q_1, v, q_2) :: T$
1.16 $W \leftarrow q_2 :: W$
1.17 $Q \leftarrow q_2 :: Q$
`// compute final states, cf Owens [22]`
`// ⟦·⟧ explained in Section 4.`
1.18 $F \leftarrow \{q \in Q \mid () \in [\![q.rte]\!]\}$
1.19 **return** (Q, q_0, F, T)

RTE to DFA construction is a generalization of classical DFA construction. Our particular DFA is a special case of that D'Antoni and Veanes [6] describe called *symbolic finite automata*. Algorithm 1 outlines the DFA construction using the Brzozowski derivative, and Figure 1 illustrates such a constructed DFA given an RTE. There are several parts of Algorithm 1 which deserve further explanation, making the topic interesting to research.

(1) The $\mathcal{A}$ needed for Algorithm 2 is computed as a call to $1^{st}(r)$ on line 1.7, discussed below.
(2) Canonicalization of an RTE on line 1.8, discussed below.

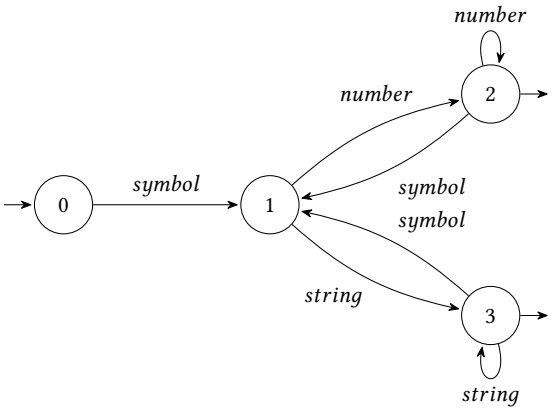


Figure 1: Deterministic finite automaton representing the expression: $(symbol \cdot (number^+ \cup string^+))^+$

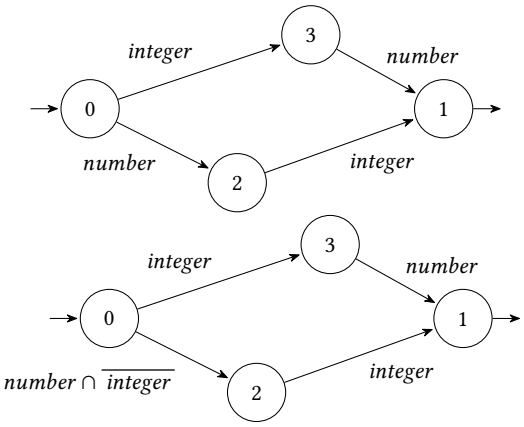


Figure 2: Finite automata representing: $(integer \cdot number) \cup (number \cdot integer)$**, (Top: non-deterministic, Bottom: deterministic)**

(3) Computation of the MDTD, a central contribution of this paper, and explained in Section 3.
(4) Computation of $\partial_v r$ on line 1.7, explained in Section 4.

On line 1.7, we call the MDTD function. The argument (as we'll in Section 3) is a set of types. We could pass the set of all type specifier mentioned in the RTE: {*symbol*, *string*} for Expression (1), and {*integer*, *number*} for Expression (2). This would give a correct answer, but most of the derivatives computed would be $\emptyset$, so most computation time would be wasted. Instead, an important optimization explained in [16] is to only consider *relevant* types. For example, Figure 1 shows that *number* is not relevant to the transitions at state 0. On line 1.7, $1^{st}(r)$ references a procedure for deciding a priori, which types are relevant. We do not discuss this optimization more in this paper as it has no effect on the MDTD algorithm.

Some amount of canonicalization is necessary (1.8). As mentioned above, Brzozowski [5] argues that this process eventually terminates provided a reasonable amount of *canonicalization* is performed on the computed expressions.

2.4 Determinism

Figure 2 illustrates the distinction between deterministic and non-deterministic finite automata. In order that the automaton be deterministic, we must be assure that the set of transitions leaving any given state contain no overlapping types. For example, state 0 in Figure 2 (Top) has exiting transitions *number* and *integer*, which are not disjoint types—a situation which we must avoid. On the other hand, state 0 in Figure 2 (Bottom) has transitions *integer* and $number \cap \overline{integer}$, which are indeed disjoint types.

Definition 2.1 (MDTD). Suppose we are given a finite set of type specifiers, $\mathcal{A} = \{A_1, A_2, \ldots A_n\}$ The set $\mathcal{X} = \{X_1, X_2, \ldots, X_m\}$ is called the *maximal disjoint type decomposition* of $\mathcal{A}$, if the following hold.

(1) **Union invariance**:
$X_1 \cup X_2 \cup \ldots \cup X_n = A_1 \cup A_2 \cup \ldots \cup A_n$.
(2) **Disjointness**:
If $X_i, X_j \in \mathcal{X}$, with $i \neq j$, then $X_i \cap X_j = \emptyset$.
(3) **Refinement**:
If $v \in \mathcal{X}$ and $\mu \in \mathcal{A}$, then either $v \subseteq \mu$ or $v \cap \mu = \emptyset$.

In Figure 2 (bottom) we express the MDTD (maximal disjoint type decomposition) of $\{\top, integer, number\}$ as $\{integer, number \cap \overline{integer}, \top \cap \overline{number}\}$. As is common convention, the graph omits the transition labeled $\top \cap \overline{number}$, because it leads to the so-called *sink* state, indicating a state of rejection as opposed to acceptance.

This kind of partition, illustrated in Figure 3, ensures that any object encountered in a candidate sequence is a member either of exactly one type in $\mathcal{X}$ or is a member of no type in the decomposition. Figure 3 expresses the MDTD as $\{X_1, X_2, \ldots, X_9\}$. Notice there is one X_i for each bounded disjoint area in Figure 3.

If $\top$, the universal type, is included in the input, $\mathcal{A}$, then the output $\mathcal{X}$ will also include the region outside A_1, *i.e.*, $\overline{A_1} \in \mathcal{X}$.

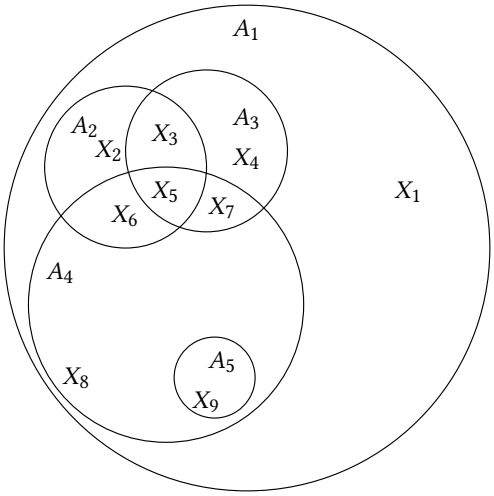
We have proven in [16] that a MDTD exists and is uniquely determined for any finite set of types.

3 A NEW MDTD PROCEDURE

We present the procedure shown in Algorithm 3 to compute the MDTD. The procedure returns two values, $\mathcal{X}$ and $\mathcal{S}$, where $\mathcal{X}$ is the actual set of types comprising the type decomposition, and $\mathcal{S}$ is metadata which can be reused to make the Brzozowski derivative (Section 4), more efficient and more accurate. The actual metadata is a mapping from each type specifier, v in $\mathcal{X}$, to two sets: a set of factors (super-types) of v and a set of disjoint types of v.

3.1 The subtypep function, in Common Lisp

In Common Lisp, programmatic reasoning about types is done in term of two so-called *type specifiers*. The type specifiers *t* and *nil* refer respectively to the universal type (containing all possible Common Lisp object) and the empty type (containing no objects). The intersection and union of two types can be specified using the *and* and *or* types; *e.g.*, `(or string (and integer (satisfies`



Disjoint Set	Derived Expression	Factors	Disjoint Types
X_1	$A_1\overline{A_2}\ \overline{A_3}\ \overline{A_4}$	A_1	A_2, A_3, A_4, A_5
X_2	$A_2\overline{A_3}\ \overline{A_4}$	A_1, A_2	A_3, A_4, A_5
X_3	$A_2A_3\overline{A_4}$	A_1, A_2, A_3	A_4, A_5
X_4	$A_3\overline{A_2}\ \overline{A_4}$	A_1, A_3	A_2, A_4, A_5
X_5	$A_2A_3A_4$	A_1, A_2, A_3, A_4	A_5
X_6	$A_2A_4\overline{A_3}$	A_1, A_2, A_4	A_3, A_5
X_7	$A_3A_4\overline{A_2}$	A_1, A_3, A_4	A_2, A_5
X_8	$A_4\overline{A_2}\ \overline{A_3}\ \overline{A_5}$	A_1, A_4	A_2, A_3, A_5
X_9	A_5	A_1, A_4, A_5	A_2, A_3

Figure 3: Example of Maximal Disjoint Type Decomposition: $\mathcal{X} = \{X_1, X_2, \ldots, X_9\}$ is the MDTD of $\mathcal{A} = \{A_1, A_2, \ldots, A_5\}$. Derived expressions are intersections of types from $\mathcal{A}$ or complements thereof.

evenp))). And types can be complemented (inverted) using the *not* type; *e.g.* (not integer).

In Common Lisp, the subtypep function can be used to determine the subtype relation. The behavior of subtypep can be understand by the following three cases:

(1) The Common Lisp expresison, (subtypep integer number), returns two values t,t; the first t indicates that the subtype relation is validated (*integer* ⊆ *number*) while the second t indicates that the subtype relation was proven to be true.
(2) (subtypep number integer) returns two values nil,t; the nil indicates that the subtype does not hold (*integer* ⊈ *number*) while the t indicates that the subtype relation was proven to be false.
(3) If Common Lisp cannot determine whether the subtype relation holds, subtypep returns nil,nil. (subtypep (satisfies oddp) integer) returns two values nil,nil; the first nil has no meaning because the second nil indicates that the subtype relation was neither proven nor disproven.

Algorithm 2: Compute MDTD of given $\mathcal{A}$.

Input: $\mathcal{A}$: a set of type designators
Output: $(\mathcal{X}, \mathcal{S})$: partition and metadata
2.1 **begin**
2.2 $\quad\mathcal{S} \leftarrow \{(\top, \{\top\}, \{\bot\})\}$ // working list of triples
2.3 $\quad\mathcal{X} \leftarrow \{\top\}$ // working list of disjoint types
2.4 $\quad$**for** $\mu \in \mathcal{A}$ **do**
2.5 $\quad\quad$**for** $(v, f, d) \in \mathcal{S}$ **do**
2.6 $\quad\quad\quad\mathcal{S} \leftarrow \mathcal{S} \setminus \{(v, f, d)\}$
2.7 $\quad\quad\quad$**if** $\mu \cap v = \emptyset$ **then**
2.8 $\quad\quad\quad\quad\mathcal{S} \leftarrow (v, f, \mu :: d) :: \mathcal{S}$ // v, μ disjoint
2.9 $\quad\quad\quad$**else if** $v \subseteq \mu$ **then**
2.10 $\quad\quad\quad\quad\mathcal{S} \leftarrow (v, \mu :: f, d) :: \mathcal{S}$ // $v \cap \overline{\mu} = \emptyset$
2.11 $\quad\quad\quad$**else**
// $\mu \cap v$ and $\overline{\mu} \cap v$ partition v
2.12 $\quad\quad\quad\quad v_1 \leftarrow \mu \cap v$
2.13 $\quad\quad\quad\quad v_2 \leftarrow \overline{\mu} \cap v$
2.14 $\quad\quad\quad\quad\mathcal{X} \leftarrow (\mathcal{X} \setminus v) \cup \{v_1, v_2\}$
2.15 $\quad\quad\quad\quad\mathcal{S} \leftarrow (v_1,\ \mu :: f,\ d) :: \mathcal{S}$ // $v_1 \subseteq \mu$
2.16 $\quad\quad\quad\quad\mathcal{S} \leftarrow (v_2,\ f,\ \mu :: d) :: \mathcal{S}$ // v_2, μ disjoint
2.17 $\quad$**return** $(\mathcal{X}, \mathcal{S})$

Algorithm 3: expand-1: Helper function for MDTD. A *triple* consists of a derived expression, list of factors, and list of disjoint types as in Figure 3 (Bottom).

Input: μ : a type designator
Input: (v, f, d): a triple
Output: a set of one or two triples
3.1 **begin**
3.2 $\quad$**if** $\mu \cap v = \emptyset$ **then**
3.3 $\quad\quad$**return** $\{(v, f, \mu :: d)\}$
3.4 $\quad$**else if** $v \subseteq \mu$ **then**
3.5 $\quad\quad$**return** $\{(v, \mu :: f, d)\}$
3.6 $\quad$**else**
3.7 $\quad\quad$**return** $\left\{\left(\mu \cap v, \mu :: f, d\right), \left(\overline{\mu} \cap v, f, \mu :: d\right)\right\}$

An expression such as (subtypep integer nil) as whether *integer* ⊆ ∅, *i.e.*, whether the integer type is empty. To ask about the disjoint relation, we ask whether the intersection is empty: (subtypep (and integer string) nil) asks whether (*integer* ∩ *string*) ⊆ ∅.

The Common Lisp specification allows subtypep to return nil,nil under several circumstances, most notably in cases involving the satisfies type in which case it is often impossible to determine, but also when it deems an accurate determination to be too costly in terms of computation time.

3.2 MDTD in Common Lisp

Algorithm 2 is restated more succinctly reduce and mapcan as in Figure 4. We pass the local function, expand, to reduce. The expand function uses mapcan to iterate a curried version of expand-1 (Algorithm 3) across $\mathcal{A}$. Each successive call to mapcan further refines

```
(defun mdtd (A)
  (labels ((expand-1 (mu triple) (...))
           (expand (acc mu)
             (mapcan (lambda (triple)
                       (expand-1 mu triple))
                     acc)))
    (let ((S (reduce #'expand A
                     :initial-value '((t (t) (nil))))))
      (values (mapcar #'car S) S))))
```

Figure 4: The `MDTD` code expressed using `mapcan`, `reduce`, and `expand-1` from Algorithm 3. The variables `S` and `M` correspond respectively to the variables $\mathcal{S}$ and $\mathcal{A}$ from Algorithm 3.

the *current* partition by intersecting appropriate type specifiers with μ, its complement $\overline{\mu}$, or both.

A subtle but crucial feature of our `mdtd` implementation is that even if `subtypep` returns *dont-know* (either during a subtype check or disjoint check) we nevertheless construct a well-formed partition of the space. This certainty is because in the worst case, algorithm lines 2.15, 2.16, and 3.7 partition the type, v, into $v_1 = v \cap \mu$ and $v_2 = v \cap \overline{\mu}$. If v and μ are disjoint (despite `subtypep` returning *dont-know*), then $v_1 \subseteq \emptyset$. If $v \subseteq \mu$, then $\overline{\mu} \cap v \subseteq \emptyset$. The consequence is that the computed set, $\mathcal{S}$ might contain type specifiers which specify the empty type, even though we have failed to detect the fact that the types are empty.

3.3 Sample Run

We detail the computation of the types in Figure 3. Figure 5 shows the computation tree. Each call to `mapcan` creates one horizontal level of the tree. The computation starts with the top type, denoted $\top$. Each subsequent level partitions each of the type specifiers in the previous level by intersecting with μ, $\overline{\mu}$ or both. If μ is disjoint with the type in question or a supertype of the type in question, the previous level's type is inherited to the next level.

The second level of the tree intersects $\top$ with A_1 and $\overline{A_1}$. The third level intersects each of $\left\{A_1, \overline{A_1}\right\}$ with A_2 and $\overline{A_2}$.

$$A_1 \cap A_2 = A_2$$
$$A_1 \cap \overline{A_2} = A_1 \cap \overline{A_2}\ .$$

Since $A_2 \subseteq A_1$, line 2.10 is reached. No intersection computation is necessary because the result would either be the empty type or the same type we started with:

$$\overline{A_1} \cap A_2 = \emptyset$$
$$\overline{A_1} \cap \overline{A_2} = \overline{A_1}$$

Similar reasoning continues with the fourth and fifth levels.

3.4 Sketch of Proof of Correctness

We do not present a formal proof, but rather we sketch an informal argument. A formal proof will come in a future publication. To sketch this proof, we'd like to show that the set of types specified by the bottom-most row of Figure 5 is indeed the MDTD of the types $\{A_1, A_2, A_3, A_4, A_5\}$. We need to show three things from Definition 2.1, which we address in Sections 3.4.1, 3.4.2, and 3.4.3.

3.4.1 Union invariance. By induction: To understand that the union of the types specified at level n (of Figure 5) is the same as the union of the types at level $n+1$, we see that each v at level n corresponds either to the same type at level $n+1$, or for some type μ, two types

$$\begin{aligned} v_1 &= v \cap \mu \\ v_2 &= v \cap \overline{\mu} \\ v_1 \cup v_2 &= (v \cap \mu) \cup (v \cap \overline{\mu}) \\ &= v \cap \left(\mu \cup \overline{\mu}\right) \\ &= v \cap \top = v \end{aligned}$$

So refining the partition moving from level-n to level $n+1$ preserves the union, which at the top level is $\top$, or A_1 in the case that $\top$ is not included in the MDTD input.

3.4.2 Disjointness. To understand that the types specified at each level are disjoint, we assume (an inductive proof) that the types at level n are disjoint and prove that the types at level $n+1$ are disjoint. We know this, because each type, v, at level n corresponds either to the same type at level $n+1$ or to two types, v_1 and v_2, where

$$\begin{aligned} v_1 &= v \cap \mu \\ v_2 &= v \cap \overline{\mu} \\ v_1 \cap v_2 &= (v \cap \mu) \cap (v \cap \overline{\mu}) \\ &= (v \cap v) \cap (\mu \cap \overline{\mu}) \\ &= v \cap \emptyset = \emptyset \end{aligned}$$

So we see that v_1 and v_2 are disjoint. If two types, v_1, v_2 at level $n+1$ are derived from the different level-n parents, v_3, v_4 respectively, then we know that v_3 and v_4 are disjoint by inductive hypothesis. Thus there exist μ_1 and μ_2 such that

$$\begin{aligned} v_1 &= v_3 \cap \mu_1 \\ v_2 &= v_4 \cap \mu_2 \\ v_1 \cap v_2 &= (v_3 \cap \mu_1) \cap (v_4 \cap \mu_2) \\ &= (v_3 \cap v_4) \cap (\mu_1 \cap \mu_2) \\ &= \emptyset \cap (\mu_1 \cap \mu_2) = \emptyset \end{aligned}$$

Thus all the types specified at level $n+1$ are disjoint.

