Proceedings of the

17th European Lisp Symposium

Vienna, Austria
May 6 — 7, 2024

Federal Ministry
Republic of Austria
Finance

SISCOG

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

Preface

Message from the Program Chair

The 17^{th} European Lisp Symposium was held in Vienna, in the beautiful settings of the Sky Lounge, on top of a building near the Danube canal with a terrace with a 360-degree panorama of the city.

The conference attracted an audience far beyond Europe, including Japan, India and the USA. The ELS series in fact is carrying over a tradition of conferences on Lisp that started in 1980 with the first ACM Conference on Lisp and Functional Programming. I was there and listened to John McCarthy who gave a keynote. He had reasons to be pleased for the survival of Lisp for 21 years and mentioned the fact that practical applications could be written in Lisp like Multics Emacs. And Emacs today is in reverse where Lisp still survives as the language for building extensions. McCarthy thought that the language had good prospects with the creation of a variety of programming libraries and the potential for checks of program correctness. Lisp was in fact conceived by McCarthy as a language with a mathematical base to be used to express "proofs in the artificial language as short as informal proofs", as he wrote in the famous proposal for the Darthmouth Workshop in 1956, where the discipline of Artificial Intelligence was founded. Indeed he "proposed to study the relation of language to intelligence", which sounds quite foresighted in this period when Large Language Models dominate the scene.

Similar lively discussions were held among the participants in Vienna about why the opportunities that McCarthy was seeing have not materialized. One argument was indeed about the importance of a wide availability of libraries, which for example have attracted programmers to Python. The difference with Lisp was that all libraries had to be written in the language itself,

requiring too much effort, while in Python one could easily write wrappers, even automatically with tools like SWIG.

Another big chance for Lisp was as an embedded extension language, like in Emacs or Autocad. Guile was an attempt in that direction for general use within the GNU project, but unfortunately the big opportunity of a scripting language for the Web was stolen by Java and JavaScript.

Notwithstanding these setbacks, the participation in Vienna showed that the language still attracts programmers, who dedicate some of their extra time to developments and experimentation.

The keynote by Markus Triska argued strongly that a symbolic programming languages like Lisp or Prolog is essential for AI applications that need to be deployed within public administrations. He gave as example the government service Grants4Companies, which his team has built to help finding and verifying the elibility of a company to apply for grants.

Stavros Macrakis gave an insightful keynote about hype cycles full of enjoyable anectodes. He debunked several myths: for example he mentioned symbolic mathemathics like Macsyma. Symbolic algebra is an attractive application for AI, but nevertheless it has limitations, since for example algebraic solutions are only available to polynomial equations up to the fourth degree: beyond that one must use numerical approximate solutions. The other arguments that the symbolic solutions provide valuable insights is also questionable, since the formulas become so complex that they are hard to interpret. In the meantime hardware progress has made solutions more practical, even NP complete problems can be solved with a fairly large number of variables.

Julian Padget in his keynote discussed the issue of bias, that arises in algorithmic models built on data by means of machine learning techniques. He gave a critical account of how bias is defined even in certain official or standardisation documents. Bias is inherent in data and essential to the ability to make decision, therefore it cannot be removed: one can only strive to reduce the presence of unwanted bias. But this is not a straightforward direction, because removing a certain bias might introduce some other bias. Hence fighting bias should be considered as a continous process to be part of the system life cycle of a software system similarly to maintenance or debugging.

Overall the ELS 2024 was a lively venue with interesting presentations as well as discussions with the participants.

I wish to thank Didier Verna, for his efforts in setting up the conference and helping in chairing it, to Philipp Marek and the local organizers for the perfect arrangements and to the sponsors: the Federal Ministro of Finance of Austria and SISCOG.

Message from the Local Chair

I welcome you all to another round of ELS - this time in the city of Vienna, where historical events have taken place, and this here will be no less important.

Amidst the espionage history embedded in these very streets, let us inspect our variable bindings and bind our forms together, so that the whole of our works might not just be printed, recycled, and garbage collected, but may be compiled to add another level of precious understandings to the boring concepts we get asked by better-sleeping people: “You know computers, can you fix my microwave?”

But putting the jokes aside – it is in these few hours, shortly before ELS is about to begin, that I already gladly look back at the great help I had organizing it; and I know that we all (the local, the lexical, and the dynamically bound) organizers hope you all enjoyed the show and will look back with comfort and pleasure at the two days we spent on the 12th level floor of the Sky Lounge.

Looking forward to seeing you again next year, wherever we might be!

Organization

Symposium Organizer

- Didier Verna, EPITA Research Laboratory, France

Programme Chair

- Giuseppe Attardi, University of Pisa, Italy

Local Chair

- Philipp Marek, BRZ, Vienna Austria

Virtualization Team

Georgiy Tugai
Yukari Hafner

Programme Committee

Ambrose Bonnaire-Sergeant	Untypable LLC
Frédéric Peschanski	LIP6, Sorbonne Université, Paris, France
Jay McCarthy	UMass Lowell
Jim Newton	EPITA Research Laboratory, France
Kai Selgrad	OTH Regensburg
Mark Evenson	not.org
Michael Raskin	LaBRI/CNRS UMR 5800, University of Bordeaux, France
Robert Smith	HRL Laboratories LLC
Robert P. Goldman	SIFT LLC
Stefan Monnier	Université de Montréal, Québec

Sponsors

We gratefully acknowledge the support given to the 17th European Lisp Symposium by the following sponsors:

