

# Proceedings of the 14<sup>th</sup> European Lisp Symposium

Online, Everywhere May 3 – 4, 2021

Marco Heisig (ed.)



RavenPack



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

# Preface

# Message from the Program Chair

Welcome to the 14<sup>th</sup>th European Lisp Symposium!

I hope you are as excited for this symposium as I am. For most of you, these are the first minutes of the symposium. But the reason why we have this symposium, the website, the Twitch setup, and all the exciting talks and papers, is because of dozens of other people working hard behind the scenes for months. And I want to use this opening statement to thank these people.

The most important person when it comes to ELS is Didier Verna. Year after year, Didier takes care that other people are taking care of ELS. Thank you Didier!

Then there is the programme committee — brilliant and supportive people that spend countless hours reviewing each submission. These people help turn good papers into great papers, and these people are the reason why our conference proceedings are such an interesting read. Thanks everyone on the programme committee!

And then there are the local chairs — Georgiy, Mark, and Michał — that spent a large part of the last week refining our technical setup. Without our local chairs, we just couldn't have this online ELS. Thank you Georgiy, Mark, and Michał!

And finally, we have the true stars of the symposium, our presenters. These people spent weeks working on their papers, preparing slides, and recording their talks. And they will also be here live during each talk to answer questions or to discuss ideas. Thanks, dear presenters!

I had the honor of working with all these people over the last few months. And while doing so, I actually learned something new about Lisp, too. I learned that the best part about Lisp is not the uniform syntax, or the macros, or the ability to redefine functions at run time. The best part about Lisp is the Lisp community. And as long as we have such a great community, I am very optimistic about the future of Lisp.

In this sense, I wish you an inspiring symposium and happy hacking!

Röthenbach an der Pegnitz, May 1, 2021 Marco Heisig

# Message from the Local Chairs

During the previous ELS, we have wished for an European Lisp Symposium organized live, just like before the pandemic. These wishes did not yet have a chance to materialize, so it is our need - but, most importantly, also an honor - to invite you to the second online European Lisp Symposium!

People who have attented the previous year should feel at home; we have not made many modifications to the technology behind the conference. The video feed is streamed via Twitch, just like last year; it is possible to view it via VLC without using the official Twitch web client. The Twitch feed will start an hour before the opening ceremony to let everyone tune in ahead of time; we will be available to help with any technical issues.

The textual Twitch chat is going to be bridged to #elsconf on Freenode for people who would prefer to participate in the chat via IRC.

We have an official Jitsi instance (along with an unofficial backup) for breakout rooms, chats during coffee and lunch breaks, and the get-together after the conference. Links to the official break rooms will be posted during breaks, but new rooms can be created by all participants on demand.

This year, we also have a small charity auction and print sales for supporting various Lisprelated organizations; please stay tuned for the start of the auction during the conference's opening, and for the finish during the closing ceremony.

We wish you a good and satisfying Symposium, free of technical issues. See you during the conference!

Georgiy Tugai, Mark Evenson, Michał Herda

# Organization

# **Programme Chair**

• Marco Heisig, FAU Erlangen-Nürnberg, Germany

# **Local Chair**

- Michał Herda
- Mark Evenson, RavenPack
- Georgiy Tugai, Configura

# **Programme Committee**

- Breanndán Ó Nualláin, Machine Learning Programs, Netherlands
- Christophe Rhodes, Google, UK
- David McClain, SpectroDynamics LLC, USA
- Evrim Ulu, Istinye University, Turkey
- Ioanna Dimitriou, Igalia, Germany
- Irène Durand, LaBRI University of Bordeaux, France
- Jim Newton, EPITA Research Lab, France
- Jonathan Godbout, Google, USA
- Kai Selgrad, OTH Regensburg, Germany
- Kent Pitman, Harvard Medical School, USA
- Matthew Flatt, University of Utah, USA
- Michael Sperber, DeinProgramm, Germany
- Olin Shivers, Northeastern University, USA
- Paulo Matos, Igalia, Germany
- R. Matthew Emerson, toughtstuff LLC, USA
- Robert Smith, HRL Laboratories, USA
- Stefan Monnier, University of Montreal, Canada

# Sponsors

We gratefully acknowledge the support given to the 14<sup>th</sup>th European Lisp Symposium by the following sponsors:





ECOLE D'INGÉNIEURS EN INFORMATIQUE

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

RavenPack Urbanización Villa Parra Palomeras, 29602 Marbella, Malaga Spain www.ravenpack.com

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

# **Invited Contributions**

# Staged Relational Interpreters: Running Backwards, Faster

# Nada Amin, Harvard SEAS, USA

Relational programming, as exemplified in miniKanren, strives to be a pure form of logic programming. Relational interpreters enable turning functions into relations, as well as synthesizing functions from partial specifications. However, however successful, the approach incurs an interpretation overhead. We turn functions into relations, while removing all interpretation overhead thanks to a novel multi-stage programming mechanism. While partial evaluation has a rich history in both functional and logic programming, multi-stage programming has so far only been explored in a functional/imperative setting, with many success stories in high-performance computing. Bringing multi-stage programming to relational programming, we derive a relational compiler from the relational interpreter. Via this compiler, we generate relations from functions with no interpretation overhead. In addition to useful first-order applications, we explore running staged relational programs backwards.

In this talk, I will introduce multi-stage programming in its traditional functional setting and in the new relational setting. I will explain how to derive a compiler from an interpreter using generative programming. I will showcase the resulting staged-miniKanren system with a theorem checker turned prover and various towers of interpreters for generating quines.



Nada Amin is an Assistant Professor of Computer Science at Harvard SEAS. Previously, she was a University Lecturer in Programming Languages at the University of Cambridge; a member of the team behind the Scala programming language at the Ecole Polytechnique Federale de Lausanne (EPFL), where she pursued her PhD; and a software engineer at Google, on the compiler infrastructure supporting Gmail and other Google Apps. She holds bachelor of science and master of engineering degrees from the Massachusetts Institute of Technology (MIT).

# Symbolic expressions for cyber-physical programming

# Andrew Sorensen, Australian National University, Australia

Cyber-physical programming gives programmers the ability to engage actively and proactively in an experimental and experiential relationship with complex virtual, physical and cultural systems. Cyber-physical programming offers programmers the opportunity to actively shape and control these complex systems of events - to be engaged with them. Studying these natural events is often only possible in situ, in-the-world, within their own temporal and spatial frames of reference.

Cyber-physical programming posits that a causal connection to the world is valuable and that it can be useful to privilege the present. To quote Rodney Brooks "It turns out to be better to use the world as its own model". One of cyber-physical programming's tenets is that a program's

state is not only an internal property of the machine, but is also an external property-of-theworld. This external, physical state, can be sensed and acted upon by both the machine and the programmer. The programmer is actively engaged in both perceiving and acting upon the world, directly, but also mediated through the machine. For the cyber-physical programmer, the physical environment constitutes a meaningful component of a program's state.



Andrew is an international keynote speaker, academic, performer, educator, computer scientist and software craftsman with over 20 years of industry experience. With a diverse portfolio of interests, Andrew is as likely to be found hacking code in nightclubs as programming computational physics simulations on high performance computing clusters. As a Senior Research Fellow in the Computer Science department at the Queensland University of Technology Andrew convened courses in High Performance Computing and Enterprise Systems. Andrew holds a PhD in Computer Science from the Australian Na-

tional University, as well as a Bachelor of Music and a Master of Music, both from the Queensland University of Technology.

# Producing News Analytics and Turning them into Actionable Insights

# Andrew Lawson and Inna Grinis, RavenPack, Spain

We will first look at the basics of RavenPack's news analytics system, from incoming stories to outgoing analytics and the technology involved. We will then look at several examples of how RavenPack's news volume, sentiment scores, events taxonomy, and connections functionality can be leveraged to create data insights that can be used to make more informed decisions in the finance industry and the corporate world.

# **Program overview**

# Monday, 3.5.2021

- 09:00–09:15 Welcome Message
- 09:15–09:30 Auction Start
- 09:30–10:30 Andrew Sorensen: Symbolic expressions for cyber-physical programming (keynote)
- 10:30–11:00 Coffee break
- 11:00–11:30 Michael Wessel: A Tangram Puzzle Solver in Common Lisp
- 11:30–12:00 Jim Newton and Adrien Pommellet: A Portable, Simple, Embeddable Type System
- 12:00–14:00 Lunch
- 14:00–14:30 Eric Timmons: Common Lisp Project Manager
- 14:30–15:00 Antoine Hacquard and Didier Verna: A Corpus Processing and Analysis Pipeline for Quickref
- 15:00.-15:30 Coffee break
- 15:30–16:00 Lisp in the middle: using Lisp to manage a Linux system
- 16:00–17:00 Panel Discussion: Implementations of ANSI Common Lisp
- 17:00–17:30 Lightning talks17:30 Get-together on Jitsi

# Tuesday, 4.5.2021

- 09:30–10:30 Lawson, Grinis: Producing News Analytics and Turning them into Actionable Insights (keynote)
- 10:30–11:00 Coffee Break
- 11:00–11:30 Hayley Patton: A replicated object system
- 11:30–12:00 Eric Peterson and Peter Karalekas: aether: Distributed system emulation in Common Lisp
- 12:00-14:00 Lunch
- 14:00–14:30 Bélanger, Feeley: A Scheme Foreign Function Interface to JavaScript Based on an Infix Extension
- 14:30–15:00 Robert Strandh: Call-site optimization for Common Lisp
- 15:00–15:30 Coffee Break
- 15:30–16:30 Nada Amin: Staged Relational Interpreters: Running Backwards, Faster (keynote)
- 16:30–17:00 Lightning talks
- 17:00–17:15 Auction Stop
- 17:15–17:30 Closing Ceremony
- 17:30 **Conference End**

# Monday, May 3 2021

# A Tangram Puzzle Solver in Common Lisp

Michael Wessel lambdamikel@gmail.com Lambda @ Home Palo Alto, California, USA

# ABSTRACT

In this demo, we present a Common Lisp program capable of solving arbitrary geometric tiling problems. Well-known examples of such problems are Tangram puzzles. The program is implemented in Common Lisp, using the Common Lisp Interface Manager (CLIM) for its GUI. It consists of an editor and solver. We describe and demo the program and comment on its implementation as well as employed techniques to make the approach feasible.

# **CCS CONCEPTS**

• Computing methodologies → Discrete space search; Continuous space search; Spatial and physical reasoning; • Theory of computation → Computational geometry; • Mathematics of computing → Combinatoric problems; Solvers; • Humancentered computing → Graphical user interfaces; User interface programming.

# **KEYWORDS**

Tiling problems, Tangram, Search, Common Lisp, Heuristics, Common Lisp Interface Manager (CLIM), Geometrical Reasoning, Computational Geometry, NP-Hard Problems

# ACM Reference Format:

Michael Wessel. 2021. A Tangram Puzzle Solver in Common Lisp. In Proceedings of the 14th European Lisp Symposium (ELS'21). 5 pages. https://doi.org/10.5281/zenodo.4711456

# **1** INTRODUCTION

Geometric tiling problems have always fascinated humanity. Tiling problems occur naturally in many domains, from bathroom floor designs, over new forms of matter organized in quasicrystals via Penrose tilings [13, 19], to Turing-complete Domino systems used for establishing undecidability of computational problems. Tiling problems also provide for entertaining and challenging leisure time activities in the form of dissection puzzles, most prominently, Tangram puzzles [20].

A program capable of solving arbitrary finite – and hence decidable – tiling problems could be of great practical utility. The long term goal of this endeavor is to create a program capable of solving Penrose tilings, as shown in Fig.2. However, the more modest goal of solving Tangram puzzles was selected as a first milestone on this journey. A first version of this program was running in 2003 [17]. Given the tremendous advances in computer performance since

ELS'21, May 03-04 2021, Online, Everywhere

© 2021 Copyright held by the owner/author(s).

https://doi.org/10.5281/zenodo.4711456



Figure 1: Tangram Tiles & Tangram Problems with Solutions

2003, the author was curious whether the program would be more successful in finding solutions, and hence resurrected, refursbished, and reworked the heuristics of the original program.



Figure 2: Penrose Tiling (Source: Wikipedia, CC)

The set of standard Tangram tiles and some problems and their solutions are shown in Fig.1. Only the outline (silhouette) of the problem polygon is given, and the goal is to cover (or partition) the silhouette polygon completely with the available tile polygons, such that tiles do not overlap. Naturally, tiles can can be translated, rotated, and reflected. In a standard Tangram problem, each tile must be used exactly once.

Unlike many other classic search problems (8-Queens, Othello, Checkers, ...), which are standard AI textbook examples these days [9], the geometric tiling problem addressed here is considerably more complex from an implementation point of view, given its geometric nature.

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



Figure 3: Solving Tiling Problems by Geometric Search

# 2 SOLVING GEOMETRIC TILING PROBLEMS

Given a set of tiles  $\mathscr{T}$  such as the standard Tangram tiles shown in Fig.1, and the outline (silhouette) of a tiling problem (note that the concrete arrangements of tiles that constitute its outline is unknown), one way of solving a tiling problem is by geometric search.

The search space and process is illustrated in Fig.3. Starting with the initial problem polygon as the current configuration polygon (grey root at the top in Fig.3), possible successor configuration polygons are generated by finding possible alignments of all the remaining tiles  $\mathscr{T}$  with the problem polygon  $\mathscr{C}$ . Each aligned polygon tile is then subtracted from the current configuration polygon, until it is empty, and has hence been completely covered – a solution to the tiling problem has been found. Depending on the tiling problem, for Tangram puzzles it is required that each tile must be used exactly once, but the solver can also search for solutions that admit less tiles or allow multiple copies of the same tile to be used (e.g., for Penrose tilings).

Note that the current configuration polygon is potentially split into multiple components, as illustrated with the bottom configuration in Fig.3. In our solver, a configuration  $\mathcal{C}$  hence consists of a set of (not necessarily convex) polygons without holes, and their union represents the remaining area that still needs to be partitioned:  $\mathcal{C} = \{\mathcal{P}_1, \ldots, \mathcal{P}_n\}.$ 

A central function in every search algorithm is the successor state generator. Here, given a selected tile and the current configuration, the function find-possible-alignments finds reasonable ways of aligning a tile with a configuration, and returns a set of successor configurations:  $\mathscr{SC} = \{\mathscr{C}_1, \ldots, \mathscr{C}_n\}$ ). The search is performed either depth-first or breadth-first. As a central heuristics, the successor generator will always try to align a tile with the outer edges of the current configuration – this is a strategy that humans use when solving Jigsaw puzzles, but it would probably fail for Penrose tilings.



**Figure 4: Generating Successor Configurations** 

The number of potential successor configurations  $\beta$  (branching factor) can be very large, and for given configuration  $\mathscr{C}$  with a set of (remaining) tiles  $\mathscr{T}$  given by the formula

$$\beta = \sum_{c \in \mathscr{C}} \sum_{t \in \mathscr{T}} \# \text{ find-possible-alignments}(t, c)$$

Heuristics are essential in order to make the search succeed. Three types of selection heuristics are used to advise the search as follows:

- **Component selection heuristic:** which of the component polygons is tackled next?
- **Tile selection heuristic:** which of the remaining tiles will be tried next for alignment with the selected component polygon?
- **Successor configuration selection heuristic:** for the selected component polygon and tile, which of the generated potential successor configurations (see Fig.3) will be tried next?

These heuristics are implemented as scoring functions, calculating geometric characteristic of polygons such as polygon area, number of polygon components, and compactness, which is defined as the ratio of the polygon area to the area of a circle with the same perimeter. These are passed as :key arguments to the sort function for prioritization of the generated successor configurations, i.e., the *ordered* set  $\mathscr{SC} = \{\mathscr{C}_1, \ldots, \mathscr{C}_n\}$ ). Configurations  $\mathscr{C}_i$  that yield higher scores are prioritized in the search. Two configurations are considered equal in this set if they are *congruent*.

We have implemented and experimented with an array of different geometric functions for these three types of heuristics and found a configuration that ensures that many Tangram problems can be solved in a reasonable amount of time.

The computation of successor configurations / possible alignments is illustrated in Fig.4. For a given tile  $t \in \mathcal{T}$  and configuration polygon  $c \in \mathcal{C}$ , all possible pairings of segments from the configuration polygon and the tile polygon are considered. Each such segment pair can result in a potential successor configuration. For a chosen pair of segments, the tile polygon is first translated and

Michael Wessel

A Tangram Puzzle Solver in Common Lisp



Figure 5: Tangram Problem Editor

rotated to match the orientation of the configuration segment, and then translated onto the configuration polygon. If the transformed tile polygon then happens to lie completely on the inside of the configuration polygon (note that it may also overlap or lie fully on the outside as a result of the transformation), then the transformed tile polygon is subtracted from the configuration polygon, yielding a new successor configuration polygon. Polygon subtraction might result in several (disconnected or only connected over a shared single vertex, i.e., touching in a point) component polygons.

As a further complication, note that in order to accommodate for *reflections*, the individual tiles in the tile set  $\mathscr{T} = \{\mathscr{T}_1, \ldots, \mathscr{T}_n\}$ are actually sets of polygons themselves – each tile  $\mathscr{T}_i$  consist of a set of (simple) non-congruent polygons which are its *reflected variants*. Hence,  $\mathscr{T}_i = \{\mathscr{TV}_{i,1}, \ldots, \mathscr{TV}_{i,m}\}$ , and the solver ensures that only one variant is used of each tile. Note that #  $\mathscr{T}_i = 1$  for all but the parallelogram tile.

## **3 THE TANGRAM SOLVER APPLICATION**

The GUI of the solver is written using the Common Lisp Interface Manager, CLIM [5, 18]. The standard workflow for using the program is the following:

(1) The graphical editor, see Fig.5, offers rapid construction of Tangram problems. First, an active set of (construction) tiles is determined from the "Available Tile Sets" pane. The selected tile set then appears on the right side pane of the editor, from where they can be pulled into the editor area. Once in the editor area, tiles can be moved around, rotated (by 45 degrees), reflected, etc. CLIM gestures and context menus are available on the tiles. An active adaptive grid



Figure 6: Tangram Problem Library

for proper alignment makes precise problem construction a breeze.

- (2) Once the arrangement is complete, the "Problem" button is pushed to record a Tangram problem. Only the polygon outline of the arrangement is recorded, not the tiles and their actual arrangements. The constructed problem appears in the "Available Problems" pane, see Fig.6. From there, context menus for selecting and deleting the problem are available.
- (3) To start the problem solver, first the desired options are determined (whether to compute all solutions or only one, and if all tiles, fewer or more tiles are admissible). Then, the "Solve!" button starts the engine. If "Show Thinking" is checked, the tiling process is visualized and the program's progress can be witnessed, see Fig.7. Visualizing the program at work comes with some performance penalties, but is very helpful for judging the effectiveness of the solver's heuristics. The solver can be stopped at any time. Computed solutions show up in the "Known Solutions" pane visible in Fig.7 as soon as they are found.

The usual convenience functions are offered as well; e.g., problems and solutions can be saved to (and loaded from) disk, and so forth.

## 3.1 Performance

We haven't conducted extensive performance evaluations yet, but let us consider the Tangram problem in Fig.6.

• With the heuristics enabled, an older 2012 iCore 7-2670QM 2.2 Ghz Xenial Ubuntu PC with 8 GBs of RAM finds the first (dozen) solutions after 10 seconds, see Fig.8.

ELS'21, May 03-04 2021, Online, Everywhere





Figure 7: Watching the Solver at Work

Figure 8: Unexpected Solutions found by the Solver

 With the heuristics disabled, the author stopped the search after 1.25 hours with no solution so far, nicely demonstrating the necessity of intelligent geometric heuristics for this approach, even if these heuristics are expensive to compute due to their geometric nature.

It is interesting to note that the solver frequently finds creative, new solutions to problems, i.e. solutions that are different from the original tile arrangement used to construct the problem. Search sometimes leads to surprising and unexpected solutions. Novel solutions to standard Tangram problems from the Internet, e.g. the problems shown in Fig.1, are frequently found by the solver. A concrete example of this is shown in Fig.8, where the first solutions (in the "Known Solutions" pane) are clearly different from the tile arrangement that was used for constructing the problem silhouette shown in the editor in the first place.

# **4 IMPLEMENTATION DETAILS**

The author has implemented the geometric substrate of the solver over a number of years as part of his CS master and PhD thesis' work, from 1996 to 2003 [15, 16]. This base layer encompasses CLOS classes for geometric object representation (polygons, multiplecomponent polygons, lines, chains, points, segments), and metric, topological and geometric basic functions and methods (computation of angle, length, orientation, area, and compactness computation; line intersection, point-in-polygon test, methods for polygon union, intersection and difference, etc.)

Importantly, qualitative spatial relationships (Cohn's well-known *RCC8 relationships* [2, 11, 12]) for polygons are used for determining if a pair of polygons are congruent (RCC8 relation *Equal*), and if

a candidate tile alignment is valid. Valid aligned tiles must be in a *Tangential Proper Part Inverse* relation like in Fig.3, unlike the invalid *Proper Overlap* relation from Fig.7.

All of the geometric objects can be dynamically affected by nested transformation matrices, similar to the ones found in CLIM (with-translation, with-rotation). Rotating tiles causes floating-point precision challenges, e.g., congruency tests might fail because of inaccuracies from cos, sin multiplications. Hence, floating point rounding and an epsilon equality predicate (=-eps) is employed to compensate for these inaccuracies – note that some rotations correspond to irrational numbers, which cannot be represented precisely by Common Lisp's rational numbers.

Due to high-level support functions, the main covering algorithm and its heuristics are only  $\approx 500$  Lines of Code (LOC). However, the underlying geometric substrate implementation contributes  $\approx$ 3400 LOC (not all of its functionality is being used though), plus another  $\approx 1800$  LOC for the CLIM GUI. There is also a library of general purpose helper functions over  $\approx 1000$  LOC (only very few are being used); as a rough overall estimate, the application consists of  $\approx 5000$  LOC.

The program is OpenSource under GPL3 [7], and a YouTube video of the program in action is available as well [8]. The current version requires LispWorks (6 or 7) with CLIM, and runs on Windows, Linux, and Mac. Windows and Linux executables can be found in the GitHub repository as well [7].

# 5 RELATED & FUTURE WORK

This is a demo paper, so we do not attempt to survey the scientific literature on solving dissection or assembly puzzles in depth.

Michael Wessel

A Tangram Puzzle Solver in Common Lisp

However, we would like to present some pointers for the interested reader for further study, and put our work into context and perspective.

According to [22, 23], Tangram puzzles are an example of irregular shape packing problems, which are known to be NP-hard. Analyzing the computational complexity of special classes of (dissection) puzzles is an active field of computational geometry research; for example, it was shown only recently that symmetric assembly puzzles are NP-complete [3].

A plethora of Tangram puzzle game programs and apps is available on the Internet; however, with the notable exceptions of [1, 6, 21], these programs are typically incapable of solving new Tangram problems (without resorting to pre-recorded solutions).

The earliest approach for solving Tangram puzzles was given by Deutsch & Hayes [4] in 1972. Similar to the author's program, their algorithm is geometric in nature, and geometric heuristics are proposed. An early neural network-based approach is presented in [10], and a solver utilizing raster-based mathematical morphology in [23], working on bitmap raster representations. Interestingly, geometric search is not always the preferred line of attack for Tangram solving. In our opinion, it is the most straightforward and most intuitive method though.

Our solver [17] was developed without knowledge of previous work in the area, and precedes the papers [22, 23] as well as the programs [1, 6, 21]. It presents a somewhat naïve approach, mimicking some aspects of human Tangram solving. As such, the performance is probably not as good as some of the algorithms presented in the literature, but an in-depth comparison has not been performed yet. Our program performs surprsingly well on many standard Tangram problems, but can certainly benefit from refined heuristics and future research. A more thorough comparison of our approach with algorithms and heuristics from the scientific literature should hence be undertaken in the near future. Moreover, a set of Tangram benchmark problems is available [22] and could be used for comparative performance and completeness evaluations in the future.

One limitation of our current algorithm is that it does not accommodate for Tile scaling during the alignment process. The scale of the tiles could probably be computed as the square root of the quotient of the area of the silhouette polygon and the sum of the Tangram tile areas, assuming the presented problem has a solution. Another shortcoming of our current algorithm is that it cannot deal with polygons with holes; standard Tangram problems don't require polygons with holes, but Tangrams problems with holes exist as a variant of the game.

The author intends to develop this further, and will eventually also consider more complex tiling problems than Tangrams, especially the aforementioned Penrose tilings. More complex matching rules will have to be implemented then. A thorough effectiveness evaluation of the already implemented geometric heuristics on different problems should be conducted before proceeding with more sophisticated heuristics.

Another action item is to port the application to SBCL and Mc-CLIM [14]; currently, it requires LispWorks CLIM 2.

# ACKNOWLEDGMENTS

The author would like to thank Bernd Neumann for providing the initial motivation and funding for this work [17], the anonymous reviewers for thoughtful comments, and Rainer Joswig (aka Lispm) for compiling the application with LispWorks 7.1 on a Silicon M1 Mac and streamlining the sourcecode for LispWorks application delivery.

## REFERENCES

- [1] axel7083. Tangram solver in python 3, 2021. URL https://github.com/axel7083/ tangram-solver.
- [2] A. G. Cohn, B. Bennett, J. M. Gooday, and N. Gotts. RCC: a calculus for region based qualitative spatial reasoning. *GeoInformatica*, 1:275–316, 1997.
- [3] Erik D. Demaine, Matias Korman, Jason S. Ku, Joseph S.B. Mitchell, Yota Otachi, André van Renssen, Marcel Roeloffzen, Ryuhei Uehara, and Yushi Uno. Symmetric assembly puzzles are hard, beyond a few pieces. *Computational Geometry*, 90: 101648, 2020. ISSN 0925-7721. doi: https://doi.org/10.1016/j.comgeo.2020.101648. URL https://www.sciencedirect.com/science/article/pii/S0925772120300420.
- [4] E.S. Deutsch and K.C. Hayes. A Heuristic Solution to the Tangram Puzzle. Machine Intelligence, 7:205–240, 1972.
- [5] Franz Inc. Common Lisp Interface Manager (CLIM) 2.0: User Guide, 1994.
- [6] Rachel Eaton (Invisibility17). Tangram solver written in python 3 using opency, 2019. URL https://github.com/Invisibility17/tangram\_solver.
- [7] Michael Wessel (LambdaMikel). Tangram github repository, mar 2021. URL https://github.com/lambdamikel/Common-Lisp-Tangram-Solver.
- [8] Michael Wessel (LambdaMikel). Tangram youtube demo video, mar 2021. URL https://www.youtube.com/watch?v=UUn\_np8I3zg.
  [9] P. Norvig. Artificial Intelligence Programming: Case Studies in Common Lisp.
- Morgan Kaufmann, 1992.
   [10] K. Oflazer. Solving Tangram Puzzles: A Connectionist Approach. Journal of
- Intelligen Systems, 8(5):603–616, 1993. [11] D. A. Randell, Z. Cui, and A. G. Cohn. A Spatial Logic based on Regions and Con-
- [11] D. H. Kandi, Z. Cu, and Y. G. Comin. A spatial Edge observed in Regions and Comnections. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning*, pages 165–176, 1992.
- [12] David A. Randell, Zhan Cui, and Anthony G. Cohn. A spatial logic based on regions and connection. pages 165–176, 1992.
- [13] P. Steinhardt. The Second Kind of Impossible: The Extraordinary Quest for a New Form of Matter. Simon & Schuster, 2019.
- [14] Robert Strandh and Timothy Moorey. A Free Implementation of CLIM. 10 2002. URL https://common-lisp.net/project/mcclim/static/documents/clim-paper.pdf.
- [15] Michael Wessel. Flexible und konfigurierbare Software-Architekturen für datenintensive ontologiebasierte Informationssysteme. doctoralthesis, Technische Universität Hamburg, 2008. URL http://tubdok.tub.tuhh.de/handle/11420/834.
- [16] Michael Wessel and Ralf Möller. Flexible software architectures for ontologybased information systems. *Journal of Applied Logic*, 7(1):75–99, 2009. ISSN 1570-8683. doi: https://doi.org/10.1016/j.jal.2007.07.006. URL https://www. sciencedirect.com/science/article/pii/S157086830700064X. Special Issue: Emprirically Successful Computerized Reasoning.
- [17] Michael Wessel and Bernd Neumann. Ein Tangram-spielendes LISP-Programm, 2003. URL https://www.michael-wessel.info/papers/tangram-poster.pdf. Exponat auf der "Informatik-Expo 2004", Fachbereich Informatik, Universität Hamburg.
- [18] Wikipedia. Common lisp interface manager (clim), mar 2021. URL https://en. wikipedia.org/wiki/Common\_Lisp\_Interface\_Manager.
- [19] Wikipedia. Quasicrystal, mar 2021. URL https://en.wikipedia.org/wiki/ Quasicrystal.
- [20]  $\widetilde{\mathrm{Wikipedia}}.$  Tangram, mar 2021. URL https://en.wikipedia.org/wiki/Tangram.
- [21] Ziyi Wu (Wuziyi616). Tangram solver written in python 3 using opencv, 2019. URL https://github.com/Wuziyi616/Artificial\_Intelligence\_Project1.
- [22] Fernanda Yamada and Harlen Batagelo. A comparative study on computational methods to solve tangram puzzles. In 30th Conference on Graphics, Patterns and Images (SIBGRAPI), 10 2017.
- [23] Fernanda Yamada, Joao Gois, and Harlen Batagelo. Solving tangram puzzles using raster-based mathematical morphology. In 32th Conference on Graphics, Patterns and Images (SIBGRAPI), pages 116–123, 10 2019.

# A Portable, Simple, Embeddable Type System

Jim E. Newton jnewton@lrde.epita.fr EPITA/LRDE Le Kremlin-Bicêtre, France Adrien Pommellet adrien@lrde.epita.fr EPITA/LRDE Le Kremlin-Bicêtre, France

# ABSTRACT

We present a simple type system inspired by that of Common Lisp. The type system is intended to be embedded into a host language and accepts certain fundamental types from that language as axiomatically given. The type calculus provided in the type system is capable of expressing union, intersection, and complement types, as well as membership, subtype, disjoint, and habitation (nonemptiness) checks. We present a theoretical foundation and two sample implementations, one in Clojure and one in Scala.

# **CCS CONCEPTS**

# • Theory of computation $\rightarrow$ Data structures design and analysis; *Type theory*.

### ACM Reference Format:

Jim E. Newton and Adrien Pommellet. 2021. A Portable, Simple, Embeddable Type System. In *Proceedings of the 14th European Lisp Symposium (ELS'21)*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.5281/zenodo.4709777

# **1** INTRODUCTION

## 1.1 Long-term Motivations

The motivation for this research is two-fold. In the larger sense we will lay the groundwork, so that in a future publication we will be able reason about the regularity of heterogeneous sequences [28] in programming languages which support them. By heterogeneous sequence, we mean a sequence of arbitrary, finite length for which elements are of various types such as ["Alice" 12 "Bob" 54 "Eve" -3]. Typically, such a sequence does not contain completely random data, but rather data which follows a pattern: each element of the sequence must in turn be of a *type* determined by some sort of specification.

By reason about the regularity of these sequences we mean somehow specifying regular (in the sense of regular expressions) sequences of types, and to ask questions such as whether a given sequence (at run-time) belongs to this set. The theory of finite automata [20] describes how such a question can be answered in linear time, regardless of the complexity of the expression, provided that membership of the so-called *alphabet* can be determined in constant time for any element of the sequence in question. Other forms of reasoning might be to ask (at compile-time) whether an arbitrary sequence in one such set also belongs to another similarly specified set, or to compute a specifier for the intersection of

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

ELS'21, May 03–04 2021, Online, Everywhere

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-x-xxxx-x/YY/MM. https://doi.org/10.5281/zenodo.4709777 such sets and ask whether that intersection is inhabited or empty. These latter questions can be answered by exploiting an algebra of operations defined on deterministic finite automata (DFA).

Novice, non-Java-savvy Scala [33, 34] users may be surprised that the pattern matcher can distinguish between Int and Double and between List and Array, but cannot distinguish between List[Int] and List[Double] as this distinction is obviated by Java type-erasure [6]. We see sequences of constant type as a special case of type patterns which may also appear in input data such as JSON [37] or as s-expression [27].

The theory of deterministic finite automata is a powerful tool for such verification: however, it can only handle sequences on a finite alphabet instead of arbitrary values from the infinite set of objects representable in a programming language. A reasonable intuition would thus be to consider a finite alphabet of *decidable types*: each type representing a potentially infinite set of values. However, these types may intersect and thus violate the *determinism* requirement if a value in the sequence belongs to two overlapping types that lead to different branches of a given algorithm. Because of their nice algebraic properties, we wish to employ deterministic finite automata (DFA), as opposed to non-deterministic (NFA). We must thus avoid such non-determinism at DFA construction time.

In a smaller sense, we want to be able to compute a proper partition of the type space (defined formally in Definition 3.1) in order to decompose *large* types into *smaller* non-overlapping types. By *compute a partition of the type space*, we mean to express the set of all objects in the programming language as a disjoint union of subsets, where each subset is expressed in terms of intersections, unions, and complements of more *fundamental* types.

We state clearly that in this article, we will not describe how to recognize sequences of heterogeneous types yet such as Array<Int>: this goal would be premature. In [29] we addressed a generalization of this problem, but only for Common Lisp. Our long term goal is to extend that approach to work with any programming language for which SETS has been implemented.

To do so, we need to define beforehand a simple embeddable type system (SETS) which can be used in such a recognition flow. This type system is intended to be simple, and not intended to replace that of Common Lisp, nor is it intended to be powerful enough to implement the Common Lisp type system. The type system encapsulates just enough of the Common Lisp type system needed to implement the regular sequence recognition flow as described in [28] to other languages.

# 1.2 Goals and Contributions

We will therefore present in this article the first necessary step to address the goal of type-based sequence recognition. We introduce a simple type system (SETS) that defines *fundamental* types, and

which makes it possible to compute a partition of a type space using a given set of types and set-theoretic operations  $(\cup, \cap, -, \in, \subset)$ . This simple type system will be used as a building block for later development in a future publication, but deserves careful consideration in its own right.

Our previous work addressed this problem for Common Lisp, as described in [28]; we begin in this article our goal of generalizing the method to a larger class of languages. We demonstrate the SETS type system by implementing it in three programming languages.

- *Clojure* [18, 19] was chosen because it is a lisp, and we hoped that it would provide a logical transition for a theory based in Common Lisp.
- *Scala* runs on the same JVM [25] as does Clojure, thus allows similar sequences and type reflection operations despite syntactic differences.
- Python [48] is a dynamic language that makes different assumptions about types than do JVM based languages.

All three of these languages support heterogeneous sequences, and provide a type system with reflection. The implementation in Python is in its infancy, and the details of its implementation are discussed sparingly in this article.

The current article lays a foundation by defining a portable, Simple, Embeddable Type System (SETS). The principles of SETS, defined formally in Section 3.2, are inspired by the type system of Common Lisp, which we outline in Section 2.4. We have identified some limitations of the Common Lisp type system in Section 4.5 and propose solutions in SETS. We then present two sample implementations of SETS, one in Clojure and one in Scala in Sections 4.2 and 4.3 respectively.

# 2 PREVIOUS WORK

# 2.1 Type-based Sequence Recognition

In [28] we discussed the flow for developing type based sequence recognition in the explicit context of Common Lisp [2], as Common Lisp provides a built-in type system capable of expression Boolean combinations of types of arbitrary complexity. We presented several algorithms for computing Maximum Disjoint Type Decomposition (MDTD) [30], but those algorithms make heavy use of the assumption that the underlying type system is that of Common Lisp.

A notable implementation detail of rational type expressions in Common Lisp [29] is that the type definition makes reference to the satisfies type. This use of satisfies means that in many cases, subtypep cannot reason about the type. In the current work, we define the building blocks of a type system which can be extended (without resorting to satisfies) in a future publication which will be able to more easily reason about such types.

## 2.2 Types as Boolean Combinations

Castagna *et al.* [7] argue that many programmers are drawn to the flexibility and development speed of dynamically typed languages, but that even in such languages, compilers may infer types from the program if certain language constructs exist. The authors argue that adding set theoretical operations (union, intersection, complement)

to a type system facilitates transition from dynamic typing to static typing while giving even more control to the programmer.

Researchers in early programming languages experimented with set semantics for types. Dunfield [13] discusses work related to intersection types, but omits acknowledging the Common Lisp type system. He mentions the Forsythe [39] language from the late 1980s, which provides intersection types, but not union types. Dunfield claims that intersections were first developed in 1981 by Coppo *et al.* [9], and in 1980 by Pottinger [38] who acknowledges discussions with Coppo and refers to Pottinger's paper as forthcoming. Work on union types began later, in 1984 by MacQueen *et al.* [26].

Languages which support intersection and union types [7, 8, 36], should also be consistent with respect to subtype relations. Frisch *et al.* [15] referred to this concept as semantic subtyping. In particular, the union of types A and B should be a supertype of both A and B, the intersection of types A and B should be a subtype of both A and B, and if A is a subtype of B, then the complement of B should be a subtype of the complement of A. While Coppo and Dezani [10] discuss intersection types in 1980, according to a private conversation with one of the authors, Mariangiola Dezani, theoretical work on negation types originates in Castagna's semantic subtyping.

The theory of intersection types seems to have influenced modern programming language extensions, at least in Java [16] and Scala. Bettini *et al.* [4] (including Dezani, mentioned above) discuss the connection of Java  $\lambda$ -expressions to intersection types.

The standard library of Scala language [33, 34] supports a type called Either. This type serves some of the purpose of a union type. Either[A,B] is a composed type, but has no subtype relation neither to *A* nor to *B*. Either lacks many of the mathematical properties of union; it is neither associative *Either*[*A*, *B*], *C*]  $\neq$  *Either*[*A*, *Either*[*B*, *C*]], nor commutative *Either*[*A*, *B*]  $\neq$  *Either*[*B*, *A*]. Sabin [44] discusses user level extensions to the Scala type system to support intersection and union types. Yet, Sabin [43] recommends that no one use his implementation in *real code*. Doeraene [12, 22] introduces pseudo-union types in Scala.js. Scala 3 [1, 31, 42] supports intersection and union types in a type lattice, but not complementary types. However, according to the Scala 3 specification<sup>1</sup>, the type inferencer may not consider union types while computing the least upper bound of two types.

# 2.3 Difficulties with the Subtype Relation

Grigore [17] addresses subtype decidability in Java [16]: deciding whether one type is a subtype of another is equivalent to the halting problem. Kennedy<sup>2</sup> and Pierce [24] identify several restrictions which when imposed, make the question decidable. They investigate the effect of restricting these capabilities in different ways in Java [16], Scala [32, 33], C#, and .NET Intermediate Language.

In Scala, the operator <:< attempts to compute the subtype relation, returning *true* if this relation can be proven, else *false*. Thus, the Scala <:< conflates *cannot compute* with *computed to be false*. The method isAssignableFrom defined on java.lang.Class is also available to the Scala programmer. This (decidable) method

 $<sup>^1 \</sup>rm See$  https://dotty.epfl.ch/docs/reference/new-types/union-types.html for a description of the behavior of Union Types in Scala 3

<sup>&</sup>lt;sup>2</sup>Curiously, Kennedy [24] from Microsoft Research gives citations for Java and Scala, but not for C# and .NET Intermediate Language. Perhaps authors believe the reader is more familiar with C# and .NET IL than with other languages.

A Portable, Simple, Embeddable Type System



Figure 1: Example Common Lisp types with intersection and subset relations

computes the subtype relation, returning true if the subtype relation is provided by the class hierarchy, and *false* otherwise.

Bonnaire [5] presents typed Clojure, but does not rigorously define a Clojure type, and does not implement anything as ambitious as Common Lisp's subtypep. The Clojure [18, 19] language offers an isa? predicate, which is simple and decidable but is ultimately based on isAssignableFrom which is provided by the JVM [16].

In Common Lisp [2] subtypep can be computationally intensive in general. Even in cases where the subtype relation is decidable, a Common Lisp implementation is allowed to return don't know in cases deemed to be too computationally intensive. [3, 47]

#### 2.4**Types in Common Lisp**

A detailed discussion of Common Lisp types can be found in the Common Lisp specification [2, Section 4.2.3]. A summary thereof emphasizing peculiarities of function types can be found in [28]. We present here a shorter version of that exposé, emphasizing features potentially portable to other programming languages.

In Common Lisp, a type is a (possibly infinite) set of objects at a particular point of time during the execution of a program [2]. As illustrated in Figure 1, Common Lisp programmers may take many ideas about types from the intuition they already have from set algebra. An object can belong to more than one type. Two given types may intersect, such as the case of unsigned-byte and fixnum in the figure, (unsigned-byte  $\cap$  fixnum)  $\neq \emptyset$ . Types may be disjoint such as float and fixnum, (float  $\cap$  fixnum) =  $\emptyset$ . Types may have a subtype relation such as fixnum and number, (fixnum  $\subset$  number) or more complicated relations such as (bit  $\subset$ (fixnum  $\cap$  unsigned-byte)  $\subset$  rational).