3.4.3 Refinement. If $v \in \mathcal{X}$ and $\mu \in \mathcal{A}$, we see that μ is in either the third column (Factors) or the 4th column (Disjoint types) of the table in Figure 3. This fact is guaranteed because we know that the function in Algorithm 3 was called with μ as an argument. Algorithm 3 assures that μ is prepended either to the set of factors or the set of disjoint types.

4 COMPUTING BRZOZOWSKI DERIVATIVE

We saw in Algorithm 1 that the output of `MDTD` is used as input for the Brzozowski derivative on line 1.8. In this section, we show how the metadata collected in `MDTD` helps to compute $\partial_v r$

Recall the $\partial_v r$ is the Brzozowski derivative of RTE, r, with respect to type v. The value or $\partial_v r$ is another RTE.

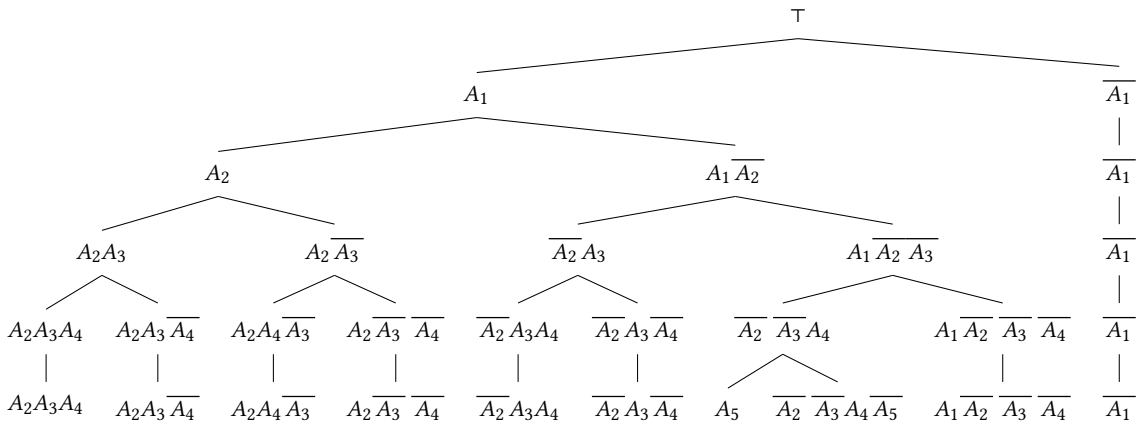


Figure 5: Computation tree computing the MDTD for the types in Figure 3

$$\partial_v\,\emptyset = \emptyset \tag{3}$$
$$\partial_v\,\varepsilon = \emptyset \tag{4}$$
$$\partial_v\,(\neg r) = \neg\partial_v\,r \tag{5}$$
$$\partial_v\,(r^*) = \partial_v\,r \cdot r^* \tag{6}$$
$$\partial_v\,(r \vee s) = \partial_v\,r \;\vee\; \partial_v\,s \tag{7}$$
$$\partial_v\,(r \wedge s) = \partial_v\,r \;\wedge\; \partial_v\,s \tag{8}$$
$$\partial_v\,(rs) = \begin{cases} (\partial_v\,r)s & \text{if } r \text{ not nullable} \\ (\partial_v\,r)s \vee \partial_v\,s & \text{if } r \text{ nullable} \end{cases} \tag{9}$$
$$\partial_v\,\mu = \varepsilon \quad \text{if } [\![v]\!] \subseteq [\![\mu]\!] \tag{10}$$
$$\partial_v\,\mu = \emptyset \quad \text{if } [\![v]\!] \cap [\![\mu]\!] = \emptyset \tag{11}$$
$$\partial_v\,\mu \quad \text{otherwise, no rule defined} \tag{12}$$

Figure 6: Recursive rules for computing Brzozowski derivative of an RTE. These rules are applied in computing $\partial_v\,r$ on line 1.8 of Algorithm 1. v (similarly μ) is a type specifier. $[\![v]\!]$ (similarly $[\![\mu]\!]$) represents the set of values comprising the specified type.

4.1 Computation Details

To compute $\partial_v\,r$, Owens *et al.* [22] suggest a recursive procedure. Newton *et al.* [18] generalized this procedure to work with Common Lisp types. Newton [16] further generalized the recursive rules of this procedure as shown in Figure 6. Rule 9 refers to *nullable*, meaning that the language of r contains the empty sequence. Owens *et al.* explain a simple decision procedure to determine whether a regular expression is *nullable*.

Some explanation is necessary to understand the notation and the implications of Figure 6. Recall that the notation $\partial_v\,r$ means that r is an RTE and v is a type. Programmatically, r is a data structure represented according to the rules of a DSL, which we summarize in Section 2.2.

The v in $\partial_v\,r$ specifies a type. Programmatically, v is represented by a Common Lisp type specifier.

Given a value of r and v, to compute $\partial_v\,r$, the rules in Figure 6 are applied recursively. There are several cases which terminate the recursion.

Rule (3): Every RTE represents a set of Common Lisp sequences. We use the symbol $\emptyset$ to represent the RTE which itself represents the empty set of sequences. Careful, the empty set of sequences is different from the set of empty sequences, denoted by ε. The derivative of $\emptyset$ is again $\emptyset$ regardless of v. The Common Lisp type specifier `nil` specifives the empty type, equivalently empty set.

Rule (4): The symbol ε represents the set of empty sequences. We sometimes represent this set as $\{()\}$; however in Common Lisp, ε includes the empty list, empty array, empty string, etc [15, 23]. The derivative of ε is $\emptyset$ regardless of v.

Rules (10) **and** (11): These rules represent the case where the RTE, μ, is known to represent specifically a set of singleton sequences,[2] *e.g.* the set of singleton sequences whose first (and only element) is an integer, or the set of sequences whose element is a string. In the notation of Figure 6, μ represents the RTE (in turn representing a set of sequences), while $[\![\mu]\!]$ represents the set comprising of the first elements of these sequences: the set of integers, or the set of strings, as opposed to the set of singleton lists of integers or set of singleton lists of strings. In Rule (11), we use $\emptyset$ to represent both the RTE containing no sequences, and also the empty type.

4.2 Complications with `subtypep`

In order to distinguish rules (10) and (11) programmatically, we must know whether one type is a subtype of another, given the type specifiers, or know whether the two specified types are disjoint. The Common Lisp function, `subtypep`, is an obvious implementation choice to make this run-time decision. However, `subtypep` sometimes returns `dont-know`. If this occurs during DFA construction, we cannot determine the value of the derivative. Thus, we must avoid this case.

[2] We can be assured that the μ represents a singleton sequence because we have eliminated all other possiblities in rules 3 through 9; *i.e.*, μ is not $\emptyset$, ε, *, negation, disjunction, conjunction, nor concatenation.

```
(defclass A1 () ())
(defclass A2 (A1) ())
(defclass A3 (A1) ())
(defclass A23 (A2 A3) ()) ; X3 U X5
(defclass A4 (A1) ())
(defclass A423 (A4 A23) ()) ; X5 U X6 U X7
(defclass A5 (A4) ())

(subtypep '(and A3 (not A2) (not A4))
          'A1)          ; returns T, T
(subtypep 'A3 'A2)      ; returns NIL, T
(subtypep '(and (and A3 (not A2) (not A4)) A1)
          nil)          ; returns NIL, NIL
```

Figure 7: Common Lisp code defining classes analogous to Figure 3, also demonstrating successful and unsuccessful calls to `subtypep`.

This weakness of `subtypep` is a significant problem for the Brzozowski derivative computation, a limitation which we alleviate with our proposed MDTD procedure. As an illustration of the problem, suppose that we have an RTE, r, representing a singleton sequence whose element has type X_4 from Figure 3, and suppose we need to compute $\partial_v r$ where $v = A_1$. We need to determine whether $\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right) \subseteq A_1$ or whether $\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right) \cap A_1 = \emptyset$. It is not syntactically obvious which (if either) is the case; there is no mention of A_1 within $\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right)$. The human (or a sufficiently intelligent `subtypep`) can of course answer this question by reasoning that A_3 is mentioned and $A_3 \subseteq A_1$. This reasoning only works if A_3 and A_1 are specified by very simple type specifiers, such as a class name. If on the other hand, either or both of A_3, A_1 are type specifiers involving Boolean combination types such as `(and ...)`, `(or ...)`, `(not ...)`, or `(satisfies ...)`, such reasoning would be less obvious and more compute intensive.

The Common Lisp code in Figure 7 defines classes which have the same disjoint and subtype relations as in Figure 3. The code contains three calls to `subtypep` to determine whether see how SBCL and CLISP handle these calls to `subtypep`. The SBCL [15] implementation responds `T, T` for the first, indicating that the subtype relation, $\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right) \subseteq A_1$. The second example returns `NIL, T`, indicating that $A_3 \not\subset A_2$. However the third call which asks whether $\left(\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right) \cap A_1\right) \subseteq \emptyset$ returns `NIL, NIL` indicating *dont-know*. We get the same results in CLISP [7].

If we use `subtypep` to answer these questions, `subtypep` is allowed (by the Common Lisp specification) to return *dont-know*. However, we do not need to rely on `subtypep` in this case, because as we also see in the third column row X_7 of Figure 3 (Bottom) that A_1 is guaranteed by construction to be a supertype of $\left(A_3 \cap \overline{A_2} \cap \overline{A_4}\right)$.

Even if the `subtypep` implementation in your particular implement of Common Lisp is intelligent enough to determine this subtype relation, doing so would necessarily be computation intensive. Our MDTD procedure avoids this redundant complexity.

5 PERFORMANCE ANALYSIS

We have taken as benchmarks, the performance analysis presented in [16, Ch 10]. In that work, Newton analyzed (ad nauseam) performance characteristics of various MDTD algorithms on various genre of input types, without conclusive results. We repeated some of those performance comparisons with the procedure presented in this paper. The experiments are summarized here. We considered the following algorithms, a subset of those presented in [16].

(1) `mdtd-bdd` – A primitive base-line algorithm using BDDs [4] as data structure to designate a type.
(2) `mdtd-graph` – A graph based algorithm also described in [19], using Common Lisp type specifiers (s-expressions) as type designators.
(3) `mdtd-bdd-graph` – Same algorithm as `mdtd-graph` but using BDDs as type designators.
(4) `mdtd-padl` – The procedure in Algorithm 2.

The reference benchmarks were divided into so-called *pools*. A pool is a set of type specifiers, chosen with similar characteristics; *e.g.*, a set of `(member ...)` types, or a set of floating point range types, or all predefined subtypes of number.

We show the results for several pools:

- **MEMBER types** – Types such as

```
(member 2 6 9 10)
(member 1 2 4 5 9)
(member 1 6 7 8)
(member 0 1 4 6 7 9 10)
(member 3 4 7 9 10)
```

- **CL combinations** – Unions and intersections of types whose name come from the `common-lisp` package. Examples include

```
(or print-not-readable structure-class)
(and simple-string bignum)
(or standard-char double-float)
(or class storage-condition)
```

- **Real number ranges** – numerical ranges of `integer`, `real`, and `float`. Examples include

```
(INTEGER 60 (79))
(REAL 1/36 47/9)
(FLOAT 55.142532 60.722794)
```

- **Subtypes of NUMBER** – Subtypes of `number` and Boolean combinations of them. Examples include

```
short-float
(and short-float (not unsigned-byte))
(or short-float unsigned-byte)
unsigned-byte
(and number (not bit))
rational
```

- **CL types** – Symbols from `common-lisp` package which designate types. Examples include

```
arithmetic-error
function
simple-condition
array
```

Running a benchmark consists of selecting successively larger sets of type specifiers from the pool in question, and calling the `mdtd` function, measuring the execution time. Figure 8 shows the benchmark results. See the legend on the bottom-right of the figure for the color scheme.

We see that for a small number of input types, the `mdtd-padl` algorithm performs poorly compared to the others, in time ranges of less than 0.1 millisecond. However, for times greater than 1 millisecond, the algorithm performs well. In the top two plots in Figure 8, `mdtd-padl` is the best performing, at least in the asymptotic case. In the bottom-most plot, CL types, we see that `mdtd-padl` performs worse by an order of magnitude. However, for most of the cases, in the middle of the figure, the performance is very good but outperformed by the BDD-based algorithm.

6 CONCLUSION

6.1 Results

In this work, we have introduced a new algorithm for computing the Maximal Disjoint Type Distribution of a given set of type specifiers. Our experiments show that the procedure is usually faster than previously reported procedures, and also provides data which makes the Brzozowski derivative easier to compute. While the improvements we have discussed here have also been applied to our RTE implements in Clojure, Scala, and Python, we have only addressed herein the aspects relating to Common Lisp.

Our MDTD procedure alleviates some of the consequences of the incompleteness and compute intensity of `subtypep`. The fact that certain dependence on direct calls to `subtypep` is elided, has the effect of eliding certain unnecessary computations, potentially making the Brzozowski derivative computation faster than it otherwise might be. In addition to computation speed, we also enable the algorithm to produce a correct (even if suboptimal) result despite having a less powerful implementation of `subtypep` in your Common Lisp implementation. The MDTD algorithm is guaranteed to compute a set of types which are disjoint; however, they may not be provably inhabited.

6.2 Perspectives

We see in Figure 8 that the BDD-based MDTD procedure outperforms `mdtd-padl`, but not significantly so. We would like to refactor the BDD-based procedure to use the approach of `mdtd-padl` but applied to BDDs rather to s-expression based type specifiers.

We have not yet extensively investigated the application of our algorithm to type-system related computations on JVM languages such as Clojure and Scala. Sometimes questions of subtype-ness and habitation/vacuity cannot be answered about JVM-based types, because we do not know at computation time which shared libraries may be dynamically loaded later in the running program. Our current model in the RTE implementation in Clojure and Scala uses a so-called *world-view*. An closed world-view means that we assume no new classes will be added, and an open world-view means we never know the entire list of subclasses of a given class. A open world-view is predicted to result in larger DFAs with more unsatisfiable transitions. However, we do not yet have data to confirm or measure this effect.

REFERENCES

[1] Seqexp: regular expressions for sequences, 2014. URL https://github.com/cgrand/seqexp.

[2] David nolen on parsing with derivatives, 2016. URL https://www.youtube.com/watch?v=FKiEsJiTMtI.

[3] Ansi. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.

[4] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986.

[5] Janusz A. Brzozowski. Derivatives of Regular Expressions. *J. ACM*, 11(4):481–494, October 1964. ISSN 0004-5411. doi: 10.1145/321239.321249. URL http://doi.acm.org/10.1145/321239.321249.

[6] Loris D'Antoni and Margus Veanes. The power of symbolic automata and transducers. In *Computer Aided Verification, 29th International Conference (CAV'17)*. Springer, July 2017. URL https://www.microsoft.com/en-us/research/publication/power-symbolic-automata-transducers-invited-tutorial/.

[7] Bruno Haible. Gnu clisp, 2010. URL https://clisp.sourceforge.io.

[8] Philip Hazel. PCRE - Perl Compatible Regular Expressions, 2015. URL www.pcre.org.

[9] Mikko Heikkilä. Malli, metosin, 2022. URL https://github.com/metosin/malli.

[10] Rich Hickey. The Clojure Programming Language. In *Proceedings of the 2008 symposium on Dynamic languages*, page 1. ACM, 2008.

[11] Rich Hickey. A History of Clojure. *Proc. ACM Program. Lang.*, 4(HOPL), June 2020. doi: 10.1145/3386321. URL https://doi.org/10.1145/3386321.

[12] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. ISBN 0321455363.

[13] Matthew Might, David Darais, and Daniel Spiewak. Parsing with derivatives: A functional pearl. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, page 189–195, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308656. doi: 10.1145/2034773.2034801. URL https://doi.org/10.1145/2034773.2034801.

[14] Alex Miller. spec guide, 2022. URL https://clojure.org/guides/spec.

[15] William H. Newman. Steel Bank Common Lisp User Manual, 2015. URL http://www.sbcl.org.

[16] Jim Newton. *Representing and Computing with Types in Dynamically Typed Languages*. PhD thesis, Sorbonne University, November 2018.

[17] Jim Newton and Adrien Pommellet. A portable, simple, embeddable type system. In *European Lisp Symposium*, Online, Everywhere, May 2021.

[18] Jim Newton, Akim Demaille, and Didier Verna. Type-Checking of Heterogeneous Sequences in Common Lisp. In *European Lisp Symposium*, Kraków, Poland, May 2016.

[19] Jim Newton, Didier Verna, and Maximilien Colange. Programmatic Manipulation of Common Lisp Type Specifiers. In *European Lisp Symposium*, Brussels, Belgium, April 2017.

[20] Martin Odersky and Matthias Zenger. Scalable component abstractions. In *Sigplan Notices - SIGPLAN*, volume 40, pages 41–57, 10 2005. doi: 10.1145/1103845.1094815.

[21] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The Scala language specification, 2004.

[22] Scott Owens, John Reppy, and Aaron Turon. Regular-expression Derivatives Re-examined. *J. Funct. Program.*, 19(2):173–190, March 2009. ISSN 0956-7968.

[23] Christophe Rhodes. User-extensible Sequences in Common Lisp. 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. URL http://doi.acm.org/10.1145/1622123.1622138.

[24] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011. ISBN 1906966141, 9781906966140.

[25] Edmund Weitz. *Common Lisp Recipes: A Problem-solution Approach*. Apress, 2015. ISBN 978-1-4842-1177-9.