BMF
Bundesministerium
Finanzen
Austria
www.bmf.gv.at/

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

Monday 6 May 2024

08:30–09:45		**Registration, Badges, T-Shirts, Meet and Greet**
09:45–10:00		Welcome Message
10:00–11:00	Keynote	Markus Triska The Need for Symbolic AI Programming Languages in the Public Sector
11:00–11:30		**Coffee break**
11:30–12:30	Keynote	Stavros Macrakis Is the Hype Cycle Real?
12:30–14:00		**Lunch**
14:00–14:30	Research Paper	Daphne Preston-Kendal R7RS Large Status and Progress
14:30–15:00	Demo	Andrew Sengul The Medley Interlisp Revival
15:00–15:30	Demo	Anders Hoff Lisp Query Notation – A DSL for Data Processing
15:30–16:00		**Coffee break**
16:00–16:30	Research Paper	Philipp Marek, Bjoern Lellmann, Markus Triska Grants4Companies: The Common Lisp PoC
16:30–17:00	Demo	Marco Heisig An Introduction to Array Programming in Petalisp
17:00–17:30		**Lightning Talks**

Tuesday 7 May 2024

08:30–09:30		**Registration, Badges, T-Shirts, Meet and Greet**
09:30–10:30	Keynote	Julian Padget Bias is a bug; but not as we know it!
10:30–11:00		**Coffee break**
11:00–11:30	Research Paper	Gábor Melis Adaptive Hashing
11:30–12:00	Research Paper	Arthur Evensen Period Information Extraction: A DSL Solution to a Domain Problem
12:00–12:30	Demo	Didier Verna The Quickref Cohort
12:30–14:00		**Lunch**
14:00–14:30	Research Paper	Shubhamkar Ayare py4cl2-cffi: Using CPython's C API to Call Python Callables from Common Lisp
14:30–15:00	Demo	Eitaro Fukamachi Qlot, a Project-Local Library Installer
15:00–15:30	Demo	Robert Mayer, Thomas Östreicher Murmuel and JMurmel
15:30–16:00		**Coffee break**
16:00–16:30		**Lightning Talks**

Contents

Invited Contributions

Bias is a bug; but not as we know it!

Julian Padget, University of Bath, UK

Algorithmic model construction is now accepted technology. Using some data to train a model is commonplace and machine learning has percolated down to the first-year CS curriculum. Testing such models is quite difficult, because conventional approaches learned from conventional programming provide only limited coverage. Furthermore, metrics offer big picture performance but may disguise edge cases. One significant worry is that such systems exhibit differential treatment of individuals or groups because the algorithm has identified an attribute relationship in the data that does not align with the system's business requirements. This is typically referred to (wrongly!) as a 'biased' output. We start by examining the language of bias in algorithmic models and argue that (unwanted) bias is a (latent) bug. However, this bug typically has complex causes, as well as the possibility of morphing over time into bias that does align with the requirements. In consequence, we will continue by exploring how consideration of bias can be incorporated into the system life cycle and put forward some strategies for thinking about bias-related debugging.

Is the hype cycle real?

Stavros Macrakis, Amazon OpenSearch, Cambridge, MA, USA

Some technologies have grown steadily and undramatically over the years; others have been transient successes, or have been relegated to narrow application areas. How can we benefit from the long-term perspective to understand what might happen with today's most hyped technologies?

The Need for Symbolic AI Programming Languages in the Public Sector

Markus Triska, Austrian Federal Ministry of Finances

Rising expectations in public IT-services lead to increasing implementation complexity at a time where many of the programmers who initially built these services retire. The cost and complexity of building reliable e-government services also depend on the used programming languages. We would greatly benefit from better technologies to create and maintain IT-services that let us flexibly state and reason about laws and regulations on which administrations are based. Our recent experiences in the department V/B/5 of the Austrian Federal Ministry of Finance indicate that symbolic AI programming languages such as Lisp and Prolog are well suited for this purpose. The e-government service Grants4Companies is a recent application of such technologies in the public sector, and is made available to companies via the Austrian Business Service Portal.

Monday, 6 May 2024

The Medley Interlisp Revival

Andrew Sengul
andrew@interlisp.org
Interlisp.org

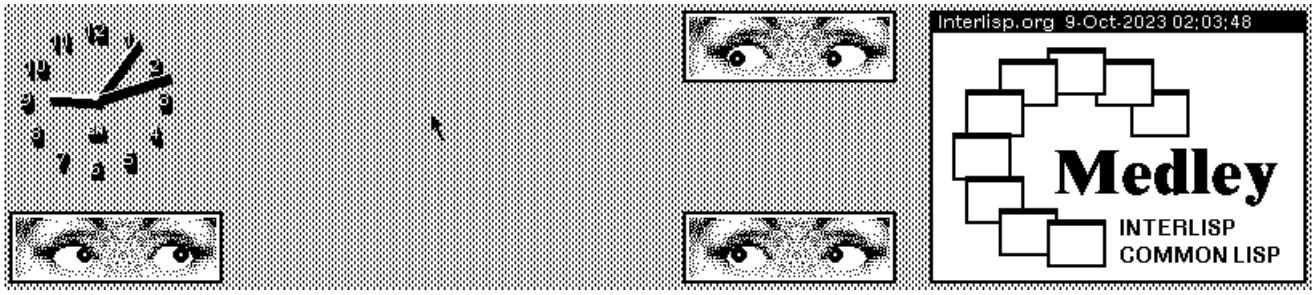

Figure 1: A screenshot from Medley featuring compact graphical software applications.