Types are never explicitly represented as objects by Common Lisp. Instead, a type is referred to indirectly by so-called type specifiers, which designate types. The symbol string and the list (and number (not bit)) are type specifiers [2]. Since types are not objects specified in the language, but rather designated by specifiers, it becomes challenging to reason about types. I.e., programmers must construct algorithms to determine whether designated types are inhabited vs. empty, and to efficiently represent types designated by composing type specifiers in various ways.

Two important Common Lisp functions pertaining to types are typep and subtypep. The function typep, a set membership test,



Figure 2: Example hierarchy of hosted types. Arrows point from subtype to supertype.

is used to determine whether a given object is of a given type. Type specifiers indicating compositional types are often used on their own, such as in the expression (typep x '(or string (eql 42))), which evaluates to true either if x is a string, or is the integer 42. The function subtypep, a subset test, is used to determine whether a given type is a recognizable subtype of another given type. The function call (subtypep T1 T2) distinguishes three cases: (1) that T1 is a subtype of T2, (2) that T1 is not a subtype of T2, or (3) that the subtype relationship cannot be determined. Newton [28] and Valais [46] discuss scenarios which fall into this third case.

#### PORTABLE AND EMBEDDABLE 3

Here we present a type system which exhibits some notable features of the Common Lisp type system, in particular, union, intersection, and complement types, type membership and subtype predicates. The type system we present is intended to be implementable in other languages, and should allow run-time code to reason about sets of values. We present our assumptions with respect to the host language in Section 3.1, an abstract definition of SETS in Section 3.2, explain some subtleties of the subset relation in Section 3.4, then discuss different subset algorithms in Section 3.5. Sections 4.2 and 4.3 present sample implementations in Clojure and Scala respectively.

# 3.1 Hosted Types

We are interested in embedding a simple type system into a host language. We will assume that said host language provides a set of fundamental types which we will accept as given as well as a distinct set  $\Upsilon_0$  of designators for those fundamental types. Just as Common Lisp distinguishes between types and type specifiers, we will also distinguish between the types (sets of values) and their designators (syntactical elements). E.g. consider  $\Upsilon_0 = \{symbol, string, number, integer, float, bignum, fixnum\}$ as shown in Figure 2. See Section 4.1 for a discussion of parameterized types such as List<Int>.

ELS'21, May 03-04 2021, Online, Everywhere

We further assume that the host language provides membership and subset relations with respect to  $Y_0$ . The membership relation must be decidable and properly implemented in the host programming language. For example, we must be able to determine that "hello world"  $\in$  string and that 1.2  $\notin$  integer.

We also require that the programming language provide a subtype query mechanism with respect to  $\Upsilon_0$ . For example, in Figure 2, we see that *fixnum*  $\subset$  *integer*  $\subset$  *number* and that *string*  $\not\subset$  *integer*. Even if the subtype relation is sometimes undecidable, we intend to exploit the cases when it is decidable. See Section 2.3 for further discussion of the decidability of the subtype relation.

In Common Lisp, the typep and subtypep functions are built-in, and symbols implement type designators (called type specifiers in Common Lisp) for built-in types.

In Clojure, the functions instance? and isa? correspond respectively to the Common Lisp functions typep and subtypep as far as low level types are concerned. In Python, the functions are isinstance and issubclass.

In Scala, a class is an object that belongs to a sort of meta-class, java.lang.Class, and that class has methods named isInstance and isAssignableFrom. We have access to the (meta-)class object via that classOf[] syntax.

```
// membership test in Scala
classOf[java.lang.String]
.isInstance("hello world")
// subtype test in Scala
classOf[java.lang.Number]
.isAssignableFrom(classOf[java.lang.Integer])
```

# 3.2 Formal definition of SETS

Different programming languages make different assumptions about types. We assume a type system in accord with Definition 3.2.

Definition 3.1 (type, type space). Let  $\Sigma$  denote the set of all values expressible in a given programming language. A *type space* is the set  $2^{\Sigma}$  of subsets of  $\Sigma$ . Each subset of  $\Sigma$  is called a *type*.

Definition 3.2 (simple embedded type system). A simple embedded type system (SETS) consists in a set  $\Upsilon$  of type designators, two operators  $\in$  and  $\subset$ , and a type designation function  $[\![\,]\!] : \Upsilon \to 2^{\Sigma}$ . Intuitively, each  $v \in \Upsilon$  designates a type which is the set  $[\![v]\!]$ . Formally, the set  $\Upsilon$  and the function  $[\![\,]\!]$  are defined inductively, using the following atomic elements:

- Hosted types: let Y<sub>0</sub> be a set of *native type designators* of the host language, such that to each v ∈ Y<sub>0</sub>, a *native type* σ<sub>v</sub> can be matched. Then Y<sub>0</sub> ⊂ Y and ∀v ∈ Y<sub>0</sub>, [[v]] = σ<sub>v</sub>.<sup>3</sup>
- Universal type: there is a symbol  $\top \in \Upsilon$ , and  $\llbracket \top \rrbracket = \Sigma$ .
- **Empty type**: there is a symbol  $\perp \in \Upsilon$ , and  $\llbracket \perp \rrbracket = \emptyset$ .
- Singletons: ∀a ∈ Σ, there is a matching element {a} ∈ Υ, and [[{a}]] = {a}.
- **Predicates**: for any decidable function f mapping  $\Sigma$  to  $\{true, false\}$ , there is a symbol  $\mathfrak{Sat}_f \in \Upsilon$ , and  $\llbracket \mathfrak{Sat}_f \rrbracket = \{x \in \Sigma \mid f(x) = true\}.$

And the following constructors:

Jim E. Newton and Adrien Pommellet

- Union: if  $v_1, v_2 \in \Upsilon$ , then  $v_1 \cup v_2 \in \Upsilon$  and  $\llbracket v_1 \cup v_2 \rrbracket = \llbracket v_1 \rrbracket \cup \llbracket v_2 \rrbracket$ .
- Intersection: if  $v_1, v_2 \in \Upsilon$ , then  $v_1 \cap v_2 \in \Upsilon$  and  $\llbracket v_1 \cap v_2 \rrbracket = \llbracket v_1 \rrbracket \cap \llbracket v_2 \rrbracket$ .
- Complement: if  $v \in \Upsilon$ , then  $\overline{v} \in \Upsilon$  and  $\llbracket \overline{v} \rrbracket = \Sigma \setminus \llbracket v \rrbracket$ .

Moreover, the relations  $\in$  and  $\subset$  on  $\Upsilon$  should verify the following properties:

- The operator ∈ defines a *decidable* membership relation between values and type designators. Formally, given v ∈ Υ and a ∈ Σ, a ∈ v holds if only if a ∈ [[v]], and a ∈ v̄ if and only if a ∉ [[v]].
- (2) There is a partial subset relation  $\subset$  on types. Given  $v_1, v_2 \in \Upsilon$ ,  $\llbracket v_1 \rrbracket \subset \llbracket v_2 \rrbracket$  if and only if  $v_1 \subset v_2$ . However, this subset relation may be *undecidable*. See Sections 2.3 and 3.4.  $\triangle$

The  $\mathfrak{Sat}_f$  type is capable of defining many of the same sets definable by other types in SETS by clever enough choices of f. However, doing so will limit reasoning power. For example, given two predicates f and g it is not possible in general to determine whether  $\mathfrak{Sat}_f \subset \mathfrak{Sat}_q$  or whether  $\overline{\mathfrak{Sat}_f} \cap \overline{\mathfrak{Sat}_q}$  is inhabited.

We do not specify how the programming language represents type designators: possibly as part of the language specification [45, Chapter 4] as in Common Lisp, or as an Algebraic Data Type (ADT) [21] implemented as an add-on library.

## 3.3 Relating SETS to Common Lisp

SETS as described in Section 3.2 is based on the type system of Common Lisp but simplified, and omitting some features which make the subtype relation difficult to compute. For example, we omit interval (such as (float -1.0 (2.0))), mod, and member types as well as composed forms such as (vector double-float 100). Whereas Common Lisp defines a member type, SETS represents such types as a finite union of singleton types.

The hosted types of SETS correspond to the types designated by the atomic forms of the type specifiers mentioned Figure 2-4 of Section 4.2.3 in the Common Lisp specification [2] as well symbolic names of Common Lisp classes, conditions, and structures. List forms of types mentioned in Figure 4-3 of Common Lisp specification Section 4.2.3 are omitted in SETS, such as (cons float), (function (t)), and (simple-array \* 4).

The universal and empty types of SETS are respectively designated by t and nil in Common Lisp. Singleton types in SETS are represented by eql in Common Lisp. Predicates in SETS are represented by satisfies in Common Lisp. Union, intersection, and complement in SETS are represented respectively by or, and, and not in Common Lisp.

The  $\in$  operation in SETS is implemented by the Common Lisp typep function. The  $\subset$  operation in SETS is implemented by the Common Lisp subtypep function, which returns two values. A second such value of nil means the subtype relation undecidable; otherwise a first value of true or false correspond to the decidable true and false.

# 3.4 A Challenging Subtype Relation

Note that the subtype relation is the atomic function for detecting intersection, disjointness, and emptiness [3, Sec 6]. We need these

<sup>&</sup>lt;sup>3</sup>Recall from Section 3.1 that we axiomatically assume that this makes sense.

A Portable, Simple, Embeddable Type System

checks in order to minimize a partition of the type space. Although we use functions such as disjoint? and inhabited? in our code, these are merely semantic wrappers around subtypep?, at least from the API perspective, if not from the internal implementation perspective.

However, the subset relation may not always be *decidable*. Thus, the decision function in SETS should be allowed to return *dont-know* in a manner appropriate to the host language, although we would like it to be as decisive as possible. We discuss here three scenarios when such an *indecision* might occur.

The most obvious situation involves  $\mathfrak{Sat}$ . Given two arbitrary decidable predicates f and g, it is not possible to know whether  $\mathfrak{Sat}_f \subset \mathfrak{Sat}_g$ . This is a direct consequence of not being able to decide whether a given recursive language is a subset of another.

A second scenario involves a hidden empty set. Consider disjoint sets *A* and *B*.  $A \cap C \not\subset B$  if and only if  $A \cap C \neq \emptyset$ . However, it may be impossible to decide whether  $A \cap C$  is empty, as it is undecidable to determine whether a given recursive language is empty.

Consider a third scenario featuring two types *A* and *B* that are disjoint such as the number types Double and Long in the JVM. We wish to confirm that  $Long \subset Double$ . Note that  $A \subset \overline{B}$  if  $A \cap B = \emptyset$  and  $\overline{A} \neq B$ . If we want to determine whether  $A \subset \overline{B}$ , testing whether  $A = \overline{B}$  would be a mistake: we would have to test both  $A \subset \overline{B}$  and  $\overline{B} \subset A$ , thus introducing an infinite loop in our subtype procedure.

We consider the subtype procedure inaccurate if it returns *dont-know* in a decidable case. We use the term *inaccurate* deliberately after considering alternative terminology. The only way we have of knowing whether the subtype relation is satisfied is to determine so (or not so) algorithmically. Given two algorithms, X and Y, if X determines that  $A \subset B$  or that  $A \notin B$ , but Y returns *dont-know* then we consider Y less accurate then X in this regard. Thus, in some cases, it is algorithmically impossible to distinguish between an *inaccurate* answer and an *undecidable* problem.

Our current solution to the question of whether  $Long \subset \overline{Double}$  is to include an admittedly inaccurate special case in the subtype procedure which checks whether it is dealing with hosted types and their complements. We may then use known properties of hosted types, as in Algorithm 1 of Section 4.4, to determine that  $\overline{Double} \notin Long$ . We may however not be able to directly handle relations involving more complex types such as  $\overline{\{0\}} \subset \{1\} \cup \overline{\{0\}}$ .

Undetermined cases can cause a cascade effect in increasing the complexity of a procedure. Indeed, consider two types A and B. Their standard partition is defined as  $\{A \cap B, A \cap \overline{B}, \overline{A} \cap B, \overline{A} \cap \overline{B}\} \setminus \{\emptyset\}$ . The automata-theoretic approach, alluded to in Section 5.2, heavily relies on such partitions. The maximum size of this partition is 4, but various subtype relations may decrease it: as an example, if  $A \subset B$  then  $A \cap \overline{B} = \emptyset$ . Given that the size of standard partitions can grow exponentially with the number of types involved, it is of the utmost importance to keep it as small as possible. Thus an inaccuracy in the algorithm may cause unnecessary performance problems when constructing finite automata.

# 3.5 Computing the Subtype Relation

The specification of SETS does not impose any particular algorithm. We discuss here three known approaches which have been used to compute subtypep in Common Lisp: (1) Baker's algorithm, (2) Binary Decision Diagrams, and (3) Symbolic Boolean manipulation.

Baker's [3] algorithm (approach 1) was presented anew by Valais [47] in attempt to make it more understandable. At a high level, the algorithm attempts to represent every type designator by a bit-mask, allowing union, intersection, and complement to be computed by Boolean operations of bit-vectors. Equivalent types have the identical bit-masks; the empty type has an all zero bit-mask. Baker's algorithm decides whether *A* is a subtype of *X* by computing the bit-wise AND of *a* and  $\overline{x}$  given a bit-vector *a* of *A* and a bit-vector *x* of *X*. If the result is 0, then  $A \subset X$ . The ECL [41] Common Lisp compiler uses a variant of Baker's algorithm.

We [30] presented Binary Decision Diagrams (BDDs) (approach 2) as computation tools for manipulating Common Lisp type specifiers. BDDs are a powerful tool for manipulating Boolean functions. A BDD can represent a type specifier, and Boolean operations between BDDs compute canonical forms for union, intersection, and complement types.

Approach 3 employs a symbolic Boolean manipulation approach. A type designator represents an abstract syntax tree (AST) for a Boolean expression. Computing the subtype relation involves checking a long but inexhaustive series of conditions, some necessary, some sufficient, and some both necessary and sufficient. If a necessary condition fails, false is returned. If a sufficient condition succeeds, true is returned. If all these conditions fail to decide the result, then dont-know is returned. The SBCL[40] Common Lisp compiler's implementation of subtypep relies on a similar yet augmented procedure [23].

The sample implementations of SETS presented in Section 4 rely on the third approach. In the Scala implementation (Section 4.3) of SETS, dynamic dispatch is used on the class of the type designator object to direct the computation away from tests known to be irrelevant. In the Clojure implementation (Section 4.2), type designators are represented by s-expressions, which are examined by an ad-hoc pattern matcher which determines according to the first element (and, or, not, etc) which irrelevant checks it should eliminate. Examples of such conditions are:

- *necessary and sufficient*:  $\overline{A} \subset \overline{X}$  if and only if  $X \subset A$ .
- *sufficient*:  $X \subset A \cup B$  if  $X \subset A$  or  $X \subset B$ .
- sufficient:  $A \cap B \subset X$  if  $A \subset X$  or  $B \subset X$ .
- *necessary*:  $X \subset A \cap B$  only if  $X \subset A$  and  $X \subset B$ .

We must also take into account the computational complexity of the solution. Our representation of a type designator is an expression tree. Some operations on such a tree have exponential complexity in terms of the height of the tree. It is possible to minimize the depth of the tree by converting the expressions to DNF (disjunctive normal form) or CNF (conjunctive normal form), but the conversion itself has an exponential worst case complexity. Yet, conversion to a normal form can significantly increase the accuracy of the decision procedures, as we will demonstrate in Section 5.1.

# **4** IMPLEMENTATIONS OF SETS

Three implementations of SETS are currently available, one in Clojure as described in Section 4.2, a second in Scala as described in Section 4.3, and a third which is currently under development in Python. The Python implementation (called Genus) is publicly available at https://gitlab.lrde.epita.fr/jnewton/python-rte, but we do not discuss specific details thereof in this article.

# 4.1 Java Subclass vs. SETS Subtype

In order to avoid confusion later, we clarify here that types in SETS are not the same as what a Java programmer might otherwise think of as a type, nor the Scala or Clojure programmer.

The Clojure and Scala implementations of SETS both fundamentally manipulate the same low-level objects within the JVM, albeit the derived classes of objects expressible in each language are different. Both at the Scala level and the Clojure level, no substantial difference is made between classes and interfaces—a fact which might be unfamiliar to the Java programmer. The Scala and Clojure implementations of SETS treat interfaces and classes identically in the sense that a type (in the SETS sense) is a set of objects which share a particular java.lang.Class in a particular list of super classes.

If  $C_C$  is a class in Java, then SETS associates with  $C_C$  the set of all (Scala or Clojure) objects whose java.lang.Class contains  $C_C$  in its list of super classes. Similarly, if  $C_I$  is an interface in Java, then SETS associates with  $C_I$  the set of all objects whose java.lang.Class contains  $C_I$  in its list of super classes. In this sense  $C_I \subset Object$  in SETS even though a Java programmer might insist that  $C_I$  is not a subclass of *Object*. That is to say, when we speak of subtype-ness in SETS, we are strictly referring to whether one set of objects is a subset of another set, not whether Java considers the classes as subtypes.

In order to avoid confusion, we state explicitly that we have not extensively experimented with parameterized Java types such as List<String>. Java type-erasure makes some such introspection impossible, but we do not know to what extent (if at all) it is possible to query such types at run-time via the Java reflection API. Furthermore, we do not know to what extent a user of Genus can confuse the system by using such parameterized types.

# 4.2 Genus, SETS in Clojure

The Clojure implementation of SETS is called Genus. The documentation is available publicly, at https://gitlab.lrde.epita.fr/jnewton/ clojure-rte/-/blob/master/doc/genus.md. The source code is available as part of a larger project called clojure-rte at https://gitlab. lrde.epita.fr/jnewton/clojure-rte. Readers familiar with the Common Lisp type system will find Clojure Genus to be intuitive, and will recognize that we are, in user space, imposing a simplified version of the Common Lisp type system onto Clojure.

4.2.1 *How Clojure-Genus is used.* The API of Genus consists of an implicit data structure which implements the type designators, following the pattern described in Definition 3.2. In Clojure Genus, a type designator is one of the following:

• Hosted types: A symbol, s for which (resolve s) has type java.lang.Class. *E.g.*, String or clojure.lang.Symbol.

- Universal: :sigma.
- Empty: :empty-set.
- **Singletons**: A list of the form (= x). *E.g.*, (= 0).
- **Predicates**: A list of the form (satisfies f). f is a symbol resolving to a function. *E.g.*, (satisfies odd?).
- Union: A list of the form (or ...). Operands are type designators. *E.g.*, (or String (satisfies keyword?)).
- **Intersection**: A list of the form (and ...). Operands are type designators. *E.g.*, (and Long (satisfies pos?)).
- **Complement**: A list of the form (not x). Operand is a type designator. *E.g.*, (not (or Long String)).

As required by Definition 3.2, two functions typep and subtype? on type designators implement the  $\in$  and  $\subset$  operators. The typep (binary) function can be used as in (typep "hello" '(or String clojure.lang.Symbol)), and returns true or false.

The subtype? function is more complicated; it accepts two arguments A and B, and returns one of true, false, or :dont-know. If subtype? proves that  $A \subset B$ , it returns true; if it proves that  $A \not\subset B$ , it returns :dont-know.

In addition to subtype? two other functions are provided for reasoning about types via type designators:

- disjoint?, returns true, false, or :dont-know indicating whether two types are disjoint, *i.e.* their intersection is empty.
- inhabited? returns true, false, or : dont-know indicating whether the type is not empty.

The disjoint? and inhabited? functions are not required by the SETS specification. We have implemented them here to facilitate the implementation of the subtype predicate. In fact, given a fully functional, self-contained implementation of subtypep?, disjoint? and inhabited?, if needed, could be implemented in terms of subtype?. Two types are disjoint if their intersection is a subtype of the empty type. A type is inhabited if it is not a subtype of the empty type. Special care would need to be taken for the case that subtype? returns dont-know.

4.2.2 How Clojure-Genus is implemented. The implementation of the public API (including the functions typep, subtype?, disjoint?, and inhabited?) is around 1600 lines of Clojure code. Space does not permit a full explanation of all the code. We invite the reader to peruse the code. We will summarize some details of the functions typep and disjoint?.

The typep function is implemented as a multi-method, where each method implements the type membership decision for a particular type designator. Two such methods are shown here.

```
(defmethod typep 'not [a-value [_a-type t]]
  (not (typep a-value t)))
(defmethod typep 'and [a-value [_a-type & others]]
  (every? (fn [t1] (typep a-value t1)) others))
```

Whereas methods implement typep directly, other functions are implemented with a *shadow* function; e.g., -subtype?, -disjoint, and -inhabited. These are not methods in the normal Clojure sense, but rather are called according to a Common Lisp-like method-combination [35]. The method-combination calls each of the methods in turn. Each method attempts to prove sufficient

Jim E. Newton and Adrien Pommellet

A Portable, Simple, Embeddable Type System

conditions or disprove necessary conditions as explained in Section 3.5 and thus return true, false, or :dont-know. The methodcombination continues to call the methods until one is conclusive, returning :dont-know as a last resort. The disjoint? function is an exception. Its method-combination assures that (disjoint A B) returns the first conclusive value of either (-disjoint? A B) or (-disjoint B A), or :dont-know as a last resort.

(defmethod -disjoint? 'or [t1 t2]
(cond (not (gns/or? tl))
:dont-know
;; sufficient
(every? (fn [t1'] (disjoint? t1' t2 false))
(rest t1))
true
;; necessary
<pre>(some (fn [t1'] (not (disjoint? t1' t2 true)))</pre>
(rest t1))
false
:else :dont-know))
<pre>(defmethod -disjoint? :subtype [sub super]</pre>
<pre>(cond (and (subtype? sub super false)</pre>
(inhabited? sub false))
false
:else :dont-know))

In the -disjoint? 'or method above, the code asks whether the first argument called is a type designator of the form (or ...). If it is not of this form, then :dont-know is returned. Otherwise other necessary and sufficient conditions are tested. If they are inconclusive, :dont-know is returned, indicating to the caller to continue with other methods of the method combination.

Two tests are made (in the 2nd and 3rd clause of the cond). First, a sufficient condition examines each of operand (or ...) to detect whether all of them designate a type which is disjoint from t2; if so, then the types are deemed to be disjoint. Second, a necessary condition examines each operand of (or ...) to detect whether at least one of them is definitely *NOT* disjoint from t2; if such a type is found, then we conclude that that t1 and t2 are not disjoint. Finally, if the tests were inconclusive we return :dont-know.

The -disjoint : subtype method shown above checks an expensive but general condition. The method attempts to detect whether the two designated types are in a subtype relation. If  $A \subset B$  then the types are not disjoint, except in the case that  $A = \emptyset$ . Since  $\emptyset$  is disjoint from every type, including itself, we must assure that the type in question is inhabited, via a call to the inhabited? method.

We show the implementation of this method to partially justify the reason we have included the inhabited? method in our protocol. During our implementation of Clojure Genus we found the existence of inhabited? simplifies the logic of certain subtype and disjoint decisions, even if not strictly necessary. In Common Lisp (not (subtypep A nil)) serves a similar purpose, understanding the caveat that if Common Lisp (subtypep A nil) returns a second value of nil then (not (subtypep A nil)) still evaluates to *true*, even though A may not really be inhabited. The method-combination assures that once the method above with dispatch value 'or returns true or false, then the method whose dispatch value is :subtype is averted.

# 4.3 Genus, SETS in Scala

The Scala Genus implementation is about 1500 lines of Scala code– roughly the same size as the Clojure version. The code is available publicly at https://gitlab.lrde.epita.fr/jnewton/regular-typeexpression/-/tree/master/cl-robdd-scala/src/main/scala/genus.

In Scala Genus a type designator is implemented as an abstract class named SimpleTypeD (simple-type-designator), and several leaf classes which extend the abstract class. There is one leaf class per type designator, *e.g.* the case class SAnd defines the type designator implementing intersection, and analogously SOr for union. There are also two singleton objects STop and SEmpty representing the top and bottom types in the type lattice. The class SAtomic serves to implement the hosted types, interfacing the Genus system with the class system of the JVM. Finally, the classes SEq1 (singletons), SCustom (predicates), SNot (complements) complete the list of cases required in Definition 3.2.

The class, SimpleTypeD, declares methods named typep, disjoint, inhabited, and subtypep, and each subclass overrides the methods in these protocols to implement case specific logic. The method typep returns type Boolean; however, whereas the corresponding methods in Clojure return true/false/:dont-know, these Scala methods return Option[Boolean] with Some(true) and Some(false), and None. That these methods return an Option as opposed to a Boolean alleviates the temptation to the programmer of treating the return value as a Boolean, as further discussed in Section 4.5.

We omit more details of the Scala implementation in this article, and invite the curious reader to download the Scala code.

## 4.4 Disjoint decision on JVM hosted classes.

In Section 3.1 we explained how the Clojure function isa? and the Scala method isAssignableFrom are used to determine the subtype relation for JVM classes which are the classes we are hosting in our simple, embedded type system.

We wish now to discuss how the disjoint decision is computed. Note that we are not trying to ask whether the two classes have a common superclass. The question, rather, is about the intersection of two sets of objects: all the objects of class  $c_1$  and all the objects of class  $c_2$ . We ask whether it is provable that those sets have no common elements. A simple example is that if *A* and *B* are two distinct *final* classes. Then there is no object which is a member of both classes simultaneously. I.e., the set of objects of class *A* is disjoint from the set of objects of class *B*. The question is trickier when considering interfaces, abstract classes, and other classes.

Some readers may object to the fact that we refer to Java interfaces and Java classes both as classes. In fact, within the JVM both interfaces and classes are themselves objects whose type is java.lang.Class, thus we unapologetically refer to them as classes. Various *flavors* of Java classes may be distinguished using the Java reflection API which is available both from Clojure and also Scala. The Clojure interface is a bit easier so we start there. The Clojure clojure.reflect library provides a function

type-reflect which returns a data structure (a map) with a :flags field. We use the value of this field to distinguish between four cases: :interface, :final, :abstract, and :public. The code is shown here.

(defn class-primary-flag [t]					
( <mark>let</mark> [r	(refl/type-	-reflect	(fi	ind-class t))	)]
(cond	(contains?	(:flags	r)	:interface)	:interface
	(contains?	(:flags	r)	:final)	:final
	(contains?	(:flags	r)	:abstract)	:abstract
	:else				:public)
	(defn clas (let [r (cond	<pre>(defn class-primary-4 (let [r (refl/type- (cond (contains? (contains? (contains? :else</pre>	<pre>(defn class-primary-flag [t] (let [r (refl/type-reflect (cond (contains? (:flags (contains? (:flags (contains? (:flags :else</pre>	<pre>(defn class-primary-flag [t] (let [r (refl/type-reflect (fi (cond (contains? (:flags r) (contains? (:flags r) (contains? (:flags r) :else</pre>	<pre>(defn class-primary-flag [t] (let [r (refl/type-reflect (find-class t)) (cond (contains? (:flags r) :interface) (contains? (:flags r) :final) (contains? (:flags r) :abstract) :else</pre>

In Scala Genus, we have implemented the same capability using JVM methods directly. The Java library java.lang.reflect.Modifier is available to the Scala programmer and provides methods Modifier.isFinal, and Modifier.isInterface. These methods are sufficient as the disjoint decision treats abstract and public classes identically.

Since we can detect whether a JVM class is an interface, final, or abstract/public class, the disjoint decision follows Algorithm 1.

In Algorithm 1 we check on line 1 whether either class is a subclass of the other, including whether the two classes are equal (a potentially optimizable special case). If so, every object of one type is in the other, so they are not disjoint.

The astute reader will recognize a loophole which line 1 ignores. We are implicitly supposing the types are inhabited. If a given Java class designates a vacuous type, our assumption will be violated. An empty type is disjoint from every type, including itself. More research is needed to accommodate Java classes for which they nor any subclass thereof can be instantiated.

On line 3 we ask whether either is final, encompassing two cases: exactly one is final, or both are final. If exactly one is final, then we've already determined on line 1 that the final class does not inherit from the non-final class. So in this case they are disjoint as no object can be of both classes. Second case is if both classes are final, which by definition designate disjoint sets.

On line 5 we ask whether either is an interface, knowing that neither is final. Thus we have an interface and some other non-final class. Here we chose to return *false*, because we cannot prove that no class inherits from both of these. If both classes are interfaces, then it is possible to create another class (abstract or final) which extends both of them. If one, say *I*, is an interface, and the other, say *C*, is public or abstract, then again is it possible declare a third class which inherits from *C* and implements interface *I*.

Note that the decision at line 5 to return *false* is one which determines the semantics of our system. It might happen be true that there exists no class which includes both classes in its lineage list. If this is the case, then indeed the two designated sets are in fact disjoint. However, since new classes can be loaded at run-time, and we cannot predict the future, we chose to return *false* saying that there are cases when the two sets intersect, thus are non-disjoint. The semantics of our system are currently undefined if classes are modified at runtime or if an object is mutated in a way which changes its type. See Section 5.2 for a discussion of defining these system semantics differently.

Finally, on line 7, we are in a position that neither class is final, neither is an interface, and neither is a subclass of the other; *i.e.*, they

are either abstract classes or otherwise public classes in separate hierarchies which cannot be further mixed using class inheritance. Therefore we return *true* as they designate disjoint sets.

Algorithm 1: Compute the disjoint relation between JVM classes
<b>Input:</b> $c_1, c_2$ : two JVM classes
Output: Boolean indicating whether the classes are disjoint
1 <b>if</b> $c_1 \subset c_2$ or $c_2 \subset c_1$ <b>then</b>
2 return false
3 else if $c_1$ is final or $c_2$ is final then
4 return true
5 else if $c_1$ is interface or $c_2$ is interface then
6 return false
7 else

# 4.5 Perceived Limitations of CL Type System

return true

SETS addresses some of our perceived limitations of the Common Lisp type system. In the Genus implementations of SETS we have separated the subtype predicate from the inhabited and disjoint predicates. This distinction is not strictly necessary, as also discussed in Section 4.2.1; nevertheless we found the separation simplifies some of the logic and facilitates unit testing in some cases. For some types, such as hosted types, it is straightforward to define disjointness and habitation and to define subtype in terms of those.

In Common Lisp, the return value of (subtypep (and string A) number) depends on whether A designates an inhabited type. If A is defined using satisfies, then (subtypep (and string A) number) is apt to return dont-know. Genus allows the definition of a new type to specify the behaviors of the subtype, inhabited, and disjoint predicates with respect that that new type. In portable Common Lisp, user defined types cannot be integrated into subtypep. There is simply no facility for doing so. In Genus, subtype is defined in terms of extensible code allowing applications defining a type to specify the behavior with respect to the new type.

Another problem (feature) of cl:subtypep is that it can accidentally be used as a predicate. The user may make believe that it returns true or false. But in reality such usage is prone to error as the usage conflates dont-know with false. When (subtype A B) returns false, the caller should not assume that A is not a subtype of B. Nevertheless, this mistake is exceedingly easy to make.

Scala Genus defines subtypep with return type Option[Boolean], rather than Boolean. Although this may sometimes be an annoyance, it forces in the programmers to remember that subtypep is not a 2-way predicate. Clojure Genus, admittedly, does fall prey to this potential delusion. In Clojure, subtype? takes three arguments, the third of which is the value to return in the dont-know case, and is a gentle reminder to the programmer can still accidentally use logic like: (if (subtype? A B :dont-know) ...), which will follow the true path even if :dont-know is returned, because :dont-know is considered a *truthy* value in Clojure.

A Portable, Simple, Embeddable Type System

	(1)	(2)	(3)	(4)
	Scala	Scala	Clojure	Clojure
	default	inhabited	default	inhabited
accuracy %	66.6	57.8	49.4	52.9
accuracy DNF %	81.0	92.0	61.7	83.0
gain %	18.0	35.5	20.2	33.0
loss %	3.6	1.4	7.9	2.4
m 11 4	1.			

Table 1: Accuracy results of subtypep

# 5 CONCLUSION

## 5.1 Results

We have conducted experiments to measure the accuracy of the SETS algorithm in Clojure and Scala. In both implementation host languages the test proceeds similarly. We select two type designators, td1 and td2, by constructing them at random but limiting them to a maximum depth. Next we test whether td1 is a subtype of td2. At each iteration we remember whether this test returns true, false, or dont-know. Thereafter, we canonicalize the two type designators to DNF, dnf1 and dnf2; and perform the subtype test on the new type designators. We repeat this procedure 10000 times and tally the results.

The use of randomly generated test cases without regard to likelihood is a common practice in property based testing [14].

Column 1 of Table 1 summarizes the results when run in Scala. Canonicalizing to DNF form gives an apparent 18% increase in accuracy. However, we did notice that in 3.6% of the cases, subtypep was able to determine the relation on the non-normalized forms, but was lost that ability after converting to DNF.

These results are highly dependent on the generator of randomized type designators. We observed that about 50% of the time, a type designator was generated for which it was impossible to determine its habitation, and that td1 and td2 were equivalent types 20.5% of the time. In the second test we restricted the generator to produced pairs of type designators which are inhabited and not equivalent. Results of this second test are shown in column 2.

Columns 3 and 4 show similar results for Clojure Genus. We notice that the percentages are different for Clojure as compared to Scala. The subtype decision procedure is roughly 20 to 30 percent more accurate if the type designators are canonicalized. However, there is a small set of cases (2 to 8 percent) where the subtype relation is computable before canonicalization but not after.

Recall that by *accurate* we are not referring to whether the subtype relation can be determined, but rather whether one algorithm is able to determine it while another algorithm reports *dont-know*.

To help the reader understand some reasons why the testing results differ for the two Genus implementations, we offer several comments. (1) The random type generator for the intersection and union types in the Clojure implementation, always generates binary operations, while the Scala generator generates from 1 to 5 arguments, thus the random sample of types is different. (2) The subtype? in Clojure diverges from the corresponding code in Scala because they have been developed at different times. An audit of the code is needed to verify that the two implementations have exactly the same sets of necessary and sufficient conditions. (3) The set of base types in the Scala Genus code base includes user defined classes, to model the fact that Scala programs may make heavy use of user defined types; whereas the Clojure Genus code base has no such test cases, to model the fact that user defined types are much rarer in Clojure programs.

# 5.2 Perspectives

In the next phase of our research, we will present a theory describing how symbolic finite automata [11, 49] can be used in pattern recognition of heterogeneous sequences of types belonging to SETS. In order to accept such sequences, we will consider finite automata whose transitions are labeled by SETS. This possibly infinite alphabet prevents a direct determinization process. However, the Boolean operations provided by SETS will make it possible to computate an appropriate partition that allows us to make these automata finite and deterministic. This work already exists [28] for Common Lisp, and we consider it important to generalize to a wider class of programming languages with a sound theoretical foundation.

In Section 4.3 we defined the semantics of our subtype search to allow for classes to be loaded at compile time. It would be interesting to experiment with a model which assumes a disabled class loader. Our current research involves investigating the question in the Python Genus implementation of which we can walk down the class graph, determining definitively whether two given classes have a common subclass. We have not yet investigated whether the JVM reflection API allows us to perform such a downward search for classes currently defined.

In Section 4.1 we stated that the results are undefined if the Genus user employs parameterized Java types such as Array[Int]. We would like to characterize to which extent, if any, such types could be used, or at least whether such types could be detected and explicit exceptions thrown rather than leaving the behavior undefined.

As experimental support for our symbolic finite automata theory, we will implement regular type expressions (RTEs) for Clojure and Scala, and eventually also for Python. RTEs will allow us to specify and efficiently recognize regular patterns in sequences, as was alluded to in Section 1.1. The implementation in Clojure is well underway and is available in its preliminary form.

Baker's [3] algorithm contains many assumptions about the Common Lisp type system, such as how to handle ranges, complex numbers, and CLOS objects. SETS is by definition, a simpler type system; nevertheless, it is currently unclear how to enhance the SETS specification to make it possible to implement Baker's algorithm. We consider it a matter of ongoing research.

### REFERENCES

- 1] Nada Amin. Dependent Object Types. unknown, page 134, 2016.
- [2] Ansi. American National Standard: Programming Language Common Lisp. ANSI X3.226:1994 (R1999), 1994.
- [3] Henry G. Baker. A Decision Procedure for Common Lisp's SUBTYPEP Predicate. Lisp and Symbolic Computation, 5(3):157–190, 1992. URL http://dblp.uni-trier.de/ db/journals/lisp/lisp5.html#Baker92a.
- [4] Lorenzo Bettini, Viviana Bono, Mariangiola Dezani-Ciancaglini, Paola Giannini, and Betti Venneri. Java & Lambda: a Featherweight Story. Logical Methods in Computer Science, 2018. URL http://www.di.unito.it/~dezani/papers/bbdgv18.pdf. to appear.
- [5] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for clojure, 2018.

- [6] Gilad Bracha. Generics in the java programming language. July 5, 2004. URL http://www.cs.rice.edu/~cork/312/Readings/GenericsTutorial.pdf.
- [7] G. Castagna and V. Lanvin. Gradual Typing with Union and Intersection Types. Proc. ACM Program. Lang., unknown(1, ICFP '17, Article 41), sep 2017.
- [8] Giuseppe Castagna and Alain Frisch. A Gentle Introduction to Semantic Subtyping. In Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '05, pages 198-199, New York, NY, USA, 2005. ACM. ISBN 1-59593-090-6. doi: 10.1145/1069774.1069793. URL http://doi.acm.org/10.1145/1069774.1069793.
  [9] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional Characters of
- [9] M. Coppo, M. Dezani-Ciancaglini, and B. Venneri. Functional Characters of Solvable Terms. *Mathematical Logic Quarterly*, 27(2-6):45–58, 1981. doi: 10.1002/ malq.19810270205. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/malq. 19810270205.
- [10] Mario Coppo and Mariangiola Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. Notre Dame Journal of Formal Logic, 21 (4):685-693, 1980. doi: 10.1305/ndjfl/1093883253. URL https://doi.org/10.1305/ ndjfl/1093883253.
- [11] 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/.
- [12] Sébastien Doeraene. Pseudo-union types in scala.js, August 2015. URL https: //www.scala-js.org/news/2015/08/31/announcing-scalajs-0.6.5/.
- [13] Joshua Dunfield. Elaborating Intersection and Union Types. SIGPLAN Not., 47 (9):17–28, September 2012. ISSN 0362-1340. doi: 10.1145/2398856.2364534. URL http://doi.acm.org/10.1145/2398856.2364534.
- [14] George Fink and Matt Bishop. Property-based testing: A new approach to testing for assurance. SIGSOFT Softw. Eng. Notes, 22(4):74–80, July 1997. ISSN 0163-5948. doi: 10.1145/263244.263267. URL https://doi.org/10.1145/263244.263267.
- [15] Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM, 55(4):19:1–19:64, September 2008. ISSN 0004-5411. doi: 10.1145/1391289. 1391293. URL http://doi.acm.org/10.1145/1391289.1391293.
- [16] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. The Java Language Specification, Java SE 8 Edition. Addison-Wesley Professional, 1st edition, 2014. ISBN 013390069X, 9780133900699.
- [17] Radu Grigore. Java generics are turing complete. CoRR, abs/1605.05274, 2016. URL http://arxiv.org/abs/1605.05274.
- Rich Hickey. The clojure programming language. In Proceedings of the 2008 symposium on Dynamic languages, page 1. ACM, 2008.
   Rich Hickey. A history of clojure. Proc. ACM Program. Lang., 4(HOPL), June 2020.
- [19] Kich Hickey, A history of cloude. Floc. ACM Flogram. Lang., 4(HOFL), Julie 2020. doi: 10.1145/3386321. URL https://doi.org/10.1145/3386321.
- [20] 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.
- [21] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages, HOPL III, page 12–1–12–55, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595937667. doi: 10.1145/1238844.1238856. URL https://doi.org/10.1145/1238844.1238856.
- [22] Lionel Parreaux Jim Newton, Sébastien Doeraene. Union types in scala 3, feb 2020. URL https://contributors.scala-lang.org/t/union-types-in-scala-3/4046.
- [23] Douglas Katzman. Email conversation with SBCL developer about the undocumented subtypep implementation with SBCL, February 2021.
- [24] Andrew Kennedy and Benjamin C. Pierce. On decidability of nominal subtyping with variance. In International Workshop on Foundations and Developments of Object-Oriented Languages (FOOL/WOOD), January 2007. URL https://www.microsoft.com/en-us/research/publication/on-decidability-ofnominal-subtyping-with-variance/.
- [25] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. The Java Virtual Machine Specification, Java SE 8 Edition. Addison-Wesley Professional, 1st edition, 2014. ISBN 013390590X.
- [26] David MacQueen, Gordon Plotkin, and Ravi Sethi. An Ideal Model for Recursive Polymorphic Types. In Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, POPL '84, pages 165–174, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. doi: 10.1145/800017.800528. URL http://doi.acm.org/10.1145/800017.800528.
- [27] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367199. URL https://doi.org/10.1145/367177.367199.
- [28] Jim Newton. Representing and Computing with Types in Dynamically Typed Languages. PhD thesis, Sorbonne University, November 2018.
- [29] Jim Newton, Akim Demaille, and Didier Verna. Type-Checking of Heterogeneous Sequences in Common Lisp. In *European Lisp Symposium*, Kraków, Poland, May 2016.
- [30] Jim Newton, Didier Verna, and Maximilien Colange. Programmatic Manipulation of Common Lisp Type Specifiers. In *European Lisp Symposium*, Brussels, Belgium, April 2017.