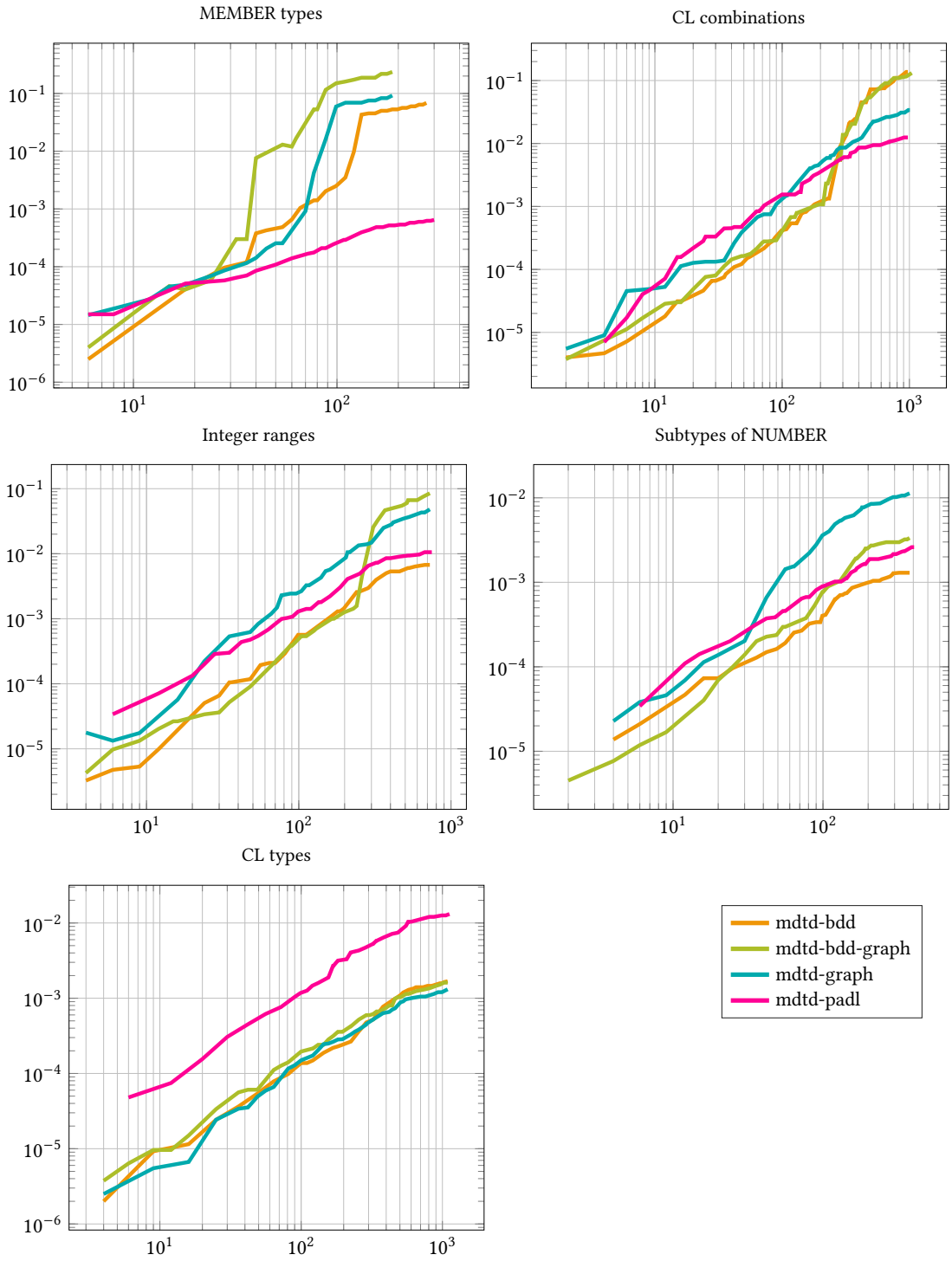


Figure 8: Results of benchmark experiments: Lower is better. Each graph has number of given types as x-axis, and average computation time in seconds, as y-axis. The pink/magenta curve indicates the results for `mdtd-padl`, that being the algorithm described in this paper.

GRASP: An Extensible Tactile Interface for Editing S-expressions

Panicz Maciej Godek
godek.maciek@gmail.com

ABSTRACT

GRASP is a Scheme-based extensible computational environment designed to work with S-expressions on touch screens. It features a powerful extension mechanism as well as a subsystem for handling gesture-based input. It is implemented in Kawa Scheme, and can be compiled as an Android application as well as run on a desktop windowing environment and inside of terminal emulators.

GRASP is still a work-in-progress application, so the purpose of the demo is:

(1) to show the current state of the application;
(2) to convey the ultimate vision behind GRASP;
(3) to present the development plan and methodology, and optionally:
(4) to describe the hitherto history of the development.

The presentation of GRASP in this paper is written as if all of its features were already implemented. The omissions are presented in a separate section.

CCS CONCEPTS

• Software and its engineering → Integrated and visual development environments; • Human-centered computing → Visualization toolkits; Ubiquitous and mobile computing systems and tools; • Computing methodologies → Graphics input devices.

KEYWORDS

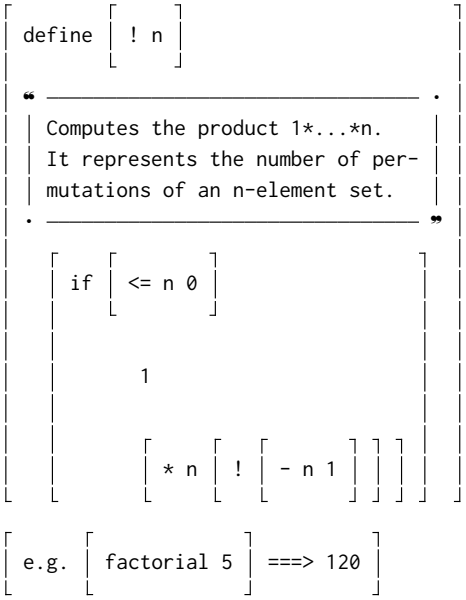
visual programming, touchscreen-based editing, interactive programming, structual editing

ACM Reference Format:
Panicz Maciej Godek. 2023. GRASP: An Extensible Tactile Interface for Editing S-expressions. In Proceedings of the 16th European Lisp Symposium (ELS'23). ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.7816633

1 THE CONCEPT OF GRASP

GRASP[1] is a tactile-first structural editor for S-expressions. Its design is based on representing S-expressions as nestable boxes. The boxes are rendered so that their left and right edge resemble – respectively – opening and closing parentheses.

When displayed in a terminal, a Lisp program edited in GRASP might look like this[2]:



which corresponds to the following program text:

```
(define (! n)
"Computes the product 1*...*n.
It represents the number of per-
mutations of an n-element set."
  (if (<= n 0)
      1
      (* n (! (- n 1)))))
(e.g. (factorial 5) ===> 120)
```

The left and right parentheses play different roles in tactile editing: the left parenthesis is used for moving (if pressed once) or copying (if pressed twice) an expression, whereas the right parenthesis is used for resizing an expression.

An expression which is currently being moved can be deleted by throwing it off the surface quickly. Likewise, moving a finger quickly while the expression is being resized causes the box to be spliced into its parent (this feature is sometimes referred to as "pulling-the-rug splicing").

In addition to boxes, GRASP offers four other types of objects: atoms, texts, extensions and comments.

[1] GRASP is an open-source project hosted at https://github.com/panicz/grasp

[2] Some recordings presenting various prototypes of GRASP can be watched on the author's youtube channel: https://www.youtube.com/channel/UCt4u6WQDy2yjXz6eXCcyijQ

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

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.7816633

Atoms are things like symbols, numbers, characters or Boolean values in Lisp. They support touch gestures in a similar way as the left parenthesis of a box: single touch causes them to be dragged, whereas double touch causes their copy to be dragged.

The text type corresponds to strings. They are displayed inside boxes with quotation marks on their corners. The roles of the quotation marks are analogous to the left and the right parenthesis: the left one can be used to move the text within the expression tree, remove it or copy, while the right one can be used to change the shape of a text.

Comments in the Scheme programming language come in three flavors, all of which are supported by GRASP:

- line comments, which span until the end of a given line;
- block comments, which are similar to text;
- expression comments, which comment out a single expression.

Comments are invisible to the operations on the document, such as `car` or `cdr`. Other than that, line and block comments are similar to text.

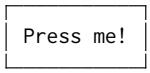
The last type of objects supported by the editor are extensions. The list of extensions is open-ended. Expressions are sometimes referred to as "magic boxes", because they are boxes which define their own rules of interaction.

A simple example of an extension is a button. If it is loaded, the expression

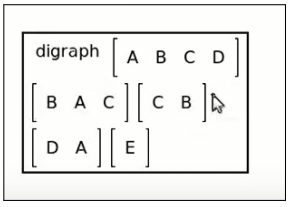
```
(Button label: "Press me"
        action: (lambda () (WARN "button pressed")))
```

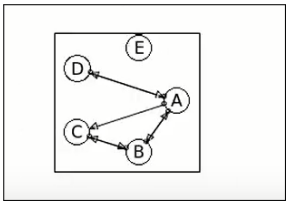
can be rendered as a button, and responds to touch events with the invocation of its action callback.

The terminal client of GRASP would display it in the following manner:



A more advanced extension – coming from an earlier prototype of GRASP – allows to display graphs represented in the form of neighbour list as an actual graph:





Extensions are meant to be user-definable, but the exact API for defining them is subject to an ongoing research.

Some desired extensions for GRASP include:

- a drawing editor
- a graph visualizer/editor
- a visual evaluator
- a function plotter

and many others.

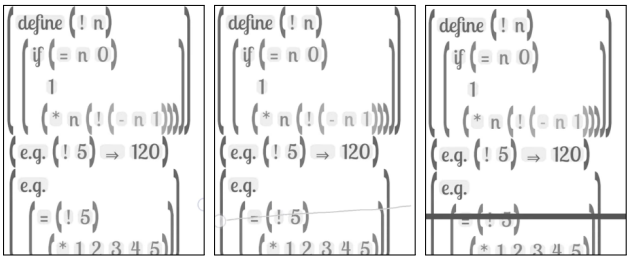
1.1 Gesture-based input

Since devices with touch screens often lack a proper keyboard, and usually display regrettable keyboard substitutes on their screens as needed, GRASP attempts to find a more ergonomic alternative.

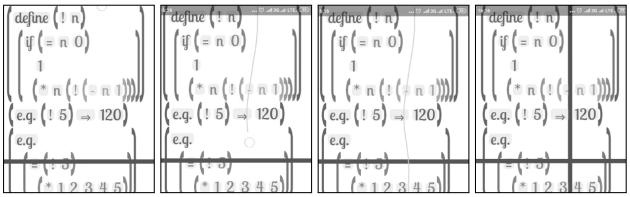
One idea is gesture-based input: the user draws a shape on the screen, and if the shape is recognized, an appropriate action is performed.

By default, the following shapes are recognized:

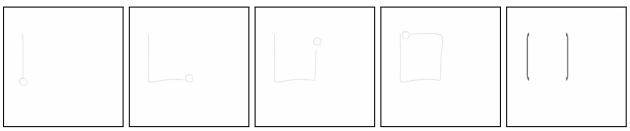
- horizontal line, which splits the panes it's drawn over vertically into halves (similar to `C-x 2` in Emacs)



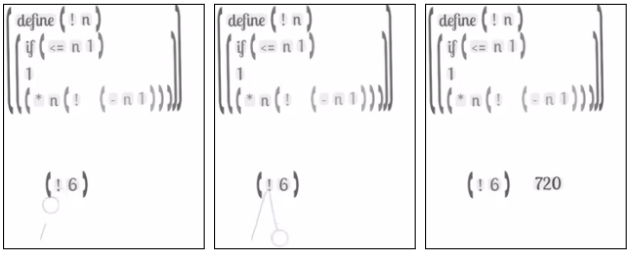
- vertical line, which splits the panes below horizontally into halves (similar to `C-x 3` in Emacs)



- box gesture, which creates a new box in the document it's drawn over



- wedge symbol, which causes the expression below its blade to be evaluated (similar to `C-x C-e` in Emacs' Lisp interaction modes)



Since many touchscreen-equipped devices also feature accelerometers, GRASP also lets define motion-based edit operations – for example, shaking a device might result in reindenting the source code.

1.2 Keyboard input

Even though GRASP focuses on tactile editing and on mobile devices, a lot of effort has been put into making it a pleasant keyboard editing experience.

GRASP features a flexible key binding mechanism, which unites the input systems from its target environments (Android, terminal and windowing systems).

By default, it provides the "Common User Access" keyboard bindings (ctrl-z for undo, ctrl-c for copy etc.) and it allows users to use keyboard arrows to navigate cursor over the active document.

Keyboard editing is context-sensitive, so for example pressing the `#\[` key creates a new box, unless the cursor is located on a text element, in which case the `#\[` character is inserted verbatim into text.

Also, extensions are free to interpret most of the pressed characters as they please.

2 STRUCTURAL EDITING

The documents in GRASP are considered mutable, and the editing of a document occurs by means of mutating their tree structure.

However, all these mutations are inter-mediated by explicit Edit operations. Each such operation has its inverse, which on one hand is used to implement the undo mechanism, and on the other – can be perceived as an interesting "document editing assembly language".

At the moment of writing this text, the language consists of the following (invertible) operations:

```
(Move from: Cursor to: Cursor with-shift: int)
(Insert element: (either pair HeadTailSeparator)
        at: Cursor)
(Remove element: (either pair HeadTailSeparator)
        at: Cursor with-shift: int := 0)
(ResizeBox at: Cursor := (the-cursor)
           from: Extent
           to: Extent
           with-anchor: real)
(InsertCharacter list: (list-of char)
                 after: Cursor := (the-cursor)
                 into: pair := (the-document))
(RemoveCharacter list: (list-of char)
                 before: Cursor := (the-cursor))
(SplitElement with: Space
              at: Cursor := (the-cursor))
(MergeElements removing: Space
               at: Cursor := (the-cursor))
```

Some of these operations are pairwise inverse (e.g. `Insert` and `Remove` or `SplitElement` and `MergeElements`), while others are self-inverse (e.g. `Move` or `ResizeBox`).

More details can be found in the source code of GRASP.

It is imaginable that some future version of GRASP could observe the actions performed by user and the structure of the document, and suggest certain operations based on previous actions (resembling Excel's auto-fill feature).

3 CURRENT PROGRESS

Although this paper could leave a different impression, at the moment of writing (February 2023) GRASP isn't yet a usable application, as:

- it doesn't let users open or save files
- it doesn't let users split or scroll the screen
- it doesn't let users evaluate expressions
- it doesn't support the basic gestures
- the extension mechanism isn't available

In certain areas, it also seems to have similar shortcomings:

- it doesn't support displaying nor editing comments
- although it should display improper lists correctly, editing them has not been tested well

Fortunately, there's still some time before European Lisp Symposium, which takes place late in April. Currently, the author envisions two milestones for the project:

(1) to reach the point that would let GRASP be used for developing itself
(2) to support extension mechanism and focus on the development of particular extensions

The author believes that reaching milestone 1 before ELS might be possible. A more detailed plan is the following:

- support for keyboard editing (mostly done)
- support for displaying and editing comments (they are already handled by parser)
- support for vertical keyboard movement (currently works somewhat but is a kludge)
- support for loading and saving files
- support for screen splitting and scrolling
- support for syntactic extensions provided by Kawa that are used in GRASP
- tests and bug fixes

4 RELATED WORK

The strongest source of inspiration for GRASP has been Emacs[14], and the Scheme interaction mode provided by the Geiser package. One motivation for the development of GRASP was the desire to share experience of Lisp interaction mode outside the world of Emacs, with possible improvements. (Some fundamental shortcomings of Emacs were pointed out with the announcement of Project Mage in the January of 2023[11].)

The desire to add interactive visual extensions was born when the author attempted to extend the idea of "evident programming" to the domain of computational geometry and graph algorithms.

However, the same idea was independently conceived by Leif Andersen, who implemented it in Dr Racket, and then

created a browser-based IDE called visr.pl (for Clojure). Leif also provided a very good explanation of the idea in a youtube video [1].

Interactive visual syntax is also a key feature of the Polytope editor developed by Elliot Evans. Polytope is a dedicated editor for JavaScript [8].

There are many similarities between GRASP and the Boxer environment developed at MIT in the 1980s by Andrea DiSessa and Harold Abelson [4]. Recently there have been efforts to resurrect Boxer within the Boxer Sunrise project run by Antranig Basman and Steven Ghitens [5]. However, building the project requires LispWorks, and pre-built snapshots are only released for MacOS X. Also, despite being written in Lisp, Boxer itself is not a Lisp interpreter.

There used to be a Boxer-inspired "integrated Scheme programming environment" called Bochser, developed by Michael Eisneberg in the 1980s at MIT[3].

Despite similarities, Bochser is a very different system than GRASP, and with very different goals.

Eisenberg's thesis contains a reference to another thesis, which presents Franklyn Turbak's "visual and manipulable model for Scheme programs" called GRASP[15]. It has even less in common with the system presented here than Bochser.

There are other interesting experiments in the area of representing programs. One example is the Fructure editor developed by Andrew Blinn for the Racket programming language (the editor itself is implemented in a purely functional way, using Racket's "big-bang" library)[6].

Another is OrenoLisp designed by Yasuyuki Maeda with the purpose of artistic live music coding[12].

There's a fun representation of ClojureScript programs as nested circles invented by Ella Hoeppner for her Vlojure editor[10].

Katie Bell created a browser-based structural editor for Python called SplootCode[2].

A lot of work concerning data visualization has been happening around the Smalltalk distribution called Pharo, and in particular its spin-off called Glamorous Toolkit, developed by Tudor Girba and his associates[9].

There's also a Visual Studio Code plug-in called "Debug Visualizer" developed by Henning Dieterichs[7]. It lets visualize various data structures during the execution of programs, and is available for the majority of mainstream programming languages.

While the scene of structural editing tools seems to be flourishing, the same cannot be said about development tools for mobile devices - most of existing tools seem to be shrinked versions of PC-based development environments and require external keyboard for comfortable work.

The only tool which stands out from this crowd that the author of this work knows about is MobileCode (formerly medc) developed by Mark Mendell[13], which is a vim-inspired touchscreen-based editor for C-like languages, capable of collapsing procedures and blocks of code.

ACKNOWLEDGEMENTS

The author would like to thank Shriram Krishnamurthi for reading the first draft of this work. He is also grateful to Andrew Blinn for his spiritual support throughout the whole development process, and the whole online scene of people interested in making programming a better experience for the future generations.

The terminal version of GRASP owes the use of Unicode box-drawing characters (rather than slashes and backslashes) to Job van der Zwan. After seeing the first prototype, Manuel Simoni recommended not to grow the boxes vertically with the increasing nesting level.