ABSTRACT

The Medley Interlisp revival is a project to restore Medley Interlisp for use on modern computers. Interlisp began as a Lisp environment for researchers sponsored by DARPA, and after gaining display capabilities it was renamed Interlisp-D. Xerox spun out sales and development of Interlisp-D with the "Medley" software release, which eventually became the product name. Medley development ended in the 1990s and was revived in 2021 by a team including some of the original PARC developers. Their effort is aimed at both preserving the Interlisp software created in the past and expanding the scope of what these tools can do to further realize the promises of interactive, graphically augmented development.

CCS CONCEPTS

• **Software and its engineering → Open source model**; **Software evolution**; *Maintaining software*; *Documentation*; Software reverse engineering; • **Theory of computation → Interactive computation**; • **Human-centered computing** → *Interaction design theory, concepts and paradigms.*

KEYWORDS

Lisp, Interlisp, GUI, interactive programming, software archaeology, software restoration, education, nonprofit, FOSS

ACM Reference Format:
Andrew Sengul. 2024. The Medley Interlisp Revival. In *Proceedings of the 17th European Lisp Symposium (ELS'24).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.11090093

ELS'24, May 6–7 2024, Vienna, Austria

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.11090093

1 INTRODUCTION

Interlisp.org, a US non-profit organization, is working to resurrect and restore the Medley Interlisp system and has made progress in modernizing and enhancing it. Medley Interlisp was the last version of the Interlisp system developed by Xerox Palo Alto Research Center (PARC). Accomplishments thus far have included reducing barriers to entry, making it easier to build versions for modern computing platforms, the creation of an online Medley system usable through a web browser, and one-click installers for major operating systems. The Medley software has been released under a free open source, downloadable at https://interlisp.org.

The system's original release came near the end of a fruitful period for interactive software development stretching from the late 1960s through the early 1990s. In the mid-1980s, Xerox PARC tapered off development of Medley and moved the project to Xerox AI Systems (a Xerox subsidiary). In the late 1980s XAIS moved the system to Envos, which closed shortly thereafter and led to a company called Venue acquiring the rights to Medley Interlisp. Circa 2018, Ron Kaplan and Nick Briggs resurrected Medley Interlisp and began transporting it to modern computing platforms to support work in natural language research.

In 2020, the Medley Project was formed with the goal of modernizing the Interlisp ecosystem, opening the way for present-day developers to experience it on modern computing platforms like Windows, MacOS and Linux. Interlisp.org was formed to organize these efforts, provide versions of Medley Interlisp to interested users and provide a repository for source code and documentation. Interlisp.org has been successful in this endeavor as Medley Interlisp now runs on the indicated platforms as well as through a browser-accessible online interface and on ARM systems. A Docker container release is available for added portability. Interlisp.org has also resurrected several applications including ROOMS, Notecards, LOOPS and others. It has assembled an online Zotero repository collecting Lisp documentation along with Interlisp papers, books, and technical material. Selected source code is also available for contributed programs.

2 THE PROJECT

Interlisp.org received the source code for Medley Interlisp and its applications from Venue Corporation. Interlisp.org consists of a group of volunteers – original developers, former users, computing historians interested in software preservation, and people interested in software archaeology. This group has focused on:

- Modernizing Medley's infrastructure and source code;
- Adapting the system to run on modern platforms;
- Reducing the barrier to entry for new users;
- Resurrecting and restoring Medley Interlisp applications of historical interest; and
- Conducting outreach to potential users, students, software historians, and others to foster better understanding of how symbolic computing evolved.

Notable is Medley's support for two Lisp dialects: Common Lisp (CL) and Interlisp. These are implemented as compilers for the respective dialects supporting interaction through a read-evaluate-print loop, or REPL. While the functions in these dialects are implemented in different ways their data structures are identical, so numbers, symbols, linked lists and some other data types can be shared between the two dialects. This makes it possible to create blended applications in which data is passed between functions written in either dialect.

2.1 Adaptation to Modern Platforms

Interlisp.org is engaged in restoring the Medley Interlisp ecosystem, including tools, utilities, and applications, and provides public versions of source and binary code for modern computing platforms along with documentation. All of these artifacts are available through Interlisp.org's GitHub and Zotero repositories. The open source version for modern computing platforms consists of:

- Maiko: the emulation software that implements the Interlisp Virtual Machine;
- Medley: source and compiled versions of Medley Interlisp, its tools and utilities, and selected applications;
- Installers for modern computing platforms: Windows 10+, MacOS and Linux, including WSL and ARM; and
- https://online.interlisp.org: a browser-accessible online version of Medley Interlisp.

Additionally, Interlisp.org provides public access to its Zotero repository, which contains a collection of documentation for Interlisp and other Lisp dialects.

2.2 Reducing Barriers to Entry

Easy access to Medley Interlisp will help (re)introduce potential users to Interlisp in a way that allows them to explore its features while building their expertise. Since Interlisp was developed before current conventions for mouse and window-based interaction as well as Unicode standards for text encoding, Medley has been modernized to give users a look and feel that will be more familiar in the context of today's computer interfaces.

Interlisp.org created an online version of Medley Interlisp using Docker and Amazon Web Services, providing for users to experience the system through a web browser without running any of the Medley system's components on their local computing platform. When they are ready, if they choose they can install a version of Medley Interlisp onto a local computer system and download files created using the online platform to be used on the same system.

During 2023, online Interlisp had 428 registered users accounting for 1,685 sessions, along with 2,588 anonymous guest sessions.