- [31] Martin Odersky. Dotty Documentation, 0.10.0-bin-SNAPSHOT, August 2018. URL http://dotty.epfl.ch/docs/reference/overview.html.
- [32] Martin Odersky and Matthias Zenger. Scalable component abstractions. volume 40, pages 41-57, 10 2005. doi: 10.1145/1103845.1094815.
- [33] Martin Ödersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.
- [34] Martin Odersky, Lex Spoon, and Bill Venners. Programming in Scala: A Comprehensive Step-by-step Guide. Artima Incorporation, USA, 1st edition, 2008. ISBN 0981531601, 9780981531601.
- [35] 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. URL http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps. Downloadable version at url.
- [36] David J. Pearce. Rewriting for sound and complete union, intersection and negation types. In Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2017, Vancouver, BC, Canada, October 23-24, 2017, pages 117–130, 2017. doi: 10.1145/3136040.3136042. URL http://doi.acm.org/10.1145/3136040.3136042.
- [37] Felipe Pezoa, Juan L. Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, page 263–273, Republic and Canton of Geneva, CHE, 2016. International World Wide Web Conferences Steering Committee. ISBN 9781450341431. doi: 10.1145/2872427.2883029. URL https: //doi.org/10.1145/2872427.2883029.
- [38] G. Pottinger. A type assignment for the strongly normalizable lambda-terms. In J. Hindley and J. Seldin, editors, To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism, pages 561–577. Academic Press, 1980.
- [39] John C. Reynolds. Design of the Programming Language Forsythe. Technical report, 1996.
- [40] Christophe Rhodes. SBCL: A Sanely-Bootstrappable Common Lisp. In Robert Hirschfeld and Kim Rose, editors, *Self-Sustaining Systems*, pages 74–86, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-89275-5.
- [41] Juan Jose Garcia Ripoll. ECL 9 release notes, may 2003. URL https://mailman. common-lisp.net/pipermail/ecl-devel/2003-May/000288.html.
- [42] Tiark Rompf and Nada Amin. Type Soundness for Dependent Object Types (DOT). SIGPLAN Not., 51(10):624–641, October 2016. ISSN 0362-1340. doi: 10.1145/3022671.2984008. URL http://doi.acm.org/10.1145/3022671.2984008.
- [43] Miles Sabin. Miles sabin via twitter. URL https://twitter.com/milessabin/status/ 953713425124818949?lang=en.
- [44] Miles Sabin. Unboxed union types in Scala via the Curry-Howard isomorphism, June 2011. URL http://milessabin.com/blog/2011/06/09/scala-union-types-curryhoward/.
- [45] Guy L. Steele, Jr. Common LISP: The Language (2nd Ed.). Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.
- [46] Leo Valais. SUBTYPEP: An Implementation of Baker's Algorithm. Technical report, EPITA/LRDE, July 2018. URL https://www.lrde.epita.fr/wiki/Publications/ valais.18.seminar.
- [47] Léo Valais, Jim Newton, and Didier Verna. Implementing baker's SUBTYPEP decision procedure. In *12th European Lisp Symposium*, Genova, Italy, April 2019.
  [48] Guido Van Rossum and Fred L Drake Jr. *Python tutorial*. Centrum voor Wiskunde
- [48] Guido Van Rossum and Fred L Drake Jr. Python tutorial. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.
- [49] Margus Veanes, Nikolaj Bjørner, and Leonardo de Moura. Symbolic automata constraint solving. In Christian G. Fermüller and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, pages 640–654, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16242-8.

# **Common Lisp Project Manager**

Eric Timmons etimmons@mit.edu CSAIL, Massachusetts Institute of Technology Cambridge, MA, USA

# ABSTRACT

In this paper we describe and demonstrate the Common Lisp Project Manager (CLPM), a new addition to the Common Lisp dependency management ecosystem. CLPM provides a superset of features provided by the Quicklisp client, the current de facto project manager, while maintaining compatibility with both ASDF and the primary Quicklisp project distribution. These new features bring the Common Lisp dependency management experience closer to what can be achieved in other programming languages without mandating a particular development workflow/environment and without sacrificing Common Lisp's trademark interactive style of development.

# **CCS CONCEPTS**

• Software and its engineering → Software configuration management and version control systems; Software libraries and repositories; Application specific development environments.

## ACM Reference Format:

Eric Timmons. 2021. Common Lisp Project Manager. In *Proceedings of the* 14th European Lisp Symposium (ELS'21). ACM, New York, NY, USA, 6 pages. https://doi.org/10.5281/zenodo.4716466

# **1 INTRODUCTION**

One way to manage the complexity of software is to reuse code by packaging up a project and using it as a dependency in another. Unfortunately, this ends up introducing another level of complexity: you have to manage your dependencies and their versions as well! In the Common Lisp world, this management is usually performed by three interacting pieces: Another System Definition Facility (ASDF) [2] to ensure all dependencies are compiled and loaded in the right order and the version of each is sufficient, the Quicklisp client [7] for installing the dependencies locally and configuring ASDF to find them, and a Quicklisp distribution containing an index that lists available projects and metadata about them.

While effective and widely used, this combination of components is missing many features that are taken for granted in other programming language specific package management ecosystems. In this paper we describe the Common Lisp Project Manager (CLPM)<sup>1</sup>, a potential replacement for the Quicklisp client in the Common

ELS'21, May 03–04 2021, Online, Everywhere

https://doi.org/10.5281/2enodo.4/16466

Lisp dependency management ecosystem that provides many of these missing features.

We begin by first defining some terminology used throughout the paper. We then provide an overview of the tasks a dependency management solution must perform. Next we highlight several of the key features that CLPM implements, along with the overarching design philosophy. We then describe the high-level design of CLPM. Last, we provide information on how to obtain CLPM and several examples on how to use it.

# 2 TERMINOLOGY

In this section, we describe some of the terminology used throughout the rest of the paper. While the broad ideas are general, we do focus on Common Lisp and particularly on projects that use ASDF.

- **Project** Primary development unit of code. Typically corresponds to the contents of a single version control system (VCS) repository, such as git.
- **Release** A snapshot of a project. Typically identified using a version number, date, or VCS commit identifier. May contain multiple systems.
- **System** Defined via ASDF's defsystem. Describes components (typically source code files) to be compiled/loaded, the version of the system, and dependencies on other systems.
- **Dependency** A tuple  $\langle f, s, v \rangle$ . f is a feature expression, s is a system name, and v is a version constraint. States that if f is satisfied when evaluated against \*features\*, then a system with the name s is required whose version satisfies v.<sup>2</sup>

**Source/Index** A collection of project and system metadata.

# **3 DEPENDENCY MANAGEMENT OVERVIEW**

Broadly speaking, there are three tasks that any dependency management solution must perform: dependencies must be resolved, installed, and built. In this section, we explain each task, describe the Common Lisp tools involved in each (when using the current de facto workflow), and then provide further grounding by briefly explaining what tools are used for each task in selected other language ecosystems.

In the building phase, every dependency must be compiled and/or loaded, in the correct order. Additionally, there should be some feedback if dependencies are not met (for instance, system A needs version 2 of system B, but only version 1 is available). In current practice, ASDF performs this function. It uses system definitions to determine what dependencies any given system. Then it finds every dependent system, checks the version constraints, produces a plan that contains an ordered set of operations to perform, and executes it.

<sup>&</sup>lt;sup>1</sup>CLPM is in the process of being renamed from the Common Lisp Package Manager. While in most communities "package" typically refers to some installable unit of software, it unfortunately collides with the use of "package" to describe symbol namespaces in Common Lisp

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

<sup>© 2021</sup> Copyright held by the owner/author(s). https://doi.org/10.5281/zenodo.4716466

 $<sup>^2\</sup>mathrm{Currently},\mathrm{ASDF}$  only supports version constraints that specify a minimum version number.

However, ASDF has no support for installing dependencies; they must be placed on the file system or in the Lisp image by another tool. While it could be done manually, this installation phase is routinely performed by the Quicklisp client. The client fetches tarballs from the internet, unpacks them to the local file system, and configures ASDF so that it can find them.

Deciding which releases need to be installed is called dependency resolution. A highly manual approach to dependency resolution is to install a single release, try building it, see what's missing, install another release, and repeat. However, the process can be sped up using an index that specifies a list of known systems and their direct dependencies. The index can then be consulted recursively for each direct dependency (and their dependencies, and their dependencies, and so on) to produce a complete list of releases that need to be installed, before even attempting to build anything. This is the approach taken by Quicklisp. The client downloads a set of files containing this index from any distributions that it is configured to consult (such as the primary distribution, also named "quicklisp", hosted at https://beta.quicklisp.org/dist/quicklisp.txt).

While we have described each of these tasks as happening in distinct stages, in reality the tasks can be jumbled together. For instance, ql:quickload attempts to resolve and install all dependencies before invoking ASDF operations. However, it's possible that the Quicklisp distribution's index is missing some information (for example, if a local project is used that has no info in the index). Therefore, ql:quickload handles conditions that ASDF signals corresponding to missing dependencies by another round of resolution and installation.

## 3.1 Comparison to Other Languages

To further ground these concepts, we briefly describe the components responsible for each of these tasks in the Python and Ruby ecosystems.

In Python [18], building is mostly performed by the Python interpreter itself. When a package is imported, it is byte compiled if needed and then loaded into the interpreter. Pip [17] is the tool predominately used for dependency installation and resolution. During resolution, pip uses metadata gathered from the Python Package Index (PyPI) [10] and a setup.py file for packages not in the index.

In Ruby [16], building is performed by a combination of the gem tool [11] and the Ruby interpreter. Gem handles building native extensions (foreign libraries) while the Ruby interpreter loads required packages much like Python's interpreter. Dependency resolution and installation is typically handled with a combination of gem and bundler [4]. Gem is largely used for installing releases locally, while bundler is used for project specific installs. Both gem and bundler can consult indices such as the one hosted at https://rubygems.org/.

# 4 FEATURES AND PHILOSOPHY

CLPM has a set of features that, while commonly found in language specific package managers for other languages, have not seen wide adoption in the Common Lisp community. The most important of these include:

(1) the ability to download over HTTPS,

- the ability to reason over dependency version constraints during resolution,
- (3) the ability to manage both global and project-local contexts,
- (4) the ability to "lock" (or "pin", "freeze", etc.) dependencies to specific versions and replicate that setup exactly on another machine,
- (5) the ability to compile to a standalone executable with a robust command line interface (CLI) for easy interfacing with shell scripts and other languages,
- (6) and the ability to install development releases directly from a project's VCS.

While CLPM is not the first to implement most of these features for Common Lisp, we believe it is the first to do so in a complete package that also places minimal constraints on the workflow of the developer using it. For example, the quicklisp-https [14] project adds HTTPS support to the Quicklisp client, but it does so at the cost of requiring Dexador [8] (and all of its dependencies, including foreign SSL libraries) are loaded into the development image.

Another example is the qlot [9] project. It adds project-local contexts (but not global), locking, a CLI, and installing directly from VCS. However, it still requires that the Quicklisp client is installed and loaded. Until very recently its executable and CLI required Roswell [15]. To date its executable is not distributed and must be built locally.

To our knowledge, no other solution exists that attempts to include version constraints during resolution. To wit, Quicklisp's index format completely elides the version constraint in its dependency lists. So no project manager that uses only Quicklisp style indices for dependency resolution can ensure that a system's version constraints are satisfied.

As just alluded, the most important guiding principle of CLPM is that it should place minimal constraints on developer workflow. An example of this principle in action is illustrated by how CLPM is distributed. For Linux systems, a static executable is provided that runs on a wide variety of distributions with no dependencies on foreign libraries (some features, such as VCS integration, require other programs, such as git, to be installed). For MacOS systems, a binary is provided that requires only OS provided foreign libraries. And on Windows, CLPM is distributed using a standalone installer and again depends only on OS provided foreign libraries.

Perhaps the second most important principle is that CLPM should be highly configurable, yet provide a set of sane and safe defaults. A concrete example of this is shown by CLPM's behavior when installing or updating projects. By default, CLPM will describe the change about to be performed and require explicit user consent before making the modification. However, this behavior can be changed for developers that like to live on the edge or otherwise have complete trust in CLPM and the projects they are installing.

# 5 DESIGN

CLPM participates in both the installation and resolution phases of dependency management. It leaves building completely up to ASDF. In this section we discuss the design of CLPM and how it completes both of these tasks. First, we describe the overall architecture, including the three main CLPM components. Second, we describe some of the benefits the architecture provides and how Common Lisp Project Manager

CLPM leverages them. Third, we describe where CLPM locates the metadata needed for dependency resolution. Last, we describe global and project-local contexts in CLPM.

# 5.1 Architecture

CLPM is split into two user facing components: the worker and the client, as well as one internal component: the groveler. The worker is a standalone executable that is responsible for all the heavy lifting. The worker interacts with the network to download releases and metadata, performs dependency resolution, unpacks archives, and manages contexts. The worker is implemented in Common Lisp and distributed both as a precompiled executable (static executable for Linux) and source code for those that want to compile it themselves. It exposes both a CLI interface and a REPL interface. The CLI interface allows for easy integration with tools such as shell scripts and continuous integration services. The REPL interface is used primarily by the CLPM client.

The client is a small system, written in portable Common Lisp, with ASDF/UIOP as its only dependency. The client is meant to be loaded into a Lisp image to facilitate the interactive management of dependencies and development of code. It does this by exposing a set of functions corresponding to the operations the worker can perform as well as integrating with ASDF's system search functions. Additionally, it has functionality to remove itself from an image if it is no longer required (such as when dumping an executable). In order to interact with the worker, the client spawns a new worker process, starts its REPL, and they communicate back and forth with S-expressions.

The last component is the groveler. Ideally users never interact directly with this component. Instead, it is used by the worker to gather the metadata needed for dependency resolution from systems that are not present in any index. For example, the groveler is used to extract dependencies from development versions of projects.

An aspect that makes extracting this metadata difficult is ASDF system definition files can contain arbitrary code and can require that certain dependencies be loaded before a system can even be correctly parsed. As such, it may not be possible to extract this metadata without running arbitrary code.

To address this, the groveler consists of a small set of portable Common Lisp functions that the worker loads into a fresh Common Lisp instance. Once loaded, the groveler and worker communicate via S-expressions, with the worker specifying which systems to load and extract information from and the groveler reporting back the information as well as any errors. If there is an error, the worker addresses it by recording any missing dependencies in the dependency resolution process, loading them into the groveler, and trying again. The worker keeps track of the systems loaded in the groveler and starts a new groveler process if needed (e.g., the groveler has v1 of system A loaded, but a dependency determined later in the resolution process requires v2).

Additionally, CLPM has experimental support to run the groveler in a sandbox. This sandbox has reduced permissions and cannot write to much of the file system. This sandbox is currently implemented using Firejail [6] on Linux, but we desire to add support for other OSes and sandbox solutions.

# 5.2 Dependencies and Non-portable Code in the Worker

One benefit of the worker/client split is that it enables the worker to freely leverage both dependencies (Common Lisp and foreign) and non-portable implementation features.

There are two benefits with respect to dependencies in the worker. First, the worker can reuse code without worrying about interfering with the code the user wishes to develop. This interference may manifest itself in many ways, including package nickname conflicts, version incompatibilities (CLPM needs version x of a system, but the user's code needs version y), or image size (e.g., the user's code uses few dependencies and they care about the final size of the image for application delivery).

Second, the worker can use foreign libraries that it is unlikely the user needs loaded for their development. A prime example of this is libssl, which CLPM uses on Linux and MacOS systems to provide HTTP support. A second-order benefit of this approach is that CLPM can statically link foreign libraries into the worker so that the user does not even need to have them installed locally (if they install a pre-compiled version of CLPM at least).

The ability for the worker to freely use dependencies has an additional knock-on effect: the development of CLPM can help improve the state of Common Lisp libraries at large. To date, the development of CLPM has resulted in at least ten issues and merge requests being reported to upstream maintainers, eight of which have been merged or otherwise addressed. Additionally, it has been the motivating factor behind several contributions to SBCL to improve musl libc support.

The worker routinely needs to perform operations that require functionality beyond the Common Lisp specification. For example: loading foreign libraries, interfacing with the OS to set file time stamps when unpacking archives, and network communication to download code. While portability libraries exist for many of these features, they are not perfect and do not necessarily extend support to all implementations [12]. The worker/client split allows us to choose a small number of target implementations and focus our testing and distribution efforts using only those implementations while not worrying about restricting what implementation the user uses to develop their code.

Currently, CLPM targets only SBCL [3] for worker support. This is due to SBCL's broad compatibility with OSes and CPU architectures. We look to extend worker support to at least one other implementation before CLPM reaches v1.0. The next implementation targeted by the worker will likely be ABCL [1] due to the ubiquity of the Java Virtual Machine.

## 5.3 Indices

Project indices are used to advertise projects and metadata about those projects, such as their released versions, what systems are available in each release, and the version number and dependencies of those systems. The most widely used project index in the Common Lisp community is the primary Quicklisp distribution. CLPM is tested against this index and has full support for interacting with it.

While there are other Quicklisp-like project indexes in the wild that work with the Quicklisp client, such as Ultralisp [5], CLPM may not work with all of them. This incompatibility is largely due to a lack of formal specification as to what constitutes a Quicklisp distribution, including what files are required and their contents. For instance, at the time of writing CLPM does not work with the Ultralisp distribution because Ultralisp does not publish a list of all its historical versions, unlike the primary Quicklisp distribution (see: https://beta.quicklisp.org/dist/quicklisp-versions.txt).

As discussed above, the Quicklisp distributions strip the version constraints from dependencies. However, CLPM actually supports reasoning over those constraints. This was the main motivation to develop and support Common Lisp Project Indices (CLPI). CLPI is part of the overarching CLPM project and seeks to fully document an index format as well as add features that CLPM can take advantage of that are missing from Quicklisp distributions. CLPI adds fields to record both a system's version (Quicklisp indices provide only the date at which the project snapshot was taken) and the version constraints of its dependencies.

A second major difference is that projects and systems are the toplevel concepts in CLPI instead of distribution version. Additionally, CLPI defines the majority of its files to be append-only. These properties allow CLPM to be more efficient in terms of network usage as only the metadata for projects potentially needed in the context are transferred and metadata on new releases can be fetched incrementally over time.

### 5.4 Contexts

CLPM manages both global and project-local contexts. A context is defined as a set of project indices in use, a set of constraints describing the projects (and their versions) that should be available in the context, and a set of project releases that satisfy both the explicit constraints and the implicit constraints added by every project in the context (i.e., transitive dependencies).

Each global context is named by a string. CLPM provides tools that generate ASDF source registry configuration files so that you can add these global contexts to your default registry and have access to all projects installed in them without the need for CLPM at all (after installation, at least).

A project-local context (also known as a *bundle*) is named by a pathname that points to a file containing the first two elements of a context (the indices and constraints). After this context is installed, all of the context information is located in a file next to the file that names the context. The names of these files are typically clpmfile and clpmfile.lock. Both of these files are designed to be checked into a project's source control and contain enough information to reproduce the context on another machine.

Both CLPM's CLI and the client provide commands to add new constraints to a context, update an entire context so that every project is at the latest release that satisfies all constraints, or update just a subset of the projects in the context. Additionally, the CLI provides commands that can execute other, arbitrary, commands in an environment where ASDF is configured to find only the projects installed in the desired context.

Global contexts are largely inspired by Python virtual environments created by the virtualenvwrapper project [13]. Project-local contexts are largely inspired by Ruby's Bundler project [4].

# 6 USE

In this section we provide some examples of how to use CLPM. We focus on project-local contexts (bundles) as we believe these are a more broadly useful feature than global contexts.

# 6.1 Installing CLPM

Tarballs (and MSI installers) of the most recent CLPM release, along with up to date documentation, can be found at https://www.clpm. dev. Windows users merely need to run the installer. Linux and MacOS users need to only unpack the tarball in an appropriate location (typically /usr/local/). If you wish to install from source, the CLPM source code (as well as its issue tracker) can be found on the Common Lisp Foundation Gitlab instance at https://gitlab. common-lisp.net/clpm/clpm.

After installing CLPM, it is recommended that you configure ASDF to find the CLPM client. To do this, simply run clpm client source-registry.d and follow the instructions printed to the screen.

Last, you may wish to consider loading the CLPM client every time you start your Lisp implementation. You can determine the currently recommended way of doing so by running clpm client rc and reading the printed instructions. The remainder of this section assumes you have CLPM installed and have a REPL where the CLPM client is loaded.

# 6.2 Configuration

CLPM reads its configuration files from the ~/.config/clpm/ folder on non-Windows OSes and %LOCALAPPDATA%\clpm\config\ on Windows. The file clpm.conf contains most of the configuration, but the defaults should be sufficient for this demo.

The file sources.conf contains a list of sources for CLPM to use when finding projects and their metadata. For this demo, it will be easiest if we populate this file with a pointer to the primary Quicklisp distribution. Place *only one* of the following forms in sources.conf.

This form uses the primary Quicklisp distribution directly. Note that we are fetching it over HTTPS. All of the primary Quicklisp distribution's files are served over both HTTPS and HTTP, even though the Quicklisp client itself can only use HTTP.

(:source "quicklisp"

:url "https://beta.quicklisp.org/dist/quicklisp.txt"
:type :quicklisp)

This form uses a mirror of the primary Quicklisp distribution. This mirror exposes the same data, formatted using the CLPI specification. Using this source will result in CLPM performing some operations faster (because it can download only the needed data). However, this source may take some time to be updated after a new version of the Quicklisp distribution is released.

```
("quicklisp"
```

:url

"https://quicklisp.common-lisp-project-index.org/"
:type :ql-clpi)

Common Lisp Project Manager

# 6.3 Downloading the Demo System

We have a simple project designed for use in CLPM demos. It is located at https://gitlab.common-lisp.net/clpm/clpm-demo. Clone it anywhere on your file system and checkout the els-21 branch.

# 6.4 Creating a Bundle

We will now create a bundle for the demo system. The simplest possible bundle is one that uses a single project index and specifies that all constraints come from .asd files. To create such a bundle you need to perform two actions. First, a clpmfile must be created. This can be done at the REPL by evaluating:

```
(clpm-client:bundle-init
```

#p"/path/to/clpm-demo/clpmfile"

:asds '("clpm-demo.asd" "clpm-demo-test.asd"))

After evaluating this, you will have a clpmfile that looks similar to the following:

```
(:api-version "0.3")
```

(:source "quicklisp"

:url "https://beta.quicklisp.org/dist/quicklisp.txt"
:type :quicklisp)

(:asd "clpm-demo.asd")

```
(:asd "clpm-demo-test.asd")
```

Notice that the first statement in the clpmfile declares the bundle API in use. This allows for the file format to evolve over time while maintaining backward compatibility. Second, notice that every directive is simply a plist. This makes it trivial for any Common Lisp project to read and manipulate the file.

After the clpmfile is created, the bundle dependencies must be resolved and installed. To do this, evaluate the following at the REPL:

(clpm-client:install

:context #p"/path/to/clpm-demo/clpmfile")

CLPM does not believe in modifying a context without explicit permission from the developer. As such, the client will by default produce a condition before it performs any modifications. This condition describes the actions that are about to be performed and has two restarts established for it: one to approve and perform the changes, and one to abort. Therefore, upon evaluating the above form, you will be dropped into the debugger to approve the changes. Of course, this behavior can be customized.

This will create a file clpmfile.lock. This file should not be modified by hand. It contains all the information necessary for CLPM to reproduce the bundle on another machine, including a complete dependency graph.

# 6.5 Activating the Bundle

Once a bundle has been installed, the next step is to activate it. The activation process configures ASDF so that it can locate all systems in the bundle. The activation process also optionally integrates the CLPM client with ASDF. This allows CLPM to notice when you try to find a system that is not present in the bundle and can provide options on adding it to the bundle. This integration is enabled by

ELS'21, May 03-04 2021, Online, Everywhere

default and discussed more next. You can activate the bundle by evaluating:

(clpm-client:activate-context

#p"/path/to/clpm-demo/clpmfile")

The CLPM CLI has an equivalent command that configures ASDF using only environment variables. This feature is particularly useful when running scripts or when running tests via CLI. For example, to run an SBCL process where ASDF is configured to use only the systems in the bundle *without* also needing to have the client loaded, simply run clpm bundle exec -- sbcl in the same directory as the clpmfile.

You can now load the demo system by evaluating:

(asdf:load-system "clpm-demo")

# 6.6 Modifying the Bundle

The most common modification made to a bundle is to add more dependencies to it. If your system acquires a new dependency you have two options on how to add it to the bundle. The first option is to explicitly reinstall the bundle using install. This will find any new dependency and install it, while trying its best to not change the installed versions of the projects already in the bundle.

The second option is to just use asdf:load-system to reload your system. If you have the client's ASDF integration enabled, the client will notice that the system is missing from the bundle and take action. The default action is to signal a condition informing the developer of the situation with several restarts in place. Following CLPM's guiding principles, however, this behavior can be modified to, for example, automatically install the dependency à la the Quicklisp client.

To see this in action, open the file clpm-demo-test.asd and add "fiveam" to the :depends-on list. Then try to run the system's test suite by evaluating (asdf:test-system "clpm-demo"). Fiveam was originally not part of the bundle so ASDF signals a condition when it tries to load it. However, we can use the reresolve-requirements-and-reload-config restart to install fiveam and then see the tests pass.

# 6.7 Using Development Versions

Now, let's say that fiveam's author has added a really cool new feature that you want to start using before it's released into Quicklisp. You can easily do this by telling CLPM to install fiveam directly from git. Simply add the following directive to your clpmfile.

(:github "fiveam"

:path "lispci/fiveam"
:branch "master")

Then evaluate (clpm-client:install). Note that you do not have to specify the context if you have already activated it. If you evaluate (asdf:test-system "clpm-demo") again, you can see

that it loads fiveam from a different path than it did previously.

## 6.8 Developing a Dependency

Last, let's say that you would like to add a new feature to fiveam and use it in clpm-demo. You should not modify any files installed by CLPM, as they may be reused between projects and the .git

folder is stripped from any dependencies installed directly from their git repo.

Instead, you should clone the project you want to work on yourself and tell CLPM to use that checkout instead of the one it installs. To demonstrate this, clone fiveam (https://github.com/lispci/fiveam) so that it is next to the clpm-demo repository. Then, create the file .clpm/bundle.conf inside the clpm-demo repository with the following contents:

```
(version "0.2")
```

```
((:bundle :local "fiveam")
```

```
'../fiveam/")
```

This tells CLPM to not install fiveam (as you have already done it) and to use the git repository located at .../fiveam/ (relative to clpm-demo's clpmfile) when resolving dependencies.

A clpm-client: hack function that performs these steps automatically is currently being developed. It is slated for inclusion in CLPM v0.5.

#### **CONCLUSION AND FUTURE WORK** 7

We have presented CLPM - the Common Lisp Project Manager. CLPM introduces many features to the Common Lisp community that are taken for granted in other programming language specific package managers. Key among these features are HTTPS support, a standalone CLI to the worker, global and project-local context management, and lock files. Additionally, CLPM adds these features without forcing a particular development practice or environment and without sacrificing "Lispy-ness" or interactive development.

We plan to continue developing CLPM and continue adding useful features. Two planned features of note are an extensible architecture for client/worker communication and the ability to add scripts from an installed project to a user's PATH. The former would enable a myriad of new configurations, including CLPM workers deployed as persistent daemons that communicate with clients over network sockets or setups that are Dockerized or otherwise isolated from other processes on the system.

In addition to improvements to CLPM itself, we aim to continue contributing back to the upstreams of our dependencies. Notably, we are in the process of interacting with the ASDF developers to add support for more expressive version numbers and dependency version constraints in defsystem forms.

Last, it is our strong preference to enable developers to use the complete power of CLPM without introducing a split in the community with regards to project indices. Therefore, we would like to take ideas and lessons learned from CLPI and integrate them into Quicklisp distributions.

## REFERENCES

- [1] Armed Bear Common Lisp. https://abcl.org/.
- [2] ASDF Another System Definition Facility. https://common-lisp.net/project/ asdf/.
- [3] Steel Bank Common Lisp. http://www.sbcl.org/.
- [4] André Arko and Engine Yard. Bundler: The best way to manage a Ruby application's gems. https://bundler.io/.
- [5] Alexander Artemenko. Ultralisp Fast Common Lisp Repository. https://ultralisp. org/ [6] Firejail Authors. Firejail Security Sandbox. https://firejail.wordpress.com/.
- [7] Zach Beane. Quicklisp. https://www.quicklisp.org/beta/.
  [8] Eitaro Fukamachi. Dexador. https://github.com/fukamachi/dexador/, .

- [9] Eitaro Fukamachi, Olot, https://github.com/fukamachi/glot/, .
- Python Software Foundation. The Python Package Index. https://pypi.org/. [10]
- Chad Fowler, Rich Kilmer, Jim Weirich, et al. Rubygems. https://github.com/ [11] rubygems/rubygems.
- Nicolas Hafner. Common Lisp Portability Library Status. https://portability.cl/. [12] Doug Hellmann, virtualenvwrapper, https://virtualenvwrapper.readthedocs.io/ [13] en/latest/
- [14] SANO Masatoshi. Quicklisp-HTTPS. https://github.com/snmsts/quicklisp-https/,
- [15] SANO Masatoshi. Roswell. https://github.com/roswell/roswell/, .
- Yukihiro Matsumoto. Ruby. https://www.ruby-lang.org/ [16]
- The pip developers. pip The Python Package Installer. https://pip.pypa.io/en/ [17] stable/
- Guido Van Rossum and Fred L. Drake. Python 3 Reference Manual. CreateSpace, [18] Scotts Valley, CA, 2009. ISBN 1441412697.

# A Corpus Processing and Analysis Pipeline for Quickref

Antoine Hacquard EPITA Research and Development Laboratory Le Kremlin-Bicêtre, France antoine.hacquard@lrde.epita.fr

# ABSTRACT

Quicklisp is a library manager working with your existing Common Lisp implementation to download and install around 2000 libraries, from a central archive. Quickref, an application itself written in Common Lisp, generates, automatically and by introspection, a technical documentation for every library in Quicklisp, and produces a website for this documentation.

In this paper, we present a corpus processing and analysis pipeline for Quickref. This pipeline consists of a set of natural language processing blocks allowing us to analyze Quicklisp libraries, based on natural language contents sources such as README files, docstrings, or symbol names. The ultimate purpose of this pipeline is the generation of a keyword index for Quickref, although other applications such as word clouds or topic analysis are also envisioned.

# **CCS CONCEPTS**

• Information systems → Information extraction; Retrieval effectiveness; Presentation of retrieval results; • Software and its engineering → Software libraries and repositories.

# **KEYWORDS**

Natural Language Processing, Indexing, Documentation

### **ACM Reference Format:**

Antoine Hacquard and Didier Verna. 2021. A Corpus Processing and Analysis Pipeline for Quickref. In *Proceedings of the 14th European Lisp Symposium (ELS'21)*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.5281/zenodo. 4714443

# **1** INTRODUCTION

Common Lisp [21] is a dialect of the Lisp family of programming languages. It was standardized in 1994 by the American National Standards Institute. It is an industrial-strength, multi-paradigm language. Languages in the Lisp family are among the very few to be homoiconic [8, 13], a property through which both introspection and intercession are achieved in a relatively homogeneous and simple way. The dynamic and highly introspective nature of Lisp makes it straightforward to extract information about the program structure and components, for example, for documentation purposes. Additionally, Common Lisp lets the programmer attach

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

ELS'21, May 03-04 2021, Online, Everywhere

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-2-9557474-5-2.

https://doi.org/10.5281/zenodo.4714443

Didier Verna EPITA Research and Development Laboratory Le Kremlin-Bicêtre, France didier@Irde.epita.fr

so-called "docstrings" (documentation strings) to almost all software components: variables, functions, classes, *etc.* When available, docstrings are a valuable source of information that can also be extracted very easily from an existing program.

# 1.1 The Paradox of Choice

In a somewhat paradoxical way, the technical strengths of the language bring drawbacks to its community of programmers [20, 24]. Lisp usually makes it so easy to "hack" things away that every Lisper ends up developing his or her own solution, inevitably leading to a *paradox of choice*. The result is a plethora of solutions for every single problem that every single programmer faces. Most of the time, these solutions work, but they are either half-baked or targeted to the author's specific needs and thus not general enough. Furthermore, it is difficult to assert their quality, and they are usually not (well) documented.

In this context, an important tool, community-wise, is Quicklisp. Quicklisp is both a central repository for Common Lisp libraries (there are currently around 2000 of them) and a programmatic interface for it. With Quicklisp, downloading, installing, compiling and loading a specific package on your machine (dependencies included) essentially becomes a one-liner. What Quicklisp doesn't solve, however, is the documentation problem.

# 1.2 Quickref

Quickref [22, 23] is a global documentation project for the Common Lisp ecosystem. It generates reference manuals for libraries available in Quicklisp automatically. Quickref is non-intrusive, in the sense that software developers do not have anything to do to get their libraries documented by the system: mere availability in Quicklisp is the only requirement.

Quickref works by introspecting libraries, and generating corresponding documentation in Texinfo format. The Texinfo files may in turn be converted into human-readable documentation, for example in PDF or HTML. Quickref may be used to create a local website documenting your current, partial, working environment, but it is also used in production, to maintain a global public website of technical reference manuals for all Quicklisp libraries. The site is kept in sync with Quicklisp.

# 1.3 Library Access

In order to provide access to the two thousand or so reference manuals available on the website, Quickref provides two indexes: a library index and an author index. The former is most likely to be used when looking for a library in particular, while the latter is probably only useful for people wanting the check out the generated documentation for their own work.

Suppose however that someone is looking for some functionality, without any prior idea or knowledge about which library may be appropriate. Quickref, as it is right now, is impractical for such a mining task, hence the idea of enriching it with a keyword index, a word cloud, *etc.* In order to generate such things automatically, it is necessary to process and analyze each library's *corpus*, that is, the bits of textual information providing some description of functionality (README files, docstrings, sometimes even symbol names, *etc.*). Fortunately for us, Declt, the reference manual generator on which Quickref is based, makes it very easy to access the corpuses in question. The purpose of this paper is to describe the natural language processing pipeline that we are currently building into Quickref to analyze the extracted corpuses, and ultimately provide library access by functionality.

Given the universal availability of very efficient internet search engines these days, one may wonder whether an indexing project specific to Quickref is really needed or pertinent. The following remarks answer that question.

First of all, a general search engine doesn't know about such or such library's availability in Quicklisp. On the other hand, a local index will necessarily point to readily-available libraries only. Next, and as opposed to search engines considering plenty of, and indiscriminate information sources, our indexing process is based on each library's documentation only. Therefore, it will have a natural tendency to favor well documented ones, which can be an important factor, when choosing which tool to use in your own project.

Finally, and beyond providing new kinds of indexes, other applications of this project could be envisioned later on, such as topic analysis, distribution, and visualization (a topography of the centers of interest in the Lisp community, of sorts).

# 1.4 Pipeline Overview

Figure 1 depicts the pipeline used to process and analyze the corpuses extracted from each library by Declt.

- (1) Each corpus is first *tokenized*, that is, split into chunks which usually (but not necessarily) correspond to words. The tokens are then *tagged*, meaning that they are associated with their syntactical class (noun, verb, *etc.*). After this stage, we are able to filter specific token classes (*e.g.* retain only nouns, verbs, *etc.*).
- (2) Next, the retained tokens are *stemmed*, meaning that their lexical root is extracted, and used to attempt matching with a canonical form found in a dictionary. This process is called *lemmatization*. After this stage, only the canonicalized known lemmas (*i.e.*, found in said dictionary), are retained.
- (3) A TF-IDF (Term Frequency / Inverse Document Frequency) value is computed for every such lemma. This value is a statistical indication of how relevant each lemma is to the corresponding library. Only the most pertinent ones are kept around (the exact number of such retained lemmas may vary).
- (4) Finally, the (possibly intersecting) sets of most pertinent keywords describing each library are aggregated in order to produce the desired output (keyword index, word cloud, *etc.*).

It is worth mentioning right away that in this pipeline, two out of four blocks (the first two) are *pre-processing* steps, devoted to sanitizing the corpuses, while only stages three and four actually perform the job of information processing. The importance of pre-processing in this pipeline is due to TF-IDF working on syntactic tokens only, without any semantic information. For example, without pre-processing, tokens such as "test", "tests", and "testing" would be treated independently, as if they meant different things.

At the time of this writing, the first three blocks in this pipeline are fully operational. Keyword aggregation, on the other hand, is a difficult problem, and the aggregator block is still subject to experimentation. Also, note that we intend, at a later time, to release the code of each block as independent, open-source libraries.

The remainder of this paper is organized as follows. Sections 2 to 5 provide a more in-depth description and discussion of the tokenizer / PoS-Tagger, stemmer / lemmatizer, and TF-IDF blocks respectively. Section 6 describes the challenges posed by the keyword index generation problem, the experiments already conducted, and some possible ideas for further experimentation.

# 2 POS-TAGGING

PoS-Tagging (for "Part-of-Speech" tagging) is a technique allowing to determine the syntactic class of words, that is, whether they are common nouns, verbs, articles, *etc.* The syntactic classes of words may be important information to perform semantic analysis of a corpus for different reasons. For example, some categories of words, like determinants, convey very little or no useful meaning at all, so we want to filter them out early, rather than carrying them around until the TF-IDF block makes the same decision (although for a different reason: they appear frequently, but everywhere). Also, in the aim of generating a keyword index, it may be interesting to experiment with different sets of retained information, such as only nouns, nouns and verbs, *etc.* 

### 2.1 Implementation

There are many ways to implement a PoS-Tagger, notably with HMMs (Hidden Markov Models), unsupervised learning, or machine learning [9]. In the Common Lisp ecosystem, we are aware of one PoS-Tagger library, namely "Tagger" [5], written by Xerox in 1990, which uses HMMs.

HMMs are statistical Markov Models used to learn an *unknown* Markov Process with hidden states, by observing another process, known this time, and depending on it. HMMs are widely used in PoS-Tagging to disambiguate syntactic classification. The biggest problem of PoS-Tagging is that a word can have several syntactic classes associated with it, depending on the context. For example, the word "can" may be either a verb, or a noun (as in "soda can"). Using HMMs, a PoS-Tagger first learns the probability of a certain sequence of syntactic classes occurring. Then, it disambiguates unknown words by using the syntactic class sequence with the highest probability.

Suppose for example that after an article such as "the", the class probabilities for the next word are 40% noun, 30% adjective, and 20% number. When seeing "The can", a PoS-Tagger will thus correctly classify "can" as a noun.

A Corpus Processing and Analysis Pipeline for Quickref

ELS'21, May 03-04 2021, Online, Everywhere



Figure 1: Keyword index generation pipeline

Because HMM-based PoS-Tagging needs a word's surrounding context in order to decide on its syntactical class, it must appear very early in the pipeline, namely, before such contextual information is removed. On the other hand, the input of the tagger needs to have been tokenized already. Therefore, the tokenization and tagging steps are tightly coupled, which is why they appear as a single block in our pipeline.

The aforementioned Tagger library happens to offer a powerful and highly customizable tokenizer, and a PoS-Tagger linked with it. The tokenizer uses an automaton to parse sentences, and can be customized with rational expressions. Our two biggest customizations on the tokenizer were to accept dashes in tokens rather than considering them as separators (otherwise, words like "commandline" would have been split), and to add a rational expression to recognize URLs as unique tokens (many URLs exist in our corpuses, even in plain text documentation).

The Tagger library poses a problem however: it accepts ASCII characters only. In the majority of the cases, this is not so much of a problem because the natural language in use is almost exclusively English, and a purely ASCII text encoded in Unicode remains readable as-is. On the other hand, some libraries do have special characters in their README files, breaking the tagger. A good example of this, is the April [18] library, which compiles a subset of the APL language. Many special characters in there are APL tokens, which are not ASCII.

In order to solve this problem, we pre-process all of our documents with "Free Recode", an open source tool to transliterate files between many encodings. Non-ASCII characters are replaced with interrogation marks. This side-effect actually has little or no impact on our pipeline at all, because our current corpus contains only English documentation. Most of it is already plain ASCII, and the few non-ASCII characters we found were either fancy "prettification" of README files (*e.g.* smileys), or in code snippets.

## 2.2 Tests and Results

In order to get an early feedback on the behavior of this preprocessing stage, we ran our complete pipeline in different PoS-Tagger modes. Sample results are presented in Table 1. That table displays the 10 keywords appearing the most frequently after the TF-IDF block (all libraries included). The numbers in parentheses are the number of libraries associated with each keyword. In all cases, the Tagger library's tokenizer is used. When tagging is turned off, there is no filtering on the syntactic classes of the tokens. Otherwise, the table presents results when only common nouns, or a combination of common nouns and verbs are retained.