REFERENCES

[1] Leif Andersen, Michael Ballantyne, Mathias Felleisen, Adding Interactive Visual Syntax to Textual Code, Proceedings of the ACM on Programming Languages, Volume 4, Issue OOPSLA, Article No.: 222, pp 128, https://doi.org/10.1145/3428290,
presentation: https://www.youtube.com/watch?v=8htgAxJuK5c
defense talk: https://www.youtube.com/watch?v=l0GfMs82PvU
online IDE: https://visr.pl
[2] Katie Bell, SplootCode, https://splootcode.io
[3] Michael Eisenberg, Bochser: An Integrated Scheme Programming System, MIT 1985, https://boxer-project.github.io/boxer-literature/theses/Bochser,AnIntegratedSchemeProgrammingSystem(Eisenberg,MITMSc,1985).pdf
[4] Andrea DiSessa, Harold Abelson, Boxer: A Reconstructible Computational Medium, MIT 1986, https://web.media.mit.edu/~mres/papers/boxer.pdf
[5] Antranig Basman, Steven Ghitens, Boxer Sunrise Project https://github.com/boxer-project/boxer-sunrise
[6] Andrew Blinn, Fructure: A Structure Editing Engine in Racket source code: https://github.com/disconcision/fructure
2019 RacketCon presentation: https://www.youtube.com/watch?v=CnbVCNIh1NA
[7] Henning Dieterichs, Debug Visualizer for Visual Studio Code https://marketplace.visualstudio.com/items?itemName=hediet.debug-visualizer
[8] Elliot Evans, Polytope, https://elliot.website/editor/
[9] Tudor Girba, Glamorous Toolkit, https://gtoolkit.com
[10] Ella Hoeppner, Vlojure: A New Way to Write Clojure, presentation: https://www.youtube.com/watch?v=1OcAUhe3E1E
online IDE: https://vlojure.io/
[11] Dmitrii Korobeinikov, Emacs is Not Enough, Project Mage, 2023, https://project-mage.org/emacs-is-not-enough
[12] Yasuyuki Maeda, OrenoLisp, https://www.youtube.com/watch?v=RuU0HI-paik
[13] Mark Mendell, MEDC project website: https://medc.mark.dev/ presentation: https://vimeo.com/641790697
[14] Richard Stallman, EMACS: The Extensible, Customizable Display Editor, 1981, https://www.gnu.org/software/emacs/emacs-paper.html
[15] Franklyn Turbak, GRASP: A Visible and Manipulable Model for Procedural Programs, MIT 1986 https://cs.wellesley.edu/~fturbak/pubs/turbak-masters-thesis.pdf

Tuesday, 25 April 2023

Demonstration: A stepper for Armed Bear Common Lisp (ABCL)

Alejandro Zamora Fonseca

ale2014.zamora@gmail.com

ABSTRACT

In this paper a stepper tool for the Armed Bear Common Lisp (ABCL) implementation is proposed, describing its features and implementation related details. ABCL does not currently have a stepper and the addition of one can help improve the quality of the code designed to run in the implementation, and also it can be useful to assist in the debugging process of Common Lisp (CL) portable code.

CCS CONCEPTS

• **Software and its engineering** → *Software notations and tools*;

KEYWORDS

Common Lisp, stepper, debugging

ACM Reference Format:

Alejandro Zamora Fonseca. 2023. Demonstration: A stepper for Armed Bear Common Lisp (ABCL). In *Proceedings of the 16th European Lisp Symposium (ELS23).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.7815887

1 INTRODUCTION

A stepper is a tool that allows to control and follow step by step the execution of a subprogram. Many programming languages provide stepping tools as part of its debugging mechanisms. CL is not an exception and includes the `step` macro in its standard to be optionally implemented.

ABCL does not include a stepper and this fact has been mentioned as one of the reasons that detract the implementation from being a proper contemporary CL implementation, see [1].

This paper introduces a stepper to address this issue in ABCL. I will describe its features, examples of use and some implementation details.

ELS23, Apr 24–25 2023, Amsterdam, Netherlands

https://doi.org/10.5281/zenodo.7815887

The code for this tool is shared as open source in the form of a pull request on ABCL's own code in its Github repository (see [2]).

2 RELATED WORK

Recently, João Távora [3] made a great summary of the state of art of stepping in CL and presented a visual and portable stepper module (sly-stepper) for his IDE Sly.

Unfortunately sly-stepper does not support ABCL and the other efforts mentioned in that paper appear to be incomplete or difficult to integrate into any CL implementation.

3 DESIGN AND IMPLEMENTATION

In my opinion, a stepper is needed for ABCL to improve the quality of the implementation, to bring more debugging tools for users of ABCL or other CL systems, and to lay the groundwork for integrations with external IDEs like Sly, Slime and others. Having a simple stepper in text mode shipped with the implementation, would also allow developers to not depend on any external tools for this task, regardless of the environment in which the implementation will run.

In this section I'll describe details in the design and implementation of the stepper and its features.

This step tool is integrated in the evaluation code in ABCL's main evaluator which is, at its core, an interpreter. The evaluator works by traversing the successive subforms in interpreted code after the macroexpansion, but it cannot go inside compiled functions, which are executed from its Java bytecode. For this reason the stepper will not enter either into compiled code.

Internally, when the user runs the `step` macro, the interpreter first sets itself to stepping mode and will allow the user to proceed to step through each sub-form according to her needs. The stepping related code is implemented to be called from selected stages of the evaluator. And finally after the form is executed, the stepping mode is disabled.

The following diagram illustrates an overview of the stepper arquitecture. The stepper hooks created in the middle of the evaluator were used to call the component that implements the logic of the options in the stepper (`Handle Stepping`), based on the result of the component `Step in symbol ?`.

Both components manipulate the internal states of the stepper in the evaluator (Lisp.java), which are mainly two flags to control the `step` and `next` features.

Most of the code for this tool was done in Lisp (abcl-stepper.lisp), taking advantage of the nice API that ABCL provides for developers to interact between Java and Lisp. Namely, the ability to create `Primitive` methods in Java that can be easily called from Lisp as functions was essential. On the other hand, the constructions to call Java objects and methods from Lisp were also useful.

See Figure 1.

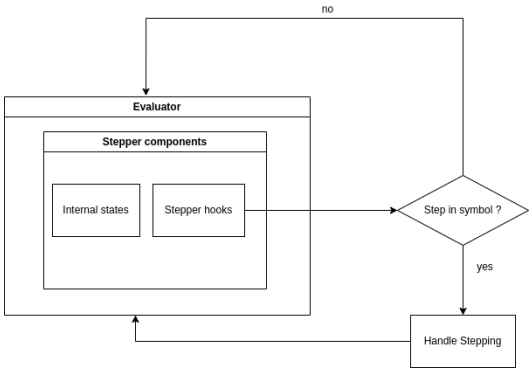


Figure 1: The arquitecture of the stepper

This approach was used due to its simplicity, as opposed to others that need to instrument the subforms of the code to step, in order to perform the stepping. Taking advantage of the built-in evaluator makes it possible to step into any interpreted code without needing to instrumenting it.

On every stage of the stepping process, the user will be prompted with a screen like the one in Listing 1.

```
We are in the stepper mode
Evaluating step 1 -->
(TEST)
Type ':?' for a list of options
```

Listing 1: The head of the prompt.

If the user presses `:?`, the system will show a simplified help with a list of the features present in the stepper, see Listing 2

```
Type ':l' to see the values of bindings on the local environment
Type ':c' to resume the evaluation until the end without the stepper
Type ':n' to resume the evaluation until the next form previously selected
↪  to step in
Type ':s' to step into the form
Type ':i' to inspect the current value of a variable or symbol
Type ':b' to add a symbol as a breakpoint to use with next (n)
Type ':r' to remove a symbol used as a breakpoint with next (n)
Type ':d' to remove all breakpoints used with next (n)
Type ':w' to print the value of a binding in all the steps (watch)
Type ':u' to remove a watched binding (unwatch)
Type ':bt' to show the backtrace
Type ':q' to quit the evaluation and return NIL
Type ':?' for a list of options
```

Listing 2: Minimal help option.

Now the rest of the options will be described in the following subsections.

3.1 Locals bindings

The `:l` option will display the local bindings for variables and functions in the current environment passed to the current form to evaluate, a typical response would look like as described in Listing 3:

```
Showing the values of variable bindings.
From inner to outer scopes:
N=2
Showing the values of function bindings.
From inner to outer scopes:
FLET1=#<FUNCTION #<(FLET FLET1) {3ACE0BC7}> {3ACE0BC7}>
```

Listing 3: Local bindings option.

3.2 Continue to the end

The continue `:c` option will, basically, ignore the stepping process and perform the evaluation of the form without any stop.

3.3 Stop at next marked symbol

The next `:n` feature allows to stop the stepper only when the interpreter is analyzing one of the symbols specified in the list of `stepper::*stepper-stop-symbols*` or any of the exported symbols presented in any of the list of packages specified in `stepper::*stepper-stop-packages*`. These variables will have initially the value `NIL` and, if left unchanged, `next` will behave almost exactly as `continue`. It is useful when we want to step over large or complex code and avoid stepping every form in order to jump only to the interested ones. This feature will be explained more in detail in next sections.

In the middle of the stepping process it is possible to change the value of the variable `stepper::*stepper-stop-symbols*`, using the options `:b`, `:r` and `:d`. The `:b` option allows to add a symbol to `stepper::*stepper-stop-symbols*`, option `:r` will remove a symbol in `stepper::*stepper-stop-symbols*` and the option `:d` will remove all the symbols in the aforementioned variable.

3.4 Step into the form

The step `:s` functionality is the most basic operation in the stepper, it will step into the current form until the evaluation ends. It can even step into ABCL internal functions.

3.5 Inspect variables

This feature (`:i`) allows one to inspect the content of a variable or binding, present in the current environment. It will first ask the user to type the symbol to inspect and proceed to print its value.

Some screens as quick examples (Listing 4)

```
Type ':?' for a list of options
:i
Type the name of the symbol: *some-var*
NIL
Type ':?' for a list of options
:i
Type the name of the symbol: x
3
```

Listing 4: Inspect variable option.

3.6 Show backtrace

The `:bt` option provides the ability to print the current backtrace, which is useful for analyzing the evaluation path until the current stepping point.

3.7 Watch and (un)watch

The feature `watch` allows to follow the values of a variable in all the steps, the user can add a variable to watch by typing `:w` and when prompted, the symbol to watch. After that, the user can remove the variable from being watched by using the `:u` option and entering the same symbol.

3.8 Quit evaluation

The quit `:q` feature will abort the evaluation in the stepper and return NIL. This is useful to avoid running the remaining forms in the code when the user wants to leave the stepper, especially if the rest of the program is performing expensive operations.

3.9 Examples of usage

This subsection explains in details, by using some examples, the features of the current stepper. The examples shown here will be using the pure ABCL's REPL but will behave the same if you use the shell buffer in Emacs for a slightly more comfortable development environment.

First let's examine the `inspect` feature combined with the `step` feature.

In this example we can see how the inspect feature is used and it retrieves correctly the values for the lexical variable `x` and the special variable `*some-var*` which is rebinded in the code. The values are shown using `cl:print`. Listing 5

As a second example the use of the `list locals` feature will be illustrated. Here, we can observe that the list of local bindings include variable and function bindings and they are showed from inner to outer scopes, for that reason the value of variable `a` is first `2` and later `1`. See Listing 6

In the previous example the use of the feature `continue` was shown too, allowing to complete the evaluation of the form turning off the stepper.

The following stepper session will be used to explain the `next` and `quit` features. They allow to stop the execution only in designed symbols. It behaves like if we were adding

```
CL-USER(1): (require :asdf)
NIL
CL-USER(2): (require :abcl-contrib)
NIL
CL-USER(3): (require :abcl-stepper)
NIL
CL-USER(4): (defparameter *some-var* 1)
*SOME-VAR*
CL-USER(5): (defun test ()
  (let ((*some-var* nil)
        (x 3))
    (list *some-var* 3)))
TEST
CL-USER(6): (stepper:step (test))
We are in the stepper mode
Evaluating step 1 -->
(TEST)
Type ':?' for a list of options
:i
Type the name of the symbol: *some-var*
1
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 2 -->
(BLOCK TEST
  (LET ((*SOME-VAR* NIL) (X 3))
    (LIST *SOME-VAR* 3)))
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 3 -->
(LET ((*SOME-VAR* NIL) (X 3))
  (LIST *SOME-VAR* 3))
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 4 -->
(LIST *SOME-VAR* 3)
Type ':?' for a list of options
:i
Type the name of the symbol: x
3
Type ':?' for a list of options
:i
Type the name of the symbol: *some-var*
NIL
Type ':?' for a list of options
:c
step 4 ==> value: (NIL 3)
step 3 ==> value: (NIL 3)
step 2 ==> value: (NIL 3)
step 1 ==> value: (NIL 3)
(NIL 3)
```

Listing 5: Step and inspect features.

breakpoints for the stepping process. Let's look at the stops in this example. The stepper is stopping in the call with the symbol `'step-next::loop-1` because it was added to `stepper::*stepper-stop-symbols*`. It is also stopping in the call with the symbol `'step-next::loop-3` because that symbol was exported in the package `next-step` and the symbol was added to `stepper::*stepper-stop-packages*`. `'step-next::loop-2` is skipped when using `next` because it is not present in any of the lists of symbols mentioned before.

We can observe as well the use of the feature `quit`. After the use of it, the evaluation was stopped before the initialization of the special variable `step-next::*test-next-var*` and therefore it is not bound yet after complete the stepping process. See Listing 7

If we observe the second stepper call in the `test-next` function, we can see the use of the `:b` and `:r` features. Using the `:b` option adds a breakpoint to the symbol `step-next::loop-2`, the breakpoint to `step-next::loop-1` is removed using the

```
CL-USER(7): (stepper:step (flet ((flet1 (n) (+ n n)))
                    (let ((a 1))
                      (let ((a 2))
                        (+ (flet1 3) a)))))
We are in the stepper mode
Evaluating step 1 -->
(FLET ((FLET1 (N) (+ N N)))
  (LET ((A 1))
    (LET ((A 2))
      (+ (FLET1 3) A))))
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 2 -->
(LET ((A 1))
  (LET ((A 2))
    (+ (FLET1 3) A)))
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 3 -->
(LET ((A 2))
  (+ (FLET1 3) A))
Type ':?' for a list of options
:s
We are in the stepper mode
Evaluating step 4 -->
(+ (FLET1 3) A)
Type ':?' for a list of options
:l
Showing the values of variable bindings.
From inner to outer scopes:
A=2
A=1
Showing the values of function bindings.
From inner to outer scopes:
FLET1=#<FUNCTION #<(FLET FLET1) {238E109B}> {238E109B}>
Type ':?' for a list of options
:c
step 4 ==> value: 8
step 3 ==> value: 8
step 2 ==> value: 8
step 1 ==> value: 8
8
CL-USER(8):
```

Listing 6: Local bindings feature.

option `:r` and, after executing the command `:n`, the stepper stops this time at `step-next::loop-2` instead of `step-next::loop-1`. See Listing 8.

The following example exhibits the use of the `backtrace` feature. This option allows to visualize the full evaluation path to the point of the program being analyzed by the stepper. See Listing 9.

The last example presents the `watch`(`:w`) feature which permits to monitor the values of a variable in the stepping process. In the code sample it can be seen the succesive values of the variable `x` and how they are removed after the `unwatch`(`:u`) option is applied. See Listing 10.

4 CONCLUSION

A first functional stepper for ABCL has been introduced. It can help users to dig into the guts of every complex code to help find the root cause of errors.

I think that even knowing that this is the first version, it can be usable and offer an alternative to debug complex systems built using ABCL or even portable CL code.

```
CL-USER(8): (defpackage step-next (:use :cl))
#<PACKAGE STEP-NEXT>
CL-USER(9): (in-package :step-next)
#<PACKAGE STEP-NEXT>
STEP-NEXT(10): (defun loop-1 (a b)
  (loop :for i :below a
        :collect (list a b)))
LOOP-1
STEP-NEXT(11): (defun loop-2 (a)
  (loop :for i :below a
        :collect i))
LOOP-2
STEP-NEXT(12): (defun loop-3 (n &optional (times 1))
  (loop :for i :below times
        :collect times))
LOOP-3
STEP-NEXT(13): (defun test-next (n)
  (loop-1 (1+ n) n)
  (loop-2 (1- n))
  (loop-3 n 3)
  ;; quit (q) here
  (defparameter *test-next-var* (loop :for i :below (expt 10 6)
                                      :collect i)))
TEST-NEXT
STEP-NEXT(14): (push 'loop-1 stepper::*stepper-stop-symbols*)
(LOOP-1)
STEP-NEXT(15): (export 'loop-3)
T
STEP-NEXT(16): (push 'step-next stepper::*stepper-stop-packages*)
(STEP-NEXT)
STEP-NEXT(17): (stepper:step (test-next 7))
We are in the stepper mode
Evaluating step 1 -->
(TEST-NEXT 7)
Type ':?' for a list of options
:n
We are in the stepper mode
Evaluating step 2 -->
(LOOP-1 (1+ N) N)
Type ':?' for a list of options
:n
step 2 ==> value: ((8 7) (8 7) (8 7) (8 7) (8 7) (8 7) (8 7) (8 7))
We are in the stepper mode
Evaluating step 3 -->
(LOOP-3 N 3)
Type ':?' for a list of options
:q
NIL
STEP-NEXT(18): (assert (not (boundp '*test-next-var*)))
NIL
STEP-NEXT(19):
```

Listing 7: Next and quit features.

```
STEP-NEXT(19): (stepper:step (test-next 7))
We are in the stepper mode
Evaluating step 1 -->
(TEST-NEXT 7)
Type ':?' for a list of options
:b
Type the name of the symbol to use as a breakpoint with next (n): loop-2
Type ':?' for a list of options
:r
Type the name of the breakpoint symbol to remove: loop-1
Type ':?' for a list of options
:n
We are in the stepper mode
Evaluating step 2 -->
(LOOP-2 (1- N))
Type ':?' for a list of options
:n
step 2 ==> value: (0 1 2 3 4 5)
We are in the stepper mode
Evaluating step 3 -->
(LOOP-3 N 3)
Type ':?' for a list of options
:q
NIL
STEP-NEXT(20):
```

Listing 8: Next and quit features.