3 INTERLISP IN PERSPECTIVE

The history of Interlisp is interwoven with the early history of artificial intelligence as many AI researchers had access to DEC PDP-10/DECsystem-10 computers which could run relatively large programs on a platform with 256K words of real memory [1]. Interlisp provided a residential programming environment in which the software development functions of edit, compile, link, and execute could be performed without leaving the Interlisp environment [16], along with a suite of tools for writing documentation.

In Interlisp, a user edits and evaluates Lisp objects that reside in an image in memory. The code is saved to files that are more like code databases than traditional source files. Users do not modify the files but load their contents into memory to edit, compile, and execute. The File Manager provides a simple interface that saves code in a memory image to disk, along with archiving unsaved code changes when a user closes a session so unfinished work can be resumed later. The beginning of a file specifies metadata describing the "file environment" and readtable associated with the file. The File Manager coordinates the development tools and readable code management tasks. It notices the changes to Lisp objects edited with SEdit or manipulated in memory, tracks what changed functions and objects need to be saved to symbolic files, and carries out the actions for building programs such as compiling or listing them (working in some ways like Unix's make).

This model is unique even today, with a few programming systems like Smalltalk and Symbolics Genera offering similar capabilities but nothing exactly matching Medley's model. Although Interlisp was not the first version of Lisp (there were several on mainframes and early minicomputers), it is safe to say that it influenced most succeeding versions. Medley's development coincided with the creation of the Common Lisp standard and its design influenced decisions by the CL committee, which included Medley developer Larry Masinter. Records of their thought process can be found in their email threads [8]. Interlisp was implemented on top of a virtual machine, preserving vertical integration, and many of its utilities were also written in Interlisp. Conversely, most other Lisp systems were written in imperative programming languages like C for performance reasons.

3.1 A Model of Interactive Software Development

Medley Interlisp was created to provide a computing environment for research into and development of large-scale applications. To this end, Xerox PARC users as well as others developed tools to facilitate software development through an interactive graphical user interface (GUI) in a collaborative environment [6]. Among these utilities were MasterScope, Spy, DWIM, CLISP, and LOOPS. Over 100 such tools are collected in the LispUsers library of contributed software, some of which have yet to appear in modern software development toolsets.

Numerous utilities were developed at institutions apart from Xerox PARC, such as NASA, several DARPA contractors and commercial firms. We are searching to identify these tools, acquire source code where possible, adapt them to run on modern computing platforms and make them available along with appropriate licenses and documentation through the GitHub and Zotero repositories.

4 MEDLEY INTERLISP ECOSYSTEM

Components of the Medley Interlisp Ecosystem include:

- Maiko: the emulator software for the Interlisp virtual machine;
- Medley Interlisp: the Interlisp source code and its utilities and tools;
- Applications: including several Interlisp applications, such as ROOMS, Notecards, LFG, STRADS, IDA, as well as other Lisp system applications; and
- Documentation: a comprehensive collection of books, papers, technical memoranda, and manuals regarding Interlisp, and extending to other Lisp variants.

Interlisp.org welcomes contributions of source code and documentation to add to its repositories. The following sections briefly describe these components and work done to restore them.

4.1 Maiko

Maiko is the emulator implementing the virtual machine within which Interlisp runs.[1] Maiko was initially developed by Fuji Xerox but acquired by Xerox PARC, which continued to maintain and enhance it. Written in Kernighan and Ritchie C, Interlisp.org developers have modernized it, making it ANSI C compatible. A few of these modifications included resolving issues of signed vs. unsigned characters, adding prototypes for functions, ensuring all parameters have types, fixing some incorrect translations of Lisp code to C code and optimizing all virtual machines' opcodes.

A major goal of this modernization process was to facilitate moving the Medley Interlisp ecosystem to modern computing platforms. Numerous programming changes were made to ensure that Maiko could run on Windows 10/11, recent MacOS versions and Linux and WSL-based platforms, as well as in the form of a hosted public installation accessible through web browsers.

4.1.1 Running Natively on Windows 10/11. Previously, running on Windows required the use of the Medley Docker container or WSL. Both involved significant effort to set up along with knowledge not possessed by most Windows users. Native support was developed using Cygwin and SDL2, allowing the use of a one-click installer in the form of an `.exe` file.

4.1.2 Support for AArch64. The build scripts for the Maiko virtual machine were extended to support the AArch64 (ARM) platform. This effort established a model for generating build scripts for other platforms, such that any system which has an ANSI C compiler can host a version of Medley Interlisp.

4.1.3 Major Platform Installers. Installing Medley was formerly a multi-step process requiring a degree of expertise with the administrative tools of a given platform. A single-step installer using a "one-click" approach was developed for MacOS, Windows (native), Windows running WSL and Cygwin and many Linux distributions, allowing a user to quickly and easily install a Medley release and do meaningful work.

[1] https://github.com/Interlisp/maiko

4.2 Medley Interlisp

Medley Interlisp includes the basic functions implementing the Interlisp language and environment as well as a selection of development tools. Because these utilities were written in Interlisp, many were found to run without major changes once the basic system became operational.

4.2.1 Common Lisp Support. With the groundswell of support for Common Lisp in the mid-1980s, Xerox PARC extended the Interlisp environment to support Common Lisp, specifically as of Common Lisp: The Language, Version 1 (CLtL1), along with CLOS and CL's condition system. The infrastructure supporting Interlisp and Common Lisp is fully integrated such that functions from both dialects are available within the same system in separate software packages, albeit with some rough edges we are working out.