All tokens	Only nouns	Nouns and Verbs		
lisp (110)	library (175)	test (110)		
test (63)	file (150)	file (108)		
message (51)	function (138)	license (107)		
common-lisp (51)	license (133)	library (107)		
name (49)	value (117)	function (92)		
file (48)	document (117)	name (81)		
value (47)	package (114)	value (81)		
stream (46)	name (114)	package (78)		
function (46)	test (102)	stream (69)		
server (45)	project (101)	load (69)		
Table 1. Tan 10 harmanda an/ different armta stie filtere				

Table 1: Top 10 keywords w/ different syntactic filters

We observe that even without tagging, we don't see "noisy" words such as articles appearing in the top 10 list. That is because at the end of the pipeline, the TF-IDF pass will detect that such words, being frequent basically everywhere, are in fact not specific to any library in particular. On the other hand, the PoS-Tagger would help filtering those words earlier in the pipeline. It is also apparent that PoS-Tagger helps filtering out uninteresting tokens such as "lisp" or "common-lisp". Indeed, these end up being filtered out as either proper nouns (as in "Lisp is a ..."), or adjectives (as in "a Lisp library").

Whether to keep verbs around, or only common nouns, remains an open question. Verbs may contain useful information for describing what a library does. For example, it is likely that a library for unit testing will make frequent use of the word "test" both as a noun, and as a verb. If we keep both around, the final weight of "test" as a unique lemma will increase (which is a good thing in that particular case, and is in fact visible in Table 1). This will also happen every time a verb and a noun are slightly different, but are lemmatized identically. On the other hand, many verbs are also uninteresting ("be", "get", "come", etc.), and it is impossible to know in advance whether their distribution across all libraries would be such that the TF-IDF block would filter them out. Finally, there are also problematic cases in which a verb and a noun convey different meanings, which would hinder the accuracy of our results. One possible solution to this problem would be to tag nouns and verbs in order to keep them as separate entities, but as mentioned before, there are also cases where keeping them separate is undesirable.

# 3 STEMMING

Stemming is the process of reducing a word to its root, or canonical form in the linguistic sense, notably by removing prefixes or suffixes.

No stemmer	Porter	Snowball	Lancaster
node	node	node	nam
server	elem	elem	parsable
test	src	src	node
stream	parse	server	src
template	stream	parse	byte
event	server	stream	stream
trivial	trivial	see	trivia
x	byte	trivial	el
connection	test	test	x
image	х	byte	test

Table 2: Stemmer-dependent results for the final index

Although stemming does not constitute a block in our pipeline *per se*, it still is an important part of the process, for reasons that will become apparent in the next section.

Because a stemmer removes everything but the linguistic root of a word, the resulting "stem" may not be a complete word at all. This is a potential problem for us, because in the end, we want an index composed of actually existing words, so the stems themselves can't always be used directly.

## 3.1 Implementation

Many stemming algorithms exist, and they are usually quick and straightforward to implement. The two most popular approaches are based, either on rule systems, or on training of stochastic algorithms [7]. The rule-based approach offers a better trade-off between simplicity of implementation and quality of the produced stems, so this is the approach we favor.

Figure 2 illustrates a typical use-case of a rule-based stemmer. There are usually two categories of rules: transformation rules and deletions rules. A transformation rule transforms a prefix (respectively, a suffix) into a simpler version. A deletion rule deletes the prefix (respectively, the suffix).

Three notable suffix stemmers exist in the literature: Porter [15], Snowball [16] (*a.k.a.* Porter 2), and Lancaster [14]. These stemmers are well suited to process English, as most of the word variations occur at their end in this language. We implemented the three of them in Common Lisp, and we used NLTK [3] as a reference point for debugging our implementations. NLTK is the most well known, and *de facto* standard Python library for NLP (Natural Language Processing), and incorporates a large number of stemmers. Note that we are aware of only one pre-existing Common Lisp implementation of a stemmer [6], a Porter stemmer, more specifically. We still decided to write our own because it is rather straightforward, and also because the NLTK implementation, which we want to follow, sometimes departs from the original specification in ways that would have been difficult to implement in the existing Common Lisp implementation, which is not very flexible.

# 3.2 Tests and Results

In order to get an early feedback on the behavior of stemming, we ran our complete pipeline which each of them, and also without stemming at all, that is, using the output of the PoS-Tagger directly. Sample results are presented in Table 2. We notice a global improvement of the final index when stemming is used. Indeed, interesting words (such as "parse") are brought up, while less interesting ones (such as "x") are brought down. We also notice that the results with the Lancaster stemmer are not so good: many final words are in fact not actual words. This is due to the fact that Lancaster is a "strong" stemmer: it has a tendency to over-stem words, which leads to the same root for words and typos. The Snowball stemmer is considered to give the best results, as it is the only one which manages to bring down "x" to *not* be in the first ten words.

# 4 LEMMATIZATION

Besides stemming, the other classical approach to word normalization in the literature is lemmatization, which consists in using the dictionary form of a word as its canonical representation (instead of a stem). The main advantage of this approach is that in the aim of building a word index, the output of a lemmatizer can be used directly, as opposed to that of a stemmer which requires reconstructing a word afterwards.

# 4.1 Implementation

Lemmatization can be implemented in many different ways. Approaches range from rule-based systems (similar to stemmers, but with more complicated rules), dictionary look-up, machine-learning, etc. As our bibliographical research didn't reveal anything satisfactory in terms of Lisp implementation of a lemmatizer (either not in Quicklisp or part of a larger library), we decided to implement our own. The approach we chose is that of dictionary look-up, as described in [10]; a solution both elegant and easy to implement. In short, a word is compared with all words in a dictionary of "lemmas", and the closest one (according to a so-called "edit distance") is chosen as its canonical form. A pre-processing step consisting of stemming the word before measuring its edit distance is discussed in the paper, and shown to give better results (hence the importance of stemming anyway). The Common Lisp library mk-string-metrics offers a set of built-in edit distances. We use this library to implement our lemmatizer.

We conducted a set of experiments in order to decide on the best combination of stemming algorithms (among the 3 described in the previous section), edit distances (we choose to only test the 5 available in mk-string-metrics but there are plenty of others in the literature [12], [4], [2], [19]), and dictionaries. The following sections report on those experiments.

# 4.2 Stemmer / Edit Distance Selection

In order to decide on which stemmer algorithm and which edit distance to use, we tested the possible combinations and counted the number of correct lemmas generated by the lemmatizer. The ground truth (*i.e.* the correct lemmas for each word) was simply found on the internet, where a lot of resources related to lemmatization exist for verifying the correctness of an implementation[1].

Table 3 shows the obtained results. These results confirm one of the paper's claims, which is that the use of a stemmer has a huge impact on the quality of the results. The other noticeable thing is that the Lancaster stemmer performs quite poorly. This, again, can be explained by the fact that Lancaster tends to produce very
A Corpus Processing and Analysis Pipeline for Quickref



Figure 2: Ruled-based stemming

	Distance				
Stemmer	Jaccard	Jaro-Winkler	Damerau-Levenshtein	Levenshtein	Overlap
None	428	659	655	656	25
Porter	840	934	970	970	132
Snowball	842	934	970	970	132
Lancaster	483	612	590	591	32

Table 3: Number of correct lemmas on a list of 1226 words

16.000 lemmas	30.000 lemmas	33.249 lemmas
clause (54)	clause (52)	library (167)
client (40)	server (38)	file (145)
server (39)	client (35)	function (134)
project (37)	hotel (34)	license (124)
value (37)	project (34)	document (113)
message (35)	value (33)	value (110)
node (34)	node (31)	name (105)
user (33)	message (29)	package (103)
test (30)	begin (29)	test (100)
begin (30)	aside (27)	project (96)

Table 4: Dictionary-dependent results for the final index

short stems, which skews the edit distance computation. Finally, we can see that the best results are obtained with Porter or Snowball stemmers, and with Levenshtein or Damereau-Levenshtein edit distances. As Snowball is an improvement over Porter, and Damereau-Levenshtein over Levenshtein, it is only logical that these four have approximately the same results. At that point, we decided to retain the Snowball stemmer, as it also performed more efficiently, time-wise, and the Levenshtein distance, because it is slightly faster than the Damereau-Levenshtein one.

#### 4.3 Dictionary Selection

For dictionary selection, we started by experimenting with two dictionaries of 16.000 and 30.000 lemmas respectively, found on the internet (unfortunately, we lost track of the source of these dictionaries in the process, but we *will* publish them later, along with the code).

Sample results are presented in Table 4 (the third dictionary / column will be described in a few paragraphs). The indexes generated with those dictionaries have a big flaw: they contain words that are not in the base corpus. The most obvious example of this is

the word "hotel" occurring at the fifth position in the second index. These words appear somewhat "magically" for a conjunction of two events: they exist in the dictionary but not in the corpus, and we're trying to lemmatize a word that is (obviously) in the corpus, but not in the dictionary. Because every word in the corpus *needs* to be matched to a word in the dictionary, such words will be lemmatized weirdly.

More specifically, lemmas are chosen by optimizing the edit distance (minimizing or maximizing it, depending on the actual distance in use), which is a *continuous* measure. This means that while a lemma will *always* be found, the edit distance may still be bad. In other words, there are times when even the best solution is a bad one.

4.3.1 White-Listing. A natural solution to this problem is to use the dictionary as some sort of "white-list", by imposing a threshold on the computed edit distance. Whenever a normalized edit distance is found to be lower than the selected threshold, the confidence in the lemmatization process is considered too low, and the word discarded from the subsequent TF-IDF statistic. Using such a threshold is a convenient way to make a distinction between words which do have a lemma in the dictionary, and words which don't (hence, words which we don't want to keep around). After some experimentation (mostly, looking at the results), we decided that a threshold of 0.8 is appropriate for making a decision.

4.3.2 *Custom Dictionary.* An even better solution to this problem would be to make sure that the dictionary we use does not contain words absent from our corpuses in the first place. This leads to the idea of generating a *custom* dictionary, from the lemmatization of the words present in our corpuses directly. Of course, creating such a dictionary leads to a bootstrapping problem, as lemmatizing our corpus would require using the dictionary we are trying to create. Thus, we need an external lemmatizer.

Here, again, we used the one from NLTK (in fact, we also tried the lemmatizer from the Stanford NLP library [11], written in Scala

No lemmatization	Lemmatization
library (96)	library (175)
file (82)	file (150)
function (79)	function (138)
data (74)	license (133)
value (71)	value (117)
license (70)	document (117)
documentation (67)	package (114)
test (65)	name (114)
name (64)	test (102)
body (57)	project (101)

Table 5: Indexes obtained with and without lemmatization

out of curiosity). The generated dictionary contains approximately 33.500 lemmas. Note that in theory, we should rebuild it every time Quicklisp is updated. Whether this is a critical issue remains to be seen however. Indeed, Quicklisp is already quite large, so the probability that an update induces a very important change in the corpus is likely to be low. On the other hand, an outdated custom dictionary may start to miss words, or contain irrelevant ones again, so it is still important to continue using the aforementioned threshold-based white-listing step.

Finally, note that with this custom dictionary, we are certain to only get lemmas existing in our corpuses, but we are not completely sure that the "technical jargon", frequent in our community's specialized version of English is fully recognized by NLTK. It is difficult to evaluate the risk of an unknown technical word being weirdly lemmatized by NLTK, but we're hopeful that if it happens at all, it remains marginal. NLTK uses the Wordnet database, which is very large, and also encodes relations between words (such as singular/plural, synonyms, etc.). A possible path to get more insight into this problem could be to evaluate the behavior of NLTK on the Common Lisp Hyperspec's glossary (which is likely to be a quite complete reference for technical jargon), and maybe adjust the reference dictionary accordingly. Another one would be to properly recognize code pieces from markup information (see Section 8). Finally, it would be highly beneficial to keep even non-Lisp jargon around. Pseudo-words such as "cmdline", acronyms such as "GUI", etc., behave just like regular words in our communities, and should probably be treated as such. How to collect them into our custom dictionary is yet another problem.

The third column in Table 4 shows the top 10 keywords obtained with this custom dictionary, and confirms that the results are better. For example, irrelevant words such as "begin" or "aside" are gone, even though our dictionary contains more lemmas in total than the two other ones.

#### 4.4 Final tests and results

In order to get an early feedback on the behavior of this preprocessing stage, we ran our complete pipeline with and without lemmatization. Recall that without lemmatization, it is the output of the PoS-Tagger which is processed by the TF-IDF block directly. Sample results are presented in Figure 5. The question of whether lemmatization is useful, and under which precise conditions remains open. In general, lemmatization is expected to be useful because it allows to treat variations on a single keyword together. On the other hand, a lemmatized keyword may not be the most informative, and we believe that this is exactly what happens with "documentation" and "document" in Figure 5. Assuming that documentation libraries (such as Quickref and Declt) are those which bring the keyword "documentation" up, it is unfortunate that in the lemmatized case, this keyword is transformed into "document" which, in fact, is less informative. This problem suggests that using an ad-hoc, carefully tuned dictionary may turn out to be important.

#### 5 TF-IDF

Even though most of the delicate work actually happens during the pre-processing phase, the heart of our pipeline consists in extracting meaningful words from our corpus. By "meaningful", we mean words which convey the most relevant and decisive information. For this task, we use the TF-IDF statistic [17].

TF-IDF is a measure that aims at reflecting the importance of a word in a document. It uses two parameters to operate: the frequency of the word in the document and the number of documents containing this word. The main idea behind this approach is that a word both frequent in a document and frequent in all documents is not very specific to the document in question, and thus, is not a good descriptor for this document. On the other hand, a word that is very frequent in one document, but which appears nowhere else, brings a great amount of information on the document in question, and can thus be used as a keyword representing it.

In the Quickref context, a "document" corresponds to the corpus of text extracted by Declt from one specific library. TF-IDF is run on each library, for which the best *x* words are retained, *x* being an adjustable parameter.

#### 5.1 Tests and Results

An important question, before running a TF-IDF on each library's corpus, is to decide on what we actually use as a corpus for each library. As mentioned before, README files and docstrings are a natural choice, but we can also think of using symbol names (of functions, variables, *etc.*) as the code is also usually explicit about what it does. We could also use ASDF's system descriptions, when provided, but we haven't tried that yet. More specifically, we ran our pipeline on the following corpus variations.

- (1) README files only.
- (2) README files, plus docstrings for all exported functionality (public API).
- (3) The above, plus the symbol names for all exported functionality. The rationale is that carefully chosen API names may be indicative of the library's purpose.
- (4) The above, plus docstrings for the library's internals (so, essentially all docstrings available).
- (5) The above, plus the library's internals symbol names (so, essentially all symbols).

Sample results are presented in Table 6. As more or less expected, it is probably not a good idea to add the documentation of a library's internals in the corpuses, as the text found there most probably deals more with the implementation of the library's functionality, than the functionality itself. This is visible, for example with the appearance of keywords such as "string", "vector", or "class" in the A Corpus Processing and Analysis Pipeline for Quickref

	README	+ Docstrings	+ Symbols
blic API	library (175)	file (148)	stream (123)
	file (150)	string (142)	file (117)
	function (138)	value (139)	value (105)
	license (133)	stream (132)	string (105)
	value (117)	object (125)	name (105)
	document (117)	name (118)	user (105)
Pu	package (114)	license (109)	object (95)
	name (114)	function (108)	test (90)
	test (102)	type (106)	type (90)
	project (101)	test (104)	error (88)
	library (175)	string (145)	string (140)
	file (150)	stream (138)	file (136)
	function (138)	file (137)	stream (133)
+ Internals	license (133)	value (118)	object (110)
	value (117)	object (114)	value (109)
	document (117)	name (102)	vector (101)
	package (114)	class (100)	class (96)
	name (114)	test (97)	test (94)
	test (102)	license (96)	function (92)
	project (101)	function (95)	message (91)

Table 6: Results for different corpus variations

top 10, which are likely to be related to typing information known statically, and advertised as such.

Even when restricting ourselves to the public API's corpus, including docstrings and / or symbol names doesn't seem to add much to the pertinence of the results. Even public docstrings are in fact likely to contain static typing information (such as "string"), function parameters descriptions (such as "object"), *etc.* In fact, we have ultimately no control whatsoever on the type, quality, or quantity of documentation (if any) provided by the developers, which makes keyword extraction a very hard problem.

# **6** AGGREGATION

An even harder problem is how to aggregate an appropriate selection of keywords coming from different libraries (probably with some overlap), into a sufficiently descriptive and pertinent index. The difficulty here comes from the fact that we would like 100% library coverage (we want every Quicklisp library to be pointed to by at least one keyword) but we also want a reasonably sized final index. How to achieve this goal is still mostly unanswered, but we have already conducted some experiments, reported below, and we also have some ideas that yet remain to be tested.

# 6.1 Histogram-Based Selection

The first approach we have experimented with is based on the crosslibrary keyword appearance histogram. For each of the retained x keywords from every library, we count the number of libraries it appears in, and we sort them by decreasing frequency. We then select the minimum number of keywords required to reach a 100% coverage. This process is very simple to implement, and as a side-effect, can also be the base for generating a word cloud. Indeed, if a keyword is representative of many libraries, it probably means that the corresponding topic is subject to a lot of activity. On the other hand, this approach also poses accuracy problems, and makes it hard to adjust the relevant parameters properly (this is where a choice on the value of *x* becomes crucial). More specifically, because we want every library to be indexed, a trade off is to be made between the number of keywords retained per library (hence, accuracy), and the size of the final index.

If, for example, we keep only one keyword per library, this keyword will indeed be very descriptive of that particular library, and so is less likely to apply to many of them. Consequently, it is very probable that the final index will be very large (at worst, one different keyword for every single library, that is, approximately 2.000).

If, on the other hand, we keep a large number of keywords for every library, there is more likelihood that the retained keywords will overlap from one library to another, letting us reach a 100% coverage faster. However, we also risk retaining keywords that are not so relevant.

Figure 3 contains plots of the library coverage (in percentage) as a function of the final index size, for different values of x, that is, when retaining different numbers of keywords per library. These plots confirm what intuition tells. When x = 50 for example, a 100% library coverage is reached with a final index of 200 keywords, but those keywords are likely to *not* be so specific. When x = 5, on the other hand, the final index will require more than 3.000 words, all probably quite relevant.

As mentioned before, because of the inherent structure of the histogram we use, the top 1 keyword will have many libraries associated with it, the next one slightly fewer, and so on. This is important, and problematic, for two reasons.

- When a user searches a library for a specific use, a keyword leading to a hundred different choices is likely to be of little help.
- (2) The number of libraries associated with a keyword is not the same for all keywords, which makes the final index somewhat heterogeneous.

This is why we also plan to investigate other approaches.

#### 6.2 Other Potential Solutions

A first alternative approach could be to sort the output of TF-IDF not by decreasing frequency, but by a pertinence factor of some sort (doing in some sense a meta-TF-IDF on top of the original one), and keep enough of the top ones to reach a 100% coverage. The pertinence factor in question could be the inverse of average ranking of a keyword in each library's top list, a normalized sum of all TF-IDF values, or any other measure yet to be thought of.

Yet another possibility would be to take a completely opposite approach, and start from the fact that in order to be usable, a keyword shouldn't point to more than, say, n = 10 libraries. We could then arrange to select all such keywords until we reach a 100% coverage (probably adjusting *n* to get a reasonably sized index in the process).

ELS'21, May 03-04 2021, Online, Everywhere

Antoine Hacquard and Didier Verna



Figure 3: Library coverage vs. final index size, for different values of x

#### 7 CONCLUSION

In this paper, we presented a natural language processing pipeline for Quickref, allowing us to analyze corpus extracted from Quicklisp libraries docstrings, README files, or symbol names. This pipeline is relatively lightweight, as it amounts to no more than 2000 lines of code.

As part of this process, we have used an existing PoS-Tagging library, and we have developed our own native Common Lisp implementations of stemming and lemmatization algorithms. As of this writing, the code is not in production yet, but we plan to cleanup, package, and release our stemmers and lemmatizer as standalone libraries in short term.

The complete pipeline, including the histogram-based aggregation approach, is currently integrated in an experimental version of Quickref, but other solutions remain to be tested before putting the whole thing in production.

#### 8 PERSPECTIVES

Apart from the aggregation problem, some other plans for future work are worth mentioning.

Our pipeline is currently unaware of any markup used in README files notably (Markdown, HTML, *etc.*). A number of specific tweaks are in place in order to remove markup tags from the corpus (for example, by recognizing and filtering URLs out during the tokenization phase). Also, in the case of frequently used markup formats (such as Markdown), the syntactic "noise" produced by the tags is likely to be filtered out as non-pertinent by the TF-IDF block, precisely because of its frequency in many libraries. Yet, it would be better to be aware of the markup formats in use, and use that information during the tagging process. The first advantage that comes to mind is to be able to properly differentiate natural language parts from code samples, in order to select what we want to keep around for TF-IDF (see Section 5.1). Correctly identifying markup tags could also be useful to spot inline code excerpts (or just words) embedded in natural language paragraphs, and give them special treatment (for instance, considering them as "technical jargon"; see Section 4.3.2). More generally, it could be interesting to think about the kind of information that other tags, such as bold or italics, provide. For instance, bold or italics may be an incentive to give more weight to the targeted textual part. Even more generally, potentially useful information can sometimes be extracted from pure, tagless, text. For example, it is customary to render Lisp references (function parameters, variables, *etc.*) in uppercase in plain docstrings.

Previously, we mentioned that we use a suffix stemmer because that is where most of the variations occur in English. We could not find any prefix stemmer in the literature, and we currently don't know if that would be worth looking for, even for English, and perhaps in combination with the current suffix one.

As far as dictionaries are concerned, we mentioned that the best results were obtained by creating our own custom dictionary with the help of an external lemmatizer. Another possibility would be to start from an existing dictionary, but keep track of missing words, and create only a custom *addition* to the original dictionary with those words. Even if we gain a little in terms of dictionary bootstrapping time, it is not very likely that this solution would get us anything more in terms of pertinence, notably because existing dictionaries are still likely to contain a lot of words that are uninteresting for us, or that actually never occur in our corpus.

Finally, one final question that could arise eventually is that of the actual language in use. Currently, we assume English (which is unlikely to pose any problem with Quicklisp), but if we even want to handle other languages, the problem will become more complicated. In particular, our current PoS-Tagger will not be usable anymore.

#### REFERENCES

- Lemma lists. https://lexically.net/wordsmith/support/lemma\_lists.html. Accessed: 2021-03-06.
- [2] L. Bergroth, H. Hakonen, and T. Raita. A survey of longest common subsequence algorithms. In Proceedings of the Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000, pages 39–48, September 2000.

#### A Corpus Processing and Analysis Pipeline for Quickref

ELS'21, May 03-04 2021, Online, Everywhere

doi: 10.1109/SPIRE.2000.878178.

- [3] Steven Bird, Edward Loper, and Ewan Klein. Natural language processing with python. https://www.nltk.org/book/, 2009.
- [4] William W. Cohen, Pradeep Ravikumar, and Stephen E. Fienberg. A comparison of string distance metrics for name-matching tasks. In Proceedings of the 2003 International Conference on Information Integration on the Web, IIWEB'03, pages 73-78. AAAI Press, 2003. doi: 10.5555/3104278.3104293.
- [5] Doug Cutting and Jan Pedersen. The Xerox part-of-speech tagger version 1.2. https://github.com/g000001/tagger, 1993. [6] Steven M. Haflich. The Porter stemming algorithm, a Common Lisp implementa-
- tion. https://github.com/varjagg/porter-stemmer, 2002.
- [7] Anjali Jivani. A comparative study of stemming algorithms. International Journal on Computer Technology and Applications, 2:1930-1938, November 2011.
- Alan C. Kay. The Reactive Engine. PhD thesis, University of Hamburg, 1969. Deepika Kumawat and Vinesh Jain. Pos tagging approaches: A comparison. [9]
- International Journal of Computer Applications, 118(6):32-38, May 2015. ISSN 09758887. doi: 10.5120/20752-3148.
- [10] Dimitrios P. Lyras, Kyriakos N. Sgarbas, and Nikolaos D. Fakotakis. Using the levenshtein edit distance for automatic lemmatization: A case study for modern greek and english. In 19th IEEE International Conference on Tools with Artificial Intelligence(ICTAI 2007), volume 2, page 428-435, Oct 2007. doi: 10.1109/ICTAI. 2007.41.
- [11] Christopher Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, page 55-60. Association for Computational Lin-guistics, 2014. doi: 10.3115/v1/P14-5010.
- [12] Andrew McCallum, Kedar Bellare, and Fernando Pereira. A conditional random field for discriminatively-trained finite-state string edit distance. In Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence. UAI2005,

Jul 2012.

- [13] M. Douglas McIlroy. Macro instruction extensions of compiler languages. Communications of the ACM, 3:214-220, April 1960. ISSN 0001-0782. doi: 10.1145/367177.367223.
- [14] Chris D. Paice. Another stemmer. ACM SIGIR Forum, 24(3):56-61, Nov 1990. ISSN 0163-5840. doi: 10.1145/101306.101310. [15] M. F. Porter. An algorithm for suffix stripping. Program: Electronic Library and
- Information Systems, 14(3):130-137, 1980. doi: 10.1108/eb046814.
- [16]
- M. F. Porter and Richard Boulton. Snowball stemmer. 2001. Gerard Salton and Christopher Buckley. Term-weighting approaches in automatic [17] text retrieval. Information Processing and Management, 24(5):513-523, Jan 1988. ISSN 03064573. doi: 10.1016/0306-4573(88)90021-0.
- [18] Andrew Sengul. April: Array programming re-imagined in lisp. https://github. com/phantomics/april, 2019.
- Syeda ShabnamHasan, Fareal Ahmed, and Rosina Surovi Khan. Approximate of Computer Applications, 120(8):26–31, Jun 2015. ISSN 09758887. doi: 10.5120/ 21247-4048.
- [20] Mark Tarver. The bipolar Lisp programmer. http://marktarver.com/bipolar.html, 2007.
- [21] ANSI. American National Standard: Programming Language Common Lisp. ANSI X3.226:1994 (R1999), 1994. [22] Didier Verna. Quickref: Common Lisp reference documentation as a stress test
- for Texinfo. In Barbara Beeton and Karl Berry, editors, *TUGboat*, volume 40, pages 119–125. TEX Users Group, September 2019.
- Didier Verna. Parallelizing Quickref. In 12th European Lisp Symposium, pages 89-96, Genova, Italy, April 2019. ISBN 9782955747438. doi: 10.5281/zenodo.2632534. [23] Rudolf Winestock. The Lisp curse. http://winestockwebdesign.com/Essays/Lisp\_
- Curse.html, April 2011.

# Lisp in the middle

using Lisp to manage a Linux system

Michael Raskin\* raskin@mccme.ru,raskin@in.tum.de Technical University of Munich Garching bei München

# ABSTRACT

In the Lisp community one can still find some nostalgia for the time of Lisp machines. The defining feature that has been since lost is having a powerful programming language as the main method of controlling the system behaviour.

Unfortunately, to the best of our knowledge, there are few modern systems that try to revive this approach. Moreover, regardless of the configuration language in use, managing the system as a whole is usually associated purely with managing a global persistent state, possibly with parts of it getting enabled or disabled in runtime.

We present a system design and a description of a partial implementation of Lisp-in-the-middle, a system based on the common GNU/Linux/X11 stack that uses Common Lisp for runtime system policy and per-user policy. We prioritise ease of achieving compatibility with niche workflows, low rate of purely maintenance changes, and minimising the unnecessary interactions between the parts of the system unless requested by user.

#### CCS CONCEPTS

• Software and its engineering → Operating systems; Application specific development environments;

#### **KEYWORDS**

operating systems, Linux, Lisp machines

#### ACM Reference Format:

Michael Raskin. 2021. Lisp in the middle: using Lisp to manage a Linux system. In *Proceedings of the 14th European Lisp Symposium (ELS'21)*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.5281/zenodo.4724166

#### **1** INTRODUCTION

Since the time when Lisp machines were in use, a variety of software has emerged that allows to use some flavour of Lisp to control some subset of the computer environment.

Among these projects, Emacs [1] probably enjoys the widest adoption. While many people use it mainly for editing text, there are plugins that add functionality ranging from email client to a window manager. Other projects used as parts of their main environment by some people are StumpWM [2] (window manager

\*The author is supported by an ERC Advanced Grant (787367: PaVeS)

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

ELS'21, May 03-04 2021, Online, Everywhere

© 2021 Copyright held by the owner/author(s).

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.

https://doi.org/10.5281/zenodo.4724166

for X11) and Guix/Guix-SD [3] with Shepherd [4] (package manager based on Guile Scheme, a service management tool based on Guile, and a GNU/Linux distribution based on them).

There are interesting projects implementing entire bare-hardware OS in Lisp, such as Mezzano [5], Movitz [6] and LOSAK [7]. To the best of our knowledge, all of them were (and Mezzano currently is) developed inside virtual machines.

We experiment with using Common Lisp for access policies and glue code in the context of runtime system management. Unlike many modern tools for system state management and system policy management, we consider the case of a user with programming experience and explicit preferences about the details of the workflows and access control. Thus we do not consider the simplicity of a language used to be important, but the expressive power of a language is an important advantage. As the user has specific preferences related to the workflows, interactions working without the need to enable them explicitly are not as important as the reduction of unexpected changes caused by the updates. Additionally, tools to explicitly restrict unwanted interactions can be useful.

We aim to provide an environment close to the range of exotic GNU/Linux distributions with some access to the Lisp-implemented system functionality to shell-based workflows.

The Lisp code in the system generally implements the answers to the two main questions. What things should happen together, and what access control is necessary for privileged operations.

For example, when opening a laptop at home, one might want multiple things to happen. After a WiFi connection is established, the instant messaging client should reconnect and set the status. If the password manager has been locked, it should ask for the master password. If email fetching is paused for some reason, it should be enabled again. Screen brightness that matches the environment should be set. At the same time, screen brightness change and WiFi reconfiguration are operations requiring root access, so there should be some policy describing under which condition these changes are allowed to happen. Both parts are usually managed via a mix of manual actions, shell scripts, and service configuration settings in various languages.

Another example requiring configurable combination of system management operations with a subset of them being privileged is launching an application in a restricted and partially isolated environment. This is desirable both from the point of view of reducing the average impact of vulnerabilities in unreasonably complex applications such as office suites and modern graphical web browsers, and from the point of view of restricting unintended interactions between simultaneously running applications or the runs of the same application.

For example, a user might want to launch a Firefox instance that only has network access via the university proxy, no access to most of the user files, and read/write access to a specified directory with article PDF files.

A large part of the impulse for experimentation was getting tired of keeping track of changed systemd default settings that needed to be reconfigured back to the previous default values or worked around. Note that such motivations raises importance of some *negative* requirements, i.e. a tool not doing undesired things is no less important than the tool having the desired features.

In the current state the Lisp-in-the-middle system replaces some configuration and tools with Lisp code and keeps the general structure of a GNU/Linux environment. As it is intended to adapt to niche workflows, it does not impose much of a global structure; it is more a collection of tools that turned out to be convenient for specific tasks. We believe that experiments with alternative system structures or alternative interaction approaches require either a massive upfront investment, or something that can be almost immediately used as a day-to-day environment and gradually extended from the inside. While the former approach might allow a much better eventual outcome in the case of success, we follow the latter one to reduce the risk. We hope that future expansion of the Lisp layer in the Lisp-in-the-middle system will lead to accumulation of tools and experience useful both for building a non-standard system on top of the standard low-level parts of GNU/Linux software stack, and for providing isolated and controllable instances of software packages required for interoperability. Examples of hard-to-replace packages are web browsers and office file format editors.

Centralised management of the system by interoperating daemons is what systemd does after expanding its scope from being an init system and daemon supervisor. Unlike systemd, we aim to provide configuration by defining or replacing the functions that make policy decisions. We also want to allow multiple replaceable daemons to manage different parts of the system without tight coupling between the daemons.

Currently the system exists as a small set of libraries and tools, some in Common Lisp, some in a mix of Nix/Shell/C, and an example system definition using these libraries and tools.

# 2 AN EXAMPLE INTERACTION: REDUCING POWER CONSUMPTION

We proceed to describe a small example of the events triggered when a user requests to reconfigure the system in the runtime. The user has unplugged the laptop from an AC power supply and plans to use the laptop without AC power supply for a significant amount time. Thus the user wishes to reduce the power consumption of the system. To achieve that, the user calls the disconnect function in an unprivileged Common Lisp image. This might be done via a REPL or via a command-line parameter to a newly started image. The disconnect function is defined in a way similar to the following.

```
(defun disconnect
```

```
(&key kill-ssh kill-wifi kill-bg (brightness 1)
      (cpu-frequency "min") kill-matrixcli
      standby-options standby kill-mounts)
(when kill-mounts
```

```
(loop for d in (directory (~ "mnt/*/")) do
```

```
(& fusermount -u (namestring d))))
(alexandria:write-string-into-file "10"
(format nil "~a/.watchperiod" (uiop:getenv "HOME"))
 :if-exists :supersede)
(! web-stream-updater-starter quit)
(uiop:run-program "rm ~/.update-web-streams-*"
 :ignore-error-status t)
(ask-with-auth
 (:presence t)
  (list
     (set-cpu-frequency , cpu-frequency)
     (set-brightness ,brightness)
     ,@(when kill-wifi `((kill-wifi "wlan0"))))
(when kill-ssh
(ignore-errors (stumpwm-eval `(close-ssh-windows)))
(! pkill "-HUP" ssh-fwd) (! pkill "-HUP" -f /ssh-fwd))
(when kill-bg (kill-background-process-leaks))
(when kill-matrixcli
 (! pkill -f /matrixcli) (! pkill -f " matrixcli"))
(! x-options)
```

(when standby (apply 'standby standby-options)))

Depending on the current needs, it might stop unnecessary background processes (SSH sessions, stuck Firefox instances in the background X session, IM clients, retrieval of web feeds), unmount network filesystems, change the refresh rate of the status bar, reduce CPU frequency and screen brightness, turn off the WiFi interface, reconfigure the X session in case an external monitor has been disconnected, and suspend the laptop to RAM. Some of these operations are performed using previously written shell scripts, some are implemented inside this function, some are called from another part of the configuration.

Some parts of the disconnect functions require privileged operations. In particular, we will consider the part corresponding to the change in the CPU frequency and the screen brightness. As the function is executed inside a non-privileged process, it asks the system management daemon to perform the privileged operations.

#### 2.1 Inter-process requests

If the disconnect function is called with the default arguments, the following form gets executed.

```
(ask-with-auth
 (:presence t)
  `(list
    (set-cpu-frequency "min")
    (set-brightness 1)))
```

The ask-with-auth macro expands to the code that connects to the system management daemon socket to send the request, possibly wrapping it into some authentication or authorisation exchange. In this section we describe how the requests are represented and sent.

The main Common Lisp system management daemon creates a listening socket with an address on the file system to receive requests. The requests are s-expressions, containing strings and numeric literals. Symbols (except NIL) are deliberately not allowed for two reasons. First, there are differences in symbol treatment between Lisps (mainly related to representing packages or modules

M. Raskin

Lisp in the middle

when serialising a symbol), while lists, strings and most numbers are represented in the same way. The identical representation across the Lisp language family should make migration (as well as using multiple policy daemons in different languages for managing different parts of the system) easier. Second, one of the obvious use cases for allowing some symbols in the requests is passing some part of the request as keyword parameters to a function. We want to discourage such an approach, because policies defined in such a way are likely to allow more than intended. Utility functions for personal use are likely to gain additional keyword parameters over time, and expanding their interface should not require complete review of all the possible requests. As a simple example, a function that converts files to a different format might eventually gain an extra keyword argument to remove original files in case of success. In that case, the right to pass it arbitrary keyword arguments might become more sensitive.

There is a special package for request handlers. The request must be a list; the first element must be a string, which is looked up (after converting to the upper case) in the package to find the handler that will be executed with the rest of data as parameters. Each connection also has a hashtable of extra data. This extra data can be used, for example, in the process of authentication of requests. The extra data can be freely read and modified by the request handlers.

For example, a request ("SET-CPU-FREQUENCY" "min") will lead to a call

(socket-command-server-commands::set-cpu-frequency context "min")

where context might contain, for example, information whether this is a part of a larger request that has been authorised by the physically present user.

For convenience, the ask-with-auth macro allows the user to pass symbols inside the requests. Each symbol is replaced with a string representing its name. Hence calling the disconnect function leads to the following request content after replacing the symbols with their names.

#### ("LIST"

("SET-CPU-FREQUENCY" "min")
("SET-BRIGHTNESS" 1))

However, we do not allow these operations to be requested by an arbitrary process without confirmation from the user, thus the conversations with the server is more complicated.

#### 2.2 Authentication and authorisation

Before executing a request the daemon usually needs to check if the policy allows such a request. The policies are defined by Lisp code executed by the daemon. The management daemon provides a few ways to either verify the client's claims about its identity or to ask the physically present user whether the command should be permitted.

The simplest and fully automatic way of authentication is verifying that the client has the claimed user ID. This is done by writing a random token into a file readable only by the target UID and asking the client process to provide the token. A simple way of authorising a request with slightly larger impact is to ask the physically present user to confirm that the request should be executed. Consider for example the ask-with-auth invocation inside the disconnect function. This macro always performs UID authentication, and additional authorisation by the physically present user is requested in this case. First the client process sends a request to confirm having access to the files only readable as user raskin.

("REQUEST-UID-AUTH" "raskin")

The server replies with the filename containing the key. The client reads the key and uses it to send an authenticated request, which requires additional authorisation via confirmation from the physically present user. Here the timeout for the user reaction is 15 seconds. The request demands to change two settings: CPU frequency and screen brightness.

```
("WITH-UID-AUTH" "JDPBTP56Y3LALB6FMA54"
("WITH-PRESENCE-AUTH" "T"
("PROGN" ("SET-CPU-FREQUENCY" "min")
("SET-BRIGHTNESS" 1))
```

15))

The daemon switches to a specially reserved virtual terminal to separate the communication between the daemon and the user from the programs in the normal user session. After a switch to a dedicated virtual terminal, the user is presented with the list of operations to confirm or cancel. If the user confirms the request, the daemon adjusts CPU frequency and screen brightness using sysfs virtual filesystem provided by the Linux kernel for interaction with drivers. Afterwards the daemon reports the success or failure to the client process.

As screen resolutions vary in a wide range, but font size configuration in plain Linux kernel-rendered virtual terminal using the fbcon driver is limited, we use a reduced interface to graphics on virtual terminals (framebuffer) provided by the Linux kernel with fbterm, a framebuffer terminal emulator with vector font support.

## 2.3 Integration with existing scripts

Sometimes an existing script in another programming language is modified to use management daemon requests instead of, for example, sudo.

We mainly use a saved Lisp image with all the relevant library code loaded and allow the scripts to just execute it passing the Lisp code for performing a request. A shell script can then include something similar to the following.

# 3 OBTAINING INFORMATION FROM THE WEB

Web browsing is performed in a tiered manner: in the best case, plain HTML is downloaded, then it is parsed using

cl-html5-parser [12], and then the parsed data is converted to a plain text representation using our Thoughtful Theridion library [13]. The download might be automated using Throughful Theridion's web page walker DSL. If the page requires Javascript to fetch the content, a background Firefox instance with Marionette remote control is started to obtain the data, save the HTML and also the corresponding text representation. In the unfortunate case where an interactive web browser is required, an isolated (typically per

site) instance of Firefox (or Chromium for sites failing to work in Firefox) is used.

The latter two cases include starting a browser in an isolated environment. This happens with the assistance of the system management daemon. As an example, evaluating the following code in an unprivileged Lisp instance will start a virtual X session, launch Firefox with full network access and limited file system access, open the ELS page, select the current year, wait for the second page to load and return the URL of the image.

```
(uiop:launch-program "xdummy :9")
(with-new-firefox-marionette
  () (() :netns nil :display 9)
  (prog1 (progn
      (marionette-set-url
        "https://european-lisp-symposium.org/")
      (ask-marionette-parenscript
         (ps:chain document document-element
           (query-selector "a.current") (click))
        :wait-ready t)
      (first
        (ask-marionette-parenscript
         (return (ps:chain
             document document-element
             (query-selector "span.imagery img")
             src)))))
      (marionette-close)))
```

While the substance of the task is the code passed to the calls to ask-marionette-parenscript, we are interested in the request to the daemon that happens due to the with-new-firefox-marionette macro.

The main part of the expansion is a call to the firefox function that launches Firefox in an isolated environment with limited file system access and, if requested, network access restricted to accessing the selected proxy. The function also tells Firefox to enable the Marionette external control framework. This framework has been initially implemented for automated testing of Firefox itself. Marionette is also used as a foundation for Geckodriver, a tool for automated testing of websites in Firefox. The instance can be used for interactive browsing or for automated interaction with some websites.

To run Firefox (or any other command) in an isolated environment, the client process sends a request to the system management daemon. We do not show a complete request here because it has too many parameters. The request contains the configuration for the files that should be available inside, which devices such as sound cards and video cameras should be made accessible to the application, what network access should be provided, etc.

#### 3.1 Sandboxing

The request to run some command in an isolated environment allows a client process to claim a fresh user ID and run some code under this UID for partial access control. It is also possible to request isolation using some of the technologies developed for Linux containers. For example, many vulnerabilities in the desktop programs when opening a malformed document have significantly less impact if the corresponding program has neither network access nor write access to anything outside the current directory. Using a unique UID also allows controlling the scope of daemons like

PulseAudio that insist to be run only once in a user session and are required by some software. This functionality permits controlling the interaction between the non-Lisp components of the system. The user has access to launching programs in isolated environ-

ments via a normal Lisp function, similar in purpose to ui op : run-program but with numerous keyword arguments to describe the desired environment and functionality available to the program.