5 FURTHER WORK

The current stepper is implemented in a way that blocks any remaining threads in the system until the stepping process

```
STEP-NEXT(20): (defun test-backtrace (x)
  (labels ((f1 (x) (f2 (1+ x)))
           (f2 (x) (f3 (* x 3)))
           (f3 (x) (+ x 10)))
    (f1 x)))
TEST-BACKTRACE
STEP-NEXT(21): (stepper:step (test-backtrace 3))
We are in the stepper mode
Evaluating step 1 -->
(TEST-BACKTRACE 3)
Type ':?' for a list of options
:b
Type the name of the symbol to use as a breakpoint with next (n): +
Type ':?' for a list of options
:n
We are in the stepper mode
Evaluating step 2 -->
(+ X 10)
Type ':?' for a list of options
:bt

(#<LISP-STACK-FRAME ((LABELS F3) 12) {758EDEFD}>
 #<LISP-STACK-FRAME ((LABELS F2) 4) {6CD6F8DD}>
 #<LISP-STACK-FRAME ((LABELS F1) 3) {3B96D1EB}>
 #<LISP-STACK-FRAME (TEST-BACKTRACE 3) {6D76BF34}>
 #<LISP-STACK-FRAME (SYSTEM::%EVAL (ABCL-STEPPER:STEP
   (TEST-BACKTRACE 3))) {64C522B}>
 #<LISP-STACK-FRAME (EVAL (ABCL-STEPPER:STEP
   (TEST-BACKTRACE 3))) {387EC7BA}>
 #<LISP-STACK-FRAME (SYSTEM:INTERACTIVE-EVAL
   (ABCL-STEPPER:STEP (TEST-BACKTRACE 3))) {356A40D7}>
 #<LISP-STACK-FRAME (TOP-LEVEL::REPL) {6DBDC651}>
 #<LISP-STACK-FRAME (TOP-LEVEL::TOP-LEVEL-LOOP) {9840CC7}>)
Type ':?' for a list of options
:c
step 2 ==> value: 22
step 1 ==> value: 22
22
STEP-NEXT(22):
```

Listing 9: Show backtrace feature

is finished. This was done by simplicity in the design and to avoid unpleasant race conditions on the internal states. Changing it to a non-blocking version, would be more flexible for users, especially when debugging systems in production.

Include other well known step features in other implementations like step-out and step-next (move to the next form avoiding step-into)

Implement the evaluation of custom expressions in the current environment.

Find a way to integrate it with Sly/Slime. Currently, when called inside Sly/Slime REPL it will only show print a message and return the form without any stepping. Also find a way to abstract the integration with any IDE.

6 ACKNOWLEDGEMENTS

I would like to thank my wife Valeria, who helped me in the general design of the features and lovingly motivated me to complete the implementation and this paper.

Also to the Common Lisp community for providing me all the software, documentation and support to every doubt I had in my learning all these years.

```
STEP-NEXT(22): (defun test-watch ()
  (let ((x 1))
    (setq x 3)
    (setq x 7)
    (setq x 21)
    x))
TEST-WATCH
STEP-NEXT(23): (stepper:step (test-watch))
We are in the stepper mode
Evaluating step 1 -->
(TEST-WATCH)
Type ':?' for a list of options
:w
Type the name of the symbol to watch: x
Type ':?' for a list of options
Watched bindings:
Couldn't find a value for symbol X
:s
We are in the stepper mode
Evaluating step 2 -->
(BLOCK TEST-WATCH
  (LET ((X 1))
    (SETQ X 3)
    (SETQ X 7)
    (SETQ X 21)
    X))
Type ':?' for a list of options
Watched bindings:
Couldn't find a value for symbol X
:s
We are in the stepper mode
Evaluating step 3 -->
(LET ((X 1))
  (SETQ X 3)
  (SETQ X 7)
  (SETQ X 21)
  X)
Type ':?' for a list of options
Watched bindings:
Couldn't find a value for symbol X
:s
We are in the stepper mode
Evaluating step 4 -->
(SETQ X 3)
Type ':?' for a list of options
Watched bindings:
X=1
:s
step 4 ==> value: 3
We are in the stepper mode
Evaluating step 5 -->
(SETQ X 7)
Type ':?' for a list of options
Watched bindings:
X=3
:u
Type the name of the symbol to (un)watch : x
Type ':?' for a list of options
:s
step 5 ==> value: 7
We are in the stepper mode
Evaluating step 6 -->
(SETQ X 21)
Type ':?' for a list of options
:s
step 6 ==> value: 21
step 3 ==> value: 21
step 2 ==> value: 21
step 1 ==> value: 21
21
STEP-NEXT(24):
```

Listing 10: Watch feature

REFERENCES

[1] Abcl manual. URL https://abcl.org/releases/1.9.0/abcl-1.9.0.pdf.
[2] Pr with the stepper code. URL https://github.com/armedbear/abcl/pull/568.
[3] João Távora. A portable, annotation-based, visual stepper for common lisp. 2020. URL https://zenodo.org/record/3742759.

Experience Report: Kandria - A Game in Common Lisp

Nicolas “Shinmera” Hafner

shinmera@tymoon.eu

Shirakumo.org

Zürich, Switzerland

ABSTRACT

In this paper we outline the experience we’ve gathered while developing Kandria using Common Lisp. Kandria is a video game in the “action RPG” genre and has been developed for Windows and Linux with the SBCL implementation. Being a video game, the project touches a unique combination of disciplines within computer science, and as such provides an in-depth and comprehensive view of the development process of a large-scale project in Lisp, and the general language ecosystem.

CCS CONCEPTS

• **Computing methodologies** → *Computer graphics*; • **Computer systems organization** → *Real-time system architecture*; • **Software and its engineering** → **Application specific development environments**; Object oriented development; Error handling and recovery; Software development methods.

KEYWORDS

Common Lisp, Games, Video Games, Computer Graphics, Experience Report

ACM Reference Format:

Nicolas “Shinmera” Hafner. 2023. Experience Report: Kandria - A Game in Common Lisp. In *Proceedings of the 16th European Lisp Symposium (ELS’23)*. ACM, New York, NY, USA, 7 pages. https://doi.org/10.5281/zenodo.7816871

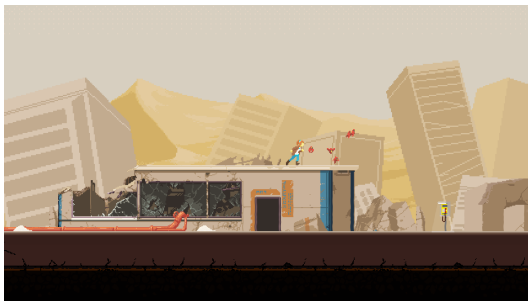


Figure 1: A screenshot of Kandria

1 INTRODUCTION

Kandria is a video game in the “action RPG” genre that was released worldwide on the Valve Steam platform for Windows and Linux in January of 2023, receiving very positive reviews. The game was developed using the Trial engine and thus relies almost entirely upon code and libraries written in Common Lisp – to our knowledge the first commercial game like this to be released.

Games lie at an intersection of many different computer science disciplines such as audio, graphics, interfaces, soft real-time, artificial intelligence, and more. As such games provide a unique challenge in combining all of these disciplines together into one product, and ultimately shipping this product to paying customers.

In this paper we will outline the challenges we faced in realising Kandria as related to Common Lisp, discuss advantages of our approach, and take a look at the work still ahead of us in expanding the capabilities of our engine to better address the requirements of even more complex games we would like to develop in the future.

In particular we will note our own experiences with a few common Boogeymen, such as the overhead of CLOS dispatch, the pause times of GC, the maturity of the library ecosystem, and the stability and size of deployed binaries.

Since this is an experience report, we cannot present any specific figures on performance characteristics, statistics on used projects, or some sort of unified thesis. Instead we hope that this insight will be valuable for readers to gain a better understanding of the complexities involved, the benefits we have identified with Lisp over other ecosystems, and, most importantly, the areas in which work still needs to be done.

All of the work we’ve done, including Trial[15], and even Kandria[10], is open source and available on our GitHub[6], in the hopes that it will inspire others to create new projects based upon them.

2 RELATED WORK

In [20] we discussed many similar points that this paper touches upon in a format more suitable for people unfamiliar with the intricacies of lisp and especially Common Lisp.

Strandh[24] proposes a new approach to improve the performance characteristics of generic function dispatch. Since we make heavy use of CLOS in our game, such improvements are very relevant.

ELS’23, Apr 24–25 2023, Amsterdam, Netherlands

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.7816871

Patton[23] are working on a new parallelisation of the SBCL[13] garbage collector to allow for faster collection and thus reduced pause times. GC pause times are a frequently reported issue in games, as they can cause unexpected latency, and thus lead to drops in the framerate.

Mäkelä[21] describe their experience in developing a game using the open source game engine Godot[8] with C# in their BSc. thesis. Godot is currently the leading open source game engine, and is striving to be a viable alternative to commercial products like Unity and Unreal.

Craighead et al.[18] cover a case study of using Unity[16], a proprietary game engine, to build a small game. Unity is currently most-used game engine for small to mid-sized games.

Nurminen[22] describe their experience developing and deploying a Common Lisp application to users.

3 LIBRARY ECOSYSTEM

In this section we will outline our general experience working in the Common Lisp ecosystem, and particularly the contributions we've made to it in order to implement Kandria.

Out of the 110 libraries Kandria depends upon, 54 were written by us. This does mean that we've spent a rather significant amount of time "yak shaving" and creating libraries to fulfil a variety of needs that were, prior to their creation, either completely unfulfilled, or unsatisfactorily so. We will go into detail on some of the more important ones in the following subsections.

While this is undeniably a lot of work, it has been done now and is available to others. Furthermore, of the other half of libraries that we were not authors of, the vast majority have been extremely stable. We only needed to supply extremely minimal patches to a select few libraries, most of which were reviewed and accepted relatively swiftly.

Among this we also count the SBCL implementation itself. While we strive to write implementation independent code in our separate libraries, for Kandria itself we decided to only focus on SBCL in order to reduce the development overhead. SBCL offers great native code performance and is available for all platform configurations that we require. And, perhaps most importantly, it is very actively maintained. Most of the issues we've encountered in releases were usually fixed within a few weeks, if not days.

We are also actively investigating the possibility of porting SBCL and Kandria to the proprietary Nintendo Switch platform, though due to non-disclosure agreements we are unfortunately not at liberty to speak of the specifics involved in that at this time.

Overall while we certainly had to create a lot of libraries to fulfil our needs, and those libraries presented a significant amount of effort to implement, we remain convinced that we were only able to implement these systems in a respectable amount of time due to the convenience factors that Lisp offers us.

We'll now touch on a few specific areas that we developed libraries for. This is by no means exhaustive, but we consider these to be the most relevant to the general community.

3.1 Math Libraries

While there is no shortage of math libraries available to Common Lisp, especially linear algebra implementations, most of those libraries focus on large scale scientific computing, often by integrating with foreign libraries such as BLAS and LAPACK. For basic computer graphics and especially games, this is overkill. Most of the linear algebra stays within the confines of 2×2, 3×3, 4×4 matrices, and 2, 3, and 4 element vectors.

It is more important to provide a very convenient interface that allows us to perform computations on these elements with adequate speed. Back when Trial started out, no linear algebra libraries with such a limited focus existed, and as such the 3d-Vectors and 3d-Matrices libraries were born. We have since extended this set of libraries to include 3d-Quaternions for common operations with rotations, and 3d-Transforms, for the convenient encapsulation of a "transform gizmo" that can represent rotation, translation, and scaling without gimbal Lock.

All of these libraries rely very heavily on macros to reduce code duplication and automatically generate code for loop unrolling and other common tactics in linear algebra code. The current versions of these libraries all work by emitting an `etypecase` for every operation, which then handles dispatch based on the provided argument types. The operation functions are inlined, such that the compiler can eliminate the dispatch altogether if the argument types are locally known. This allows us to provide a generic interface to the user that's quite convenient, while still staying competitive in performance critical sections.

Unfortunately this approach, while portable, is also riddled with issues: since every operation is inlined, this leads to explosive code growth for the compiler before it can reduce the code back down again by eliminating superfluous dispatch `etypecases`. Type inference is also much more complicated, and stack allocation is usually only possible via careful manual rewriting of the operations involved. The libraries are also limited to a single float type, meaning you can by default only create vectors with single-floats as elements.

In this case the lack of static typing facilities and lack of portable compiler hooks for integrating with type inference really hurts the compile speed, implementation clarity, and ultimate performance of the resulting code.

We have started work on a full rewrite of all libraries that take a fundamentally different approach: instead of emitting etypecases and relying on inlining, we create a sort of "template mechanism" by which we can generate all possible permutations of a singular function for all involved argument types. This gives us very tiny, but perfectly optimised base functions for all required operations. We then create a dispatcher function on top which falls back to emitting an `etypecase` on other implementations, but will hook into SBCL's `deftransform` and similar facilities to better handle expansion and type propagation. Finally we create variadic functions on top of the dispatchers which transform any possible variadic call into calls to dispatched two-argument operation functions, while retaining proper type propagation.

Ultimately this results in much more easily understandable and performing code. However, it still comes at a cost: because we cannot know ahead of time which type combinations and operations the user will actually need, we must generate all possible combinations ahead of time. This potentially results in thousands of functions being generated that are never used. We would like a compiler facility that allows us to hook into the expansion process of a function call to generate the required permutations on-demand.

While such a facility is conceivable, and making it work for unknown runtime types by delayed compilation is, too, we currently have no plans to implement such an advanced strategy. More importantly, we hope that at some point in the future implementors can come to some kind of consensus that would allow a portability library to expose a similar mechanism to SBCL's `deftransform` for faster, more convenient, type-inference-informed call expansion.

3.2 User Interfaces

When Trial initially began its development in 2016 it was directly integrated with the Qt4 UI toolkit (via CommonQt / Qtools). Since Qt4 is a rather large dependency that not only increases the deployed package size, but also invites a lot of C and C++ interoperation that can cause hard to debug issues, and has not been maintained in many years, it was quickly abandoned, however. These days Trial does not depend on a UI toolkit directly, or even a specific backend for its OpenGL use.

Aside from CommonQt, the options for a user interface in Common Lisp were, and remain, limited: GTK runs the same issues as Qt, though with even worse deployment and support aspects, LTK is far too limited and cannot integrate with OpenGL at all, McCLIM similarly does not possess an OpenGL backend or method of integration and is still very limited in its theming capabilities, and the newly established CLOG requires a browser to be shipped, which is far too heavyweight.

As a result we decided to implement our own toolkit, called Alloy. Alloy is separated into different protocols, with the core being completely independent of any rendering or input method, instead only handling the layout decisions, the input handling via a generic event protocol, and a system to dynamically react to changes in the represented data.

How visual elements can be rendered is then offloaded to several other protocols: the "Simple Rendering Protocol" which provides a basic text and shape rendering API, the "Presentations Protocol" which allows users to describe how to compose the look of a visual element via the basic shapes from the Simple protocol, and the "Animations Protocol" which allows users to describe how shapes change over time as properties of a visual element are changed.

Finally, the Simple protocol needs to be implemented by a backend, such as the "OpenGL Renderer" to actually provide a way to draw the shapes in some way, the Core protocol needs to receive input events from the surrounding context to actually react to user input, and some method of rendering and layouting text needs to be provided. Text in particular is separated out as it is a very complex topic in its own right.

This separation of concerns via protocols implemented in CLOS allows us to make Alloy far more amenable to being ported to different backends. In particular, it allows us to use Alloy in contexts that are otherwise rather unusual for UI toolkits, such as within a game where the actual operating system interaction and rendering logic cannot be directly controlled by the toolkit itself, but is instead handled by the game engine.

Usual desktop UI toolkits are rather cumbersome to style effectively, as most desktop applications are expected to present themselves in a "native look and feel", while games are expected to have rather elaborately customised and animated interfaces. Alloy's presentations protocol allows us to style the UI quite extensively. Thanks to macros we can present the user with a declarative style interface to define these behaviours with relative ease.

This protocol separation would not be possible to implement without the high degree of flexibility CLOS offers us in combining behaviours, and even without advanced techniques such as shadow mixins and metaclasses.

3.3 Audio Processing

Of the entire system, audio processing is where the toughest constraints apply, as audio is very sensitive to latency. We cannot process audio in large buffers to smooth over processing hiccups, as then sound effects would be desynchronised from their visual counterparts. Audio processing can also often be very computation intensive, and benefits greatly from vectorisation.

For these reasons we have decided to write the main bulk of our processing as a C library, rather than Common Lisp, as at the time it was the fastest way for us to get the kinds of performance constraints met and the kinds of capabilities we needed. Now with the advent of the sb-simd contrib, writing a competitive alternative (albeit constrained to SBCL) may be feasible.

In any case, our library, libmixed, follows a very strict C style and is extensible and interoperable from Lisp. You can write "segments" that process audio from Lisp as well, and integrate with the rest of the processing system neatly.

This allows us to write the more hairy parts such as audio format decoding, and audio playback in Lisp, instead, without sacrificing the performance gains in the bulk of the processing. Libmixed then takes care of unpacking and decoding audio streams, applying a variety of effects, mixing multiple audio streams together, and even resampling everything to fit into the expected, and often differing, sample rates at the input and output points.

Libmixed also includes a full introspection API, allowing us to not only put the audio processing pipeline together from Lisp, but also to interactively inspect the state of the processing segments and modify them at runtime. We can also use the extensibility to prototype new effects from Lisp before lowering them down to C should the performance requirements be strict enough.

On the Lisp side we have also crafted a higher-level system called Harmony, which makes it trivial to put together these processing pipelines, and interactively play music and sound effects back. In Kandria we use these pipelines for a variety of effects such as cross fading multiple music layers (called "horizontal mixing"), ducking the audio with a low pass filter when the player is underwater, and slowing down audio playback when the player enters a slow motion segment.

3.4 Operating System Interaction

In order to output sound and graphics, process input, and receive user information we need to interact with the surrounding operating system. On all three targets we care about, this is done via calls to a C API of some sort. Thanks to CFFI it is usually unnecessary to actually rely on an external C library to do so, and we can instead code the interaction directly in Lisp, retaining the interactive implementation and debugging.

Despite this though, the interaction still involves C and the usual memory safety perils, and depending on the interface the documentation often leaves things to be desired, especially on MacOS. So while our implementation definitely benefits from a much faster retry cycle, it is still a very arduous process to write these OS interoperation libraries.