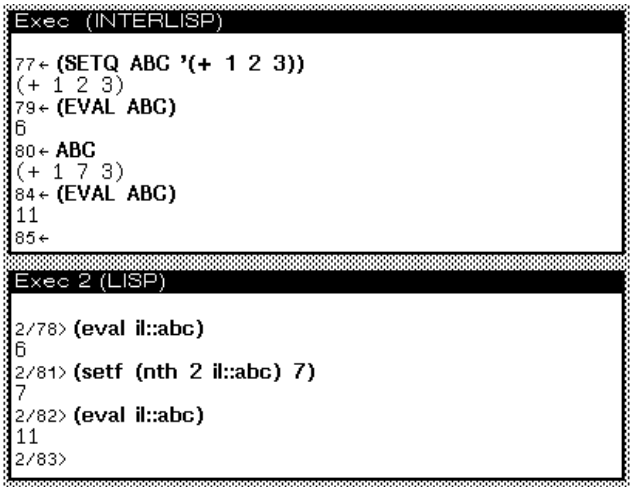

Figure 2: Code evaluated in both Interlisp and Common Lisp addressing a common data structure.

For example, Figure 2 shows two Exec windows – one using the Interlisp readtable addressed through the package `IL:` and one using the Common Lisp readtable addressed through the package `CL:`. The pictured code creates a list in Interlisp, modifies it in Common Lisp and evaluates it in both dialects.

The Common Lisp integration with the Interlisp tools was incomplete. Substantial work has made it easier to use some of the Interlisp tools like HELPSYS and MasterScope with CL. Also, some of the functions and directives in CLtL2 are not available (such as 'declaim' versus 'proclaim'). As we discover these we are determining how to provide the missing functionality, but some might not be available until later in 2024.

4.2.2 Editing and Browsing Support. Since the early 1980s Medley Interlisp has used a 16-bit internal representation of characters in

strings and atoms in the form of Xerox XCCS codes [9], which were mapped into appropriate glyphs for display and printing. We have since generalized character reading/writing functions as part of our external formatting project. If a file's external format is specified as Unicode when a stream is opened, Unicode byte sequences are read into 16-bit Unicode codes, which are translated into their XCCS equivalents before being delivered to the calling function. Support for ISO8859 and certain Japanese conventions are also provided. This system removes the need for most programmers using Medley to directly deal with character encoding.

TEdit, the text editor, was extended with Unicode support in a way providing for better efficiency, reliability, and maintainability of the system [14]. TEdit reads all characters into an internal editing buffer and creates pointers to the bytes on the file that represent those characters. It only interprets those bytes when it needs to display that section of the file, move characters from one place to another in the file, or copy them to some other application. Thus a TEdit session on a large file opens quickly and only occupies a small amount of memory.

Work on TEdit has encompassed a major portion of the modernization effort over the past three years because assumptions about the XCCS file format were threaded all through the core TEdit implementation. Every location of XCCS code usage had to be tracked down and this revealed a variety of bugs, inconsistent behaviors and maintainability issues, which required substantial refactoring. As of Spring 2024 this work is finally coming to the end, bringing significantly greater robustness and reliability to the Medley Interlisp system.

Additional changes allowed the ingestion of Xerox Alto Bravo-format files, making it possible for legacy documents to be converted to PDF through invocation of an external converter. Also, HELPSYS was extended to allow lookup and display of the Common Lisp Hyperspec and other Medley documentation.

4.2.3 UnixUtils. Medley was enhanced to allow it to reach out to the host environment platform to accomplish system-level tasks that are not available in Medley. These include ShellBrowser, which opens a URL in the specified browser, and ShellOpen, which opens a host-resident viewer for a specified file.

4.2.4 PDFStream. Medley incorporated a native imagestream implementation for producing PostScript™ hardcopy files. The PDF format is not supported as it was not yet invented when this system was developed, but an interim PDF solution from 2023 allows creating a PS file and executing a UnixUtils shell script to convert it to PDF via Ghostscript's ps2pdf utility. Medley's FileBrowser was extended to automatically open PDF files in a separate window using a host-resident PDF viewer.

4.2.5 Github Integration. Github is being used to manage the coordination of multiple developers across several time zones and countries in extending Maiko, Medley, and the tools and utilities. A major effort was the integration of Github functions with the Interlisp File Manager. GITFNS is a set of functions that includes a menu-driven interface to compare Lisp source files on a function-by-function basis, supporting Interlisp's characteristic "residential" approach to development.

4.2.6 Mouse and Keyboard Usage. Originally, Interlisp supported the three-button mice available with many Xerox computers. These have largely disappeared so the interface APIs have been extended to support mice with two buttons (as often used with Windows), one button (as with Macs) or a touchpad. Function keys on several popular keyboards have been mapped to codes emitted by Xerox-type mice to allow access to the original functionality; a "meta" key allows emulation of the middle mouse button of a three-button mouse. Work is ongoing to implement more scroll wheel and middle mouse button functionality.

Only a few keyboard models were available when Interlisp was developed and every application had a unique way of associating input keycodes with program functions. This is impossible now; users want uniform treatment, no matter what keyboard they use. Interlisp.org is working to broadly organize keyboard encoding and communication because keyboard handling is buried in several different locations within Maiko and Medley. Different utilities interpret certain keystrokes in a variety of ways.

Our goal is to build a unified keyboard model that will accommodate a large number of commercial keyboards and unify keyboard handling across tools and applications. This is likely to involve translators interpreting "niche" keyboard types as more common keyboards and/or translators from actual keycodes to an internal model. We expect this to increase the speed with which we can port Medley to different computing platforms.