#### 3.2 Window management integration

If the firefox function was called with the :stumpwm-tags keyword argument, all the windows created by this Firefox instance will be automatically assigned the specified window tags by StumpWM. These tags can be used to manipulate the windows as desired. The implementation is as follows.

GUI applications launched in isolated environments can be distinguished, for example, by the internal host name of the environment. We have expanded StumpWM support of the X11 protocol window properties and provide functions to assign window tags based on window properties. In particular, the Inter-Client Communication Conventions Manual standard provides access to the hostname of the client application and the Extended Window Manager Hints standard provides access to the process ID and hostname of the client application via X11 window properties.

On the StumpWM side we add a socket with an address on the file system. Before launching the isolated application, we evaluate code in StumpWM that establishes correspondence between the chosen hostname of the isolated environment and the desired tags.

# 4 UPDATING THE SYSTEM

An important operation on most systems is updating the system itself. When a user updates the system, there are two parts of the task: a new persistent state of the system needs to be constructed, and a subset of runtime state needs to be reinitialised based on the new persistent state. The user initialises the system update by calling full-refresh function in an unprivileged Lisp instance.

For the first part we use Nix package manager [8]. First we call Nix from the unprivileged client process to prepare the new persistent state. As Nix installs each package, from glibc to the overall system state, into its own path, asking the Nix daemon (provided by Nix) to construct the new persistent state is non-destructive and can be done without special privileges. Then the client process asks the daemon to set the newly created system state as the active system state. This is a privileged operation; unlike more frequent operations like adjusting the screen brightness here the user is asked to enter the system root password for confirmation instead of just pressing enter. This operation essentially just changes the target of one symbolic link on the filesystem.

To update the runtime state of the daemon, the client process sends the request to the daemon to exit; a new instance of the daemon is automatically started whenever the old one exits. Any data that needs to survive such restarts is stored in external SQLite databases, mostly on RAM-backed filesystems. Such external storage of the runtime daemon state reduces the risks in case of errors in policy code leading to unhandled conditions in the daemon, as

#### Lisp in the middle

well as makes it easier to share the runtime state with a different daemon. In addition to the fully functional Common Lisp system daemon, updating the system is also supported a proof-of-concept Guile system daemon.

Some of the other daemons running in the system can also be restarted after the update by normal requests to the daemon.

# 5 LISP IMPLEMENTATION LIMITATIONS AND WORKAROUNDS

Common Lisp implementations (including SBCL), as well as portability libraries such as uiop, often expect that spawning other programs will be done in a relatively limited way. There are two kinds of cases where we currently choose to use a workaround.

The first case is starting programs that need access to the terminal input. The most annoying situation is the case of a long running program that we might want to interrupt. Unix shells intended to spawn such a case have special code for handling interruptions while a program is running in the foreground. However, if the program is launched from an SBCL REPL using sb-ext:run-program or uiop:run-program, pressing Ctrl-c leads to a different, less desirable behaviour. Both the program and the Lisp session interpret Ctrl-c normally; the program exists as desired, and the REPL starts a debugger caused by an interactive interrupt. Similarly, it is also possible to break some expectations regarding the input/output stream (for example, Ctrl-d can be caught both by the program invoked and the Common Lisp implementation).

A simple way to avoid this problem is to make sure the spawned program has its own separate pseudo-terminal. Using a terminal multiplexer like screen and running interactive programs in a new screen window is usually an acceptable workaround. We provide a macro !! which interprets its argument in the same as the ! macro does, but asks screen to start the command in a new window instead of starting it directly. Of course, an obvious way to avoid that is to run workflows in a traditional shell, spawning a Lisp instance when necessary.

The second case is using special system functionality, for example file locks. For Common Lisp implementations it is more natural to implement concurrency as multiple threads in a shared address space. However, this means that system APIs intended for use in a multi-process model are not fully usable. We use existing wrapper programs for such functionality or write small helper executables in C. For example, if the use of POSIX flock function for file locking is desired, we spawn a separate process that acquires a lock, reports success when the lock is acquired, then waits for the command to release the lock.

## **6 OVERALL STRUCTURE OF THE SYSTEM**

The initial boot is performed in a way pretty similar to other GNU/Linux distributions. The customisation of this stage is done via a shell script. Once the system runs with the normal storage properly set up, a minimalistic init process sinit is launched for the core system tasks (releasing the process memory structures of the processes that terminate after their corresponding parent processes has terminated, and handling the system shutdown), and the main Lisp process is started.

The Lisp process starts the login processes on the virtual terminals, and launches the daemons (such as CUPS printing daemon and OpenSSH) according to the policy. Afterwards the system Lisp daemon starts listening on a Unix domain socket on the file system for requests. The requests are used for operations requiring special access in a way similar to the way sudo or doas are used. Using a Unix domain socket for communication instead of e.g. a local HTTP server allows us to send file descriptors. Passing file descriptors is used in the context of isolated environments.

A large part of the scripts for launching the daemons and of the daemon configuration files is generated using the code from NixOS GNU/Linux distribution.

We do not currently use any service supervision system, mostly because restricting the global interactions leads to a limited set of system-wide services with failure modes that are invisible to a service supervision system anyway.

A small library is provided for writing both the system-wide policy code and the user-level policy code. We will now describe the provided functionality.

# 7 OTHER RELATED FUNCTIONALITY

### 7.1 Authentication and authorisation

Request authorisation by the physically present user supports some extra features.

The client can provide extra text to explain to the user why the request has been sent. Sometimes authorising the request will require the user to enter a password, either their own, or belonging to a specified other user on the system (usually the root user). The password request and entry will happen in the same reserved virtual terminal to prevent abuse of input interception in the X11 protocol. The same mechanism can be used by a client process to request an application-specific password from the user without the password going through the normal input channels of the current session.

#### 7.2 Isolated environments

Our protocol for requests supports forwarding some status messages to the client as they happen without waiting for the entire request to finish. For example, when starting a command in an isolated environment, it is possible to have the standard output (or the exit code of the program) as a return value, but forward the error messages as they appear. This is based on sending the file descriptors through the socket and thus requires the socket to be a Unix domain socket as opposed to a TCP socket. The same functionality also allows to run a command-line program in an isolated environment but provide it with the input from the current pseudo-terminal.

While the client process has a lot of freedom in configuring what files are accessible inside the sandbox, it may be sensible to impose some limitations. For example the following policy says that an empty tmpfs (temporary memory-based filesystem) can be mounted wherever the requesting process wishes, but real files and directories can either be provided at their true location, or inside /tmp/ or /home/, or at a location without any well-known meaning. This restricts the possibilities for requesting access to an installed

setuid program inside the sandbox and confusing the program with fake entries inside /etc/.

```
(defun nsjail-mount-allowed-p (from to type)
  (or
     (equal type "T") (equal from to)
     (and
        (or
            (alexandria:starts-with-subseq "/home/" from))
            (alexandria:starts-with-subseq "/tmp/" from))
        (or
            (alexandria:starts-with-subseq "/home/" to)
            (alexandria:starts-with-subseq "/tmp/" to)
            (alexandria:starts-with-subseq "/tmp/" to)
            (not (find
            (second (cl-ppcre:split "/" to))
            *well-known-directories* :test 'equalp))))))
```

On the current system the risks of abusing setuid binaries are further reduced by having only 4 setuid binaries. There is some separate support for generating safe contents for specific files inside /etc/ such as /etc/passwd (but not /etc/shadow).

## 7.3 Miscellaneous Lisp utilities

For many system administration tasks, the simplest way to perform them on GNU/Linux involves invoking utility programs and parsing their output. We provide some tools to simplify manipulating shell commands, and tool-specific wrappers for some commands.

We provide a function to parse the network address list returned by the ip network configuration tool, as well as a few trivial wrappers for the most common network configuration modifications, including WiFi state control via wpa\_supplicant. We currently assume that the configuration of wpa\_supplicant is managed separately.

We wrap some of the basic Nix package management operations, basic user account manipulation operations, power management etc.

In the spirit of the standard with-open-file macro and multiple with-. . . macros provided by Lisp libraries, we implement shell-with-mounted-devices macro that attaches a file system or multiple file systems on removable devices, launches a specified shell command (by default it opens a new shell window in the current screen session) and detaches the file systems when the command finishes. Many graphical environments based on GNU/Linux offer graphical shortcuts to attach a file system on the USB drive and to launch a file manager or a terminal for that file system, but we are not aware of environments that provide an option to tie automatic disconnection of a file system with the lifetime of the corresponding terminal or file manager.

Of course, the functionality provided for lifetime tracking of a non-blocking command can be used for implementing other related functionality, such as deactivating the screen saver for the lifetime of a video player when the video player doesn't implement screen saver interaction on its own. From the low-level point of view, the lifetime tracking of the child process is based on letting it inherit an open file descriptor; in most cases the file descriptor will be inherited by any descendant processes, which we consider a desirable behaviour.

#### 7.4 Tracking process windows with StumpWM

We also provide the automatic-tagging functionality for unsandboxed applications, however this only works if the initially started process is the one creating windows. This restriction is absent when isolation is used for launching the application, because we can assign a unique hostname to the container and the application will not change it. Note that the hostname and process ID are reported by the application who can deliberately violate the protocol, but most applications use some GUI libraries that reveal the data correctly.

# 8 COMPARISON TO OTHER APPROACHES

# 8.1 Similarities and differences with Guix-SD goals

While Guix and Guix-SD also use a Lisp (namely, Guile Scheme) to configure the system, our goals are different. Guix and Guix-SD are intended to provide Lisp-based management and Lisp APIs for constructing the system. GNU Shepherd manages the system in the runtime, but it is intended for managing only the set of running daemons, and for use via command-line. It also provides APIs that mirror the command line invocations.

In our case, the goal is to experiment with controlling the runtime state of a system and with providing Lisp APIs for runtime fine-tuning as opposed to building a preconfigured system once. An example of a difference is access to privileged commands. Guix-SD uses the standard sudo mechanism and allows to specify the contents of the /etc/sudoers configuration file. Our system implements a custom protocol for requesting privileged operations, and the access policy is configured by defining Lisp functions.

Guix could have been used instead of Nix for package management; the choice of Nix here was motivated by package availability.

#### 8.2 Request authentication and authorisation

A custom protocol for checking the UID of the other process is providing functionality pretty similar to the getsockopt system call with SO\_PEERCRED option. However, we verify the current user ID of the client. In contrast, getsockopt checks the identity of the process that has initially established the connection; this could be, for example, the parent process of the current one that did not close the socket before spawning the current process. The behaviour we use is more similar to the behaviour achieved when using sudo. (As an aside, iolib does not have full support for SO\_PEERCRED.)

Our protocol allows authorising multiple operations as a batch. In this case the user will need to take an action only once, but full information about all the actions will still be provided. This differs from the typical situation with sudo where either each operation requires separate authorisation, or authenticating a session leads to all requests in the next few minutes are granted.

In many cases, the physical presence of a user is relevant: for example, a user with physical access can shutdown the system using the power button, so physical presence can be enough to initiate safe shutdown; but the same user logged in via SSH might not want to be able to shutdown the system too easily. Our solution seems to be more robust than the usual ones when a user has both SSH and physical login session and also has some programs running in a pseudo-terminal accessed from both of the sessions Lisp in the middle

(e.g. using screen). In such a setup the standard policy mechanisms of checking whether a user has any physical session, or whether the current program is a descendant of physical login do not answer the question whether the request comes from a physically present user.

### 8.3 Isolated environments

The goals of the system include running software in isolated environments. Linux namespaces support permits to construct some isolated environments even without administrator access. It is still desirable to assign single-use user IDs to different isolated environments as an extra layer of protection. However, for things like network isolation it does not matter whether the namespace has been constructed as root or as user. This allows us to run a relay for limited network access in an environment that is almost fully isolated save for the networking, then create a nested sandbox with access to the relay socket but not the host networking (figure 1).

Thus it would be convenient to have a tool with nested isolation support. This makes service-oriented isolation options, including Shepherd service sandboxing, less attractive. An intermediate option would be something like systemd-nspawn, but it requires that the entire system is managed by systemd.

Standalone options range from pretty limited unshare to fully featured options such as Firejail, nsjail, Bubblewrap, and others.

We use nsjail [9] tool for the low-level work necessary for creation of isolated environments, as it supports many various options and seems stable. For example, nsjail supports presenting specified parts of filesystem inside the isolated environment, choosing whether to use or not each kind of isolation supported by the Linux kernel, resource limits, user and group ID handling, etc. As the invocations of nsjail are generated by the Common Lisp code, flexibility and feature availability are more important than convenience of manual invocations of availability of predefined isolation profiles.

#### 8.4 Integration with shell scripts

Of course, it is possible to reimplement the authentication protocol in the language of the script. On the other hand, maintaining implementations of the protocol in multiple programming languages increases the cost of updating the protocol.

Another approach is to have a separate management daemon running with the user's permissions, and use file system access control to allow requests to such a daemon only from the client programs with the same user ID. The user daemon will execute the requests. It can have access to both the user Lisp scripts not requiring special permissions and to sending a request to the system management daemon.

A benefit of the chosen approach with starting a saved Lisp image is the possibility to see the parent process ID. Another benefit, which is only relevant for scripts running in a terminal, is that the Lisp image providing the communication with the system management daemon can write to the terminal. A possible drawback is a higher memory use, although most of the data in a saved Lisp image is mapped read-only from the image on the disk and can be shared between multiple process.

## 8.5 Package management

Most system-wide package managers such as dpkg and rpm typically contain some notion of package conflicts, leading to annoying interactions between the parts that we would prefer to keep separated and not interacting.

To have a solid foundation for managing the runtime state, we choose a tool for managing the packages that minimises unwanted interactions such as package conflicts, and ensure a requested state regardless of the previous state. If we want these properties to apply to the base system, the most natural options are Nix [8] and GNU Guix [3]. Both the Nix package manager and the Guix package manager inspired by Nix restrict the notion of package conflicts by installing each package into its own directory. That way, installation of a package is not seen as a change in the globally shared mutable system state; instead, the installed packages are treated as a garbage-collected pool of immutable data structures. Updating the system with this approach is closer to constructing a new list out of cons cells (reusing some of list elements), than to updating the elements in-place in a sequence.

Both package managers have distributions based on them, NixOS and GuixSD; in this case the entire system is defined as a package with some dependencies, allowing to keep multiple versions of the entire system and choose between them on boot without unnecessary duplication of the installed packages. This allows both to install individual packages and to reuse parts of the system (e.g. configuration files and service definitions) without running the entire distribution.

Of course some persistent data such as PostgreSQL databases needs to be converted to newer formats during some upgrades, so the abstraction is not perfect. However, the cost and the risk of a typical update of a system with some exotic packages installed are lowered significantly.

We use the Nix [8] package manager. While Guix, which has started as a rewrite of Nix into Guile, seems to be a natural choice for a package manager in a Lisp OS, we currently use Nix [8] because of a larger packaging community. In particular, Nixpkgs have better Common Lisp package coverage than GuixPkgs. However, we use only some parts of NixOS, because NixOS uses systemd.

#### 8.6 Communication with StumpWM

There are alternative ways to send commands to StumpWM. One of them relies on setting the window properties of the root window. Unfortunately, this approach has no access control and cannot handle concurrent commands. The generic SWANK [14] debugging interface is often used for a similar purpose. However, it uses a local network port that also can be accessed by all the applications granted unconstrained network access.

#### 8.7 Choice of implementation language

The overall design of the system and the supporting tools do not require specifically Common Lisp or a Lisp-family language. However, many languages in the Lisp family possess all the strong sides we consider required for comfortable use in such a setup, such as REPL and macro support.

While scsh [10] would be a natural choice for migrating Shell workflows into a REPL with a more powerful language, Common



r 5 isolation

Figure 1: Providing limited network access inside isolated environment

Lisp library availability provides an advantage for some parts of the system, as libraries such as SQLite3 FFI are readily available. Another language option could be Julia [11], which has macro support and REPL, as well as solid FFI support and a very active ecosystem. However, a higher level of activity comes with a drawback as breaking changes happen more often. A tangible benefit of the Common Lisp ecosystem is the tendency to expand the tools more often than to change them with an impact on existing use.

In any case, we want to avoid a system design that would lock in too many decisions. We want to make gradual migration between multiple different system management daemons reasonably easy, including the case when they are written in different languages (maybe even outside the Lisp family). In particular, we want to allow running two system management daemons simultaneously and making requests to both of them. Moreover, it would be nice to allow the daemons to have overlapping areas of control.

Note the system also has some small utility programs implemented in C specifically to be invoked from Common Lisp code. Such programs are useful when it is desirable to invoke low-level system functionality from a separate process, not just a separate thread, for example due to thread-safety concerns.

#### **9 FUTURE DIRECTIONS**

There are many things that need to be wrapped in a Lisp API; too many to attempt listing them.

We will introduce support for more authentication checks, such as a protocol for confirming the process ID of the program that makes a request and looking up the corresponding executable. On the one hand, this requires adding support for getsockopt with SO\_PEERCRED to some FFI library; on the other hand, the race conditions need to be evaluated to avoid the case where a program sends a request and immediately uses exec to replace itself with a different executable. This can be useful to provide some reliable context.

Implementing service supervision could be useful for some cases.

A modern computing environment often consists of multiple interacting devices. Our model already includes multiple communicating agents without complete trust between them. It would be interesting to try managing a multi-device environment using interacting policy daemons. An interface to prevent a denial of service by a misbehaving program issuing too many requests requiring user confirmation is needed.

Some alternative interfaces based on JSON or XML could be provided for easier integration with client code in different languages.

Providing an option to use lightweight VMs instead of containers, probably reusing some of the SpectrumOS [15] work, would expand the options for isolation.

Two daemons could be used to make it easier to fix broken updates without reboot and without logging in as root.

## 10 CONCLUSIONS AND LESSONS LEARNED

Using a system management daemon running Lisp policy code is a simple way for gradually increasing the part of the environment managed in runtime via Lisp, reducing the amount of necessary upfront work. Unlike existing solutions, authorisation policies for privileged system reconfiguration actions are written in a high-level general-purpose programming language (in our case, Common Lisp). Another atypical feature naturally arising from our use of an extensible protocol is the possibility to authorise multiple related actions via a single user interaction without loss of transparency. Tools developed for isolation and testing can be reused by such a system for reducing unwanted interaction between the non-Lisp system parts. We think such an approach will be an integral part of a realistic project to regain some of the benefits of Lisp OS on the modern hardware without losing the ability to interact with the full range of tools and formats considered portable.

In general we observe that the optimistic expectations about the effort of making a system managed by Lisp code the day-to-day system were not too far from truth.

We observe that implementing the system-level and user-level policies in Lisp immediately suggests convenient system-level functionality in the spirit of idiomatic Lisp code that is usually over-looked in other environments. An example is the use of with-... macros.

In a somewhat disappointing way, it turned out that for many tasks the author finds Shell workflows running Common Lisp where needed more convenient than alternative workflows based on the Common Lisp REPL.

The current code of the system is available at https://github.com/7c6f434c/lang-os.

/

M. Raskin

Lisp in the middle

# ACKNOWLEDGMENTS

The author would also like to thank the anonymous referees for their valuable comments and helpful suggestions on presentation.

#### REFERENCES

- [1] GNU Emacs project page. Retrieved on 16 February 2018. https://www.gnu.org/ software/emacs
  [2] StumpWM project page. Retrieved on 16 February 2018. https://github.com/
- stumpwm/stumpwm
- [3] Ludovic Courtès. Functional Package Management with Guix. European Lisp Symposium 2013, Madrid, Spain. Retrieved on 16 February 2018. https://arxiv. org/abs/1305.4584 [4] GNU Shepherd project page. Retrieved on 07 March 2021. https://www.gnu.org/
- software/shepherd
- [5] Mezzano project page. Retrieved on 16 February 2018. https://github.com/froggey/ Mezzano
- [6] Movitz project page. Retrieved on 16 February 2018. https://www.common-lisp. net/project/movitz/
- [7] LOSAK project page. Retrieved on 18 February 2018. http://losak.sourceforge.net/
- [8] Eelco Dolstra, Merijn De Jonge, Eelco Visser. Nix: A Safe and Policy-free Sys-tem for Software Deployment. Large Installation System Administration Con-ference 2004. Retrieved on 16 February 2018. http://nixos.org/~Eeelco/pubs/ nspfssd-lisa2004-final.pdf
- [9] nsjail project page. Retrieved on 07 March 2021. https://github.com/google/nsjail
- [10] Olin Shivers, Brian D. Carlstrom, Martin Gasbichler, and Mike Sperber. Scsh Reference Manual Retrieved on 07 March 2021. https://scsh.net/docu/html/man. html
- [11] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman. Julia: A Fast Dynamic Language for Technical Computing. Retrieved on 07 March 2021. https://arxiv.org/abs/1209.5145
- [12] cl-html5-parser project page. Retrieved on 07 March 2021. https://github.com/ rotatef/cl-html5-parser
- [13] Thoughtful Theridion project page. Retrieved on 07 March 2021. https://gitlab.common-lisp.net/mraskin/thoughtful-theridion
  [14] Superior Lisp Interaction Mode for Emacs project page. (Includes Swank debug-
- ging protocol implementation) Retrieved on 07 March 2021. https://common-lisp. net/project/slime/
- [15] SpectrumOS project page. Retrieved on 22 March 2021. https://spectrum-os.org/

# Tuesday, May 4 2021

# A replicated object system

Hayley Patton hayley@applied-langua.ge

#### ABSTRACT

We describe Netfarm, a replicated object system, in which various kinds of objects can be stored across a network. These objects are instances of schemas (themselves also objects), which describe the representation of objects, and their behaviour, using a portable bytecode. Objects affect each other by running scripts using the bytecode, which in turn produce effects on multi-sets of computed values. Arbitrary access to effects is restricted by an object capability system on the object scale, and by capability lists on a larger scale, allowing untrusted objects to communicate, and for many untrusted applications to run on nodes and client programs. Objects may exist on multiple nodes of the network to make the system fault-tolerant, as their behaviour is explicated to the system and thus reproducible. Replication of the actions generated by scripts cause Netfarm to exhibit strong eventual consistency. This programming model allows for supporting many programs on one object system, which can use efficient replicated algorithms and data structures.

### **CCS CONCEPTS**

• Computing methodologies → Distributed programming languages; • Computer systems organization → Redundancy; • Information systems → Distributed storage.

#### **ACM Reference Format:**

Hayley Patton. 2021. A replicated object system. In *Proceedings of the 14th European Lisp Symposium (ELS'21)*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.4712699

#### **1** INTRODUCTION

There is a growing trend of using decentralised networks and systems for online communication and information storage. Such systems have many *servers* or *nodes*, operated and hosted by different parties, and thus do not have central points of failure. Many of these systems exist today, with vastly different applications and network models. These systems can become stifling with their focus on individual applications; while some persistent users may work out how to re-use the infrastructure of a system for another use case, they are invariably going to find difficulties, as they are unable to replace features of the system with their own. Some systems also cannot detect errors and invalid transactions on the system sufficiently, often putting too much trust on a server or node to not misbehave on behalf of the users of that server. For example, some systems may be unable to detect *spoofed* messages, which were not

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

ELS'21, May 3-4 2021, Online

© 2021 Copyright held by the owner/author(s).

https://doi.org/10.5281/2e11000.4/12099

actually sent by their authors, but were sent by someone else with access to the server.

Our initial intentions were to create some sort of fault-tolerant and "accountable" decentralised communication application, and avoid failures caused by server failure and bad operators and users on the network. The system would be accountable in the sense that any computation made could be reproduced, so that a server publishing incorrect results of some computation (perhaps due to a hardware fault, or a malicious operator) could be detected. These intentions are well served by a *distributed* and *replicated* network.

After implementing a basic system with these properties, we found it would be convenient for the system to verify invariants on behalf of an application, and for the system to be capable of storing information for many applications. These goals culminated into a *replicated object system*, in which objects can validate their current states, refusing to be instantiated if they are invalid, and objects can communicate to add "after the fact" references to each other.

# 1.1 Distribution

We will discuss properties of existing networks in Previous Work, but an abstract understanding of how these networks are formed will be useful.

Many programs and protocols on a network have many clients and one server. Larger systems may use many servers, but usually maintain the illusion that there is only one server somehow. Either form of system is a *centralised* system, as all information is centralised on infrastructure owned by one party. Managing the infrastructure and data is also entirely the duty of that party, which may be unacceptable for some users.

A *decentralised* system partitions such duties to multiple parties. There are two general techniques for decentralisation: federation and distribution. A *federated* system has users pick a server to use as part of a kind of *identity* for information stored on the network. (This server is sometimes called a *homeserver* or *instance*.) An identity provides an unambiguous reference format for any other user of the protocol to retrieve the information. Such an identity usually is made from some pseudo-random string and the name of the server. For example, the Netfarm room on the *Matrix* messaging protocol has the identity of !YUbZBUZUxCNnDA1xsZ:matrix.org. The dependence on a server is an immediate drawback of a federated system; that server is the only source of some information, and failure of a server can make the protocol unusable for users of that server.

A *distributed* system avoids this failure mode, as the identity of information is independent of a server on which it is stored. Typically, an identity is instead formed using a cryptographic *hash* of the information; and it can be efficiently searched for using a *distributed hash table*, which provides an algorithm for searching for a node which stores some information. Using a hash to create an identity precludes mutable objects per se, as modifying the state of an object would change its hash. Mutable data must be implemented

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM. https://doi.org/10.5281/zenodo.4712699

using a different mechanism, and Netfarm provides *computed values* to simulate mutable data, by computing side effects from the rest of the state of the network.

# 1.2 Replication

Data on a networked system is either *replicated* or not replicated. In this paper, we will be specific and define a *replicated system* as one where data may be retrieved from multiple sources, and any source is as valid as another. This definition excludes some systems where one server is the *primary source* of some data, and other servers *cache* that data, where only the former server would be able to accept updates to the data. Any data can be retrieved and updated from any server in a replicated system which has a copy of the data.

To our knowledge, replicated systems must be distributed. A centralised system would not be able to replicate, and the server component of object identity in a federated replicated system would have to be useless in order for any server to be able to serve requests for any object. We have not found any non-replicated distributed systems, but replication is a useful property in itself, and sometimes less confusing to discuss (we discuss "distributed" object systems in Previous Work).

Much like decentralised and centralised systems, a replicated system exhibits better fault tolerance than a non-replicated system; if *n* servers can be used to service a request with a *p* probability of failing on one server, and failures are independent events, then there is a  $p^n$  probability that all servers fail, and a request cannot be serviced. It is thus preferable to have a replicated system, as to minimise the occurrence of faults which render the system inoperable.

However, replicated systems require careful synchronisation and consistency measures, which are further complicated by having to handle node failures, and intentional attempts to manipulate the state of a system. Such measures often lead to difficultto-understand programs, and programmer experiences not unlike debugging low-level concurrent programs. Netfarm allows a programmer to write replicated systems using a small set of features; but said features can capture the behaviour of many systems, and an object model and message passing can make the resulting program easier to comprehend and more adaptable.

# 2 PREVIOUS WORK

#### 2.1 Distributed ledgers

2.1.1 Bitcoin. A blockchain creates consensus over the ordering of a transaction log, while allowing for substantial fault tolerance. Typically, blockchains are used to handle digital currencies, or *cryptocurrencies*, but we are more interested in the consensus mechanism and programming techniques they provide. (Although such mechanisms often assume, in part, that there is some currency to compensate honest users with.)

*Bitcoin* has a simple scripting system for transactions. Scripts are written in a bytecode, which executes on a stack machine. A script is usually used to verify that the user attempting to *spend* some currency is allowed to spend it. The bytecode includes instructions for basic arithmetic, hashing and verifying signatures; but the bytecode is not Turing-complete, nor particuarly expressive. Some non-essential instructions were removed as part of a (fairly lousy) fix for an incorrect implementation of bit-shifting, which would cause nodes to crash sometimes.<sup>1</sup> The bytecode has never supported any looping; it would not be suitable for programming many other uses of the Bitcoin network.

2.1.2 *Ethereum*. The *Ethereum* network has extended the script system, to allow for clients to program *smart contracts*, which allow for implementations of many complex systems on a blockchain using a Turing-complete language, such as additional currencies, decentralised exchanges, voting systems and some games. The program in a smart contract is executed when it receives a transaction, and the program is described using a bytecode, which has access to a stack, temporary memory in the form of an octet vector, and a permanent map from 256-bit unsigned integers to 256-bit unsigned integers (as described in Chapter 9 of [19]). Contracts can communicate by calling methods of other contracts.

2.1.3 Holochain. The Holochain project can be seen as a modification of the blockchain paradigm. Instead of providing consensus on one global transaction log, a user of a Holochain network maintains their own transaction log, and transactions provide cross-references between logs. Verifying a transaction between two parties consists of verifying the logs belonging to either party [5]. By using many separate logs, such a network can scale using the partitioning provided by a *distributed hash table*. (The Ethereum developers intend to improve scaling by *sharding* transactions over 64 smaller chains [8], but nodes in a distributed hash table can scale their contribution to the network at a more fine-grained level.) Scripts for Holochain are written in the *WebAssembly* portable assembler.

2.1.4 *Proof of work.* Many blockchains use a *proof of work* technique, which makes falsifying the transaction log very computationally expensive [15]. This process requires creating *blocks* of transactions, which will only be accepted by the network if hashes of blocks have sufficiently small values. Such blocks have to be generated by *brute force*, and the act of generating blocks is called *mining*. It is assumed that only an honest majority of users would have the majority of computing power, so an honest network would be able to produce blocks faster than a user attempting to tamper with the order of transactions.

Many parties are interested in dedicating huge computer farms to mining, as they are paid with cryptocurrency when they are first to produce an acceptable block. Unfortunately, a system based on perpetually brute-forcing values, especially at this scale, has the drawback of requiring significant amounts of electrical energy. At the time of writing, [6] estimates the electrical consumption of *Bitcoin* to be somewhat greater than that of the country of Finland. There are other methods of implementing verification for blockchains, but they all require the well-behaving users of a network to have a majority of some scarce resource, such as other currency or storage.

We do not believe we require the strict consistency measures blockchains provide, so we did not use a blockchain to implement Netfarm. However, the ability for clients to program a distributed system can be very useful, so we have taken some inspiration

<sup>&</sup>lt;sup>1</sup>This is better documented in CVE-2010-5137 and https://en.bitcoin.it/wiki/Common\_ Vulnerabilities\_and\_Exposures#CVE-2010-5137.

A replicated object system

from how blockchains are programmed when implementing our programming system.

#### 2.2 Federated networks

As mentioned before, federated applications delegate data storage and maintaining identity to many servers, although a user will be dependent on one server. In general, cross-server communications only authenticate servers and not users, so servers could falsify messages and state without detection; and while federated systems provide collections of events that are general enough to support multiple forms of presentation, they are still inextensible and cannot support the implementation of new programs.

It can be observed on both systems that large numbers of users are on very few homeservers, allowing much of those networks to be unusable should only those few servers become unavailable.<sup>2</sup> There are many factors influencing the population distribution (which we attempt to examine in the introduction to [2]) but the decentralisation and fault tolerance of federated systems appears to be very questionable.

2.2.1 *Matrix*. Matrix is a federated chat protocol, which offers a convenient web client, end-to-end encryption, integration with some other protocols (such as IRC and Discord), and integration with some other tools frequently used by its users, such as receiving GitLab and Grafana notifications in a room.

It uses a *room* as an aggregation of *messages*, which is a convenient abstraction for a messaging application, but it is awkward for many other applications. This situation is not unlike, say, implementing the Common Lisp tagbody special operator using the condition system. It is certainly possible, but it is strange, and likely inefficient and difficult. Some new applications have been created, such as a script which renders a blog out of the messages in a room;<sup>3</sup> but they do not modify the semantics or structure of Matrix significantly, merely changing the presentation of a room. Instead, we provide a lower-level abstraction, a *computed value* system, but it is capable of implementing rooms (and we will describe how to implement it in *An example use*).

Furthermore, when a user is not using end-to-end encryption, it is possible for a server operator to spoof messages; claiming that a user sent a message which they had not actually sent. Even a simple authorisation mechanism, such as optionally *digitally signing* messages, would be preferable for sending sensitive announcements, where the message cannot be falsified, but it must still be published publicly.

2.2.2 ActivityPub. ActivityPub is a federated micro-blogging protocol, which is used to post short text messages, with optional visual and audio attachments. Well known implementations of ActivityPub include *Mastodon* and *Pleroma*; although it has had many implementations which change the mode of communication somewhat, such as focusing on image or video sharing, or presenting messages in an online forum layout. These deviations from ELS'21, May 3-4 2021, Online

the usual micro-blogging style, however, remain mostly cosmetic, where messages and aggregations of messages are presented to the user in different ways, and different (arbitrary) restrictions are placed on the forms of messages which a server will publish for a user.

#### 2.3 Distributed object systems and Actors

E and Spritely are examples of *distributed object systems*. Object identity is based off server identity in these systems, so it could be argued that distributed object systems of this type are really *federated* by our definitions, but users of these systems call them *distributed* anyway. It follows that they do not replicate objects. There are also separate instructions for sending messages to local objects, which cannot fail, and sending messages to foreign objects, which can fail, as network-related faults can happen.

Erlang may or may not be a distributed object system, depending on who<sup>4</sup> you ask. It is less contentious to call Erlang an *Actor system*. Actors (processes in Erlang parlance) also have node addresses in their identity, but there is only one asynchronous message sending construct [3, p. 66], which can always potentially fail. Again, processes are not replicated in Erlang, but failure detection and replacement is still a design goal of Erlang; which is instead achieved by respawning processes which crash. This sort of fault tolerance is orthogonal to replication, as it generally is used to work around a program generating an invalid state, and not to work around the disappearance of a node.

# **3 OUR TECHNIQUE**

Objects are stored across a distributed hash table, in which it is possible to store objects, retrieve objects by hash, and subscribe to observe *side effects* induced on objects. We provide a class-based object system, where objects are instances of *schemas*, which in turn are objects themselves. These schemas contain lists of slots, *computed slots*, and *scripts* which describe the behaviour of instances of a schema. A schema is an instance of the schema schema, which is an instance of itself (c.f. the Common Lisp metaclass standard-class, which is an instance of itself).

All information which a node presents has been verified and/or produced by itself, so nodes do not have to trust each other to make progress when synchronising objects. The faultable nature of the network is hidden from Netfarm objects, as nodes perform all of the computation on behalf of an object. If an object attempts to retrieve another object that the node does not currently store, the computation is paused until that object has been retrieved. We believe this may introduce some storage overhead, as objects that the node is not expected to store will be verified to satisfy requests on behalf of scripts, but this overhead has not been measured.

We use a Kademlia-based distributed hash table [13] to search and retrieve objects, but the design of Netfarm is not dependent on any object storage mechanism. The only requirements of Netfarm on the object storage mechanism are that it must be able to retrieve objects which it would not otherwise retrieve, enabling scripts to

<sup>&</sup>lt;sup>2</sup>https://social.coop/@Stoori/100542845444542605 shows the usage of ActivityPub servers on a log-log graph. https://the-federation.info/ estimates the total users of the Fediverse to be around 4 million, and the top 12 servers on https://instances.social/ list/advanced have around 1.75 million users.

<sup>&</sup>lt;sup>3</sup>This project is named the "Matrix blog" and is described on https://matrix.org/docs/ projects/client/matrix-blog.

<sup>&</sup>lt;sup>4</sup>...and also when you ask. Joe Armstrong was initially critical of object-oriented programming, leading to an infamous quote starting with "you wanted a banana"; but then his thesis supervisor had convinced him that Erlang was the only really objectoriented language, as it exhibits strong isolation and polymorphism using message passing and independent "object" processes.



Figure 1: An object is an instance of a schema, which is in turn an instance of the schema schema. We draw objects with box-and-arrow diagrams, using dashed arrows for *computed values*, and labelling arrows with the names of the slots they represent.

retrieve any referenced object on the network, and to provide some metadata about objects for synchronisation.

Objects can be verified using *digital signatures*, associating them with the users that signed them. Such signatures can be used as an authentication mechanism, which allows some actions on objects to be performed only by specific users. (Users are just objects; they do not have any special status in the Netfarm system, except that Netfarm is able to retrieve public keys from user objects.) A *key exchange* mechanism is also provided, using *elliptic-curve Diffie Hellman* (a variant of [7]) keys associated with users of Netfarm, which could be used to encrypt private objects, but it is not used by Netfarm itself.

Objects can contain references to other objects in their slots, which are usually named by hashes of the referenced objects, to create references to other objects (excluding *inbuilt objects*, which are named with symbolic names starting with inbuilt@). This would prohibit mutating non-inbuilt objects, as changing the slot values of an object would change its hash, and thus break identity, and it would prohibit creating circular references, as there would be infinite recursion when attempting to compute the hash of any object in the cycle. Side effects are instead caused by running *scripts* which produce *side effects*, adding and removing *computed values* from *computed slots* in objects, which contain multi-sets of values. These computed values allow for "after the fact" references, which do not affect the hash, and thus identity, of an object; so a client will observe the computed slots of the object change.

#### 3.1 Side effects

When designing a system where changes must be replicated over many copies of an object, it is possible that users will observe changes in different orders; so the order in which changes are applied should not affect the result, i.e. the changes are *commutative*, and the data type which is affected by said changes is a *commutative replicated data type* (or CRDT). It is also possible to use a *conflict resolution* mechanism to resolve the ordering of effects, freeing effects of having to be commutative, but we have chosen to use a conflict-free replicated data type, as conflict resolution mechanisms are frequently very specialised, and sometimes hard to get right. The actions on a computed value system are the addition or removal of a value to a computed slot (herein called a *computed value*). A computed slot is a variation of a *multiset*, which also allows for negative counts of a value, in order for actions to always be commutative. If we disallowed negative counts, for example, removing a value that was not present in a computed value set, then and then adding it would create a different count to adding a value then removing it, and so side effects would not be commutative. The computed value set is a CRDT, which makes Netfarm exhibit *strong eventual consistency*. [17] provides a proof that using CRDTs will provide eventual consistency, so we only need to show how side effects on computed values are commutative, to show that computed values can be used in a system with strong eventual consistency.

#### THEOREM 3.1. Actions on computed values are commutative.

PROOF. Recall that an action on a computed value system is either the addition of a value to a computed slot of an object, or the removal of such a computed value. We could model the state of a computed value system as a mapping of tuples  $\langle object, slot, value \rangle$  to an integer *count* of values, each tuple initialized to 0. Adding a computed value causes the count associated with the value to be incremented, and removing a computed value causes the respective count to be decremented. Incrementing and decrementing are special cases of addition ( $\lambda x.1 + x$  and  $\lambda x.-1 + x$  respectively), and addition on integers is known to be commutative. Thus our computed value system has commutative actions.

#### 3.2 Scripts

For side effects to be replicated, we must explicate the behaviour of objects to a Netfarm network. Scripts are run using a virtual machine based on the SECD machine [12] and the Smalltalk interpreter [9]. This virtual machine provides first-class functions and immutable data, and message-passing based polymorphism. The behaviour of objects is aggregated into *scripts*,<sup>5</sup> and schemas have lists of scripts, which are searched to find applicable methods. The behaviour of the virtual machine is entirely deterministic, and thus the complete object system is deterministic.

We decided to use our own virtual machine for a few reasons:

- it is not difficult to write a compiler for a mostly-functional language targeting the SECD machine,
- a virtual machine which exposes some sort of in-memory representation of data, such as WebAssembly, reduces the number of possible techniques that could be used to optimise an implementation of the virtual machine, such as *hash consing* intermediate data (apparently first performed in [10]), and passing host objects into the virtual machine without copying, and
- the virtual machine can enforce various invariants which make cross-object communication significantly less difficult, such as checking function signatures, and ensuring method dispatch works as expected.

<sup>&</sup>lt;sup>5</sup>The provider of behaviour of an actor in the Actor model is sometimes cutely named a program script. We did not know of this usage before choosing to call our programs scripts.

#### A replicated object system

A message-passing system appeared to be the simplest system which would enforce *encapsulation*: only the methods of an object can affect the object, a requisite for running multiple programs on one network, without malicious programs being able to clobber the state of other programs. Objects send messages, consisting of a method name and argument list, to other objects, and a receiving object dispatches on the method name. The receiver searches for all methods with the method name defined by scripts, and calls the first applicable method. If a method cannot be found, the process is repeated for a special does-not-understand method (much like in Smalltalk), and the method is called with the method name and arguments. Messages can be *authenticated*, as the receiving object can retrieve the sender of a message. If it is necessary to obfuscate the sender of a message, it is not difficult to create a proxy object, which re-sends all messages sent to it to another object.

When a node retrieves a new object, either by replicating it from another node, or by receiving it from a client, it sends an initialize message to the object, before it allows other nodes to read its copy of the object. The initialize method can thus be used to prevent invalid objects from being created.

Scripts can also induce a partial ordering on the creation of objects in the network. Recall that, if a script attempts to access an object that the node does not yet store, execution of the script is paused. Thus the creation of a referred object always happens before the creation of a referring object.

Using a scripting system with a Turing-complete language and a suitable CRDT could make a replicated system into a sort of "universal simulator" for replicated systems; it is possible to re-implement the core algorithms and data structures of some decentralised systems, including Matrix and Holochain, in Netfarm. However, executing arbitrary untrusted code without any safety measures is a very large security and safety problem; we must provide some boundaries on what programs may do in order to guarantee determinism and safety.