Particularly we had to write libraries to do the following:

(1) Input handling for game controllers. This is rather involved, especially on Windows, where multiple APIs need to be supported simultaneously.[1]

(2) Audio output. We've implemented several different output APIs on both Linux and Windows, as both systems can differ on their supported output interfaces depending on version and setup.[11][2][9]

(3) Native dialog boxes. In order to show emergency error boxes or prompt the user for files.[12][4]

(4) Font discovery. To search the available fonts for a matching set we need to query operating system APIs.[5]

(5) Language querying. To ensure the game launches with the user's language (if supported of course), we again need to query the environment for the preferred localisation.[14]

For graphics output and general window interaction we currently rely on the GLFW[7] C library, as it has proven extremely stable and portable. It is conceivable that we will replace this at some point to reduce C dependencies, but given that we've had zero issues arising from its use, this is rather low priority for us.

3.5 Service Integration

In order to publish a game on the Steam platform, you must integrate with their SteamWorks SDK. The actually required amount of integration is very minimal, and merely involves loading their shared library and calling a single function. However, the SteamWorks SDK offers a lot of other functionality that games can make use of, like social networking features, user generated content sharing, multiplayer systems, and more.

Unlike the prior OS interfaces, the default expected interaction mode with the SteamWorks SDK is via C++, which is quite difficult to talk with directly via CFFI. Fortunately, they also offer a raw C API, but this API is not fully documented and fraught with strange gotchas. The SDK does ship a "machine readable" description of the endpoints, structures, and types in the form of a JSON file, but as we have found this file is both incomplete and partially incorrect.

In our implementation we analyse the JSON file along with a manually supplied file with the lacking data, and a couple of the shipped static headers to automatically generate the CFFI wrapper data structures, types, constants, and functions. Since the analysis of these files is rather involved however, we have chosen not to use macroexpansion, but rather provide a separate system which emits Lisp code to a new file.

In addition to this generated interface, we also had to supply a minimal "shim" C library that handles a few select API calls that rely on structure-by-value passing, a feature which is not natively supported on most Lisp implementations at this point. Another workaround would be to use libffi, but libffi comes with its own issue, and would be another C library to depend on anyhow, so we opted for the much simpler and easier to understand alternative of writing our own minimal shim library.

In Kandria we make use of the extended services to present the user with an on-screen keyboard when they are using a controller, to read out the username for default save file naming, and to integrate with the Steam platform's "achievements" system, which players have come to expect.

4 COMMON LISP OBJECT SYSTEM

In both Trial and Kandria we make rather extensive use of CLOS, both its basic features of classes, multiple dispatch, and deep hierarchies via mixins, and the advanced features of metaclasses. For instance, our entire event handling system is simply a singular function `(handle event receiver)` which we define methods on to receive events. A central "event loop" object then just calls the handle function for every event it receives on every receiver it has in its internal list.

As described in our prior paper [19] we use metaclasses to attach "shader fragments" (code that is executed on the GPU during rendering) to classes and inherit the behaviour of these fragments together, allowing composition of rendering behaviour through inheritance as well.

Overall the use of classes as an organisational structure lends itself extremely well to games, especially in the presence of multiple inheritance and mixins. We also often use this to introduce "marker classes" that contain no behaviour or data of their own, but act as type information that other parts of the system use, such as whether a class can be instantiated by the editor, resized, should be treated as solid during collision, should not be deleted when loading a prior save state, etc.

We make use of structures over standard objects only in very select cases where the impact of the increased verbosity can be contained, and the performance requirements actually indicated that we needed to optimise. Specifically this is for the hit information during collision resolution, for the node graph that powers the pathing AI, and the glyph information used for text layouting and rendering. In those cases the initial class approach produced too much overhead with repeated slot access dispatch, so we switched to a few specific structures.

Especially with regards to propagating type information CLOS can be a hindrance, as you are prevented from declaring the argument and return types. Structures give us the advantage of fixing a return type on their accessors, propagating type information more easily and thus leading to better performing code at their call sites. There are projects that allow inlining method dispatch as well, eliminating some of the type ambiguity, but we have not seriously investigated these approaches yet.

Overall the dispatch overhead of CLOS has never been a real problem for us. Even with thousands of objects that receive events through our `handle` function, hundreds of methods attached to it, and tens of events every frame, not to mention all the other parts in the game that use CLOS, we still manage to easily hit a steady 120 frames per second on a ten year old machine.

5 PERFORMANCE & GARBAGE COLLECTION

Overall we have needed to do surprisingly little actual performance analysis and optimisation work to make Kandria run well. This is definitely in large part thanks to SBCL's quite good native code compiler and type inference systems, and the prior work we've done to design critical libraries to not be completely obscene in terms of their performance characteristics.

We have done larger scale rewrites of several subsystems in Trial, which have provided us with general performance increases, though these rewrites were primarily done to increase the code clarity and usability, with the performance being a nice bonus.

As usual, far more important than constant factor improvements like those has been reducing the asymptotic behaviour. Introducing a Bounding Volume Hierarchy to speed up spatial queries and especially collision detection from a primitive linear search per object down to a logarithmic one has definitely provided the most significant performance improvement.

While asymptotic behaviour improvements will be applicable for projects in any language, with SBCL we've had the definite advantage of being able to rely on the statistical profiler to determine hot spots in the code. This is especially advantageous in Lisp as we can turn the profiler on and off at any time, to capture exact segments that we are interested in, rather than having to capture the full run of a program and then massaging the data manually.

Newly developed tools to better visualise the data gathered by the statistical profiler, such as the "flamegraph" visualiser developed by Jan Moringen have also been invaluable in gaining a better understanding of the performance characteristics of the code.

As outlined in subsection 3.1 above, there are still areas in which Lisp struggles to be truly competitive, largely in part due to its very dynamic typing, an aspect that is otherwise very advantageous for development. We've also encountered issues in eliminating superfluous consing, as many convenient styles of writing code will prevent stack allocation and other garbage elimination.

We have managed to keep our garbage production down largely with two tricks:

- Pooling and preallocation. By storing objects in a pool and manually allocating and freeing them we can avoid allocating them at runtime, leaving less work for the GC to deal with. Since the objects are long lived, they will also be quickly promoted to later generations, lessening the work to scan other generations. The obvious downside is that we lose automatic freeing and may introduce double-use cases, but for many cases the lifetimes of the objects can be predetermined and managed relatively painlessly.
- Using `load-time-value` in lieu of stack allocation. With this trick we can allocate a "local object" that will be modified at runtime rather than allocated fresh. The downside of this approach is that whichever function provides the load-time-value object cannot be called concurrently, and the programmer must be vigilant in tracking the lifetime of the load-time-value object should it escape the dynamic extent of the function.

We are also very interested in a recent proposal to add memory arenas to SBCL, which would allow us to capture all of the objects allocated within a dynamic extent and free them immediately on exit, further lessening the burden of the global GC in cases where we know the exact lifetimes.

Aside from memory arenas, we're also very interested in recent work by Hayley Patton to parallelise SBCL's garbage collector, though it is currently unknown when this feature will become mature and be merged into mainline SBCL.

In general we have not felt that we've had to do a lot of work to keep garbage production down. On a usual system the GC seems to trigger every ten seconds or so, with no noticeable stutters caused by the pause time. We have been made aware of some rare systems on which the GC pause *does* cause visible stutters, but have not been able to identify why that should happen or what to do about it. Completely eliminating all runtime garbage production does not seem like an effective way to spend our time, however.

6 DEPLOYMENT

Finally we would like to talk about our experience actually delivering Kandria to users and the process we have developed for doing so.

The first step in the deployment pipeline is the build of the actual game binary, for which we use the ASDF build system and the Deploy[3] library. With these all game code is compiled fresh, dumped into an executable suitable for the local platform, and bundled together with the depended-upon shared libraries, producing an executable ready for deployment to target machines.

However, since the build does not happen from scratch via a platform independent compiler, we have to ensure that we use an SBCL version that has been compiled on our minimal target platform version. For instance, we build the SBCL we use for deployment in a virtual machine on an old Linux version to ensure that internal symbol version dependencies in glibc can be satisfied on target machines. We have to take similar care for all required shared libraries that are shipped. Once this version has been compiled however, we can simply invoke it on any Linux version we like to use, and don't have to build the entire system inside a VM.

We can also directly build Windows version of the game on Linux by using WINE[17]. This has worked flawlessly for us, and we imagine that it would also be possible to do the inverse on Windows by using the Windows Subsystem for Linux to build Linux versions of the games. We've also investigated the possibility of using Darling to build MacOS versions. Unfortunately Darling is not able to run SBCL as of the writing of this paper.

For all generated binaries we also make use of SBCL's core compression to reduce the binary size to about 35MB per platform. The newly introduced support for zstd also brings improved compression rates and startup speeds over the older zlib. This binary size is more than acceptable to us and is completely dwarfed by the resource files for sound and graphics.

We also make use of hooks in the Deploy library to extract these resource files from their source directory and bundle them alongside the binary when building. Paths to resource files are resolved at runtime in the engine, and are thus trivial to redirect to binary-relative paths when deployed.

Finally we have created a release system that invokes the necessary SBCL subprocesses to kick off a build, prunes out unnecessary files from the generated release, then bundles it all into a ZIP for easy delivery to testers. The system can also upload the release directly to a variety of distribution platforms including Steam, Itch.io, Keygen, and generic HTTP or FTP servers. Ultimately this allows us to compile, bundle, package, and upload new builds with a single command, drastically reducing the time and complexity involved in supplying testers and users with bugfixes.

Another aspect that is advantageous to Lisp's dynamic nature is that when an unhandled condition occurs on a target system we can, in almost all cases, still use the system well enough to gather telemetry and submit an automated crash report. This has been invaluable for us to detect rare bugs, especially ones related to uncommon system configurations that we would otherwise never have been able to catch.

7 CONCLUSION

Overall the biggest hurdle we have had in our development of Kandria has nothing to do with the language itself, but rather with the general lack of manpower behind the community. We have had to spend large amounts of time implementing, testing, and documenting auxiliary systems that have often close to nothing to do with the core process of implementing a game. This is also why we have been extremely light on topics that concern the actual game implementation process in this paper.

On the other hand, it is very clear to us that interactive development is an immense boon to the development of a game. Being able to redefine game behaviour at runtime to quickly iterate on the game content and feel is invaluable. So much so that almost every engine in use will have some form of scripting language available for game logic specifically, such that designers can iterate quickly. However, having the full stack of code available, debuggable, and performing just as well as any other part of the system is undeniably far more convenient.

We also fully recognise that while Kandria is a full game project, it is by its nature rather limited in the required processing capabilities. A lot more work is needed to support more complex games, which we intend on focusing on in the near future. However, we do not at present see any deal-breakers that would make it unfeasible to create such games using Common Lisp and the base ecosystem we have helped establish so far.

If anything the recent advances in SBCL's capabilities show us a very promising future and we are excited to make use of them to further improve the situation, and with much of the "grunt work" now done we should be able to focus our efforts onto problems more directly associated with game development instead.

8 FURTHER WORK

Currently a sizable amount of work in Kandria has not been backported into Trial for more general purpose use. We would like to extract a few of the systems and generalise them to make them available for other users.

We are also working on implementing several new subsystems in Trial to allow creating 3D games, as well. A skeletal animation system has been completed, and we're currently working on a physics subsystem. Also needed will be several spatial query structures to speed up collision testing, along with a more unified rendering subsystem to support Physics Based Rendering pipelines.

Finally we are also exploring the possibility of porting the engine to work on closed platforms such as the Nintendo Switch. This presents several challenges that we unfortunately cannot elaborate on here due to non-disclosure agreements.

9 ACKNOWLEDGEMENTS

We would like to thank the various contributors to all of the projects that have made it possible to make Kandria in the first place, and we would like to thank *you* for being beautiful and nice.

Kandria was funded in part by the Pro Helvetia Interactive Media Grant and the KPT Poland Prize Digital Dragons Accelerator.

REFERENCES

[1] A library to handle gamepad input devices, . URL https://shirakumo.org/projects/cl-gamepad.
[2] A library for audio processing and output, . URL https://shirakumo.org/projects/cl-mixed.
[3] A library to ease the deployment of common lisp binaries. URL https://github.com/shinmera/deploy.
[4] A library for file selection dialogs. URL https://github.com/shinmera/file-select.
[5] A library to search and query system fonts. URL https://github.com/shinmera/font-discovery.

Figure 2: A flamegraph of the game's main thread. Due to the high nesting not much is visible at this scale. We've identified two specific blocks belonging to collision resolution and rendering. The graph of a release build would look slightly different, as several things are optimised away for a release build.

[6] Shirakumo github. URL https://github.com/shirakumo.
[7] The glfw opengl portability library. URL https://glfw.org.
[8] The godot game engine. URL https://godotengine.org.
[9] A high-level audio processing server. URL https://shirakumo.org/projects/harmony.
[10] Kandria, an open world action rpg. URL https://shirakumo.org/projects/kandria.
[11] A c library for audio processing pipelines. URL https://shirakumo.org/projects/libmixed.
[12] A library for native message box display. URL https://github.com/shinmera/messagebox.
[13] The steel bank common lisp implementation. URL https://sbcl.org.
[14] A library to query the system for locale information. URL https://github.com/shinmera/system-locale.
[15] The trial game engine. URL https://shirakumo.org/projects/trial.
[16] The unity game engine. URL https://unity.com.
[17] The wine project. URL https://winehq.org.
[18] Jeff Craighead, Jennifer Burke, and Robin Murphy. Using the unity game engine to develop sarge: a aase study. In *Proceedings of the 2008 Simulation Workshop at the International Conference on Intelligent Robots and Systems (IROS 2008)*, volume 4552, 2008.
[19] Nicolas Hafner. Object oriented shader composition using clos. In *ELS*, pages 80–83. Shirakumo.org, 2018.
[20] Nicolas Hafner. Using a highly dynamic language for development. 2021. URL https://github.com/Shinmera/talks/blob/master/gic2021-highly-dynamic/paper.pdf.
[21] Henri Mäkelä. Development of a 3d mahjong video game in godot engine. 2021.
[22] Jukka K Nurminen. Rft design system-experiences in the development and deployment of a lisp application. In *Proceedings of the First European Conference on the Practical Application of Lisp*, 1990.
[23] Hayley Patton. Parallel garbage collection for SBCL. 2023.
[24] Robert Strandh. Fast generic dispatch for common lisp. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, pages 89–96, 2014.

Parallel garbage collection for SBCL

Hayley Patton
hayley@applied-langua.ge

ABSTRACT

We describe a parallel garbage collector which we are implementing for Steel Bank Common Lisp. The collector reclaims memory and allows for bump allocation without the collector needing to move objects, using a mark-region heap based on Immix [8]. The heap is comprised of pages, and pages are comprised of lines. We exploit the design of Immix in two ways: (i) generations are implemented without the collector moving objects or recording the generation in each object, by associating generations with lines; and (ii) conservative root finding is implemented by updating an object map only on demand, based on recording runs of contiguously allocated objects. The parallel garbage collector using one core usually is slower than the copying collector of SBCL, outperforms copying with two cores, and continues to scale with more cores.

CCS CONCEPTS

• **Software and its engineering → Garbage collection**.

ACM Reference Format:
Hayley Patton. 2023. Parallel garbage collection for SBCL. In *Proceedings of the 16th European Lisp Symposium (ELS'23).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.7816398

1 INTRODUCTION

Steel Bank Common Lisp (SBCL) uses a generational mostly-copying collector named *gencgc*. The collector also must conservatively scan the registers and stacks of the mutator when using the x86 and x86-64 instruction sets (similar to Barlett's mostly-copying collector [4]). The heap is first split into the *static*, *dynamic* and *immobile* spaces (the last only when using the x86-64 instruction set), with only the dynamic and immobile spaces ever garbage collected. Almost all objects are allocated into the dynamic space. The dynamic space is split into 32 kibibyte pages; a page may either be used for storing many small objects, or for storing part of a large object. Small objects are stored contiguously in pages. gencgc reclaims memory by copying small objects into empty pages, which causes the live objects to occupy fewer pages, and by marking large objects and pages encountered as live without copying.

There are two main inefficiencies with this approach: copying objects may require more time than marking, and the collector only uses a single core. The latter can lead to Common Lisp programs exhibiting poor scalability, to no fault of the programmer. For example, one parallel fuzz tester is *embarrassingly parallel* as tasks do not share any resources; in a tight heap, the fuzz tester runs 55 seconds of processor time over 12 cores in 5.0 seconds of real time, but the collector runs for 10.5 seconds of real time, causing the program to run for 15.5 seconds of real time, with only a 4.5× speedup.

While it is possible to improve the throughput of tracing garbage collection by increasing the size of the heap [1], it can be undesirable to use a heap much larger than the live objects, and it is not sustainable to need to use more memory in order to maintain scalability. In order to hold the impact of a serial garbage collector constant, the heap size may need to grow quadratically with regards to the number of cores used. Suppose a program processes n tasks in parallel, each task allocating r words per second and keeping m words live at any time. The system performs a garbage collection after allocating t words. A full tracing garbage collection[1] thus requires tracing nm words, and a garbage collection occurs nrt^{-1} times per second. The overall cost of tracing (which often dominates) is thus proportional to n^2mrt^{-1}. Maintaining a constant cost of collection while increasing n requires increasing t by a factor of n^2, i.e. allowing the collector to use space proportional to the square of the number of tasks to run. Assuming perfect scalability of the collector, a parallel collector can instead use n cores for tracing and a heap only n times larger, to achieve the same effect.

Increasing the heap or nursery size also decreases locality of reference, which can decrease the performance of functional programs overall [12]. However, Common Lisp programmers vary in their use of functional or in-place algorithms, and thus the significance of locality of reference may not be as large as with more functional languages.

We can also use parallelism to improve the latency of garbage collection. While our collector still stops the world in order to perform a collection, parallel garbage collectors take less real time to collect than non-parallel collectors, thus decreasing pause times.

A reader unfamiliar with garbage collection may want to consult the Memory Management Reference[2] for definitions of unfamiliar terms in this paper.

2 PRIOR WORK

Luís Oliveira parallelised gencgc [18], using an approach like that used by Marlow et al for the Glasgow Haskell Compiler [16]. In both collectors, worker threads claim pages to scan for references to objects which need to be copied. Oliveira did not achieve a large speedup by parallelising garbage collection, but he was only able to test on a dual-core machine. We suspected greater speedups could be achieved with more cores, as processors with more cores are much more accessible than at the time of development of either collector; the Steam hardware and software survey results[3] indicate that more than half of participants now have 6 or 8 cores.