4.3 Applications

Numerous applications have been built using versions of Interlisp including many early AI tools as well as ROOMS[4], NoteCards[15], a workbench for writing LFG grammars[7], Intelligent Database Assistant (IDA), LOOPS, and the Strategic Automated Discovery System (STRADS) [5]. Recently, an affiliate of Interlisp.org discovered an archive of the Stanford AI Laboratory containing sources for Doug Lenat's AM and Eurisko programs[2], written in an early version of Interlisp. Interlisp.org has demonstrated they can be loaded into a Medley Interlisp environment with minimal changes and are documenting steps to make them usable again.

Interlisp.org is also collecting open source Common Lisp programs and using them to test the Medley Common Lisp implementation, with some changes to make them easier to use, and providing them through our GitHub repository for public use. Some of these applications include ATMS, BB1 and NIKL. Work will continue throughout this year to get them running in Medley reliably and provide minimal documentation (or more, if possible).

4.4 LOOPS

The Lisp Object-Oriented Programming System (LOOPS) is unique among programming systems in that it combines four different paradigms for software development:

- Imperative/Functional Programming
- Object-Oriented Programming
- Aspect-Oriented Programming
- Rule-Based Programming

[2] https://white-flame.com/am-eurisko.html

The integration of these paradigms provides the software architect and developer with a comprehensive toolkit for building large applications [2], allowing for the choice of a data representation and problem solving approach that best meets the needs of a given application. The Medley Interlisp tools and utilities were extended to operate with the LOOPS constructs seamlessly, and Medley's interface tools allow the creation of graphical displays reflecting the values of variables to which they are attached.

Researchers at PARC developed the Truckin' game to help users understand how to program in a multiparadigm environment and visualize what was happening as the game evolved [13]. Truckin' simulated the activity of truck drivers working to make deliveries on time, accounting for geography, varying types of goods and the need to refuel [12]. LOOPS is documented in three books that address the Basic System, the Tools and Utilities, and the Rule-Based System; the latter volume details Truckin' and its development.

4.5 Documentation

Interlisp.org has access to a wide variety of documentation about Interlisp and Common Lisp, including original Xerox PARC manuals, memoranda and program and application documentation from the Computer History Museum's PARC archive. Much of this documentation was written by the original developers who already knew how to use the system and can be obtuse for new users. Interlisp.org has released additional volumes on the usage of Medley Interlisp and LOOPS since 2021, all of which are available at Interlisp.org. These include:

- Interlisp: The Language and its Usage
- Medley Interlisp: The Interactive Programming Environment
- Medley Interlisp: Interactive Programming Tools
- LOOPS Volume I: The Basic System †
- LOOPS Volume II: Tools & Utilities †
- LOOPS Volume III: Rule-Based Systems ††

More work documenting Interlisp and Common Lisp applications is planned for the next two years, with a focus on supporting new users.

4.5.1 Community Outreach. We revamped the Interlisp.org website over the past year to make it easier to navigate, offering visitors an array of options to support further Medley Interlisp development. The website provides access to most of the material that Interlisp.org has collected, with more information being added as we locate sources. Our collection of new and recovered documents extends beyond the website to the GitHub and Zotero repositories.

Interlisp.org continues its outreach to the broader Lisp and computer science community through technical presentations. Three talks were presented in 2023:

- BALISP: In March 2023, the project's efforts were presented to the Bay Area Lisp meetup group. The slides are available on the project's Google Drive[3] and the talk on Youtube[4].
- Software Preservation Network (SPN): On Nov. 2, 2023 Larry Masinter presented to the SPN Idea's Workshop technical details of our work as well as suggesting future collaborative projects across the community.
- BCS Computer Conservation Society (CCS): Steve Kaisler presented a talk entitled "Software Archaeology: The Medley Restoration Project" to the CCS Monthly Meeting on Nov. 16, 2023 in London, England. It included a brief history of Interlisp, a review of some applications, and discussion of challenges in in modernizing Medley Interlisp (some of which have been presented in this paper).

Articles on the Medley Interlisp project have appeared in The Register [11], Hackaday [10] and Hacker News [3].

†Draft available at Interlisp.org

††In progress, forthcoming

[3]https://drive.google.com/file/d/1xpXSoEnc5PPnIa7BHcionBbc8v-Nxp7N/view?usp=sharing

[4]https://www.youtube.com/watch?v=N1MobfEaoWY

ACKNOWLEDGMENTS

We acknowledge the leadership and historical perspective of some of the original developers - Larry Masinter, Nick Briggs, Frank Halasz, Ron Kaplan and the other members of Interlisp.org. We also acknowledge the estate of John Sybalsky for granting rights to use and distribute the source code for Medley Interlisp, and Eric Kaltman and his students at UC Channel Islands for efforts in organizing the Zotero repository. Funding for this effort has been provided by members of Interlisp.org.

REFERENCES

[1] Daniel G. Bobrow and Bertram Raphael. New programming languages for artificial intelligence research. *ACM Comput. Surv.*, 6(3):153–174, sep 1974. ISSN 0360-0300. doi: 10.1145/356631.356632. URL https://doi.org/10.1145/356631.356632.

[2] Daniel G. Bobrow and Mark Stefik. *The LOOPS Manual.* Xerox Corporation, Palo Alto, CA, USA, 1983.

[3] Paolo Amoroso et al. My encounter with Medley Interlisp., January 2023. URL https://news.ycombinator.com/item?id=34300806.