3.2.1 Limiting script misuse. It may be useful to run scripts to present objects and content for clients, and allow clients to interact with objects in a more convenient manner for each object. Conventional "wisdom" surrounding the state of Web browsers, which have run *applets* written in Java, Flash, and now run larger programs written in JavaScript, suggests that this is only a convenient manner to exploit a client's computer, steal their information, or use their computers to perform computing for another party without consent; but we can make these forms of misuse impossible, with little effort.

The behaviour of scripts is restricted by a *capability list*, which prohibits using some runtime features when they are inappropriate; for example, a script run by initializing an object cannot read computed values, as that would allow creating behaviour which is dependent on script execution order, and thus not commutative; but it may write them, and a script run by a client can read computed values, but cannot write computed values. No instructions are ever provided to affect the outside world, and we have implemented the interpreter in safe Common Lisp code, so it should not be possible for a script to escape its environment.

Scripts follow an *object capability model*, where they cannot construct arbitrary references, and can only be given references

by other scripts and objects. (This is not the case for users of the Netfarm protocol themselves; replication requires being able to locate all objects, to our knowledge.) Limiting the execution time given to scripts also reduces the computing wastage a malicious script can perform; object initialization must terminate in 300,000 instructions (which can currently be executed in about 5 milliseconds on a relatively new desktop processor), and an interactive client should require significantly fewer than even a million instructions per second to process information for the user; excluding graphical rendering and user input, which are better done in the host environment. There is no situation in which it is possible to perform any significant computation, and then have it transmitted off a client; nor is there a situation where a program could somehow steal confidential information, and then transmit it, nor is there a situation where a network can be misused to perform any significant computation.

## 4 AN EXAMPLE USE

We have been designing some programs to test the usability of Netfarm. One of these programs is a chat program named *Catfood*,<sup>6</sup> which requires storing messages in a way that allows relatively fast sequential access.

We use an *event stream* data structure, which provides a partial ordering of *events* added to the event stream. Such a stream consists of the stream object, and many event objects which are added to the stream.



Figure 2: A queue may be constructed using *head* and *tail* references into a linked list.

The event stream resembles how a queue data structure may be constructed from a linked list. (However, we do not remove events from an event stream; we only define adding new events and iterating over all events.) Such a queue would hold references to the first and last conses of the linked list. Similarly, an event stream contains a slot for the first event, and a *wavefront* computed slot. This wavefront provides a fast method to retrieve the latest messages; when an event is created, it is added to the wavefront, and all events observed in the wavefront are removed. Each event contains a slot with the previously observed wavefront, forming a *directed acyclic graph* of events.

With zero latency and no partitioning, this graph will be a linked list, and the wavefront always contains exactly one event. In the presence of latency or a partition, the entire wavefront may not

<sup>&</sup>lt;sup>6</sup>This program might be used to "eat our own dogfood", should we test it and entrust it with real discussions, but we aren't really dog people. After the bunny, which we used as a logo for Netfarm, the next most loved animal in the early Netfarm community was the cat.

ELS'21, May 3-4 2021, Online

be observed, and only part of the wavefront will be removed by a new event. The true wavefront thus can contain multiple events. Creating another event will return the wavefront to containing only one event. It is not necessary for the wavefront to be consolidated as such, but it will minimise the number of events that a user must sort when iterating over the events of the stream in order.



Figure 3: Two writers raced to add events to an event stream. The wavefront contained two events while they were racing; though either writer observed only their last event in the wavefront due to network latency. After they finish racing, another event is added, which stores the previous wavefront, and now the current wavefront contains exactly one event.

Netfarm schemas can be created using a special metaclass. The event class and schema could be defined with:

(defclass event ()

((last-wavefront :initarg :last-wavefront

:reader last-wavefront)

(stream :initarg :stream :reader stream)

```
(successors :computed t :reader successors))
```

(:scripts (program-from-local-file "Stream/event.scm"))

```
(:metaclass netfarm:netfarm-class))
```

(Note that program-from-local-file is defined to compile the source code of a script to a script object, as described later.)

#### 4.1 Searching for a timestamp

Some operations we may require are displaying some of the most recent messages, and displaying messages sent around a given time. These sorts of operations can be performed by traversing the graph, and using a data structure like a *priority queue* to sort events we have not traversed yet, popping them off in order.

One illustrative example is searching for an event at a timestamp; we can work our way back from the wavefront, traversing previous wavefronts, and eventually reach that event. We could choose whether most of the code for this system should be written Hayley Patton

in Common Lisp (or another host language), or be written to run on the Netfarm script machine. There are advantages to using either language; using Netfarm scripts allow for other users to provide objects subject to some extensions to a protocol, without having to modify a client, but the facilities provided by Common Lisp, including the standard library and compilers<sup>7</sup> may make programming in the host language more favourable.

A function to find an event at a particular timestamp may be defined as:

(defun find-timestamp (stream timestamp) (labels ((older? (e) (< (timestamp e) timestamp)))</pre>

```
(newer (events) (remove-if #'older? events)))
(loop with q = (make-queue (newer (wavefront stream)))
until (empty? q)
do (let ((e (pop-oldest q)))
        (when (= timestamp (timestamp e))
            (return-from find-timestamp e))
            (insert-all (newer (last-wavefront e)) q)))
```

(event-not-found)))

The performance of searching could be improved by storing the older wavefronts of previous events, forming a *skip list* of predecessors, allowing faster access to older events to search. The algorithm would otherwise remain the same.

# 4.2 Programming Netfarm

While a client can be programmed in any mixture of the host language and Netfarm scripts, to replicate side effects and verification of objects, we must write some scripts. These scripts, fortunately, can be compiled from a language which is not as tedious as SECD assembly. We could use the *Slacker* compiler<sup>8</sup>, which generates code for our virtual machine from a language which looks superficially like Scheme. The script for an event could be written like:

```
(define (curry f x)
 (lambda (y)
    (f x y)))
(define (map f xs)
 (if (null? xs)
      '()
      (cons (f (car xs))
            (map f (cdr xs))))
(define (remove-event! s e)
 (call-method s 'remove-from-wavefront e)
 (call-method e 'add-successor (self)))
(define-method (initialize)
 (let ([stream (object-value (self) 'stream)])
    (map (curry remove-event! stream)
         (object-value (self) 'last-wavefront))
    (call-method stream 'add-to-wavefront (self))))
(define-method (add-successor event)
```

(add-computed-value 'successors event))

<sup>&</sup>lt;sup>7</sup>We are considering writing a dynamic compiler for Netfarm scripts which would generate Common Lisp code, which would then be compiled to a faster representation using compile.

<sup>&</sup>lt;sup>8</sup>https://gitlab.com/Theemacsshibe/slacker-compiler

A replicated object system

Some helper functions (which probably should exist in a standard library) have had to be defined, but they demonstrate that the Netfarm script machine is capable of computing with higher order functions, and returning closures.

The Netfarm implementation represents Netfarm objects as Common Lisp instances, and schemas are translated<sup>9</sup> to Common Lisp classes. It is thus possible to dispatch on the schema of an object from Common Lisp code; the implementation uses this feature to allow for clients to provide methods to duplicate Netfarm slots with specialised representations of Netfarm values, and it has proven useful while testing Netfarm code. During manual testing, we wrote a short method to *present* an event graph visually using the *Common Lisp Interface Manager* [14], (using format-graph-from-root) to quickly verify if the graph was being modified as expected.

#### 4.3 Other decentralised log implementations

Our design is mostly based on the *state resolution graph* used in Matrix, described in [1]. Matrix maintains a similar directed acyclic graph of some events, where any event references the power level events which allow it to occur. (Power level events change the *power levels* of users, which are used to grant permissions to them, such as the ability to invite or kick other users.) The events are then topologically sorted to determine all power levels, and then other events can be verified based on the computed levels. The operation of the wavefront (which is also called the set of *forward extremities* in Matrix) is programmed as part of a Matrix server implementation, whereas the event graph in Netfarm is an ordinary application, for which node implementations do not have to have any special implementation code to support.

[11] offers a proof that the Matrix event graph is a CRDT, as well as analysis of how the data structure handles network latency and partitions. However, the implementation in Netfarm is a sort of proof by reduction, as it is possible to rewrite event graph actions into actions on a computed value system, which we have already proven is a CRDT. This proof may be considerably simpler than the aforementioned proof; but it only proves that actions are commutative, not that the data structure works as intended, or is particularly efficient.

[16] is a similar event log, which is implemented atop IPFS, based on a *grow-only set*. This log does not maintain a wavefront or previous wavefronts. Instead, events contain *Lamport timestamps*, where an event contains a timestamp greater than the timestamps of all observed previous events. To access the log sequentially, a list of events and their timestamps is retrieved; which may cause significant space overhead if only retrieving a small subset of events. However, the runtime of sufficiently large accesses will be dominated by the latency of resolving references. A balance between being bottlenecked on network latency, or bottlenecked on network throughput can be made, by allowing the observed wavefront to grow to a relatively large number of events, and only clearing the wavefront when it grows further than that. The previous wavefronts of events would then contain many events, which can be retrieved in parallel.

#### **5 CONCLUSIONS AND FURTHER WORK**

We have described how we have designed the Netfarm replicated object system, which is notable for providing a programming system, allowing many distributed programs to be implemented on one Netfarm network. We then demonstrated programming with Netfarm by designing a simple event log, demonstrating that it is possible to replicate the behaviour of other decentralised systems on Netfarm easily.

A replicated object system, which has demonstrated itself capable of implementing other decentralised systems, has several benefits. The infrastructure used to implement multiple decentralised systems and networks can be simplfied. It would only be necessary to implement each decentralised system using the object system, and then one network and one node process could be used to host all the systems. An easily accessible network would lower the cost of entry to developing and deploying a new replicated program, as already existing networks and node implementations could be reused. A replicated system designed around objects communicating with *protocols* of messages allows for more extension than in systems with protocols consisting of bare data. Still, some additional features and changes would facilitate using more applications on Netfarm, and would make the programming model provided more understandable.

#### 5.1 Protocols

A sufficiently large distributed system will contain multiple implementations of some concept. These implementations may be abstracted over by programming against protocols. However, it is very likely these implementations are going to use different protocols. [18] bluntly states this problem as "*names* don't scale well [...]. So, [...] we are looking for ways to get things to happen without [...] having to tell any object to 'just go do this'". Names are the least of our problems though; we are more likely to find protocols which have methods with completely different meanings, than the same meanings attached to different names. Some kind of inference strategy may be useful to figure out how to express one protocol in terms of another.

Netfarm does not have an inheritance mechanism, because the techniques used for inheritance can vary wildly. A subtle variation is how a *class precedence list* is computed – Common Lisp uses a simple topological sort for instances of standard-class, and Dylan and Python (among other languages) use C3 linearization. A much larger variation is how slots and methods interact; in Common Lisp and Smalltalk (among others), slots and methods are orthogonal and methods cannot override slots, but in Self and Newspeak, methods can override slots. The absence of an inheritance mechanism leaves how to implement inheritance to the user, for better or worse.

#### 5.2 Object identity

Using hashes to form the identity of objects is typical for distributed hash tables, but it can create some strange situations with mutability of any form. It suffices to say that two objects with the same initial state are considered to be the same object, as hashing either object will result in the same hash. This notion of identity can often confuse a programmer, whom is used to creating objects that have the same slot values, but are certainly different; modifying

<sup>&</sup>lt;sup>9</sup>Schemas themselves are not classes. If that were the case, the class of the schema schema/class would have to be itself, which is not possible to construct in CLOS.

ELS'21. May 3-4 2021. Online

computed values of one object should not make visible changes to another object.

It may be possible to implement a naming system, like the system described in [4], where each node names each object with a random but unique identifier specific to each node, and additionally maintains tables of what other nodes have named each object stored. The usual searching used in distributed hash tables could still be employed, by having references contain the usual content hash, as well as the name one node provided. The content hash would be used to locate nodes which likely store the object required, then the name could be translated to retrieve the correct object.

#### REFERENCES

- [1] Neil Alexander. State resolution v2 for the hopelessly unmathematical, 2020. URL https://matrix.org/docs/guides/implementing-stateres
- Applied Language. The Netfarm book, 2020. URL https://cal-coop.gitlab.io/ netfarm/documentation/.
- [3] Joe Armstrong. Making reliable distributed systems in the presence of software errors. PhD thesis, KTH Royal Institute of Technology, 2003. URL http://erlang. org/download/armstrong\_thesis\_2003.pdf.
- [4] Ganesha Beedubail and Udo Pooch. Naming consistencies in object oriented replicated systems, 1996. URL https://citeseerx.ist.psu.edu/viewdoc/download? doi=10.1.1.19.8209&rep=rep1&typ==pdf. [5] Arthur Brock and Eric Harris-Braun. Holo: Cryptocurrency infrastructure for
- global scale and stable value, 2017. URL https://files.holo.host/2017/11/Holo-Currency-White-Paper\_2017-11-28.pdf.

- Cambridge Centre for Alternative Finance. Cambridge Bitcoin electricity con-[6] sumption index, 2020. URL https://cbeci.org/cbeci/comparisons.
- Whitfield Diffie and Martin Hellman. New directions in cryptography. IEEE [7] Trans. Inf. Theor., 22(6):644-654, September 1976. ISSN 0018-9448. doi: 10.1109/ TIT.1976.1055638.
- Ethereum developers. Shard chains, 2021. URL https://ethereum.org/en/eth2/ [8] shard-chains/
- [9] Adele Goldberg and David Robson. Smalltalk-80: The language and its implementation, 1983
- [10] Eiichi Goto. Monocopy and associative algorithms in extended Lisp. University of Tokyo Technical Report TR-74-03, 1974.
- [11] F. Jacob, C. Beer, N. Henze, and H. Hartenstein. Analysis of the Matrix event graph replicated data type. IEEE Access, 9:28317-28333, 2021. doi: 10.1109/ ACCESS.2021.3058576.
- [12] P. J. Landin. The Mechanical Evaluation of Expressions. The Computer Journal, 6 (4):308-320, 01 1964. ISSN 0010-4620. doi: 10.1093/comjnl/6.4.308. URL https: //doi.org/10.1093/comjnl/6.4.308.
- [13] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the XOR metric, 2002. URL https://pdos.csail.mit.edu/~petar/ papers/maymounkov-kademlia-lncs.pdf.
- [14] Scott McKay. CLIM: The Common Lisp interface manager. Commun. ACM, 34(9): 58–59, September 1991. ISSN 0001-0782. doi: 10.1145/114669.114675. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2007.
- [15] [16] OrbitDB. Append-only log CRDT on IPFS, 2016. URL https://github.com/orbitdb/
- ipfs-log.
- [17] Marc Shapiro and Nuno M. Preguiça. Designing a commutative replicated data type. CoRR, abs/0710.1784, 2007. URL http://arxiv.org/abs/0710.1784. Viewpoints Research Institute. STEPS towards the reinvention of programming, [18]
- 2007. URL http://www.vpri.org/pdf/tr2007008\_steps.pdf. [19]
- Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger, 2020. URL https://ethereum.github.io/yellowpaper/paper.pdf.

# aether: Distributed system emulation in Common Lisp

Eric C. Peterson Eigenware Berkeley, CA, USA peterson.eric.c@gmail.com

# ABSTRACT

We describe a Common Lisp package suitable for the design, specification, simulation, and instrumentation of real-time distributed algorithms and hardware on which to run them. We discuss various design decisions around the package structure, and we explore their consequences with small examples.

#### CCS CONCEPTS

• Computer systems organization → Real-time system specification; • Computing methodologies → Distributed programming languages.

#### ACM Reference Format:

Eric C. Peterson and Peter J. Karalekas. 2021. aether: Distributed system emulation in Common Lisp. In *Proceedings of the 14th European Lisp Symposium (ELS'21)*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.5281/zenodo.4713971

#### **1** INTRODUCTION

In this paper, we consider the dual problems of modeling and designing a complex distributed system, at the levels of both hardware and software. Those working in this problem space labor under an overwhelming number of variables: the structure and properties of the network, the structure and properties of the compute nodes on the network, the precise behavior of the software on the nodes, and so on. These variables have complex interdependencies, and the relative sizes of their effects depend strongly on the ultimate application and its performance requirements. Meanwhile, building a physical system to verify the performance of any collection of settings can be prohibitively expensive. Thus, this problem is especially well-suited for simulation.

Common Lisp is exceptional at incrementally modeling complex systems whose full set of constraints is not known at start. Its features allow one to nonuniformly increase the fidelity of the model in whatever areas make the most impact. While Common Lisp is not initially well-suited specifically to distributed programming, it is both generally adaptable and organizable, so that one can make a principled account of the augmentations needed to support this style of programming and its simulation. We offer here a Common Lisp framework for the design, programming, and analysis of distributed systems which are, in a variety of senses, not fully specified at the inception of the project.

ELS'21, May 03–04 2021, Online, Everywhere

© 2021 Copyright held by the owner/author(s).

Peter J. Karalekas Eigenware Berkeley, CA, USA peter@karalekas.com

To set the stage more concretely, let us discuss the specific application which drove the initial development of this framework. In quantum computing, our domain of interest, the classical electronics which drive a typical system are a fleet of processors and programmable radio-frequency waveform generators and receivers. These are highly concurrent systems which operate on micro- or nanosecond timescales, and they must solve coordination problems to determine the next step in a program. Basic forms of coordination (e.g., time synchronization) are required for any kind of operation, but they are especially critical for quantum error correction [8, 9]. Syndrome decoding and error correction require active communication and joint processing of feedback from different areas of the quantum device in hard real-time. Meanwhile, because the practice of quantum computing is so nascent and "artisinal", the details of the device, its steering electronics, and even the decoding scheme itself are all in flux-indeed, their design has the potential to be driven by the predicted performance characteristics of a specified decoding procedure.

To analyze the behavior of such systems, it is valuable to have a flexible emulation framework available which can accommodate myriad hardware components with varying levels of detail:

- **Network** The structure, protocol, and properties should all be programmable.
- **Compute nodes** The architecture of a particular node may be specified in complete detail, e.g., its behavior can be modeled by an instruction-by-instruction emulator with fully specified timing. Similarly, the amount of computational power available to a node should be configurable.
- System composition In all aspects, it should be possible to work with heterogeneous components, e.g., different processors with different microarchitectures and different clock rates.
- **Macroscopic defaults** In all aspects, it should be possible to ignore implementation details by falling back on default behaviors. These defaults may be unrealistic in a limited-resource regime (e.g., infinite network transmission capability), but they must nonetheless be fully specified.

We designed our software package, aether [14], with these kinds of flexibility in mind: aether provides support for the time-domain emulation of components with user-defined behavior, communication among these components, the steady offloading of execution semantics from standard Common Lisp to an application-specific DSL, the debugging of such software, and the analysis of performance effects in the forms of time, compute, and network requirements.

In sequels to this paper, we describe an application to topological quantum error correction, provide an implementation using aether, and apply aether's performance analysis features [13].

*Remark* 1. aether is *not* suitable for performant concurrent programming in Common Lisp, but this is not its goal. There are many

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

https://doi.org/10.5281/zenodo.4713971

such libraries available, including cl-actors [16], cl-muproc [6], erlangen [15], memento-mori [12], and simple-actors [1], though this list is in no way exhaustive. Ours differs in that it intends to capture the performance features of distributed software running atop emulated hardware, whereas these others are concerned with performing the distributed computation itself as quickly as possible.

## 2 MANY-COMPONENT SIMULATION

The bedrock of aether is in its simulation loop. A key quality of a digital system is that its components operate against various clocks, where on each tick they enact some operation before delaying until the next tick. Additionally, information passed through one component (e.g., a network interface or a memory bus) may arrive at some other component for reading after a possible delay.

The two key data types facilitating this kind of simulation are: event A callback and a time at which to trigger it. In addition to any other side effects, the callback returns a list of future events

to reinject into the simulation. **simulation** A collection of events yet to be run and a time value, called the *horizon*, which tracks the time before which history

has been fully fixed. In particular, no events in the collection may have trigger times before the horizon value. The utility function simulation-add-event can be used to set up

the initial contents of a simulation, and simulation-run can be used to process events in time order<sup>1</sup> either until exhaustion or until a user-specified condition has been met (e.g., a timeout).

To facilitate the authoring of event callbacks, aether provides the context macro with-scheduling, which executes its body in sequence while offering the following convenience forms to collect events to be returned at the close of the macro:<sup>2</sup>

schedule Stashes an event for return at the close of the macro.
schedule\* Consumes the results of another callback and adjoins
its events to the current list of events to be returned.

**finish-with-scheduling** Exits with-scheduling, either with a specified return value or with the events collected thus far.

*Example 2.* In Figure 1, we illustrate using these constructs to generate a few callback events.

Often, the components of a simulation are properly thought of as objects, each of which has a default behavior associated with it. For example, a particular simulation component may be responsible for emulating the behavior of a processor, which on each tick decodes and enacts the instruction at its program counter. To facilitate this, the generic function define-object-handler captures the default callback behavior of an object class, and when a simulation processes an event whose "callback" is an object, its callback is instead supplied by define-object-handler.

*Example* 3. In Figure 2, we define a processor emulator's heartbeat loop by using default object callbacks.

*Remark* 4. A simulation as described above can be neatly implemented via a priority queue. However, such an implementation

#### Eric C. Peterson and Peter J. Karalekas

```
(defun g (now)
 (with-scheduling (format t "g: ~a~%" now)))
(defun f (now)
 (with-scheduling
   (format t "f: ~a~%" now)
   (schedule #'g (+ now (/ (random 4) (1+ (random 4)))))
   (schedule #'f (+ now 1))))
(let ((simulation (make-simulation)))
 (simulation-add-event simulation (make-event :callback #'f))
 (simulation-run simulation :canary (canary-until 2))
 (values))
;; f: 0
;; f: 1
;; g: 5/4
;; g: 3/2
```

Figure 1: Example event callback f which triggers itself after a fixed delay and another function g after a random delay.

```
(defstruct processor
 "State of a simple CPU."
(data-stack nil :ty
 (data-stack nil :type list)
(program-counter 0 :type (integer 0))
 (instructions
                  nil :type vector :read-only t))
(define-object-handler ((self processor) now)
  ; NOTE: passing the bare object "self" to schedule
  (schedule self (+ now (/ *processor-rate*)))
  (destructuring-bind (operator &rest arguments)
     (incf (processor-program-counter self))
    (ecase operator
       ; example definitions of instruction behavior
      (halt : halts processor execution
      (finish-with-scheduling))
      (push
              place constant on stack
      (push (first arguments) (processor-data-stack self)))
      (muli
             : stack-constant multiplication
      (push (* (pop (processor-data-stack self))
                (first arguments))
             (processor-data-stack self)))
     )))
```

Figure 2: define-object-handler used to describe the transition rule for a toy microprocessor.

turns out to be under-performant in cases of interest: for large collections of processors with identical start times and identical clock frequencies, there are very few distinct key values in the priority queue. For this reason, aether's default implementation of a simulation uses a data structure with the interface of a priority queue but whose internals take advantage of repeated keys.

### 3 MESSAGES AND MESSAGE PASSING

While we have discussed some aspects of time-domain simulation, we have not made any particular reference to the communicative aspect of distributed computation. We make the following fundamental assumption: typical communication channels are physically synchronous (although perhaps lossy), but their *access* is subject to contention and hence appears as asynchronous. Accordingly,

<sup>&</sup>lt;sup>1</sup>Events scheduled for the same time are run serially, but not according to any convention. It could be useful for debugging race conditions to expose a programmable hook when breaking such ties.

<sup>&</sup>lt;sup>2</sup>Compare SBCL's define-vop's local macro inst.

#### aether: Distributed system emulation in Common Lisp

aether's networking capabilities are split into two halves: the simulation of a "physical" layer, which processes serially-supplied data against a constant clock; and primitives which communicate over this layer, which may appear to be asynchronous as delay accumulates in the simulation. We discuss the two layers in turn.

The physical network is made up of courier objects. As an interface to the communication API, each courier maintains a set of inboxes, indexed by address, which collect packets to be retrieved. Additionally, each courier maintains a transmission queue of packets to be sent: if the destination address resides on this courier, the message is stashed in the appropriate inbox; if not, it is routed to the next courier on the transmission path. The default courier class implements a network with all-to-all connectivity: every courier on the network is one hop away from every other. We also provide a more specialized example implementation where the couriers are arranged in a grid, direct communication is only allowed between nearest neighbors, and the couriers route packets accordingly.<sup>3</sup> One informs aether which courier services a block of code by binding the dynamic variable \*local-courier\*, which allows the code to interact with the networking layer.<sup>4</sup>

The communication API then provides a point of contact between this physical networking layer and the distributed programmer. It defines the following primitives:

- **register** Opens a new inbox at the local courier and returns the key to that inbox.
- **send-message** Sends a payload to a specified address. Any message may be sent to any inbox, provided the address of that inbox is known.
- receive-message Checks the indicated inbox for messages of a specified type. If there is such a message, it binds it and executes an associated block of code. If no message is found, the receive terminates and execution resumes as before.<sup>5</sup> The key associated to an inbox is required to receive messages from it.
- **unregister** Releases an inbox resource from the courier. A released inbox address can never again be used to retrieve messages.

*Example* 5. These primitives are enough to build various convenience methods. For example, Figure 3 illustrates an implementation of a "synchronous receive", which waits to resume execution until a receive has been successful.<sup>6</sup> aether also provides

#### ELS'21, May 03-04 2021, Online, Everywhere

#### Figure 3: An example callback that performs a busy-wait.

send-message-batch, a utility which sends messages to a sequence of target addresses.

*Remark* 6. Message passing in aether comes with significant guarantees: messages are always delivered, and messages sent from the same source to the same target are received in order. Neglecting the effects of transmission delay (e.g., when working with a single courier), aether acquires the semantics of the LOCAL model of distributed computing [11]. With transmission and processing delays incorporated, aether acquires the semantics of the DECOUPLED model [3, 4].

Additionally, each message is optionally tagged with a "reply address". This is available for user-defined behavior, but it is also used automatically by the networking layer in the case that the message is sent to an address which does not exist (e.g., if unregister has since been called to close its destination inbox). In this event, the destination courier will automatically generate a "return to sender" message and send it to the reply address so as to signal the failure.

*Remark* 7. Neither the simulation framework nor the messaging framework provide a built-in mechanism for communicating with the outside world. Instead, we leave it up to the user to decide how information should be extracted from the simulation. Typical options include direct inspection of simulated objects or via a user-supplied object that converts simulation-internal messages to simulation-external side effects.

# 4 ACTORS AND PROCESSES

A robust message-passing system opens the door for actor-based distributed computing [7]. Each actor, which might be thought of as a physical processor or as a green thread on some ambient physical processor, is responsible for some component of a computation, which it completes by communicating with other actors through the network.

The basic notion of an actor is captured in aether as a process. In order to render invisible the difference between processes residing on the same ambient processor and those residing on different nodes in the network, processes ought not to modify each others' state directly. Instead, each process registers an inbox with the courier to serve as its "public" address, and processes communicate with one another via these inboxes. The processes then proceed to act at regular intervals, with the pace of each set by its process-clock-rate. On each such tick, a process services inbound requests at its public address through user-defined handlers and makes progress on its own computations through user-defined procedures.

<sup>&</sup>lt;sup>3</sup>More explicitly, each destination address consists of a courier ID and an inbox ID registered at that courier. In the gridded example, a courier's ID is its coordinate position in the grid, and couriers route packets according to logic such as: if the requested address's *x*-coordinate is to the left of this one, forward this packet to the courier to the immediate left, so that the routing table is procedurally generated. <sup>4</sup>In practice, this binding is done automatically.

<sup>&</sup>lt;sup>5</sup>More exactly, the body of receive-message can carry different branches which specialize on different message types. The inbox is first scanned for instances of the first message type. By default, the entire inbox is considered, and if one or more messages is found, the indicated place is bound to the first (i.e., oldest) matching message, the associated block of code is executed, and then control exits from receive-message without trying its other branches. By toggling the peruse-inbox? argument, the user can tell it to only consider the first message in the inbox, rather than the entire inbox. If no message of the first type is found, then it proceeds to search for messages of the second type. Message processing continues in this way until all the message types are exhausted. In the case of exhaustion, receive-message executes an otherwise block if one is available, and then cedes control to its caller either way.

<sup>&</sup>lt;sup>6</sup>In the next section, we discuss a standard implementation within aether of waiting for receive-message to complete.

Service handlers are specified using define-message-handler, which takes as arguments the process (specialized by type), the message (specialized by type), and the current clock value. The body of a handler might access or modify the process's state, cause it to jump to a new code segment to perform a computation, generate messages, and more. The main constraint is that handlers function as "interrupts", in the sense that they are given a fixed amount of time (viz., one tick) to perform computation before they are required to terminate. Apart from the individual specification of these handlers, define-message-dispatch specifies the ordered list of service handlers that a process employs to act on messages received at its public address (by repeatedly calling out to receive-message). Each handler is optionally guarded by a predicate that is computed from the current state of the process, so that services can be selectively enabled.

The computational trajectory of a process is captured by its "command stack", which is a stack of commands yet to be performed, each specified as a cell of shape (COMMAND-NAME . ARGS). On each tick, a process automatically pops the next command off of its stack and performs the specified action, whose behavior is set out by define-process-upkeep. Typical command behaviors include:

**Local state modification** The command might modify the local state of the process, e.g., by changing some of its slot values.

**Message-passing** The command might send new messages or draw waiting messages out of inboxes.

- Intermediate computation The command might perform intermediate computation, e.g., collating values received from different network sources.
- **Procedure continuation** The command might push more items onto the command stack (viz., using process-continuation). This has the effect of delaying the command that was to be processed immediately next, in favor of performing some other command (or sequence of commands) first.<sup>7</sup>

In this way, the command stack simultaneously serves the purpose of a traditional call stack, a program counter, and executable memory.<sup>8</sup>

Because define-process-upkeep describes a region of code which belongs to a particular active process, we again have the opportunity to define some utility macros.

- **sync-receive** Busy-waits for a message of a specified type to arrive on a specified inbox. Its syntax is identical to that of receive-message.
- with-replies Gathers a family of responses, potentially from many different inboxes, by busy-waiting until they all arrive. It then proceeds with the computation housed in its body.

## Eric C. Peterson and Peter J. Karalekas

(defclass processor (process)

(:documentation "State of a simple CPU."))

(define-process-upkeep ((self processor) now) (HALT)
 (process-die))

(define-process-upkeep ((self processor) now) (PUSH value)
 (push value (process-data-stack self)))

(define-process-upkeep ((self processor) now) (MULI value)
 (push (\* value (pop (process-data-stack self)))
 (process-data-stack self)))

(define-message-dispatch processor

#### Figure 4: Re-implementation of Figure 2 using a process. The program-counter and instructions slots have been absorbed into the command stack.

(defclass arithmetic-server (processor) ())

```
;;; act like a server
```

- (define-process-upkeep ((self arithmetic-server) now) (START)
   "This process sits in a loop, waiting to serve."
   (process-continuation self `(START)))
- (define-process-upkeep ((self arithmetic-server) now) (EMIT address)
   "Generic instruction for emitting a message."
   (let ((result (pop (process-data-stack self))))
   (send-message address (make-message-rpc-done :result result))))

;;; factorial-specific functionality

```
(defstruct (message-factorial (:include message))
   "Message which requests a factorial computation."
   (n nil :type (integer 0)))
(define-process-upkeep ((self arithmetic-server) now) (FACTORIAL n)
```

"Computes n!, leaves result on the data stack."
(cond
 ((zerop n)
 (process-continuation self `(PUSH 1)))
 (t

;; advertises the factorial handler to externals
(define-message-dispatch arithmetic-server
 (message-factorial 'handle-message-factorial))

# Figure 5: Extends the processor defined in Figure 4 to allow it to serve requests for computing factorials.

*Example* 8. In Figure 4 we revise the example given in Figure 2 to use the process framework, and in Figure 5 we extend it to service interprocess communication requests to perform a simple computation. We also give a fresh example in Figure 6 of a distributed algorithm solving a standard problem in graph theory.

<sup>&</sup>lt;sup>7</sup>For instance, one may implement a while loop by testing the loop predicate and, conditionally on the test coming back positive, pushing the loop body and the while loop back onto the command stack.

<sup>&</sup>lt;sup>8</sup>While not set in stone, this unusual design choice is a direct consequence of the flexibility we have elected for in process semantics. In order to make use of a program counter to track program state, one needs the ability to jump (so, in particular, to return) to an arbitrary instruction location. However, in the case where the granular behavior of a process has not been fully specified, we permit Common Lisp itself to shoulder its execution semantics, and it is difficult to jump to the middle of a Lisp-defined procedure. To accommodate this, we essentially limit jumps to occur only at the end of each procedural step. In effect, this becomes a form of continuation-passing style: each step in a procedure organizes only the code needed to continue its computation.

#### aether: Distributed system emulation in Common Lisp

(defclass process-coloring (process) ((stopped? ...) (color ...) (neighbors ...)))

(defstruct (message-color-query (:include message)))

```
(define-rpc-handler handle-message-color-query
  ((process process-coloring)
   (message message-color-query)
   now)
  (process-coloring-color process))
```

```
(define-message-dispatch process-coloring
(message-color-query 'handle-message-color-query))
```

```
(define-process-upkeep ((process process-coloring) now) (START)
 (process-continuation process `(QUERY)))
```

(with-replies (replies) listeners
 (setf stopped? (not (member color replies))))))))

(define-process-upkeep ((process process-coloring) now) (IDLE) (process-continuation process `(IDLE)))

Figure 6: Randomized algorithm for 3-coloring a line which uses only local network structure [11, 17]. Leverages some features of the standard library discussed in Section 5.

*Remark* 9. There is no centralized directory service by which processes on a network can discover one another. Instead, processes become aware of each others' existence in three ways:

- The user can directly intervene (e.g., before simulation start) and write a process's public address into the slot of another.
- (2) When a process spawns a subprocess, it can record the subprocess's public address.
- (3) Addresses can be sent over channels, so that processes can tell each other about addresses of which they're aware. (So, a programmer could write their own directory service.)

# 5 A STANDARD LIBRARY

All of this scaffolding permits us to write actor-based distributed algorithms with relative ease. Such algorithms follow certain standard patterns to enable easy interprocess coordination, and aether provides built-in support for some of these patterns in the form of a standard library.

#### 5.1 Remote procedure calls

One such pattern is the remote procedure call: one process queries one or more other processes, then awaits the result (or, at least, a signal that they have finished) before using it to continue its own computation. This is typically implemented in terms of aether primitives as: ELS'21, May 03-04 2021, Online, Everywhere

- (1) The caller registers an inbox, the "private reply inbox".<sup>9</sup>
- (2) The caller forms an RPC request with the private inbox listed as the return address and sends it to the public address of the remote process.
- (3) The caller then performs a blocking receive, waiting to find a response in the private inbox.
- (4) The remote process services the request, which triggers the desired computation.
- (5) At the conclusion of the computation, the remote process prepares a message-rpc-done, which it sends back to the private address.
- (6) Upon receiving that message, the caller unregisters the reply inbox and proceeds with its own computation.

This involves some boilerplate code on both sides of the protocol, the caller and the callee. On the side of the caller, we have:

with-replies Manages the blocking receive. It also supports the automatic unboxing of the replies it receives (e.g., by mapping #'message-rpc-done-result over the message objects), waiting on multiple mailboxes in parallel, and gracefully handling return-to-sender replies.

On the side of the callee, we have:

- **define-rpc-handler** Extends define-message-handler by testing whether the return address of the inbound message is nonnull and, if so, returning the result of its body computation via a message-rpc-done envelope.
- define-message-subordinate Defines a non-interrupting message handler. A limitation of define-message-handler is that it functions akin to an interrupt: it cannot perform any operations that take nontrivial simulation time, and at its completion control passes back to the next frame on the call stack. In order to perform computations which require simulation time (e.g., if they need to perform RPC calls of their own), the handler can install call stack frames of its own, interrupting whatever procedure was being worked on previously and blocking it from continuing until the call resolves. As this blocking behavior can also be undesirable, define-message-subordinate provides an alternative: it spawns a parallel thread to handle the message, with all the features of define-process-upkeep available (e.g., with-replies).

#### 5.2 Broadcast and convergecast

We also provide tools for common patterns in the domain of distributed graph algorithms. In that context, processes correspond to problem graph nodes, and communication channels correspond to problem graph edges. As an algorithm progresses, nodes communicate and form graph structures (e.g., rooted trees), and these structures may have to internally coordinate to take further action. This coordination is conveniently accomplished via *broadcast* and *convergecast* protocols [2, 5].

The goal of a broadcast is to announce a message across a network, e.g., an instruction to enact some effect. When a process receives a broadcast message, it performs an associated action, calculates the next set of targets (e.g., all adjacent nodes except

<sup>&</sup>lt;sup>9</sup>This automatically associates the reply message with the particular query which generated it, where we may otherwise have had to distinguish between several inbound replies at the process's public address.

the sender), and then forwards the message along to those targets.<sup>10</sup> Thus, a broadcast can be thought of as a way to recursively act on a distributed graph structure. To facilitate this operation, we provide an extension to the standard message handler called define-broadcast-handler, which pushes a BROADCAST command onto the command stack before executing the body of the handler. This command forwards the broadcasted message to the next round of targets, optionally awaiting their reply and acknowledging the broadcaster in the form of a message-rpc-done.

When a process wants to gather some information from across the network, it can do so using convergecast. This protocol is similar to broadcast, with an additional step: each process reports the result of performing its action, and as reports are sent back to the convergecast originator, they are summarized by a user-specified computation in order to produce a final result. As a trivial example, convergecast can be used to count the number of nodes in a distributed tree, where each node reports the number 1 and the reports are summed as they are relayed back to the root.

### 5.3 Locking

Some process interactions are nonatomic, in the sense that they might interleave with other process interactions. Nonatomicity is why a distributed algorithm achieves a speedup over a sequential algorithm: a completely atomic system is essentially indistinguishable from a sequential system. Nonetheless, atomicity also remains a valuable property, as it is often critical to the correctness of an algorithm. One then wants to confer atomicity onto those sections of the algorithm which require it for correctness, called critical sections, and to otherwise work nonatomically to take advantage of parallel process interactions. Any mechanism by which we can confer controlled atomicity is called a lock. The broadcast lock, a cousin of Lamport's mutual exclusion mechanism [10], and a special case of the more general broadcast operation described previously, is a particular form of locking which appeared often in our application work. As such, we have included it as part of aether's standard library.

To begin the protocol, the process which wishes to hold the lock calculates the set of clients it must lock and sends them each a lock request. Each client either replies with a failure, or it gathers locks from its sub-clients, and so on. When each client hears successful replies from each of its sub-clients, it opens a new, private inbox and sends that address to the lock holder as its signal of success. Upon success, the lock holder then does its critical work; upon failure, it knows to avoid the critical work, perhaps to try again later. In either case, it unwinds any client locks it did acquire by sending a finish message to the private addresses, then awaiting signals from each that they have finished.

The subclass process-lockable of process implements this algorithm within aether. The relevant hooks are as follows:

**BROADCAST-LOCK** A command used to initiate lock requests.

- **message-lock-request** A message type used to deliver lock requests, replete with message handlers.
- **process-lockable-targets** A generic function used to calculate the clients on which to acquire recursive locks.

#### Eric C. Peterson and Peter J. Karalekas

```
(defclass writer (process-lockable)
 ((transmit-list ...)
                 ...)))
  (target
(defclass reader (process-lockable)
 ((receive-list ...)))
(defstruct (message-write (:include message))
 pavload)
(define-process-upkeep ((process reader) now) (START)
  (process-continuation process `(START)))
(define-process-upkeep ((process writer) now) (START)
 (unless (endp (writer-transmit-list process))
    (with-slots (target) process
      (process-continuation process
                             (BROADCAST-LOCK (,target))
                             (TRANSMIT)
                             (BROADCAST-UNLOCK)
                             `(START))))))
(define-process-upkeep ((process writer) now) (TRANSMIT)
  (let ((next (pop (writer-transmit-list process))))
    (unless (or (process-lockable-aborting? process)
                (null next))
      (send-message (writer-target process)
                    (make-message-write :payload next))
     (process-continuation process `(TRANSMIT)))))
(define-rpc-handler handle-message-write
   ((process reader) (message message-write) now)
  (push (message-write-payload message)
        (reader-receive-list process)))
(define-message-dispatch writer
(define-message-dispatch reader
 (message-lock
                 'handle-message-lock)
  (message-write 'handle-message-write))
(defmethod process-lockable-targets ((process reader))
 nil)
```

Figure 7: Writer processes mutually exclude their transmission sequences to a commonly held reader process by first acquiring its lock.

*Example* 10. In Figure 7, we provide an example of a pair of processes that use a broadcast lock to manage a race condition. Several writer processes which want to transmit data to a single reading process first acquire a lock on it before transmitting their multi-part payloads, retrying if their lock is denied. In this way, the reading process receives the payloads without interleaving, though they may arrive in any order relative to one another.