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

https://doi.org/10.5281/zenodo.7816398

[1] We believe the same relation would hold for a generational collector, but the cost model would be more complicated.
[2] https://www.memorymanagement.org/
[3] The survey results are accessible at https://store.steampowered.com/hwsurvey; the participants are users of the Steam game distribution service, so it is possible that the results are biased towards more powerful computers.

We attempted to replicate Oliveira's collector in 2022, but the collector did not work reliably, nor did it achieve any substantial speedup when it did work. Collector threads contended heavily on a lock while acquiring new pages to copy objects into; Marlow et al had experienced similar behaviour, and reduced the frequency of locking by having collector threads acquire more pages at once. We instead opted to implement a non-moving collector, eliminating altogether the need to support fast allocation by many threads.

3 MARK-REGION COLLECTION

3.1 Heap structure

The heap structure is based on the two-layer structure of Immix [8], consisting of 32 kibibyte *pages* consisting of 128 byte (or 16 word) *lines*. The page size may be changed, but the scavenging algorithm constrains the line size, as described in Section 4. As with gencgc, pages may either store small objects or part of a large object, but objects larger than three quarters of a page may reside on a single "large" page, to prevent the allocator from trying to search for large holes in small pages.

The collector reclaims memory at the granularity of lines; a line is considered either entirely live and not reusable, or entirely dead and reusable. The collector also marks all lines that an object occupies when tracing the object, ensuring live lines are not reclaimed later. As objects allocated together tend to die together [24], the garbage collector is still effective at reclaiming memory despite this inaccuracy, and the heap produces little internal fragmentation. The mutator allocates objects contiguously into unused lines, providing for locality of reference between objects allocated contemporaneously.

The collector relies on four additional tables stored outside of the heap: object map and mark bitmaps, and line metadata and card table bytemaps, each $\frac{1}{128}$ of the heap size on 64-bit platforms, leading to approximately 3.1% space overhead. As objects are aligned to two words (or 16 bytes), it is only necessary to store a bit for locations spaced 16 bytes apart. A byte consists of eight bits, so the locations for 128 bytes of heap fit in one byte of bitmap. The bytemaps are sized to have the same scale (of metadata bytes to heap bytes) as the bitmap, to simplify traversals of the heap, as described in Section 4.

3.2 Tracing

The collector has four stages. The collector first marks the *roots*, such as local and global variables. Then the collector *scavenges* objects in older generations to find references to new objects; such references require the collector to retain those new objects. The collector then *traces* the heap, marking every object which is transitively reachable from a marked object. The collector finally *sweeps* the heap, resetting the internal state of the collector and allowing the memory used by dead objects to be reused. The scavenging and sweeping passes can be parallelised by giving each collector thread its own section of the heap to process, but it is not as trivial to parallelise the tracing pass.

Parallel tracing is performed using the design by Ossia et al [19]. Grey objects (which have already been marked, but need to be traced by the collector) are stored in a set of *grey packets*, with each packet storing a sequence of references to grey objects. Each worker thread has an *input packet* of objects that the thread is going to trace, and an *output packet* of objects that were discovered by the thread during tracing. As collector threads read and write packets entirely sequentially, threads may use software prefetching to avoid waiting for objects to be loaded from main memory. We store packets in a stack, using a lock to protect the stack; Ossia et al used a lock-free list, but we did not observe a significant decrease in performance by using a lock. Locking has been also used in other well-used collectors; the Garbage-First collector for Java[4] used a lock, suggesting that locking the stack does not impact performance in Java.

The collector allocates packets outside the heap, directly acquiring memory using the `mmap` system call.[5] In order to avoid serialisation induced by the kernel updating the memory map, we implemented an arena allocator, which allocates chunks of packets sized to increasing powers in two. At the end of the collection, the entire contents of the arena are considered unused. In order to avoid allocations in subsequent collections, we retain chunks which have been used recently, but we `munmap` chunks which have not been used recently, so that the collector does not needlessly retain chunks which are seldom used. To avoid having to protect a global free list from concurrent access, packets are reused in thread-local lists. We confirmed Ossia et al's observation that packets use little memory, at most using 3.7 MB while running the boehm-gc benchmark in a 4 GB heap (as described in Section 6).

4 NON-MOVING GENERATIONAL COLLECTION

Many approaches to *generational* garbage collection rely on representing generations using ranges of addresses in the heap. For example, gencgc associates a generation with each page in the heap. We avoid moving objects where possible, so attempting to partition the heap using addresses prevents the collector from reclaiming much memory; if a page were promoted to an older generation, any free space on the page could not be used for allocating new objects (which necessarily are in the youngest generation), and could not be reused until the page is entirely unused.

Demers et al suggested that it is not necessary to represent generations this way, however, and that associating a generation with each object suffices [9]. We are left with the problem of where to store the generations associated with each object. All types of objects that are not cons cells have a *header* word in SBCL, which can store a generation number[6], but there is no free space in a cons cell to store a generation number. We may instead store generation numbers in a table external to the heap, as we do with the mark bitmap. Using a table separate from the heap also reduces the amount of memory which must be scanned, as generation numbers are stored compactly, instead of being mixed with other data in the heap. Scanning the table also eliminates the need to walk most objects in the heap.

[4] See https://github.com/openjdk/jdk/blob/jdk-21+8/src/hotspot/share/gc/g1/g1ConcurrentMark.cpp#L175-L211

[5] As the collector runs in a signal handler, it is necessary to avoid using libc functions, and SBCL generates a "raw" system call when possible.

[6] Indeed this is how the *immobile space* in SBCL works, as it does not need to store cons cells.

Making such a table space-efficient would be difficult, however, as a generation number would be required for every location that an object could reside at. SBCL uses eight generations by default, requiring three bits of storage per location. If a full byte were reserved for each generation number, the generation table would require a $\frac{1}{16}$ space overhead. Immix peculiarly used a table of bytes (which we will call a *bytemap*) for storing marks for lines, rather than a table of bits (a bitmap), in order to avoid using atomic operations to mark lines as live. We may instead use the bytes in the bytemap to store generations and line marks; as we only collect whole lines, parts of a line cannot be reused, so all objects on a line must share the same generation. To implement conservative root finding, the line bytemap is also used to record which lines have been freshly allocated.

Demers et al also raised the issue of *card pollution*. Generational collectors often use a card map [23] in order to find old objects which have been updated, and may now contain references to new objects. Cards are segments of the heap, similar to pages, although cards are usually smaller than pages. The compiler inserts *write barriers* into mutator code, which cause the mutator to mark a card as *dirty* when the mutator stores a reference in that card. If cards are larger than lines, then newer and older objects may exist in the same card. It is not possible for the collector to discern writes to newer and older objects on the same card, so writes to newer objects in a card may cause the collector to needlessly scavenge older objects in the same card. We avoid this imprecision by making cards the same size as lines, so that cards can only store objects in the same generation.

Another benefit to making cards the same size as lines is that various operations on all the bitmaps can be performed more efficiently using *single instruction-multiple data* instructions, which many instruction sets have. It is convenient to use SIMD comparison instructions on the bytemaps, as *true* is represented as all bits set in a byte, and *false* as all bits cleared[7]. Such results can be treated as sets of object locations where tests on the line and card bytemaps succeed, and bitwise-and may be used to perform logical conjunction of tests. For example, code computing $map \wedge (generation > g) \wedge (card = dirty)$ for every byte of the object map, line bytemap and card bytemap will correctly compute a bitmap with bits set where objects which are dirty and are older than generation g reside.

5 CONSERVATIVE ROOT FINDING

The SBCL compiler does not record which registers and stack locations are used for tagged and untagged values, when targeting the x86 and x86-64 instruction sets, so a collector must be *conservative* when scanning the stack and registers to find root references into the heap; the collector must be able to identify which values identify objects in the heap, and which do not. While SBCL does not use *interior pointers*, which keep an object live without pointing to the start of the object, we implemented interior pointers to see how complicated implementing interior pointers would be.

[7]For example, the SSE2 instruction `pcmpeqb` effectively computes `(map 'vector (lambda (a b) (if (= a b) #xFF #x00)) A B)` for two vectors A and B. In practise we rely on the auto-vectorisation of C compilers for portability; GCC and Clang successfully generate vectorised code for x86-64 (with SSE2 and AVX2), ARM (with SVE) and RISC-V (with the Vector extension).

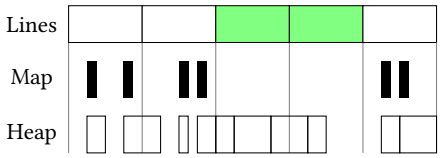


Figure 1: Most objects are not contiguous in memory, but objects allocated after the last collection are contiguous in *fresh* lines (light green).

If objects are stored sequentially in pages, the collector can scan a page of memory sequentially to find the object which a pointer references [5]; SBCL currently uses this approach. We cannot use this approach, however, because objects are not stored sequentially in pages, and so attempting to scan sequentially will likely read garbage data.

Another approach is to record positions of objects into an *object map* bitmap when allocating, as suggested by Shahriyar et al [21]. Without interior pointers, it suffices to check the bit in the object map corresponding to a pointer, to determine if the pointer points to an object in the heap. With interior pointers, the bitmap must be scanned backwards to find a set bit, and then the size of the corresponding object found may additionally be checked, in order to confirm that the pointer does indeed point inside the object. The scan is rather fast when employing *bit parallelism* and checking entire words of bits at a time. However, setting bits in the bitmap slows down the mutator, and it is also difficult to update the bitmap without dedicated instructions; Shahriyar et al use the x86 `bts` instruction to set a bit in a bitmap, but many other instruction sets such as ARM and RISC-V do not have a similar instruction.

We instead use a hybrid of the two approaches, where an object map is used, but it is not updated by the mutator directly. While Immix does not store objects contiguously on pages, objects are allocated contiguously in smaller runs. An example of this behaviour is depicted in Figure 1. Our allocator marks lines it uses for allocation as *fresh*, and when the collector encounters a conservative reference into a fresh line, the collector finds the start and end of the enclosing run, and computes the object map for that run of objects. As most objects die young [22] and few objects are referenced from the stack, the collector does not have to compute much of the object map. It is also thus not necessary to produce a fast instruction sequence to update the object map, as the object map is seldom written into, and the object map is not written into by the mutator. We have observed that, at most, the parallel fuzz tester requires the collector to compute the object map for about 600 kilobytes of heap on average.

This approach might increase pause times, as object map computation is done all at once, rather than spread out across the execution of the application. We have not observed any effect on pause time in practise, but a concurrent collector may opt to process conservative roots concurrently with the mutator.

6 PERFORMANCE

We improve performance by using multiple cores to perform garbage collection work, but it is also important that the performance with a single core is not made much worse by the use of a more scalable

parallel algorithm. Using many more cores for a small performance gain would lead to very poor efficiency, which is usually undesirable. Focusing on only using more cores to achieve performance is likely futile in any case, as the scalability of tracing is often limited by the shape of the heap being collected [3]. Thus we report results for a variety of thread counts: we test the baseline performance of the collector with only one thread, realistic configurations with 2 and 4 threads, and the limit with 12 collector threads. We test with a Ryzen 1950X processor, with 12 cores provided to the virtual machine used. Each configuration of each benchmark was run 30 times, and we report the average of all the runs.

We test with commit `4e71abc577180c0276cb14b31a69dc0d2eb84694` of our fork of SBCL accessible at https://github.com/no-defun-allowed/swcl, with the immobile space disabled for both collectors, as we currently cannot use it with the parallel collector. Enabling the immobile space makes the mutator much faster running Regrind; we expect a similar speedup would be achieved when the immobile space works with the parallel collector.

We use our own benchmark suite as we could not find any other suite which was appropriate. In particular, the *cl-bench* suite is popular, but does not generate much of a load for the garbage collector. We could decrease the size of the heap in order to cause more frequent garbage collections, but more of the heap would still fit in the large caches of modern processors. There are also no latency-sensitive benchmarks in cl-bench, nor any benchmarks which use multiple threads. The benchmark suite itself is accessible at https://github.com/no-defun-allowed/gc-benchmarks/tree/v1.

6.1 Throughput benchmarks

Figure 2 contains the results of the throughput benchmarks. The benchmarks are:

- **boehm-gc**: A benchmark from *cl-bench*[8] which allocates many binary trees of various sizes, with the k parameter (which affects the size of trees allocated) increased to 24 from the original default of 18. Binary trees of each stage of the benchmark die simultaneously, and fragmentation is negligible. Serial mark-region collection lags copying collection somewhat, and any parallel collection out-performs gencgc. The benchmark also runs in a smaller heap when using the mark-region collector.
  Despite the formerly discussed efforts to reduce mutator overhead, some overhead still exists when using the mark-region collector. We suspect that the smaller cards cause more cache misses, slowing down the mutator.
- **regrind-interpret**: The parallel fuzz tester for the *one-more-re-nightmare* regular expression compiler[9], running using 12 worker threads. In order to make the benchmark more reproducible, it was modified to generate the same sequence of regular expressions to test, and to use static load balancing. It allocates lots of very short-lived objects. gencgc is comparable in performance to parallel mark-region collection with two threads, owing to the lower mutator time. The mark-region does not perform well with a heap smaller than 5GB, and all but the 12-thread configuration are outperformed by gencgc with a heap larger than 5GB. Very few objects survive a nursery collection, causing the scavenging and sweeping passes of the mark-region collector to take most of collection time.
  This benchmark exhibits a more asymptotic mutator performance than *boehm-gc*, with the mutator performance varying with the heap size when using mark-region collection. The performance appears to vary due to the mutator needing to acquire more pages in tight heaps (as in Figure 3). A partly used page fits fewer new objects than an entirely free page, so more partly filled pages must be acquired to allocate; boehm-gc does not produce any partly used pages. While the collector can reuse partly used pages without moving, it incurs some time overhead in doing so. (We thus have lost some scalability to the allocator, but it is not as bad as if we had used a parallel copying collector.)
- **regrind-compile**: The same fuzz tester, with the same modifications made as with regrind-interpret, but using the compiler of SBCL rather than the interpreter. (Using the compiler is much slower than the interpreter, so the fuzz tester is configured to perform much less work.) It allocates longer-lived objects, requiring much more time to trace. The variation of mutator time is greater than in regrind-interpret.

6.2 Latency benchmarks

One benchmark we test, named **ring-buffer** is pathological for copying collectors [13], demonstrating a substantial difference in work performed by non-moving and copying collector algorithms. The benchmark involves a ring buffer of small unboxed arrays, each array containing a one kilobyte "message", with each new message being a new allocation from the heap. Each live message must be copied by a copying collector, although no pointers to trace are discovered in doing so.

As depicted in Figure 4, the mark-region collector performs much better by not having to copy messages, and also allows running in smaller heaps. When running in a 2GB heap and with gencgc, the program spends 50% of processor cycles in the C `memcpy` function. While the benchmark is derived from a real program which had this pathological behaviour, we do not think it is a good model of most programs. Almost all objects allocated in the benchmark have no pointers, are somewhat large, and survive several nursery collections.

Kandria, a commercial game written in Common Lisp [14], is used in a more complex benchmark. The game uses a mixture of object-oriented code and numerical code with unboxed arrays. The game is configured identically to the retail version, using a 4GB heap. We played the same part of the game for a few minutes with each collector configuration[10], and record the distribution of how long it takes to produce each frame (*frame times*), and the distribution of pause times in the **kandria** benchmark. As the game

[8] https://gitlab.common-lisp.net/ansi-test/cl-bench/-/blob/master/files/boehm-gc.lisp
[9] https://github.com/telekons/one-more-re-nightmare/blob/master/Tests/regrind.lisp

[10] We would prefer to be able to replay a *capture* of inputs to the game, in order to have the game run more deterministically, but we were not able to get the capture to be replayed reliably.

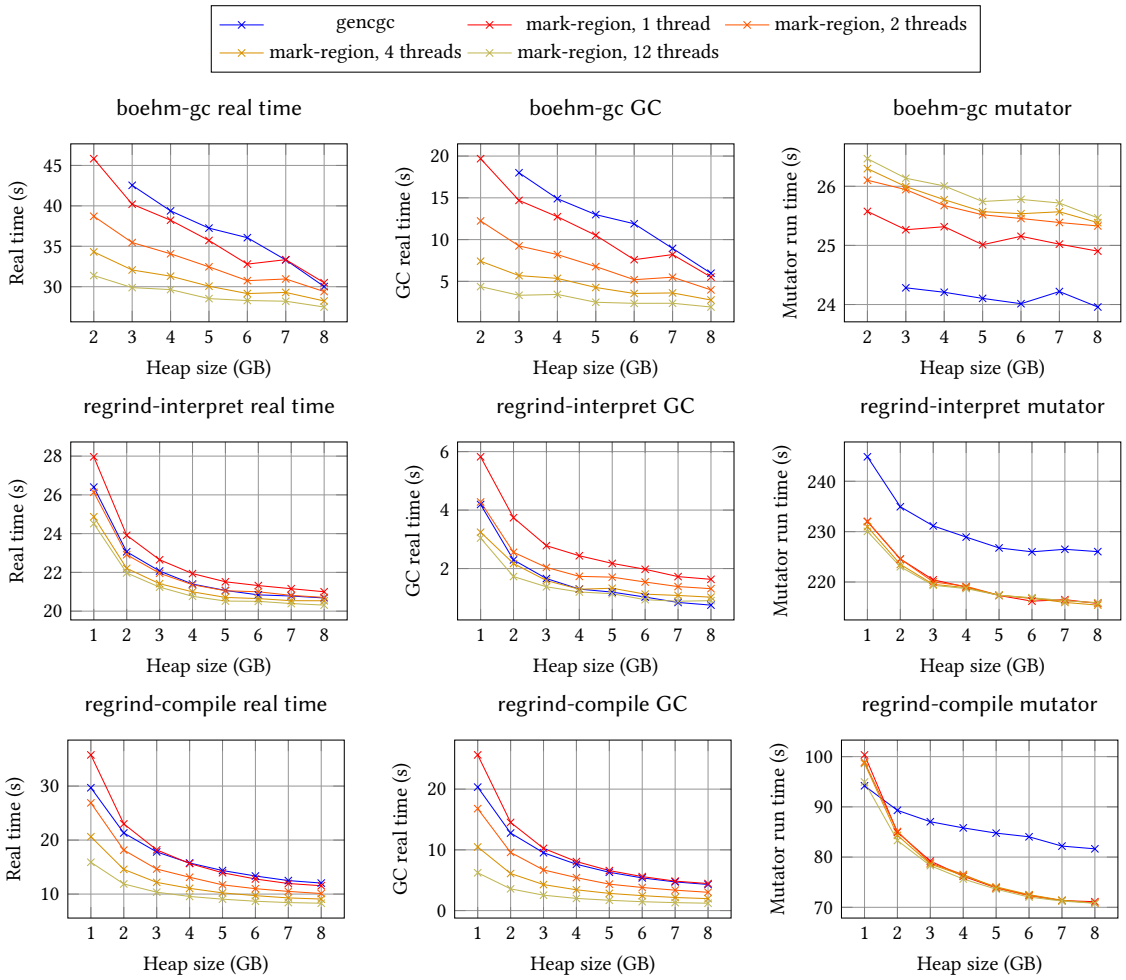


Figure 2: Results of the throughput-oriented benchmarks.