[4] D. Austin Henderson and Stuart Card. Rooms: the use of multiple virtual workspaces to reduce space contention in a window-based graphical user interface. *ACM Trans. Graph.*, 5(3):211–243, Jul 1986. ISSN 0730-0301. doi: 10.1145/24054.24056. URL https://doi.org/10.1145/24054.24056.

[5] Stephen H. Kaisler. A knowledge based system for geopolitical analysis. In *Proceedings of the 57th MORS Symposium.* Military Operations Research Society, Jun 1989.

[6] Stephen H. Kaisler. *Medley Interlisp: The Interactive Programming Environment.* Interlisp.org, Palo Alto, CA, USA, 2021.

[7] Ronald M. Kaplan and John T. Maxwell. *LFG Grammar Writer's Workbench.* Xerox Corporation, Palo Alto, CA, USA, Mar 2003.

[8] David Moon, Kent M. Pitman, Larry Masinter, Brad Miller, Scott Fahlman, Warren Harris, and Jon L White. Issue: Eval-other (version 1), 1988. URL https://github.com/masinter/parcftp-cl/blob/main/cl/cleanup/old-mail/eval-other.mail.

[9] Greg Nuyens. *Font/Character documentation.* Xerox Corporation, Palo Alto, CA, Mar 1986.

[10] Maya Posch. Reviving Interlisp with the Medley Interlisp project, July 2023. URL https://hackaday.com/2023/07/09/reviving-interlisp-with-the-medley-interlisp-project/.

[11] Liam Proven. Revival of Medley/Interlisp: Elegant weapon for a more civilized age sharpened up again, November 2032. URL https://www.theregister.com/2023/11/23/medley_interlisp_revival/.

[12] Mark Stefik. Truckin' and the knowledge competitions, 2017. URL https://www.markstefik.com/?page_id=359.

[13] Mark Stefik, Daniel G. Bobrow, Sanjay Mittal, and Lynn Conway. Knowledge programming in loops: Report on an experimental course. *The AI Magazine*, pages 3–13, 1983.

[14] Xerox Artificial Intelligence Systems. *Interlisp-D: A Friendly Primer.* Xerox Corporation, Pasadena, CA, USA, Nov 1986.

[15] Xerox Special Information Systems. *NoteCards™ Release 1.2 Reference Manual.* Xerox Corporation, Pasadena, CA, USA, Apr 1985.

[16] Warren Teitelman. Interlisp. *SIGART Bull.*, page 8–9, dec 1973. ISSN 0163-5719. doi: 10.1145/1056786.1056787. URL https://doi.org/10.1145/1056786.1056787.

Lisp Query Notation—A DSL for Data Processing

Anders Hoff
inconvergent@gmail.com

ABSTRACT

This paper introduces Lisp Query Notation (LQN). A Common Lisp library, DSL and command-line utility for manipulating text files, and structured data such as JSON and csv, and Lisp Source code.

First we introduce the motivation and design principles. Then we present the LQN syntax, and demonstrate how to use LQN as a library to manipulate data structures in CL. Moreover we demonstrate how to use LQN from the command-line. After describing several operators and their syntax in more detail we finally describe a few possibilities for improvements and further work.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**.

KEYWORDS

demonstration, command-line utility, data processing, domain specific language, structured data, functional programming, common lisp

ACM Reference Format:
Anders Hoff. 2024. Lisp Query Notation—A DSL for Data Processing. In *Proceedings of European Lisp Symposium (ELS'24).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/11001584

1 INTRODUCTION

Lisp Query Notation (LQN) [1] is a Domain Specific Language (DSL), Common Lisp (CL) library, and command-line utility for text and data processing. It draws inspiration from other well-known text processing tools, such as Sed, AWK, and jq[2]. In particular LQN mimics jq's tacit style and chaining of operations.

We start by describing the motivation and main features of the query language. Further we introduce the LQN CL library, before we make the seamless transition to using the LQN terminal commands. Finally we comment on some implementation details and challenges; performance improvements; and potential further work.

2 MOTIVATION, DESIGN AND IMPLEMENTATION

LQN started as an exercise. The primary motivation beyond that is to develop a terse, but intuitive language that makes it fast and convenient to write small (sometimes throw-away) programs; primarily on the command line. Where all—or most—of the processing can be done in the same language. It should also be possible to fall back to conventional CL when LQN is incomplete, inconvenient or insufficient.

It should handle common data formats such as plain text, JSON, csv and Lisp data. Moreover it should handle tasks encountered by e.g. data scientist, data engineers, and when making Generative Art. All of which are practices that require data wrangling. The latter is particularly relevant as we use CL for our art practice. As such it is convenient to export, import and process data in a format native to Common Lisp.

2.1 Compiler

The current compiler performs all code generation in a single pass. Because of this inherent simplicity the core of the compiler is only about 400 lines of code. The syntax is flexible enough that mixing the LQN syntax with CL and other libraries does not appear to present much friction.[3]

2.2 Internal Data Representation

In order to handle multiple data formats LQN always loads all input data into native CL objects. Primarily vectors and hash-tables. E.g. text files are read into vectors of strings; whereas JSON is read into vectors and hash-tables, depending on the structure.

```
[ { "id": "1",
    "objs": [ { "obj": "Ball",
                "is": "round" } ] },
  { "id": "2",
    "msg": "Hi",
    "objs": [ { "obj": "Box",
                "is": "empty" },
              { "obj": "Yak",
                "is": "shaved" },
              { "obj": "Computer" } ] },
  { "id": "3",
    "msg": "Hello!",
    "objs": [ { "obj": "Paper" },
              { "obj": "Bottle",
                "is": "empty" } ] } ]
```

Listing 1: Contents of `dat.json`

2.3 CL Library

In the first example we use the function, jsnloadf, to load some JSON from a file. The contents of dat.json are in listing 1.

```
* (in-package :lqn)
* (jsnloadf "dat.json")
> #(#<HASH-TABLE :COUNT 2 {1793}>
```

[1] https://github.com/inconvergent/lqn (v. 2.0.1)
[2] https://jqlang.github.io/jq/

[3] So far LQN has only been tested in SBCL on Ubuntu 22.04 LTS, with a limited number of other libraries.