## 6 INSTRUMENTATION

Much of the value in writing simulation software is having a lowcost environment in which to do software development: one can implement complex software and verify at least some of its behaviors before testing it on a true physical system, which may itself be expensive to build. As previously discussed, another large source of value comes from the opportunity to design the physical system itself: if the simulation is faithful enough to give an estimate of resource consumption, then its analysis can be used to set hardware performance requirements. Toward both of these goals, aether

<sup>&</sup>lt;sup>10</sup>We use the word "broadcast" to capture the recursive, potentially multi-target nature of this operation. The message need not be forwarded to *all* targets on the network.

aether: Distributed system emulation in Common Lisp



Figure 8: (top) Weighted problem graph (10 nodes, 17 edges) with minimum-weight spanning tree indicated by red solid edges. (bottom) Output of the print-message-report logging mechanism when running the GHS algorithm [5] on the above graph. In particular, we find a message total of 85, which is indeed less than the upper bound of 200.

provides tools for analyzing the behavior of programs that use its simulation framework.

In sequel work, we describe an elaborate application of these tools to a hardware design problem in quantum computing.

## 6.1 Logging

A significant challenge in designing distributed software is diagnosing errors that arise: the effect of an error can be very far from its source, both in terms of position in the code base as well as in simulation state. To help with this, aether provides a structured logging service which aggregates log entries in the form of Lisp objects. Many of the aether primitives automatically emit these log messages, and log-entry allows the programmer to emit additional user-defined messages. Because these messages are retained as Lisp objects, they are relatively easy to query and filter, allowing the programmer to "zoom in" on particular process interactions in a sea of other information.

*Example* 11. In Figure 8 we provide an example of using some of the built-in logging facilities to verify that an implementation of the GHS algorithm runs using fewer messages than the predicted upper bound of  $5N \log N + 2E$ . See Appendix A for implementation details.

## 6.2 Dereferencing

Part of the conceit of distributed programming is that processes are unable to modify each others' states except by communication through agreed-upon protocols. To enforce this within a simulated context, where a programmer could use the host language's ability to inspect objects directly, we make a concerted effort to retain only the public addresses of processes. These are decoupled from the processes that own them, and hence they prevent their owner processes from being directly manipulated by the host language.

While necessary for the basic semantic validity of a distributed application, one frequently wants to dissolve exactly this barrier when debugging that application. Diagnosing and tracing errors ELS'21, May 03-04 2021, Online, Everywhere

almost always requires inspection of the states of *several* processes but which processes can only be known at runtime, and even then they are known only by their public address. We provide in aether an optional debugging utility for marrying public addresses to their owners, enabling this critical form of state tracing.

#### 6.3 Call tracing

In addition to understanding in detail the behavior of particular interactions, it is also useful to understand the aggregate statistics of interactions, as this has a direct impact on algorithm performance. To this end, aether provides a "tracing" mechanism that records a set of performance statistics for interprocess queries, e.g., number of targets in a batch send, or time spent waiting for a query to resolve (and, so, distinct from behavioral logging). This, too, can be further instrumented to suit a user's needs.

#### 7 AVENUES FOR IMPROVEMENT

We close with some comments about the framework's shortcomings and indicate clear areas for improvement.

#### 7.1 Granular processor simulation

While we have used the time-domain simulation features of an aether-like library to emulate the behavior of a family of processors at the microarchitecture level, we have not done so in the same project where communication and high-level algorithms are relevant. Because of this, we have not explored what a microarchitecturespecific DSL for algorithm specification looks like, having instead relied on Common Lisp to set process command semantics. One would surely learn a lot by pursuing this, and it would probably motivate nontrivial changes in the definition of an aether process, and perhaps to the framework generally.

Additionally, without a concrete specification of the "compute" power available to each process (even in an example!), it is impossible to faithfully simulate compute-bound behavior of systems. Instead, the simulations with aether described in this document tacitly assume that they are I/O-bound, so that the compute power does not matter relative to the network delay.

#### 7.2 **Procedure delineation**

In current practice we break "procedures" into sequences of "commands" in a way that is pleasing to the conscious programmer, but unstructured and unenforced from within the framework. It would be preferable to have a language-level method for delineating procedures, as well as for calling and returning from them.

Another side effect of manual manipulation of procedure bodies is that the validity of the command stack is not checked at compile time. Whereas the Common Lisp compiler can check whether a static procedure call comports to its type signature, we have not provided aether with the ability to warn a programmer whether the commands to be injected are defined at all, much less whether they match any advertised type signatures.

In a different but related direction, the locking system is not robust to malicious use. Without further programming intervention, a locked process continues to respond to all messages, including in particular messages sent by actors other than the lock holder and messages which may belong to some other critical section. Better

segmentation of procedures is not a prerequisite for improving on these problems, but it would make that work simpler.

# 7.3 A middleware DSL

Apart from setting out the details of a microarchitecture, it is a separate problem to specify a useful DSL for writing distributed algorithms in an actor-based framework. There are many obvious constraints to consider here, including flexibility, user-friendliness, and compiler-friendliness for mapping onto any potential microarchitecture embodiment. We have only begun to describe such a DSL here in the form of the framework around process.

A less obvious and extremely promising aspect of designing a DSL is the possibility of introducing a type system suitable for analyzing the interactions of processes and enabling a compiler targeting a microarchitecture to reason about programs and optimize them. A likely key starting point for such a project is a "closed world" hypothesis: process-based typing systems are difficult to govern when the processes can receive arbitrary queries from unknown third parties, but if the ingress points for such third parties are explicitly marked, initial progress seems possible.

#### 7.4 Granular network components

Network specification is somewhat underexplored. While it is currently very featureful—one can enact nontrivial routing strategies on nontrivial networks—it has not ever been pressured to become a "faithful" emulation of real networked electronics. For instance, aether presently measures the concept of network pressure by the packet count a node is processing, but the size and contents of those packets are not taken into account. Serialization and deserialization strategies would increase the fidelity of this aspect of the simulation. In a different direction, phenomena such as network noise (so: dropped packets, unpredictable delays, and so on) are not modeled. Modifying the simulator to support this, modifying the messaging system to be robust against it, and exposing a networking layer where the user is not obligated to use the message-passing system would each be a valuable project.

# ACKNOWLEDGEMENTS

The authors owe debts of gratitude to several collaborators, friends, and family. Separately and at different times, Charles Hadfield and Sergio Boixo both suggested that we take an interest in a particular form of topological quantum error correction, which ultimately led to this project and its sequels. We are grateful to Colm Ryan for providing valuable guidance early in the course of this project, and to Erik Davis and Michael J. Fischer for feedback on a draft of this manuscript. We would also like to thank Jade Lilitri for keeping our spirits "up". Lastly, the first author would like to thank Samrita Dhindsa for all manner of support as this work was carried out.

#### REFERENCES

- simple-actors The actor model implemented with closures. https://github.com/ j3pic/simple-actors. Accessed: 2020-09-01.
- [2] James Aspnes. Notes on theory of distributed systems, 2020.
- [3] Armando Castañeda, Carole Delporte-Gallet, Hugues Fauconnier, Sergio Rajsbaum, and Michel Raynal. Making local algorithms wait-free: the case of ring coloring. *Theory Comput. Syst.*, 63(2):344–365, 2019. doi: 10.1007/s00224-017-9772-y. URL https://doi.org/10.1007/s00224-017-9772-y.

- [4] Carole Delporte-Gallet, Hugues Fauconnier, Pierre Fraigniaud, and Mikaël Rabie. Distributed computing in the asynchronous LOCAL model. *CoRR*, abs/1904.07664, 2019. URL http://arxiv.org/abs/1904.07664.
- [5] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. ACM Trans. Program. Lang. Syst., 5 (1):66–77, 1983. doi: 10.1145/357195.357200. URL https://doi.org/10.1145/357195. 357200.
- [6] Klaus Harbo et al. CL-MUPROC Erlang-inspired multiprocessing in Common Lisp. https://common-lisp.net/project/cl-muproc/. Accessed: 2020-09-01.
   [7] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular
- [7] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In Nils J. Nilsson, editor, Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973, pages 235–245. William Kaufmann, 1973. URL http: //ijcai.org/Proceedings/73/Papers/027B.pdf.
- [8] A Yu Kitaev. Quantum computations: algorithms and error correction. Russian Mathematical Surveys, 52(6):1191–1249, dec 1997. doi: 10.1070/rm1997v052n06abeh002155. URL https://doi.org/10.1070% 2Frm1997v052n06abeh002155.
- [9] A. Yu. Kitaev. Quantum Error Correction with Imperfect Gates, pages 181– 188. Springer US, Boston, MA, 1997. ISBN 978-1-4615-5923-8. doi: 10.1007/ 978-1-4615-5923-8\_19. URL https://doi.org/10.1007/978-1-4615-5923-8\_19.
- [10] Leslie Lamport. A new solution of Dijkstra's concurrent programming problem. Communications of the ACM, 17(8):453-455, 1974.
- [11] Nathan Linial. Locality in distributed graph algorithms. SIAM J. Comput., 21(1): 193–201, 1992. doi: 10.1137/0221015. URL https://doi.org/10.1137/0221015.
- [12] Kat Marchán et al. memento-mori Robustness through actors, for Common Lisp. https://github.com/zkat/memento-mori. Accessed: 2020-09-01.
- [13] Eric C. Peterson and Peter J. Karalekas. A distributed blossom algorithm for minimum-weight perfect matching. *In preparation*, 2021.
- [14] Eric C. Peterson, Peter J. Karalekas, et al. dtqec/aether Distributed system emulation in Common Lisp. https://github.com/dtqec/aether. Accessed: 2021-04-23.
- [15] Max Rottenkolber et al. eugeneia/erlangen Distributed, asychronous message passing system for Clozure Common Lisp. https://github.com/eugeneia/erlangen. Accessed: 2020-09-01.
- [16] Naveen Sundar et al. naveensundarg/cl-actors An actor system for Common Lisp. https://github.com/naveensundarg/Common-Lisp-Actors. Accessed: 2020-09-01.
- [17] Jukka Suomela. Distributed Algorithms (online textbook). https://jukkasuomela. fi/da/, 2014.

#### A IMPLEMENTATION OF GHS ALGORITHM

Below we have an implementation of a canonical distributed algorithm using aether's process framework. As indicated by the docstrings, each command or message handler definition maps directly to a section of the pseudocode algorithm in the appendix of the GHS paper [5]. The snippet is truncated for brevity—for the full implementation, please refer to the file distributed-mst.lisp in the tests/examples directory of the aether repository [14].

```
(define-process-upkeep ((node fragment-node) now) (START)
  Section 1: Response to spontaneous awakening
  (process-continuation node `(START))
  (when (eql (slot-value node 'node-state) ':SLEEPING)
    (process-continuation node `(WAKEUP))))
(define-process-upkeep ((node fragment-node) now) (WAKEUP)
   Section 2: Procedure WAKEUP.
 (let ((address (process-public-address node)))
  (with-slots (adjacent-edges find-count fragment-level) node
      (when (eql (slot-value node 'node-state) ':SLEEPING)
        (let ((minimum-weight-edge
                 (find-minimum-weight-edge adjacent-edges)))
          (with-slots (edge-state)
               (gethash minimum-weight-edge adjacent-edges)
             (setf edge-state ':BRANCH find-count 0 fragment-level 0
                   (slot-value node 'node-state) ':FOUND)
            (send-message minimum-weight-edge
```

#### aether: Distributed system emulation in Common Lisp

```
(define-message-handler handle-msg-connect
    ((node fragment-node) (message msg-connect) time)
  "Section 3: Response to receipt of Connect() on edge j."
(let ((address (process-public-address node)))
(with-slots (fragment-level fragment-weight node-state) node
       (with-slots (edge level) message
          (with-slots (edge-state edge-weight)
              (gethash edge (slot-value node 'adjacent-edges))
            (cond
               ((< level fragment-level)
                (setf edge-state ':BRANCH)
                (send-message edge (make-msg-initiate
                                          :edge address
:level fragment-level
                                          :state node-state
                                          :weight fragment-weight))
               (when (eql node-state ':FIND)
 (incf (slot-value node 'find-count))))
((eql edge-state ':BASIC)
                (send-message address message))
               (t
                (send-message edge (make-msg-initiate
                                         :edge address
                                         :level (1+ fragment-level)
:state ':FIND
                                          :weight edge-weight))))))))
(define-message-handler handle-msg-initiate
     ((node fragment-node) (message msg-initiate) time)
  "Section 4: Response to receipt of Initiate(L, F, S) on edge j."
(let ((address (process-public-address node)))
     (with-slots (adjacent-edges find-count) node
       (with-slots (edge level state weight) message
          (setf (slot-value node 'best-edge)
      (slot-value node 'best-weight)
                                                              nil
                                                              most-positive-fixnum
                  (slot-value node 'fragment-level)
                                                            level
                  (slot-value node 'fragment-weight) weight
                  (slot-value node 'in-branch)
(slot-value node 'node-state)
                                                              edge
                                                              state)
          (let ((edges (find-specific-edges adjacent-edges
                                                     :desired-state ':BRANCH
                                                     :not-edge edge)))
            (flet ((payload-constructor ()
                        (make-msg-initiate :edge address :level level
                                                :state state :weight weight)))
               (send-message-batch #'payload-constructor edges))
          (loop :repeat (length edges) :when (eql state ':FIND)
      :do (incf find-count)))
(when (eql state ':FIND)
             (process-continuation node `(TEST))))))
(define-process-upkeep ((node fragment-node) now) (TEST)
    Section 5: Procedure TEST
  (with-slots (adjacent-edges test-edge) node
     :desired-state ':BASIC)))
        (setf test-edge minimum-weight-edge)
        (if minimum-weight-edge
            (send-message test-edge
                             (make-msg-test
                              :edge (process-public-address node)
            :level (slot-value node 'fragment-level)
:weight (slot-value node 'fragment-weight)))
(process-continuation node `(REPORT))))))
(define-message-handler handle-msg-test
  ((node fragment-node) (message msg-test) time)
"Section 6: Response to receipt of Test(L, F) on edge j."
(let ((address (process-public-address node)))
     (with-slots (adjacent-edges test-edge) node
       (with-slots (edge level weight) message
          (with-slots (edge-state) (gethash edge adjacent-edges)
            (cond
               ((> level (slot-value node 'fragment-level))
               ((send-message address message))
((/= weight (slot-value node 'fragment-weight))
(send-message edge (make-msg-accept :edge address)))
```

(t (when (eql edge-state ':BASIC) (setf edge-state ':REJECTED)) (if (or (null test-edge) (not (address= test-edge edge))) (send-message edge (make-msg-reject :edge address))
(process-continuation node `(TEST)))))))) (define-message-handler handle-msg-accept ((node fragment-node) (message msg-accept) time)
"Section 7: Response to receipt of Accept on edge j." (with-slots (adjacent-edges best-edge best-weight test-edge) node (with-slots (edge) message (with-slots (edge-weight) (gethash edge adjacent-edges)
 (setf test-edge nil) (when (< edge-weight best-weight) (setf best-edge edge best-weight edge-weight)) (process-continuation node `(REPORT)))))) (define-message-handler handle-msg-reject ((node fragment-node) (message msg-reject) time) 'Section 8: Response to receipt of Reject on edge j.' (with-slots (adjacent-edges) node (with-slots (edge) message (with-slots (edge-state) (gethash edge adjacent-edges)
 (when (eql edge-state ':BASIC) (setf edge-state ':REJECTED))
 (process-continuation node `(TEST)))))) (define-process-upkeep ((node fragment-node) now) (REPORT) Section 9: Procedure REPORT. (with-slots (best-weight find-count in-branch test-edge) node (when (and (zerop find-count) (null test-edge)) (setf (slot-value node 'node-state) ':FOUND) (send-message in-branch (make-msg-report :edge (process-public-address node) :weight best-weight)))) (define-message-handler handle-msg-report ((node fragment-node) (message msg-report) time) Section 10: Response to receipt of Report(w) on edge j." (with-slots (best-edge best-weight find-count in-branch) node (with-slots (edge weight) message (cond ((not (address= edge in-branch)) (decf find-count) (when (< weight best-weight) (setf best-weight weight best-edge edge))
(process-continuation node `(REPORT)))
((eql (slot-value node 'node-state) ':FIND) (send-message (process-public-address node) message)) ((> weight best-weight) (( ungert angle continuation node `(CHANGE-ROOT)))
((and (= weight best-weight) (= weight most-positive-fixnum))
(process-continuation node `(HALT)))))) (define-process-upkeep ((node fragment-node) now) (CHANGE-ROOT)
 "Section 11: Procedure CHANGE-ROOT." (let ((address (process-public-address node))) (with-slots (adjacent-edges best-edge fragment-level) node (with-slots (edge-state) (gethash best-edge adjacent-edges) (cond ((eql edge-state ':BRANCH) (send-message best-edge (make-msg-change-root :edge address))) (t

ELS'21, May 03-04 2021, Online, Everywhere

(send-message best-edge

(make-msg-connect :edge address :level fragment-level))

(setf edge-state ':BRANCH))))))

(define-message-handler handle-msg-change-root ((node fragment-node) (message msg-change-root) time) "Section 12: Response to receipt of Change-root." (process-continuation node `(CHANGE-ROOT)))

# A Scheme Foreign Function Interface to JavaScript Based on an Infix Extension

Marc-André Bélanger Université de Montréal Montréal, Québec, Canada marc-andre.belanger@umontreal.ca

#### ABSTRACT

This paper presents a JavaScript Foreign Function Interface for a Scheme implementation hosted on JavaScript and supporting threads. In order to be as convenient as possible the foreign code is expressed using infix syntax, the type conversions between Scheme and JavaScript are mostly implicit, and calls can both be done from Scheme to JavaScript and the other way around. Our approach takes advantage of JavaScript's dynamic nature and its support for asynchronous functions. This allows concurrent activities to be expressed in a direct style in Scheme using threads. The paper goes over the design and implementation of our approach in the Gambit Scheme system. Examples are given to illustrate its use.

### CCS CONCEPTS

- Software and its engineering  $\rightarrow$  Interoperability; Compilers;

# **KEYWORDS**

Foreign function interface, Macros, JavaScript, Scheme

#### **1 INTRODUCTION**

In this paper we relate our experience designing, implementing and using a Foreign Function Interface (FFI) in the context of a Scheme implementation hosted on JavaScript. Our system avoids the cumbersome syntax and boilerplate declarations found in typical FFIs and offers a lightweight interface that is both easy to use and expressive.

Cross-language interoperability is a desirable feature of any language implementation. It allows building applications using multiple languages and expressing each part with the most appropriate language. Important factors in the choice of language are the availability of libraries and APIs for the tasks to be done. In a Scheme implementation running in a web browser the support of a JavaScript FFI opens up many interesting possibilities such as accessing the Document Object Model (DOM) and handling events in Scheme code. Marc Feeley Université de Montréal Montréal, Québec, Canada feeley@iro.umontreal.ca

FFIs are notoriously implementation-dependent and code using a given FFI is usually not portable. Consequently, the nature of FFI's reflects a particular set of choices made by the language's implementers. This makes FFIs usually more difficult to learn than the base language, imposing implementation constraints to the programmer. In effect, proficiency in a particular FFI is often not a transferable skill.

In general FFIs tightly couple the underlying low level data representation to the higher level interface provided to the programmer. This is especially true of FFIs for statically typed languages such as C, where to construct the proper interface code the FFI must know the type of all data passed to and from the functions. As a simple example, here is a program using Gambit Scheme's C FFI[9] to interface to the C library's 1 dexp(x, y) function computing  $x \times 2^y$ :

(c-declare "#include <math.h>") ;; get ldexp C prototype (define ldexp (c-lambda (double int) double "ldexp")) (println (ldexp 10.0 -3)) ;; prints 1.25

Note the use of type specifiers in the c-lambda to indicate the type of the arguments (double and int) and result (double). The FFI defines some mapping between the Scheme data and the C types, and raises an error for incompatible types (e.g. the Gambit C FFI will raise an error when a C double is expected and a value other than a Scheme inexact real is used). There is even wider variation in how different FFIs handle more complex constructs like variadic functions, higher order functions, multiple return values, pointers, arrays, structures, classes, continuations and threads (a notorious hard case is interfacing to the C library's qsort function which uses universal pointers and a callback). FFIs are also difficult to use from the REPL and interpreted code, if at all possible.

There is an opportunity to simplify the interfacing code when the host language is a dynamically typed language that supports dynamic code evaluation (i.e. eval). By interfacing through *expressions* rather than the *function* level, we can leverage a Scheme reader extended with infix notation support to generate and evaluate host language expressions. Our work proposes a new FFI design based on those ideas for interfacing Scheme and JavaScript that is easy to use in the common case yet also supports more complex use cases including asynchonous execution and threads. Our design enables *natural* cross-language programming that programmers can pick up quickly. We first go over our design and its implementation, followed by an exposition of potential uses.

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

ELS'21, May 03-04 2021, Online, Everywhere

<sup>@</sup> 2021 Copyright held by the owner/author(s). https://doi.org/10.5281/zenodo.4711424

https://doi.org/10.0201/2010d0.111121

#### 2 SYNTAX AS INTERFACE

Programmers using an FFI are expected to be knowledgeable in both the foreign and native languages; in our case JavaScript and Scheme respectively. We use the term *native* as the opposite of *foreign* and **not** to mean machine code.

Some Lisp/Scheme FFIs cast the foreign constructs to a parenthesized syntax to make them usable within the native language. As an illustrative example, JScheme[1] interfaces to Java using functions whose names follow the "Java Dot Notation" containing special characters: (.method obj ...) for calls to methods, (Constructor. ...) for calls to constructors, Class.field\$ for static field members, etc. For example: (.println System.out\$ "hello") performs the same operation as the Java code System.out.println("hello").

This decoupling of syntax and semantics can be confusing to the programmer who must essentially write semantically foreign code but with a more or less contrived mapping of foreign constructs to the native syntax (e.g. there are 9 different but similar looking Java Dot Notation rules in JScheme). This adds an intellectual burden and prevents cutand-paste of existing foreign code snippets and idioms into native code. Our point of view is that the foreign language's syntax should be preserved as much as possible in native code in order to express naturally the foreign constructs which don't map well to the native language, such as method calls and accessing member fields. Moreover, the difference in syntax helps distinguish which of the two languages is being used at various points in the program. This works particularly well in languages such as Scheme and JavaScript that have easily distinguishable syntaxes: prefix and infix notation.

#### 2.1 Scheme Infix eXtension

In our FFI, infix expressions within Scheme code are considered to be foreign code. A single backslash must precede the infix expression to *escape* temporarily from the prefix syntax. As a simple example the following native code will print the current day of the week twice to the JavaScript console:

```
(define date \Date().slice(0, 15))
(define day (substring date 0 3))
\console.log(`day)
```

```
\console.log(`(substring \Date().slice(0, 15) 0 3))
```

The first line calls out to JavaScript to retrieve the date as a Scheme string (stripped of the time of day using the JavaScript slice method). The second line extracts the day of the week with a Scheme call to substring. The third line reads the Scheme variable day and uses the JavaScript console.log method to print it out. The last line is similar but with an inline Scheme call to substring. Note the use of a backquote to switch back to the prefix syntax temporarily within the infix form. The expressions marked with a backquote are evaluated in the Scheme environment whereas the rest of the infix form is evaluated in the JavaScript global environment. This is why the identifiers Date and console refer to JavaScript globals and the identifiers date, day, and substring refer to Scheme variables. As shown in the last line it is possible to nest prefix and infix forms. The FFI's implementation is simplified by Gambit's existing reader which supports reader macros and which has a default setup to invoke the Scheme Infix eXtension (SIX) parser when a backslash is encountered. After a complete infix form is read the reader continues parsing using the prefix syntax. Note that the Gambit reader does not use \ to escape symbols as in Common Lisp and some other Scheme implementations. Vertical bars are the only supported symbol escaping syntax.

Similarly to other reader macros, the SIX parser constructs an s-expression representation of the infix form's AST. This representation can easily be viewed by quoting the infix form and pretty printing it, for example (pp '\console.log(`day)) prints:

The system achieves the JavaScript FFI semantics by defining a six.infix macro, as explained in the next section.

The grammar supported by the SIX parser is given in the Gambit user manual and the details are mostly uninteresting here. Two aspects are nevertheless noteworthy.

First of all the grammar combines syntactic forms from multiple languages including C, JavaScript, Python and Prolog (without implementing any of those grammars fully) and has a few extensions, such as `X to parse `X using the prefix syntax. The choice of using ` to switch back to prefix is motivated by the fact it is seldom used in the grammars of infix based languages. It is also evocative of Scheme's quasiquote notation, but with \ and ` in place of ` and ,.

The infix operators have the precedence and associativity of the JavaScript language. The SIX parser was originally designed to be used in an undergraduate course on compilers to easily construct ASTs. Even though the parser supports multiple syntaxes, it is problematic to force the parser to restrict the syntax to a subset, or to extend the grammar itself, as this introduces a phasing problem. It would require introducing a read-time mechanism (such as Racket's **#lang** feature[10]) to select the grammar, which is something we want to avoid so that a source file can combine code for multiple host languages (possibly guarded by a cond-expand dependent on the compilation target).

Secondly, supporting infix forms within prefix forms required a few syntactic concessions related to identifier syntax and whitespace handling. Whitespace is significant when it is outside of infix form parentheses and braces. The expression (list 1+2 3+4-5) evaluates to (3 2) whereas (list (1 + 2) 3+4 -5) evaluates to (3 7 -5). This is important to keep the syntax lightweight. In an earlier version of the SIX parser whitespace was not significant and infix forms were required to end with a semicolon, but this was less visually pleasing, so the current parser makes the semicolons optional outside of infix braces.

The use of a Scheme identifier immediately after a ` can cause issues because Scheme allows many more characters in a symbol than in JavaScript. Consequently the SIX parser will have the surprising behaviour of treating  $\x-f(y)$  as an expression containing a reference to the identifier x-f. The programmer can circumvent this issue by using (x - f(y)), (x)-f(y), or  $\x-x(x)-f(y)$  that unambiguously reference x.

#### 2.2 The six.infix Macro

As with any FFI there is a need to bridge the semantic gap between the native and foreign languages. An important aspect is the conversion of values between the languages so that a value generated in the Scheme code gets converted to the appropriate value in JavaScript. Conversion from JavaScript values to Scheme values is also needed for returning results back to Scheme. Conversions in both directions are also needed to allow JavaScript code to call Scheme procedures. Scheme procedures and JavaScript functions are themselves considered to be data with a mapping to the other language.

The FFI must also bridge the semantic gap of the control features of the languages. Specifically, Gambit Scheme supports proper tail calls, first class continuations, and threading (SRFI-18 and SRFI-21) thanks to a green thread scheduler implemented in Scheme using first class continuations. JavaScript does not support these features. However it does offer asynchronous functions which are similar to threading in that they allow concurrent activities. The details of the mapping between languages is explained in Section 4.

The FFI's semantics are implemented using a suitable definition of the macro six.infix. This macro traverses the SIX AST of the expression and extracts all the Scheme expressions wrapped in a quasiquote. These are the *Scheme parameters* of the expression. These Scheme parameters are given names by generating JavaScript identifiers (\_\_\_1, \_\_\_2, ...). The SIX AST gets translated to a string of JavaScript code representing an asynchronous function whose parameters are these identifiers and where the body of the function computes the expression using these identifiers. As an example, the SIX expression \(`s).repeat(10), that can be used to repeat a Scheme string s 10 times, leads to the creation of this JavaScript function:

```
async function (___1) { return ___1.repeat(10); }
```

Using the standard data mapping this JavaScript function will be converted to a Scheme procedure p and (`s).repeat(10) expands to code that performs the call  $(p \ s)$ .

There is a concern with this approach related to phasing. The six.infix macro is expected to work in compiled code and also through the REPL and Scheme's eval. This is solved by a JavaScript eval of the function definition to create dynamically the JavaScript function and the corresponding Scheme procedure. This dynamic evaluation is required because at Scheme compilation time the JavaScript environment has not yet started running (for example the Scheme program could be compiled on a desktop computer ahead of time).

Scheme	JavaScript
#!void	undefined
()	null
#f/#t	false/true
fixnum, e.g. 12	number, e.g. 12
flonum, e.g. 1.2	_Flonum
bignum, e.g. <b>99999999</b>	_Bignum
ratnum, e.g. $1/2$	_Ratnum
cpxnum, e.g. 1+2i	_Cpxnum
character, e.g. <b>#\a</b>	_Char
pair, e.g. (1 . 2)	_Pair
string, e.g. "abc"	_ScmString
symbol, e.g. abc	_ScmSymbol
keyword, e.g. abc:	_ScmKeyword
structure, port, table,	_Structure
vector, e.g. <b>#(1 2)</b>	Array, e.g. [1,2]
u8vector, e.g. <b>#u8(1 2)</b>	_U8Vector
other homogeneous vectors	
f64vector, e.g. <b>#f64(1.2 3.4)</b>	_F64Vector
procedure of n parameters	parameterless function

# Figure 1: GVM's representation of the Scheme types in JavaScript

To avoid a call to the JavaScript eval at every evaluation of a given six.infix call site, the expansion uses a simple caching mechanism to remember the Scheme procedure obtained during the first evaluation. A Scheme box is used as a cache. It initially contains a string of the JavaScript function definition, and on the first execution it is mutated to contain the result of calling the JavaScript eval on that string and converting the result to a Scheme procedure. This is done by the procedure ##host-function-memoized which takes the box as its sole parameter and returns the Scheme procedure. So to be precise, \(`s).repeat(10) expands to:

((##host-function-memoized '#&"async function (\_\_1) { return \_\_1.repeat(10); }") s)

with **##host-function-memoized** defined as:

```
(define (##host-function-memoized descr)
 (let ((x (unbox descr)))
  (if (string? x)
        (let ((host-fn (##host-eval-dynamic x)))
            (set-box! descr host-fn)
            host-fn)
            x)))
```

# **3 GAMBIT VIRTUAL MACHINE**

Before discussing in more detail the implementation of the FFI it is important to briefly go over the Gambit Virtual Machine (GVM) which is the Gambit compiler's intermediate language. Thanks to this abstraction it is possible to retarget the compiler with moderate effort, and indeed there are backends for several languages both high-level (C, JavaScript, Python, ...) and machine languages (x86, arm, riscv, ...). In

the case we are concerned with here, the compiler translates GVM instructions to JavaScript.

To support the GVM each Scheme type is mapped to a corresponding representation as a JavaScript type. The mapping for some types is direct when a JavaScript type supports the same operations. This is the case of **#f**, **#t**, (), **#!void**, vectors, and fixnums which are mapped to **false**, **true**, **null**, **undefined**, **Array**, and numbers respectively. It is important for performance to use JavaScript numbers as a representation of fixnums so that operations on small integers can be done without an extra layer of boxing/unboxing.

Figure 1 gives a list of the Scheme types and their representation in JavaScript. All the JavaScript classes supporting the GVM are prefixed with \_. In most cases a class is used to group the information related to a type, for example a \_Pair contains the car/cdr fields of a Scheme pair and a \_Char contains the Unicode code of a Scheme character. Scheme strings are not mapped to JavaScript strings because those are immutable. Instead a \_ScmString contains an Array of Unicode codes. Scheme symbols are not mapped directly to the (relatively new) JavaScript Symbol type because the GVM stores a hash code and other information in symbols.

Scheme procedures are mapped to parameterless JavaScript functions. What is peculiar about this mapping is that the JavaScript function is used to represent a control point in the code, similarly to a code label in assembly language. These control point functions take no parameters and return a new control point function or null. Jumping from one point in the code to another is the job of the *trampoline* which is the loop while (pc) pc = pc(); where pc is the current control point function. A control point function can jump to a new control point by returning this new control point to the trampoline which will transfer control to it. This approach is needed to support tail calls properly and also to perform strict checking of the parameter count (JavaScript does not check or report parameter count mismatches). When Scheme code calls a procedure the parameters, a parameter count and a return address (another control point function) will be stored in registers and the stack of the GVM (JavaScript global variables) before returning the procedure's control point function to the trampoline to jump to it.

Control point functions, the trampoline and an explicit representation of the stack are the basic elements needed to implement closures, continuations and threads similarly to implementing these types in machine language. The thread type is a structure with several fields, one of which is the continuation of the thread. When a thread needs to be suspended, its current continuation is captured and stored in the thread structure so that it can be invoked later when the thread's execution needs to resume. Mutexes and condition variables are structures which contain a queue of threads, which are the threads blocked on them. A thread scheduler implemented in Scheme keeps track of a queue of runnable threads and moves threads out of this queue when the threads block on a mutex or condition variable. The scheduler is preemptive, forcing the current runnable thread to the end of the runnable thread queue when it has been the currently

Scheme		JavaScript	
#!void	$\longleftrightarrow$	undefined	
#f/#t	$\longleftrightarrow$	false/true	
fixnum, flonum	$\longleftrightarrow$	number, e.g. 12, 1.2	
bignum, e.g. <b>99999999</b>	$\longrightarrow$	number, e.g. <b>99999999</b>	
ratnum, e.g. $1/2$	$\rightarrow$	number, e.g. 0.5	
character, e.g. <b>#\a</b>	$\longrightarrow$	number, e.g. 97	
exact integer, e.g. 42	$\leftarrow -$	BigInt, e.g. 42n	
string, e.g. "abc"	$\longleftrightarrow$	string, e.g. "abc"	
symbol, e.g. abc	$\rightarrow$	string, e.g. "abc"	
keyword, e.g. abc:	$\rightarrow$	string, e.g. "abc"	
vector, e.g. <b>#(1 2)</b>	$\longleftrightarrow$	Array, e.g. [1,2]	
()	$\longrightarrow$	Array, e.g. []	
pair, e.g. (1 . 2)	$\rightarrow$	Array, e.g. [1,2]	
pair, e.g. (1 2 3)	$\longrightarrow$	Array, e.g. [1,2,3]	
table	$\rightarrow$	object, e.g. <b>{a:1,b:2}</b>	
u8vector	$\longleftrightarrow$	Uint8array	
other homogeneous vectors			
f64vector	$\longleftrightarrow$	Float64Array	
procedure of n parameters	$\longleftrightarrow$	function of n parameters	

Figure 2: FFI mapping of types between Scheme and JavaScript

running thread for more than a small time interval (typically 0.01 second).

# 4 FFI MAPPING OF TYPES

The FFI defines the mapping of types between the native and foreign languages. The mapping is designed to be convenient and intuitive to allow commonly used values to be be mapped to the other language to what is expected by a programmer, and be consistent. The conversions need not have a link with the GVM's mapping of Scheme to JavaScript types, which was chosen to achieve good execution speed of pure Scheme code. The mapping is given in Figure 2.

# 4.1 Simple Types

The conversion functions which implement this mapping, the JavaScript functions \_scm2host and \_host2scm, are called when there is an inter-language call when converting the parameters and the result. It is desirable for values to be invariant when they are sent to an identity function in the other language (i.e. that the round-trip does not change the value in the sense of equal?). However this is not possible for all values. The Scheme values #!void, #f, #t, strings and homogeneous vectors are bidirectionally mapped to the JavaScript values undefined, false, true, strings and typed arrays respectively, so they have ideal round-trip behaviour. Scheme vectors are bidirectionally mapped to be recursively converted. So the round-trip behaviour of vectors/Arrays will depend on the round-trip behaviour of their elements.

Numbers need to be mapped carefully because JavaScript has two numerical types, *number* and BigInt, that correspond to Scheme's inexact reals and exact integers respectively.
However, they are not consistently used that way in typical code (for example JavaScript arrays are almost never indexed with BigInt which is a fairly recent addition to the language). For that reason it is more convenient for Scheme exact integers to be mapped to JavaScript numbers. When a JavaScript number is converted to Scheme, it will become a fixnum value if it has an integer value falling in the fixnum range, otherwise (if it has a fractional part or is outside the fixnum range) it becomes a flonum value. When a JavaScript BigInt is converted to Scheme, it will become an exact integer (either a fixnum or bignum depending on its value). Scheme bignums and rationals are also mapped to numbers. Scheme characters are mapped to the number that is their Unicode code.

Scheme symbols and keywords are converted to JavaScript strings. Scheme pairs and lists are converted to JavaScript Arrays with recursively converted elements.

## 4.2 Procedures

Scheme procedures are mapped bidirectionally to JavaScript functions and they accept the same number of parameters. In the conversion from one language to the other, calls to the appropriate conversion functions are added to convert the parameters and the result. In other words, when a Scheme procedure p is converted to the JavaScript function f, a call of f in JavaScript must pass JavaScript values that will be converted to the corresponding Scheme value for processing by p. When p delivers its Scheme result it will be converted to JavaScript and returned for the call to f. The situation is similar for a JavaScript function that is converted to Scheme.

Asynchronous functions and the Promise type were added to JavaScript to avoid the deeply nested Continuation Passing Style (CPS), aka. "callback hell", that commonly occurs when using CPS to perform asynchronous processing. In a language with threads, such as Gambit Scheme, asynchronous processing can instead be expressed in a direct style using threads that wait for the availability of the next piece of data or event. Our FFI implements a mapping of JavaScript promises and asynchronous functions to Scheme threads, making asynchronous processing easier to use.

It is important to realize that, due to the presence of threads, Scheme procedures may take an arbitrary long time to complete if they block the current thread on an I/O operation or mutex or condition variable until some event unblocks the thread and allows the procedure to return. So for a smooth integration with the JavaScript execution model, Scheme procedures must be mapped to JavaScript asynchronous functions. Similarly, a JavaScript asynchronous function may take an arbitrarily long time to deliver a result, so if Scheme code calls a JavaScript asynchronous function it may cause the current Scheme thread to effectively block. However, this must not happen deep inside JavaScript code because in that case the Scheme thread scheduler itself would be unable to continue scheduling runnable threads (in effect the scheduler itself would be blocked).

This is solved by using the *Promise API* and a JavaScript to Scheme callback that notifies the Scheme thread scheduler

when a promise is *settled* (either *fulfilled* with a value or rejected with an exception). An asynchronous JavaScript function f is converted to a Scheme procedure p that ends with a call to the **##scm2host-call-return** procedure that receives the promise result of the asynchronous function. The Scheme thread must wait for the promise to be settled. This is achieved with a mutex that is initially in a locked state and that the Scheme thread tries to lock (thus blocking at that point). When the promise is settled a Scheme callback is called which stores the result (in the mutex *specific* field) and unlocks the mutex, allowing the Scheme thread to determine if the result is normal or an error. The following code shows how this synchronization is implemented:

## // JavaScript side

7

```
function _when_settled(promise, callback) {
```

```
function onFulfilled(value) {
    // call the Scheme callback asynchronously
    _async_call(false, false, // no result needed
                callback.
                [_host2scm([value])]);
 }
 function onRejected(reason) {
   // call the Scheme callback asynchronously
    _async_call(false, false, // no result needed
                callback,
                [_host2scm(reason.toString())]);
 }
 promise.then(onFulfilled, onRejected);
;; Scheme side
(define (##scm2host-call-return promise)
  (let ((mut (make-mutex)))
    ;; Setup mutex in locked state
    (mutex-lock! mut)
```

;; Add callback for when promise is settled (when-settled ;; defined in JS as above promise (scheme ;; pass-through (see next section) (lambda (result) ;; callback (mutex-specific-set! mut result) ;; wake up waiting Scheme thread (mutex-unlock! mut))))

```
(mutex-lock! mut) ;; Wait until settled
(mutex-unlock! mut) ;; Avoid space leak
(let ((msg (mutex-specific mut)))
  (if (vector? msg)
                         ;; Promise was:
      (vector-ref msg 0) ;; fulfilled
      (error msg)))))
                         ;; rejected
```

Non asynchronous JavaScript functions can be encountered by \_host2scm in a variety of situations, including converting data structures containing functions and global functions such as alert and fetch. The above code is a slight simplification of the actual code which must also handle calling a non asynchronous JavaScript function which (typically) does not return a promise. This is done by dynamically testing the type of ##scm2host-call-return's parameter to determine if it is a promise.

Because the handling of SIX expressions creates a definition of a JavaScript asynchronous function, a call to that function always returns a promise. The expansion of the six.infix macro will contain a Scheme call of the JavaScript asynchronous function converted to Scheme. Consequently the Scheme thread will implicitly wait for the asynchronous JavaScript processing to complete before continuing. This decouples the control flow of the Scheme thread scheduler and the JavaScript task queue, allowing other Scheme threads to run while the asynchronous call is executing.

A similar decoupling is necessary for Scheme procedures that are converted to JavaScript asynchronous functions. When called, the JavaScript function creates a promise and an **Array** packaging the Scheme procedure to call, the parameters and a JavaScript callback, and calls <u>\_async\_call</u> to add this **Array** to a *callback queue*. A dedicated *callback loop* Scheme thread reads this queue, performs the corresponding Scheme call and settles the promise accordingly (fulfilled or rejected depending on whether a Scheme exception was raised) by calling the JavaScript callback.

## 4.3 Pass-Through Types

In some cases it is not desirable for values to be converted implicitly according to the previously described rules. An important case is when a value created by one language needs to be stored by the other language for passing back to the originating language unchanged at a later time. For this purpose the FFI defines two *pass-through* types which are treated specially by the conversion functions, represented by the \_Scheme and \_Foreign JavaScript types. These types simply box a Scheme and JavaScript value respectively. In Scheme a\_Scheme value is constructed with the procedure call (scheme val). In JavaScript a \_Foreign value is constructed with the function call foreign(val).