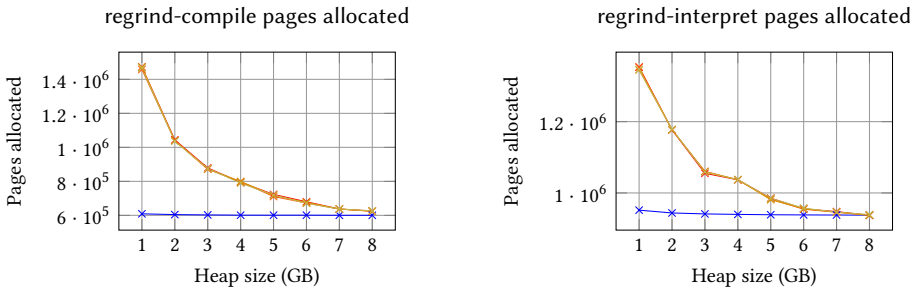


Figure 3: The number of pages acquired in order to allocate small objects.

requires low-latency graphical input and output, Kandria was run on the author's desktop computer, with a Ryzen 5900X processor and RX 580 graphics card.

As depicted in Figure 5, parallel garbage collection slightly reduces the size of the tail of pause times. However, the parallel collector does not improve pause times substantially. We also compared frame times to some time limits (in Table 1): the 16ms time limit tests if the game can run smoothly at 60 frames per second, and shorter time limits approximate the same target while using a

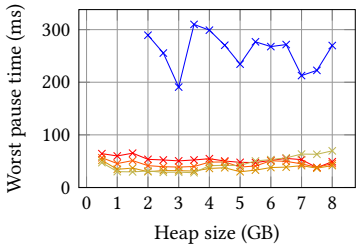


Figure 4: Worst pause times running ring-buffer.

Table 1: The percentages of frame times which exceed various time limits, for varying collectors and thread counts. (Mark-region is abbreviated to MR.)

Limit	gencgc 1 thread	MR 1 thread	MR 2 threads	MR 4 threads
16ms	0.22%	0.28%	0.27%	0.26%
12ms	0.28%	0.38%	0.35%	0.34%
8ms	0.43%	0.52%	0.50%	0.51%
6ms	2.15%	2.30%	2.21%	2.23%

slower computer. The mark-region collector violates the time limits more often than gencgc, regardless of the number of threads used.

Very few objects survive garbage collection in Kandria, with less than a megabyte surviving out of a 200 megabyte nursery. The scavenging and sweeping passes dominate collection time, due to doing work proportional to the number of used pages. We observed at one point that there were older objects occupying 160MB of memory spread across lines occupying 310MB of memory, in turn spread across pages occupying 660MB of memory. This fragmentation causes the collector to scan much more metadata than is strictly necessary. As a result, scavenging took 2.9 milliseconds on average, tracing took 1.9 milliseconds, and sweeping took 1.6 milliseconds. Kandria thus appears to represent a pathological case for non-copying collectors.

7 CONCLUSIONS AND FUTURE WORK

When using a single core, our mark-region garbage collector is only somewhat slower than the copying collector of SBCL, despite our collector never moving any objects. Using additional cores to collect in parallel allows our collector to significantly outperform the copying collector, and non-moving collection appears to be simpler to correctly parallelise than copying collection.

The collector is not ready to be used in production yet, lacking support for the immobile space of SBCL, and lacking any kind of compaction. We are also considering extending the collector to operate *concurrently* with the mutator, which is simpler without needing to copy objects.

7.1 Immobile space

SBCL has an additional *immobile space* which resides in the lowest 2^{32} bytes of the address space, and does not move. The immobile space stores layouts of "instance" objects to reduce the size of the headers of instance objects, and stores symbols to reduce the size of code referencing symbols. The immobile space is managed by a different marking algorithm, and by the TLSF allocator [17] rather than a bump allocator. We haven't succeeded in getting the immobile space collector to work with the mark-region collector yet, but it should be used in a garbage collector used in production.

As the parallel collector used for most of the heap (in *dynamic space*) does not move, it is tempting to simplify the heap and use the same collector for immobile space. But allocations into immobile space are infrequent, and objects in immobile space can never be compacted (except when saving a core file), so it is worthwhile to proactively avoid fragmentation by using the more complex TLSF allocator.

7.2 Compaction

Heap fragmentation, due to our collector not moving objects, leads to more pages being used than necessary. While the SBCL process can effectively reuse holes in pages, the space is unusable by other processes on the same computer. The lack of compaction also affects the size of *core files*; for example, the `sbcl.core` core file using the copying collector is 36 megabytes large, but the file is 248 megabytes large using the mark-region collector, as the mark-region collector never compacted the heap during bootstrapping. Compaction can also coalesce free space into full pages, reducing the number of pages that the mutator must acquire. Compaction may help to regain some locality of reference, if objects are compacted into fewer pages. Compaction can also reduce the amount of metadata that scavenging and sweeping need to scan, which may be particularly useful for Kandria.

We have begun to implement an algorithm for *incremental* compaction [6] which only moves part of the heap at the time. We select pages with few lines used starting from the end of the heap, and copy their contents into holes in the start of the heap. Collector threads record references to the selected pages while tracing, so that those references can be fixed up after copying has been performed. Unlike the mixture of marking and copying that Immix performs in one pass, having separate passes allows marking to be performed concurrently, while compacting is done in a (hopefully shorter) stop-the-world pause.

The algorithm may not perform well with many generations, however, as we cannot identify all references to older objects when performing a collection of a younger generation, and so we cannot move older objects correctly. It may be helpful to analyse if the many generations used by SBCL are beneficial; many generational garbage collectors with good performance only use a young generation and an old generation, and all objects could be moved when collecting the old generation with just two generations.

7.3 Concurrent tracing

The collector could be made concurrent by following the Ossia et al design. The Ossia et al collector did not support generations, but used the card map to detect modifications of any object while the collector is tracing. In order to support generations, another card table storing only the locations of old-to-new references would need to be maintained by the collector. Both card tables would need

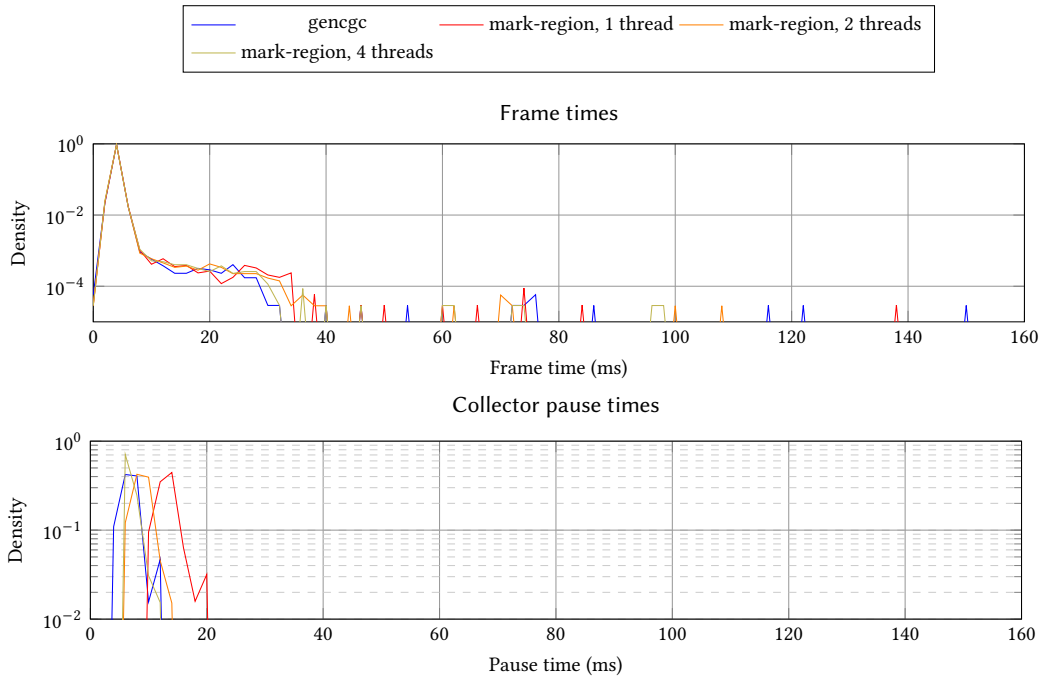


Figure 5: Frame times and collector pause times running Kandria.

to be consulted to find all old-to-new references for collection of younger generations.

It is possible to make compaction concurrent, typically by using a *read barrier* [2], which allows for ensuring the mutator only accesses pointers to copied objects, but using a read barrier induces more time overhead on the mutator. Recently Zhao et al have suggested that the latency which a user of the application experiences is better served by improving throughput, rather than focusing on further minimising pause times [25]. As we intend to compact infrequently, application latency may not be greatly affected by compaction pauses.

7.4 Other ways to collect

There are other approaches to make garbage collection more performant on multi-core computers, which we believe should be reconsidered. For brevity we will only discuss *reference counting* and *thread-local garbage collection* in some depth.

While reference counting has been seen as inferior in performance to tracing, it has been optimised with *coalescing* [15] to greatly reduce the number of updates to reference counts, combined with the Immix heap layout to provide better locality of reference [20], and recently the LXR collector [25] has been shown to outperform other garbage collectors for Java in both throughput and latency. Updating reference counts in a coalescing reference counting collector can be embarrassingly parallel, unlike tracing. Reference counting cannot collect cycles however, so infrequent tracing to collect cycles is necessary; but if tracing is infrequent, it will not harm scalability too much. It has also been observed that old objects are less often modified than young objects in Java [7]; if a similar observation holds for Common Lisp programs, using coalesced reference counting to reclaim old objects may work well, with fewer updates to reference counts contributing to a stop-the-world pause.

Another approach to improving the scalability of tracing is to use *thread-local* garbage collection, for which designs for immutable objects in ML [10] and mutable objects in Java [11] exist. Each thread has its own private nursery, which may be collected independently and without synchronisation, allowing for high scalability. Thread-local collections also may improve latency, as *global* collections which require all threads to be stopped are less frequent. Locality of reference may also be improved, as thread-local nurseries can be small; and cache ping-pong effects are minimised, as threads never need to access cache lines for the nurseries of other threads.

The latter design, which does not require objects to be moved, could benefit from a mark-region heap as described in this paper; in particular, the mutator can still utilise bump allocation, even though global objects cannot be moved out of partly used pages without performing a global collection. A similar kind of sweeping can be used for sweeping during a local collection, by copying the mark bitmap to the object bitmap only for local objects, thus preserving global objects which are not collected.

ACKNOWLEDGMENTS

We would like to thank Stas Boukarev and Douglas Katzman for helping us get up to speed with the interactions between the garbage collector and the rest of SBCL. Steve Blackburn and Kunal Sareen had discussed Immix with us, leading to the approaches to implementing generations and conservative root finding described in

this paper. Nicolas Hafner helped us get auto-vectorisation to work, and helped us navigate the Kandria source code. Paul Khuong and Larry Masinter helped us design the internal memory manager used by the garbage collector. Cliff Click helped us make sense of our benchmark results with parallel programs. Grindwork Corporation provided us with access to a virtual machine for benchmarking. Jan Moringen, Kunal Sareen, Elijah Stone, and Robert Strandh provided comments on early versions of this paper.

REFERENCES

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25(4):275–279, June 1987. ISSN 0020-0190. doi: 10.1016/0020-0190(87)90175-X.

[2] Henry G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, April 1978. ISSN 0001-0782. doi: 10.1145/359460.359470.

[3] Katherine Barabash and Erez Petrank. Tracing garbage collection on highly parallel platforms. In *Proceedings of the 2010 International Symposium on Memory Management*, ISMM '10, page 1–10, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450300544. doi: 10.1145/1806651.1806653.

[4] Joel F. Barlett. Mostly-copying garbage collection picks up generations and C++. Technical report, Digital Western Research Laboratory, 1989.

[5] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. *SIGPLAN Lisp Pointers*, 1(6):3–12, April 1988. ISSN 1045-3563. doi: 10.1145/1317224.1317225.

[6] Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Kean Kuiper, and Victor Leikehman. An algorithm for parallel incremental compaction. In *Proceedings of the 3rd International Symposium on Memory Management*, ISMM '02, page 100–105, New York, NY, USA, 2002. Association for Computing Machinery. ISBN 1581135394. doi: 10.1145/512429.512442.

[7] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, page 344–358, New York, NY, USA, 2003. Association for Computing Machinery. ISBN 1581137125. doi: 10.1145/949343.949336.

[8] Stephen M. Blackburn and Kathryn S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, page 22–32, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781595938602. doi: 10.1145/1375581.1375586.

[9] Alan Demers, Mark Weiser, Barry Hayes, Hans Boehm, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, page 261–269, New York, NY, USA, 1989. Association for Computing Machinery. ISBN 0897913434. doi: 10.1145/96709.96735.

[10] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *POPL 1993: 20th symposium Principles of Programming Languages*, pages 113–123. ACM, 1993. doi: 10.1145/158511.158611.

[11] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for Java. *SIGPLAN Not.*, 38(2 supplement): 76–87, Jun 2002. ISSN 0362-1340. doi: 10.1145/773039.512439.

[12] Henrique Ferreiro, Laura Castro, Vladimir Janjic, and Kevin Hammond. Kindergarten cop: Dynamic nursery resizing for GHC. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, page 56–66, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342414. doi: 10.1145/2892208.2892223.

[13] Jim Fisher. Low latency, large working set, and GHC's garbage collector: pick two of three, 2016.

[14] Nicolas "Shinmera" Hafner. Experience report: Kandria - a game in Common Lisp. *The 16th European Lisp Symposium (ELS'23)*, 2023.

[15] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, jan 2006. ISSN 0164-0925. doi: 10.1145/1111596.1111597.

[16] Simon Marlow, Tim Harris, Roshan P. James, and Simon Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *Proceedings of the 7th International Symposium on Memory Management*, ISMM '08, page 11–20, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581347. doi: 10.1145/1375634.1375637.

[17] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, ECRTS '04, page 79–86, USA, 2004. IEEE Computer Society. ISBN 0769521762.

[18] Luís Oliveira. SBCL garbage collection. Master's thesis, Universidade de Coimbra, 2009.

[19] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. A parallel, incremental and concurrent GC for servers. *SIGPLAN Not.*, 37(5):129–140, May 2002. ISSN 0362-1340. doi: 10.1145/543552.512546.

[20] Rifat Shahriyar, Stephen Michael Blackburn, Xi Yang, and Kathryn S. McKinley. Taking off the gloves with reference counting Immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, page 93–110, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450323741. doi: 10.1145/2509136.2509527.

[21] Rifat Shahriyar, Stephen M. Blackburn, and Kathryn S. McKinley. Fast conservative garbage collection. *SIGPLAN Not.*, 49(10):121–139, October 2014. ISSN 0362-1340. doi: 10.1145/2714064.2660198.

[22] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGPLAN Not.*, 19(5):157–167, April 1984. ISSN 0362-1340. doi: 10.1145/390011.808261.

[23] P. R. Wilson and T. G. Moher. A "card-marking" scheme for controlling intergenerational references in generation-based garbage collection on stock hardware. *SIGPLAN Not.*, 24(5):87–92, May 1989. ISSN 0362-1340. doi: 10.1145/66068.66077.

[24] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, page 1–116, Berlin, Heidelberg, 1995. Springer-Verlag. ISBN 3540603689.

[25] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. Low-latency, high-throughput garbage collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, page 76–91, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392655. doi: 10.1145/3519939.3523440.

Design of an Efficient Lisp Bytecode Machine and Compiler

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

Charles Zhang
karlos@berkeley.edu

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

ABSTRACT

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

CCS CONCEPTS

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

KEYWORDS

virtual machine, compiler, bytecode

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

1 INTRODUCTION

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

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

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

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

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

2 MOTIVATION

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

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

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

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

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

3 PREVIOUS WORK

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

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

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

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

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

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

4 DESIGN

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

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

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

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

4.1 Interoperability

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

4.2 Instruction set design

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

4.3 Flat closures

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

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

Nonetheless, the flat closure approach has many desirable features:

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

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

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

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

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

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

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

Figure 1: Linked closure strategy (used in CLISP)

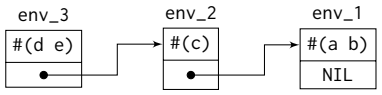
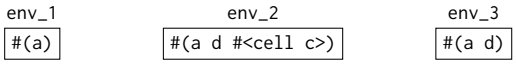


Figure 2: Flat closure strategy



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

4.4 `long` instruction prefix

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

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

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

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

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

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

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

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

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

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

4.5 Non-local exits

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

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

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

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

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

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

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

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

4.6 Fix-ups

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

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

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

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

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

5 RESULTS

5.1 Clasp build performance

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

5.2 VM Performance

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

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

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

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

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

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

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

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

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

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

6 EXAMPLE OF COMPILED CODE

6.1 Basic code

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

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

into:

```
check-arg-count-= 1
bind-required-args 1
```

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

```
const 0 ; '5
set 1
```

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

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

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

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

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

```
pop
return
```

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

6.2 Non-local exit example

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

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

The outer function compiles to the following:

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

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

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

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

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

```
  unbind
  nil
  pop
```

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

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

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

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

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

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

7 FUTURE DIRECTIONS

7.1 Trucler integration

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

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

7.2 File compilation

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

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

7.3 Conversion to IR

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

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

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

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

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

8 CONCLUSION

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

REFERENCES

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

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

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

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

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

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

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