The \_scm2host conversion function acts as the identity function when passed a \_Scheme value. Similarly the \_host2scm conversion function acts as the identity function when passed a \_Foreign value. However, when passed a \_Foreign, the \_scm2host conversion function unboxes the value to get back the JavaScript value originally passed in the call foreign(val). Similarly, when passed a \_Scheme, the \_host2scm conversion function unboxes the value to get back the Scheme value originally passed in the call (scheme val). With these rules it is possible for the programmer to achieve ideal round-trip behaviour (in the eq? sense) for any value by inserting explicit calls to scheme and foreign when the normal conversion must be disabled.

#### Marc-André Bélanger and Marc Feeley

The foreign function can also be used to bypass the implicit promise synchronization. If the programmer wants the calling Scheme thread to continue execution without waiting for the asynchronous call to complete then a **Promise** object can be returned to Scheme by wrapping it in a call to foreign. Waiting for a promise **p** to be settled is as simple as writing \`**p** as shown in the following example:

;; define a JS function that takes time to complete

\sleep=	functior	n (ms	;) {			
	return	new	Promise(funct	cion	(resolve)	{
			setl	Cimed	out(resolve	e,ms);
			});			
}						

(define p \foreign(sleep(5000))) ;; does not wait

\sleep(1000) ;; pause Scheme execution for 1 sec

#### $\ \ ;;$ waits for the remaining part of 5 secs

The scheme procedure can also be used to write JavaScript code that directly accesses the GVM's value representation. This can be useful to implement special operations or special purpose conversions of Scheme values. A \_Scheme value has a scmobj field that contains the Scheme object (more precisely its GVM representation using JavaScript objects). For example \`(scheme "abc").scmobj.codes[1] evaluates to 98, which is the Unicode code of the second character of the Scheme string "abc".

For convenience, any Scheme object not mentionned in Figure 2 is converted to a \_Scheme value. Similarly, any JavaScript object not mentionned is converted to a \_Foreign value. As a consequence, a data conversion between languages always succeeds. For example a Scheme complex number will be converted to a \_Scheme value, allowing the GVM representation to be accessed using JavaScript, as shown in the following code:

(define num 1+2i)	
(println \(`num).scmobj.real)	;; prints 1
\(`num).scmobj.imag=9	;; mutate object
(println num)	;; prints 1+9i

This shows that by accessing its GVM representation the complex number can be mutated even though it is a Scheme constant. This clearly exposes implementation details to the programmer, which is a double edged sword (useful in some contexts but dangerous if not used properly). A programmer should mainly rely on the FFI mapping shown in Figure 2 and seldom if ever use the GVM object representation details given in Figure 1 that is more likely to change in future versions of Gambit or when special compiler options are used (indeed the Gambit compiler's compactness setting may cause the use of shorter names for the fields of GVM objects).

# 5 EXAMPLES

We can now show various examples that illustrate the qualities of our design. We begin with a trivial *hello world* program and move on to more involved use cases. We hope these examples convincingly show the simplicity of use and terseness of the code interfacing Scheme and JavaScript. The reader may want to try the examples on the online Gambit Scheme REPL at https://gambitscheme.org/try/ using a *cut-and-paste* of the code shown.

# 5.1 Interfacing with the DOM

One of the most obvious use cases for the FFI is interfacing with the DOM. The browser effectively acts as a graphical user interface for Scheme, just as it would for JavaScript. As a first example, let's consider inserting a DOM node in the page to render some text, as shown in Figure 3.

```
(define msg "<h1>Hello!</h1>")
(define top "afterbegin")
```

\document.body.insertAdjacentHTML(`top, `msg)



Figure 3: Simple modification of the DOM.

This defines the HTML code to insert and then calls the **insertAdjacentHTML** method of the *body* element of the page. Note that this directly invokes the JavaScript DOM API without writing Scheme wrappers. This code should feel natural and be self-explanatory for any programmer knowledgeable in JavaScript and Scheme, a stated goal of our design. The example in Figure 3 could have been written as

(define body \document.body) ;; <-- foreign object \(`body).insertAdjacentHTML(`top, `msg)

where the *body* element is stored as a foreign object in Scheme. Such use still feels natural and allows for modularization in more involved code, such as when developing a library.

# 5.2 Event Handling

Event handlers and listeners constitute the foundation of interactive browser user interfaces. It is easy to register a Scheme procedure as a callback to an event listener. Because of the implicit mapping of Scheme procedures to JavaScript functions, any Scheme procedure can be used as a callback to process DOM events triggered on the page. The following code will track the mouse movements and log the x, y coordinates to the console as the mouse is moved.

```
(define (handler evt)
  (update \(`evt).clientX \(`evt).clientY))
(define (update x y)
  \console.log(`(object->string (list x: x y: y))))
```

\document.addEventListener("mousemove", `handler)

# Figure 4: Registering a Scheme procedure as an event listener callback.

The handler event listener callback passes the *mousemove* event's *clientX* and *clientY* coordinates to the update procedure. The latter simply logs the coordinates to the console using JavaScript's console.log.

## 5.3 Interfacing with Libraries

Modern web apps typically make use of multiple external JavaScript libraries. Our FFI allows Scheme code to easily interface to such libraries. JQuery is a widely used library that facilitates interacting with the DOM. Figure 5 is a representative example.

```
(define html
 (string-append
    "<button>Toggle visibility</button>"
    "First paragraph"
    "Second paragraph"))
```

\document.body.insertAdjacentHTML("beforeend", `html)

```
(define (toggle evt)
 (let ((hidden \$("p.hidden"))
        (visible \$("p.visible")))
        \((`hidden).removeClass("hidden")
            .addClass("visible")
            .toggle())
        \((`visible).removeClass("visible")
            .addClass("hidden")
            .toggle())))
```

```
\$("button").click(`toggle)
```

## Figure 5: Interfacing Scheme with JQuery through the FFI.

After defining and inserting the HTML, the toggle event handler is defined and assigned to the *click* event using JQuery. The event handler uses JQuery's \$ function to find the elements corresponding to a *selector*. In the example, the selectors find every element with class *hidden* or *visible*. The toggle handler uses the JQuery removeClass, addClass and toggle methods to hide or show the element in question. An element with class *hidden* will see its class change from *hidden* to *visible*, and its actual visibility toggled by JQuery's toggle method, and vice versa.

The SIX expressions in the body of the handler's let are wrapped in parentheses to allow writing a multi-line

expression which is very similar to the conventional style used in JavaScript. The pattern of selecting and mutating DOM elements is very common and forms the basis of rich user interfaces and web applications and is clearly easily achieved with our FFI design.

# 5.4 Asynchronous Updates

Asynchronous processing is a useful approach to decouple the UI and application logic. In this example we use JavaScript's fetch API to get resources from other web servers, specifically the weather reports of New-York and Miami. The program in Figure 6 uses the JavaScript fetch asynchronous function to request a JSON formatted weather report from the server forecast.weather.gov. The temperature is shown for each city and is updated every 10 seconds.

The updating is handled for each city by creating one Scheme thread per city. Each thread loops on the operations that fetch the JSON weather report, transfers the temperature to the DOM, and sleeps for 10 seconds. Note that the Scheme code hides from view inside the **fetch-json** procedure the promises and asynchronous functions that are operating at the JavaScript level.

```
(define (fetch-json url)
  \fetch(`url).then(function (r) { return r.json(); }))
(define (url loc)
  (string-append
   "https://forecast.weather.gov/MapClick.php?"
   "lat=" (cadr loc) "&lon=" (caddr loc)
   "&FcstType=json"))
(define (html loc)
  (string-append "<h3><span id='" (car loc)
"'>?</span> F -- " (car loc) "</h3>"))
(define (show-weather loc)
  \(document.body
     .insertAdjacentHTML("beforeend", `(html loc)))
  (let ((elem \document.getElementById(`(car loc))))
    (thread (lambda ()
              (update-weather elem loc 10)))))
(define (update-weather elem loc period)
  (let loop ()
    (let ((json (fetch-json (url loc))))
      \(`elem).innerText=`(temperature json)
      (thread-sleep! period)
      (loop))))
(define (temperature json)
  \(`json).currentobservation.Temp)
(for-each show-weather
          '(("New-York" "40.78333" "-73.96667")
                         "25.76000" "-80.21219")))
            ("Miami"
```

Figure 6: Asynchronously updating weather reports using threads.

# 5.5 Parallelism

Figure 7 is our last example. It shows how the use of threads for asynchronous processing can improve performance. The program starts off by defining the future and touch forms of Multilisp[11] to easily express parallelism. They are the basis of the pmap procedure which is like map but processes all elements concurrently. The rest of the code uses pmap to fetch 43 images<sup>1</sup> asynchronously and adds them to the web page. This program is an order of magnitude faster than one using plain map because it takes advantage of the inherent external parallelism in the web servers and network.

```
(define-syntax future
 (lambda (stx)
    (svntax-case stx ()
      ((future expr)
      #'(thread (lambda () expr)))))
(define touch thread-join!)
(define (pmap f lst)
                     ;; "parallel" map
  (map touch (map (lambda (x) (future (f x))) lst)))
(define memo
 (string-append
  "Scheme_-_An_interpreter_for_extended_"
  "lambda_calculus.djvu"))
(define (page n)
  (string-append
  "https://upload.wikimedia.org/wikipedia"
  "/commons/thumb/1/1e/" memo
  "/page" (number->string n) "-593px-" memo ".jpg"))
(define (fetch-blob url)
  \fetch(`url).then(function (r) { return r.blob(); }))
(define (->URL blob)
 \URL.createObjectURL(`blob))
(define (show url)
  \document.body.insertAdjacentHTML(
  "beforeend".
  "<img src='"+(`url)+"' width=200px>"))
(define images
 (pmap (lambda (n) (->URL (fetch-blob (page n))))
        (iota 43 1)))
```

(for-each show images)

Figure 7: Downloading a set of images in parallel.

# 6 RELATED WORK AND CONCLUSION

C FFIs are offered by Scheme implementations such as Racket[5], Chez Scheme[8], Larceny[12], Bigloo[14] and Gambit Scheme[9]. These essentially propose a domain-specific

<sup>&</sup>lt;sup>1</sup>The pages of the original Scheme report!

#### A Scheme Foreign Function Interface to JavaScript Based on an Infix Extension

ELS'21, May 03-04 2021, Online, Everywhere

language (DSL) to facilitate interfacing through foreign function declarations, something we wish to avoid.

FFIs to dynamically typed languages exist in languages and software such as Hop[15], Haskell[7], Kotlin[2], PharoJS[6], Pyodide[4] or Racket[5]. Of these, Haskell's *Foreign Expression Language*, PharoJS and Racket's facilities fall into the DSL category. Hop, Pyodide and Kotlin allow seemingly more natural access to JavaScript code. This is facilitated by Kotlin and Python's syntactic similarity to JavaScript. However, these methods are in essence either like writing Python to a string and passing it to Python's eval in the case of Pyodide (which is essentially CPython compiled to WebAssembly), or evocative of the C FFI function declarations in the case of Kotlin. JScheme[1] and LIPS[3] offer yet another way of interfacing with JavaScript by leveraging a *dot notation*, wherein Java or JavaScript semantics is mapped to Scheme through syntactic convention.

Hop's ability to syntactically distinguish computations that should occur on the server or the client resembles our escaping mechanism to switch between languages. This is reminiscent of quotation/antiquotation in SML[16], which allows to splice host-language expressions in foreign code. The `C (*Tick* C)[13] language also offers a mechanism using a backquote for escaping between languages which is reminiscent of our own. Racket provides facilities for modifying its reader and expander which can be used to read and execute custom languages by using the **#lang** form[10]. These features, while certainly powerful, are more complex than our solution, yet share the quality of allowing a programmer to switch back and forth between languages.

All things considered, our work distinguishes itself from other FFIs most clearly by its use of a Scheme reader extended with an infix notation parser. This allows our FFI to interface host and foreign languages at the expression level, enabling a more concise and natural style. The FFI's ability to interface JavaScript asynchronous functions with Scheme threads transparently also simplifies combining Scheme programs with asynchronous JavaScript code and libraries.

## ACKNOWLEDGMENTS

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

#### REFERENCES

- [1] JScheme Reference Manual. Retrieved March 15, 2021 from http://jscheme.sourceforge.net/jscheme/doc/refman.html.
- [2] Use JavaScript code from Kotlin | Kotlin. Retrieved March 15, 2021 from https://kotlinlang.org/docs/js-interop.html.
- [3] LIPS: Powerful Scheme based lisp interpreter in JavaScript. Retrieved March 15, 2021 from https://lips.js.org/.
- [4] Pyodide Version 0.17.0. Retrieved April 22, 2021 from https: //pyodide.org/en/0.17.0/.
- [5] Eli Barzilay and Dmitry Orlovsky. Foreign interface for PLT Scheme. In Proceedings of the Fifth ACM SIGPLAN Workshop on Scheme and Functional Programming, pages 63-74, Snowbird, Utah, 2004.
- [6] Noury Bouraqadi and Dave Mason. Mocks, Proxies, and Transpilation as Development Strategies for Web Development. In Proceedings of the 11th edition of the International Workshop on Smalltalk Technologies, pages 1–6, Prague Czech Republic,

August 2016. ACM. https://dl.acm.org/doi/10.1145/2991041. 2991051.

- [7] Atze Dijkstra, Jurriën Stutterheim, Alessandro Vermeulen, and S. Doaitse Swierstra. Building JavaScript Applications with Haskell. In Ralf Hinze, editor, *Implementation and Application* of Functional Languages, volume 8241 of Lecture Notes in Computer Science, pages 37–52, Berlin, Heidelberg, 2013. Springer. https://doi.org/10.1007/978-3-642-41582-1\_3.
- [8] R. Kent Dybvig. Chez Scheme Version 8 User's Guide. Cadence Research Systems, 2009. Retrieved March 15, 2021 from https: //www.scheme.com/csug8/.
- Marc Feeley. Gambit v4.9.3 manual, 2019. Retrieved on March 15, 2021 from http://www.iro.umontreal.ca/~gambit/doc/gambit. pdf.
- [10] Matthew Flatt, Robert Bruce Findler, and PLT. The Racket Guide. Retrieved April 22, 2021 from https://docs.racket-lang. org/guide/index.html.
- [11] Robert H. Halstead. Multilisp: A Language for Concurrent Symbolic Computation. ACM Trans. Program. Lang. Syst., 7(4): 501–538, October 1985. https://doi.org/10.1145/4472.4478.
- [12] Felix S. Klock II. The Layers of Larceny's Foreign Function Interface. In Workshop on Scheme and Functional Programming, Vancouver, British Columbia, 2008.
- [13] Massimiliano Poletto, Wilson C. Hsieh, Dawson R. Engler, and M. Frans Kaashoek. 'C and tcc: a language and compiler for dynamic code generation. ACM Transactions on Programming Languages and Systems, 21(2):324–369, March 1999. https://doi. org/10.1145/316686.316697.
- [14] Manuel Serrano. Bigloo, a Practical Scheme Compiler, March 2021. Retrieved March 15, 2021 from http://www-sop.inria.fr/ indes/fp/Bigloo/.
- [15] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a Language for Programming the Web 2.0. In *Proceedings of the First Dynamic Languages Symposium*, pages 975–985, Portland, Oregon, 2006.
- [16] Konrad Slind. Object language embedding in Standard ML of New Jersey. In *Proceedings of the Second ML Workshop*, CMU SCS Technical Report, Pittsburgh, Pennsylvania, 1991. Carnegie Mellon University.

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

# ABSTRACT

A function call in a language such as Common Lisp can be fairly costly. Not only is an indirection required so that a redefinition of the callee can take effect for subsequent calls, but several features of Common Lisp can have an even greater impact on the performance of function calls. The presence of optional parameters and/or keyword parameters requires some non-trivial argument parsing in the callee. And when the callee is a generic function, it must invoke the discriminating function in order to dispatch to the effective method that is determined by the arguments. Restrictions such as the required boxing of all arguments can make function calls slow for full-word integer and floating-point numbers.

In this paper, we propose a very general technique for improving the performance of function calls in Common Lisp. Our technique is based on *call-site optimization*, meaning that each call site can be automatically customized for the callee according to the number and the types of the arguments being transmitted to the callee. Our technique is based on the call site being implemented as an unconditional jump to a trampoline snippet that is generated by the callee according to information provided by the caller with respect to the arguments. Thus, the callee is able to fully customize the call, thereby avoiding many costly steps of a function call such as indirections, boxing/unboxing, argument parsing, and more.

# CCS CONCEPTS

• Software and its engineering  $\rightarrow$  Software performance; Runtime environments;

# **KEYWORDS**

Common Lisp, Performance, Call-site optimization

#### **ACM Reference Format:**

Robert Strandh. 2021. Call-site optimization for Common Lisp. In Proceedings of the 14th European Lisp Symposium (ELS'21). ACM, New York, NY, USA, 7 pages. https://doi.org/10.5281/ zenodo.4709958

#### 1 **INTRODUCTION**

Function calls in a dynamic language like Common Lisp can be significantly more expensive in terms of processor cycles

© 2021 Copyright held by the owner/author(s).

than function calls in a typical static language. There are several reasons for this additional cost:

- (1) With *late binding* being a requirement, i.e., the fact that functions can be removed or redefined at run-time, and that callers must take such updates into account, it is necessary to have some indirection that can be modified at run-time. Mechanisms such as compiler macros and inlining break this requirement, which is often a serious drawback to their use.
- (2)Common Lisp has a rich function-call protocol with optional parameters and keyword parameters. Keyword parameters, in particular, require some considerable run-time parsing for every call to a function that has such parameters.
- (3) In general, a function that can honor its contract only for certain types of its arguments must check such types for each call.
- (4) All objects must be *boxed* in order to be used as function arguments. For example, IEEE double-float values will typically have to be allocated on the heap, though so-called NaN-boxing [4] can eliminate that particular case. Full-word integers still require boxing, however. Similarly, boxing is required for values returned by a function.
- (5) Generic functions can be dynamically updated by the addition or removal of methods. Thus, even when the callee is a known generic function, callers can make no assumptions about which methods might be applicable.
- (6)The fact that a function can return multiple values requires the callee to return additional information about the number of return values, and callers that accept multiple values must retrieve this information in order to access the return values, and use default values when it expects more values than the callee returned.

In a typical Common Lisp implementation, item number 1 is handled by an indirection in the form of a slot in the symbol naming the function, requiring a memory access. On modern processors a memory indirect branch is more costly than a direct branch. Even if the branch-prediction logic of the processor is able to make the right decision in the indirect case, there is at least the additional cost of accessing the cache.

Item number 2 can be mitigated by the use of compiler macros. Essentially, the creator of a function with a nontrivial lambda list can also create special versions of this function for various argument lists. A call with an argument list that is recognized by the compiler macro can then be replaced by a call to such a special version, presumably with a simpler lambda list.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s). ELS'21, May 03-04 2021, Online, Everywhere

https://doi.org/10.5281/zenodo.4709958

Item number 3 can be handled by inlining, allowing the compiler to take advantage of type inference and type declarations to determine that some type checks can be elided. However, inlining has the disadvantage that a redefinition of the callee will not automatically be taken into account, thereby requiring the caller to be recompiled for the redefinition to be effective.

The problem indicated by item number 4 can be largely eliminated by the use of more than one entry point for functions, one of which would accept unboxed arguments. This technique is used by Allegro Common Lisp<sup>1</sup> which also allows for functions to return a single unboxed value.

Concerning item number 5, the main difference between function redefinition and generic-function updates is that a generic function consists of independent *effective methods*, only one of which is applicable for a particular call. To determine which effective method is applicable, in the general case some significant *generic dispatch*, based on the class or the identity of arguments, may be required.

A common technique for handling item number 6 is to recognize that most callers will use only a single return value. Then, if the callee returns no values, the register holding the first return value will nevertheless be initialized to nil. As a result, callers that use a single return value never have to test the number of values actually returned.

In this paper, we propose a very general technique for callsite optimization that can handle many of the issues listed at the beginning of this section. We plan to incorporate this technique in the SICL<sup>2</sup> implementation of the Common Lisp language.

# 2 PREVIOUS WORK

To our knowledge, no work on call-site optimization has been published in the context of Common Lisp, though some practical work exists in the form of code in certain implementations, as explained in Section 2.2.

The absence of published work is perhaps due to the many unique features of Common Lisp functions, that make the task very difficult, such as keyword arguments, generic functions with arbitrary method combinations, etc.

# 2.1 Inline caching

One technique that is fairly common is *inline caching*, pioneered by Smalltalk [2]. This technique is used to avoid repeated method selection in a particular call site. The key observation is that, for a particular call site, often the same method is concerned each time the call is made. By caching the latest method, keyed by the distinguished class argument, the system can often avoid a costlier computation.

The purpose of inline caching being to reduce the cost of finding the applicable effective method, it is not directly related to speeding up function calls, but it has the effect of making calls to generic functions faster. Robert Strandh

# 2.2 Ctors

The CMUCL<sup>3</sup> implementation of the Common Lisp language uses a technique that they call *ctors* that can be used for callsite optimization of certain functions. This optimization was introduced by Gerd Moellmann in 2002, and has since been included also in SBCL<sup>4</sup>, which is a derivative of CMUCL. In particular, in CMUCL the technique is used for the function **make-instance** which is often called with a literal class name and literal keywords for the initialization arguments. When the name of the class to instantiate is a literal, several steps in the object-initialization protocol can be simplified.

Most importantly, checking the validity of the initialization arguments can be done once and for all, subject only to added or removed methods on the functions **initialize-instance** and **shared-initialize** and to updates to the class being instantiated.

CMUCL accomplishes the optimization by replacing (using a compiler macro) the original call to make-instance by a call to a *funcallable object* that is specific to the name of the class and the literal keyword arguments given. The *funcallable instance function* of the funcallable object is updated as a result of added or removed methods and modified classes as mentioned. This technique can be used on other, similar functions. For example, slot-value is often called with a constant slot name, and this fact has been explored by SBCL.

Since the optimization is done as a manual source-code transformation, it is applicable mainly to standard functions that can not change later on. The mechanism presented in this paper can be seen as an automatic low-overhead version of ctors.

A similar mechanism (called "constructor functions") exists in Allegro Common Lisp. And the Clasp Common Lisp implementation<sup>5</sup> uses a similar mechanism for make-instance, change-class, and reinitialize-instance.

## 2.3 Sealing

Sealing is a mechanism that allows the programmer to freeze the definitions of various program elements such as classes and generic functions. The work by Heisig [5] applies to Common Lisp and can allow for certain call sites to be optimized to different degrees, from bypassing generic dispatch to fully inlining entire effective methods.

# 3 MAIN FEATURES OF THE SICL SYSTEM

In this section, we give a quick overview of the main features of our system SICL. The important aspect of our system in order for the technique described in this paper to work is that code is not moved by the garbage collector, as described below.

SICL is a system that is written entirely in Common Lisp. Thanks to the particular bootstrapping technique [1] that we developed for SICL, most parts of the system can use

 $<sup>^{1} \</sup>rm https://franz.com/products/allegro-common-lisp/$ 

<sup>&</sup>lt;sup>2</sup>https://github.com/robert-strandh/SICL

<sup>&</sup>lt;sup>3</sup>https://cmucl.org

<sup>&</sup>lt;sup>4</sup>http://www.sbcl.org/

<sup>&</sup>lt;sup>5</sup>https://github.com/clasp-developers/clasp

the entire language for their implementation. We thus avoid having to keep track of what particular subset of the language is allowed for the implementation of each module.

We have multiple objectives for the SICL system, including exemplary maintainability and good performance. However, the most important objective in the context of this paper is that the design of the garbage collector is such that executable instructions do not move as a result of a collection cycle. Our design is based on that of a concurrent generational collector for the ML language [3]. We use a *nursery* generation for each thread, and a global heap for shared objects. So, for the purpose of the current work, the important feature of the garbage collector is that the objects in the global heap do not move, and that all executable code is allocated in that global heap.

The fact that code does not move is beneficial for the instruction cache; moreover it crucially allows us to allocate different objects in the global heap containing machine instructions, and to use fixed relative addresses to refer to one such object from another such object.

# **4 OUR TECHNIQUE**

## 4.1 Function call

A function call involves a first-class object called a *function* object or a function for short. In general, a function may refer to variables introduced in some outer scope, so that the function is a *closure*. The (typically native) instructions to be executed by the function must be able to refer to such closed-over variables. But the values of such variables may vary according to the flow of control at run time. This situation is handled by a compile-time procedure called *closure* conversion whereby a static environment is determined for each function object. A function object thus consists at least of an entry point, which is the address of the code to be executed (and which is shared between all closures with the same code) and an object representing the static environment (which is specific to each function object). A function call must therefore contain instructions to access the static environment and put it in an agreed-upon place (typically a register), before control is transferred to the entry point.

This work covers function calls to functions that are *named* at the call site. The most common such case is when the name of the function appears in the operator position of a compound form. Less common cases include arguments of the form (function *name*) to some standard functions such as funcall and apply. In particular, expansions of the setf macro are often of the form (funcall (function (setf symbol)) ...) because function names like (setf symbol) are not allowed in an operator position.

In general, with such named function calls, the function associated with the name can be altered at run time, or it can be made undefined by the use of fmakunbound. For that reason, the caller can make no assumptions about the signature of the callee. This issue is solved by a standardized function-call protocol that dictates where the caller places the arguments it passes to the callee. Thus, for the purpose of this work, we define a *function* call to be the code that accomplishes the following tasks:

- (1) It accesses the arguments to be passed to the callee from the places they have been stored after computation, and puts the arguments in the places where the callee expects them.
- (2) It accesses the function object associated with the name at the call site and stores it in some temporary location.
- (3) From the function object, it accesses the static environment to be passed to the code of the callee.
- (4) Also from the function object, it accesses the *entry point* of the function, i.e., typically the address of the first instruction of the code of the callee.
- (5) It transfers control to the entry point, using an instruction that saves the return address for use by the callee to return to the caller.
- (6) Upon callee return, it accesses the return values from the places they have been stored by the callee, and puts those values in the places where the caller requires them for further computation.

In a typical implementation, a function call is generated when the code of the caller is compiled, and it then never changes. As mentioned above, for this permanent code to work, a particular *function-call protocol* must be observed, and that protocol must be independent of the callee, as the callee may change after the caller has been compiled.

Our technique optimizes function calls to functions in the *global environment* such that the *name* of the callee is known statically, i.e., at compile time. There are three different types of forms that correspond to this description and that are considered in this work:

- A function form where the operator is a symbol naming a function in the global environment, and that does not correspond to any of the following two form types. We use the term *ordinary function form* for this case.
- (2) A function form where the operator is the symbol funcall and the first argument is either a literal symbol or a function special form with a function name.
- (3) A function form where the operator is the symbol apply and the first argument is either a literal symbol or a function special form with a function name.

The first type of form can be considered as special syntax for a **funcall** form with a constant function name.

There are some other cases that we do not intend to cover, in particular a call to multiple-value-call with a named function argument. In fact, the third type of form could be generalized to cover other functions that are commonly used with a constant function argument, such as mapcar. At the moment, we are not considering such additional cases.

With our suggested technique, for these three different form types, the function call is created by the callee. We discuss each form type separately.



Figure 1: Caller, callee, and snippet in the global heap.

## 4.2 Ordinary function form

The code emitted by the caller for a function call consists of a single unconditional *jump* instruction. The target address in that instruction is altered by the callee according to its structure. The code for the function call is contained in an object that we call a *trampoline snippet*, or just *snippet* for short. The callee allocates an appropriate snippet in the global heap as described in Section 3, at some available location, and the unconditional jump instruction of the caller is modified so that it performs a jump to the first instruction of the snippet. The constellation of caller, callee, and snippet is illustrated in Figure 1. We omitted an explicit indication of a control transfer from the snippet to the callee code, because such a control transfer is not always required.

When the callee changes in some way, a new snippet is allocated and the jump instruction is altered to refer to the position of the new snippet. The old snippet is then subject to garbage collection like any other object. For the callee to be able to alter the caller this way, a list of all call sites must be accessible from the name of the callee. For an ordinary Common Lisp implementation, the symbol used to name the callee can store such a list. In SICL, this information would be kept in the data structure describing the callee in the first-class global environment [7]. Either way, to avoid memory leaks, the call site should be referred to through a weak reference.

When code containing a caller is loaded into the global environment, and that caller contains a call site that refers to a function that is not defined at the time the caller is loaded, a *default snippet* is created. The default snippet contains the same instructions that a traditional compiler would create for a call to a function that might be redefined in the future.



Figure 2: Default snippet.

Thus, the default snippet contains code to put arguments in places dictated by the calling conventions, and it accesses return values from predefined places. It also accesses the function indirectly, either through a symbol object (as most Common Lisp systems probably do) or through a separate *function cell* as described in our paper on first-class global environments [7]. The default snippet is illustrated in Figure 2. The default snippet is also used when the definition of the callee changes, as described below. A default snippet for each call site can either be kept around, or allocated as needed. The former situation is advantageous for a callee with many call sites and for callees that are frequently redefined, as it decreases the time to load a new version of the callee.

In order for the callee to be able to adapt the snippet to its requirements, the caller, when loaded into the executing image, must provide information about its call sites to the system. Each call site contains information such as:

- The name of the callee.
- The number of arguments.
- The type of each argument. If the type is not known, it is indicated as t. When an argument is a literal object, its type is indicated as (eql ...).
- For each argument, whether the argument is boxed or unboxed.
- For each argument, its location. The location can be a register or a stack position in the form of an offset from a frame pointer.
- The number of required return values, or an indication that all return values are required, no matter the number.
- In case of a fixed number of return values, for each such value, some limited information of the *type* of

Robert Strandh

each value. See below for a more elaborate explanation of the restrictions involved for this information.

- Also, in case of a fixed number of return values, for each such value, the location where the caller expects the value.
- Indication as to whether the call is a *tail call*, in which case the snippet should deallocate the frame before returning.

A callee can take advantage of this information to customize the call. The default action is to generate a snippet that implements the full function-call protocol, without taking into account information about the types of the arguments.

While our technique allows for information provided by the caller to be taken into account by the callee in various ways, the opposite direction is not generally possible. The reason is that the callee can change or be redefined in arbitrary ways, and the caller code is fixed, so it can not adapt to such changes in the callee. The only place where some limited amount of adaptation is possible is in the snippet, after the callee code returns.

A direct consequence to this one-directional dependency is that the caller can not, in general, dictate the type of the return values. The current callee will produce the values that its code dictates, no matter what the caller needs. However, it is quite advantageous to be able to return unboxed values of certain types; in particular full-word floating point numbers. For that reason, we allow some restricted type information to be provided by the caller. Thus, if the caller indicates a type other than t for some return value, it has to be one of a small number of fixed types, for example double-float, character, (signed-byte 64) and (unsigned-byte 64) (assuming a 64bit architecture). When one of these types is indicated by the caller, the meaning is that the caller requires an unboxed value of this type. Then, if the callee cannot supply such a value, code is generated in the snippet to signal an error.

When a modification is made to a callee that alters its semantics, care must be taken so as to respect the overall semantics of all callers. In particular, a callee can be removed using fmakunbound or entirely replaced using (setf fdefinition). In that case, the following steps are taken in order:

- (1) First, every call site is de-optimized, which means that a default snippet is allocated for each caller, or the kept default snippet is reused. The unconditional jump instruction is modified to refer to the default snippet. As previously explained, this snippet contains code for the full function-call protocol, and in particular, it accesses the callee using an indirection through the function cell.
- (2) Next, the callee is atomically replaced by a new function, or entirely removed by a single modification to the contents of the function cell.
- (3) The new function is attached to the list of call sites, and, depending on the nature of the new function, new

snippets can then be allocated in order to improve performance of calls to the new function.

When new snippets are substituted, actions may be needed to ensure that that processors do not use stale code. Depending on the types of processors involved, such actions include flushing instruction caches and prefetch pipelines.

The thread responsible for redefining the callee, blocks until step 1 is accomplished. Without this blocking, some callers may get the old version of the callee and some others the new version, thereby violating the overall semantics of a function redefinition. In some cases, it may be acceptable for different callers to get different versions, but in the general case, i.e., when it is observable which versions are used, it is not acceptable. Because of this requirement, redefining a function can be an expensive operation, but redefining a function is expected to be infrequent compared to calling it.

Step 3, on the other hand can be accomplished asynchronously, and even in parallel with caller threads, provided that appropriate synchronization prevents subsequent simultaneous redefinitions of the callee.

# 4.3 funcall with known function name

There are two subcases for this type of form:

- (1) The first argument is a special form quote with the argument being a symbol. This case can occur as a result of the programmer wanting to avoid capture of the function name, and make sure the name refers to the function with that name in the global environment. It can also occur in the expansion of a macro form.
- (2) The first argument is a special form function. This case typically occurs as a result of expanding a setf macro form and the function name is then of the form (setf symbol). The expansion uses funcall simply because a function name of this form can not be used as the operator of a function form. This case can also occur in the expansion of a macro form.

To handle this case, the compiler treats the symbol funcall as a special operator. If the first argument corresponds to any of the two subcases, then the call is treated in the same way as an ordinary function form. Otherwise it generates a call to the function funcall.

# 4.4 apply with known function name

As with funcall, the same two subcases exist for apply, and for the same reason. Again, the compiler treats the symbol apply as a special operator and generates a call to the function apply whenever the first argument is neither the special form quote nor the special form function.

However, the case of apply is of course more complex than that of funcall. Recall that apply takes at least two arguments. The first argument is a function designator as with funcall. The remaining arguments represent a *spreadable argument list designator*, which means that the last argument is treated as a list of objects, and the arguments to the callee are the objects in that list, preceded by the remaining arguments to apply, in the order that they appear.

A very common subcase of this case is a call to **apply** with exactly two arguments. It is used when the execution of some code results in a list of objects, and these objects must be passed as the arguments to some function, in the order that they appear in the list. For this subcase, our technique can be used to avoid the indirection to find the callee entry point as usual. But it can also be used to access the callee arguments directly from the list of objects, so as to avoid unpacking the list to locations dictated by the full call protocol.

A more interesting subcase is that of some intermediate function wanting to override some, but not all of the keyword arguments that it was passed, before calling the callee. The remaining arguments to apply are then typically keyword/value pairs. Our technique can then be used to avoid scanning the last argument to apply for these keyword arguments. Recall that the standard allows for multiple occurrences of the same keyword argument in an argument list, and that the first occurrence is then the one that is used.

Call-site information resulting from a call to apply must be indicated as such, so that the call-site manager can process the arguments as a spreadable argument list designator, rather than as an ordinary suite of arguments.

## 5 BENEFITS OF OUR TECHNIQUE

Our technique makes possible several features that are not possible when a function call is created by the caller, without knowledge about the callee.

For starters, at least one indirection can be avoided, thereby saving a memory access. When the call is generated by the caller, there must be an indirection through some kind of *function cell*, unless the callee is a function that is known never to change. This indirection is required so that a redefinition of the callee is taken into account by the next call. A typical Common Lisp implementation uses a symbol (the name of the function) for this indirection, whereas SICL uses a separate **cons** cell, but the cost is the same. With our technique, when a callee is altered, the snippet is modified. As a result, no indirection<sup>6</sup> is required. Furthermore, in SICL all functions are standard objects, which requires another indirection from the *header object* to the so-called *rack* where the entry point is stored.

A more significant benefit than saving an indirection is that argument parsing can be greatly simplified. Even in the simple case where all parameters are required, it is no longer necessary for the caller to pass the argument count, nor for the callee to check that it corresponds to the number of parameters. But the advantages are even greater in the presence of optional parameters and in particular for keyword parameters. In a typical call with keyword parameters, the keywords are literals. The argument list can then be parsed once and for all when the snippet is created, and the arguments can be directly copied to the locations required by the callee. This possibility largely eliminates the need for separate compiler macros, as the purpose of a compiler macro is precisely to take advantage of some known structure of the list of argument, in order to substitute a call to a specialized version of the callee.

The specialized function call can admit unboxed arguments. Avoiding boxing is particularly useful for applications that manipulate floating-point values that are at least the size of the machine word, say IEEE double or quadruple floats in a 64-bit system. When a general-purpose function-call protocol is used, each such argument must be encapsulated in a memory-allocated object before the call, and often, the argument will immediately be unboxed by the callee for further processing.

Return values benefit from the same advantages as arguments. Often, the number of values required by the caller is known statically. The callee can then specialize the transfer of those values to the right locations in the caller. And if the caller requires fewer values than the callee computes, the callee can sometimes be specialized so that extraneous return values do not need to be computed at all. As with arguments, return values can be unboxed, again avoiding costly memory allocations.

When the callee is a generic function, a specialized discriminating function can often be created, provided that enough type information is made available by the caller for the arguments that correspond to specializers of some methods of the generic function. In the extreme (but common) case where the callee is a slot accessor and the class of the specialized argument is known, the snippet can contain the full code to access the slot, without any need to call a particular method function.

Often, *inlining* is used to improve the performance of function calls, either by the application programmer or by the system itself. But inlining some function necessarily increases the code size of each caller of that function. Furthermore, the semantics of inlining are such that the caller must be recompiled for a modified callee to be taken into account. Our technique can often provide enough performance improvement to make inlining unnecessary. Total code size will then be smaller, and the disadvantage of inlining with respect to callee redefinition is eliminated.

Compared to the so-called *ctor* technique describe in Section 2.2, our technique is more general, since it does not involve any source-code transformations. Thus, it can be used with functions defined by the application programmer, and that can change at any point after the callee has been compiled. Furthermore, the *ctor* technique still requires at least one, probably two, indirections (one to access the funcallable object and another one to access the entry point). However, our technique in itself can of course not accomplish the entire optimization machinery required to optimize a function such as make-instance, as knowledge of its semantics is required for such optimization.

# 6 DISADVANTAGES OF OUR TECHNIQUE

The proposed technique is fairly complicated. In order for all the advantages to be had, the callee must be represented in

<sup>&</sup>lt;sup>6</sup>Though, the snippet is itself a kind of indirection, of course.

such a way that multiple versions can be created, depending on different information provided by each caller. On the other hand, most of the benefit of this technique can be obtained with a limited amount of such flexibility. Bypassing argument parsing in the presence of optional or keyword parameters will already provide great benefits. For this benefit to be as useful as possible, it is advantageous to compile a callee in two parts; one part that allows for its parameters to be positioned in any places (registers or stack frame locations) that makes the remaining code as fast as possible, and one part that parses arguments from their default locations into those places. The callee can then generate code for the snippets that moves the arguments passed by the caller to those final places.

The garbage collector must not reclaim snippets that are currently in use, and "in use" can mean that a callee has an activation record on the call stack, so that the snippet can not be reclaimed until the activation record is removed from the stack. As a result, a modification to the garbage collector is required, and code for garbage collectors is notoriously hard to get right.

The technique involves the creation of two unconditional *jump* instructions; one from the core code of the caller to the snippet and another one from the snippet back to the core code of the caller. These additional instructions must be executed, which may use up processor cycles. However, on most modern processors, unconditional jumps are very fast [6].

Finally, there may be some slightly increased probability of contention in the instruction cache, due to the fact that snippets are allocated wherever the global memory manager can fit them.

# 7 CONCLUSIONS AND FUTURE WORK

We have presented a very general technique for call-site optimization. Our technique subsumes (entirely or partially) several other techniques such as inlining, compiler macros, sealing.

Our technique is very general, and promises several advantages to function-call performance that can not easily be obtained with other techniques. The flip side is that our technique is fairly complicated and requires significant support from both the compiler and the memory manager.

Furthermore, we have not implemented the suggested technique, and the state of the SICL system is not yet such that it can be done soon. The most urgent future work, then, is to create a native SICL executable. We are probably several months away before this work can be accomplished.

# 8 ACKNOWLEDGMENTS

We would like to thank Frode Fjeld and David Murray for providing valuable feedback on early drafts of this paper. We would also like to thank Duane Rettig for his remarks, and for valuable information about several optimization techniques used in Allegro Common Lisp.

# REFERENCES

- Bootstrapping Common Lisp using Common Lisp, April 2019. Zenodo. doi: 10.5281/zenodo.2634314. URL https://doi.org/10. 5281/zenodo.2634314.
- [2] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the smalltalk-80 system. In POPL '84: Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pages 297–302. ACM, 1984. ISBN 0-89791-125-3. doi: 10.1145/800017.800542.
- [3] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 113–123, March 1993. URL http://www.acm.org:80/pubs/citations/proceedings/ plan/158511/p113-doligez/.
- [4] David Gudeman. Representing Type Information in Dynamically Typed Languages. Technical Report TR 93 27, Department of Computer Science, The University of Arizona, October 1993.
- [5] Marco Heisig. Sealable Metaobjects for Common Lisp. In Proceedings of the 13th European Lisp Symposium, ELS '20, pages 26 – 32, April 2020. URL http://www.european-lisp-symposium.org/ static/proceedings/2020.pdf.
- [6] John L. Hennessy and David A. Patterson. Computer Architecture, Sixth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 6th edition, 2017. ISBN 0128119055.
- [7] Robert Strandh. First-class global environments in common lisp. In Proceedings of the 8th European Lisp Symposium, ELS '15, pages 79 - 86, April 2015. URL http://www.european-lisp-symposium. org/editions/2015/ELS2015.pdf.