ELS'24, May 06–07, 2024, Vienna, Austria

https://doi.org/10.5281/11001584

```
    #<HASH-TABLE :COUNT 3 {1E43}>
    #<HASH-TABLE :COUNT 3 {2B33}>)
```

We see that the JSON file has been loaded into a CL `vector` with three `hash-tables`. Storing data in `hash-tables` and `vectors` internally facilitates easy data manipulation and extraction. For the LQN compiler as well as the user. We can use `ldnout` to serialize data before printing it:

```
* (ldnout (jsnloadf "dat.json"))
> #(((:ID . "1")
     (:OBJS . #(((:OBJ . "Ball")
                 (:IS . "round")))))
    ...
    ((:ID . "3")
     (:MSG . "Hello!")
     (:OBJS . #(((:OBJ . "Paper"))
                ((:OBJ . "Bottle")
                 (:IS . "empty"))))))
```

Note that `ldnout` serializes to a combination of `vectors`, and `alists` with `keyword` keys. We will use the acronym LDN—"Lisp Data Notation"—to refer to this particular way of serializing such a nested structure.[4]

The primary entry point to the LQN compiler is the `qry` macro. The following is a query that simply returns the input. As we will see shortly, _ is used to refer to the current data in any operator or context.

```
* (defvar *dat* (jsnloadf "dat.json"))
* (ldnout (qry *dat* _))
```

2.4 Queries & the Get Operator

There are several different ways to get, select or iterate data in LQN. The simplest is `(@ ...)`, which can be used to access a particular key, index or path. The optional second argument is used as a default value. Here are some examples:

```
* (qry *dat* (@ :1/msg))
> "Hi"
* (qry *dat* (@ :1/abc :missing))
> :MISSING
* (ldnout (qry *dat* (@ :*/msg)))
> #("Hi" "Hello!")
* (ldnout (qry *dat* (@ :*/msg :nope)))
> #(:NOPE "Hi" "Hello!")
* (ldnout (qry *dat* (@ :*/objs/*/obj)))
> [ [ "Ball" ],
    [ "Box", "Yak", "Computer" ],
    [ "Paper", "Bottle" ] ]
```

This makes it easy to quickly look at parts of a data structure. For brevity we omit calls to `ldnout` in the examples from now on.

2.5 Pipe, Map and Filter

Data can be chained, iterated and filtered in a few different ways. First we consider these three operators:

- `(||...)`: pipe the result of each clause to the next clause. Returns the last result. We will see that it usually isn't necessary to use pipe explicitly;
- `#(...)`: map these clauses across a `vector`. If there are multiple clauses they are wrapped in a pipe. Returns new `vector`;
- `[...]`: filter `vector` by one or more clauses. Returns new `vector`.

The following is an example where we chain a map and filter together. First we select the `id` from each item and convert it to an integer; then we drop odd values.

```
* (qry *dat* (|| #( (@ :id) int! )
                 [ evenp ] ))
> #(2)
```

We note that bare symbols (`int!`, `evenp`) inside their respective operators are called as functions with _ as the first argument. Moreover, `qry` will wrap all arguments beyond the first in an implicit pipe operator. So we get the same result if we write this:

```
* (qry *dat* #( (@ :id) int! ) [ evenp ])
```

You can also use arbitrary expressions:

```
* (qry *dat* #( (@ :id) int! (+ 10 _) )
               [ (< _ 13) ])
> #(11 12)
```

If there are multiple clauses in the filter, the default is to include items that match either clause. E.g. `[evenp (< _ 2)]` to select even numbers as well as numbers smaller than 2.

To require multiple clauses to be satisfied, use the `+@` modifier:

```
* (qry *dat* #( (@ :id) int! )
               [ +@oddp (+@< _ 2) ])
> #(1)
```

Similarly, the `-@` modifier drops items on some condition; in this case the number 1:

```
* (qry *dat* #( (@ :id) int! )
               [ oddp (-@= _ 1) ])
> #(3)
```

The full behaviour of the filter modifiers is explained in the documentation. They can be combined to some extent, but if the behaviour of the modifiers do not suit your situation, you can use regular CL, as we have seen already.[5]

2.6 LQN on the Command-line

The transition to use LQN in the terminal is virtually seamless. Currently there are three different entry points to LQN: Namely the commands `tqn`, `jqn` and `lqn`; for `txt`, JSON, and Lisp data respectively. They expect different input data formats, but they all behave in (nearly) the same way. You can always output data to any format from either terminal command, as you can see in this excerpt from the output of `tqn -h` on the command-line:

```
$ tqn -h
> Usage:
    tqn [options] <qry> [files ...]
    cat sample.csv | tqn [options] <qry>
```

[4] In time we might add support for Extensible Data Notation (`edn`), which is a little more more pleasant to look at. LDN will do for now.

[5] We also note that the behaviour of these modifiers is one of the open questions of the overall design of LQN.

```
  Options:
    -v prints the compiled qry before the
    result. For debugging.
    -j output as JSON.
    -l output to readable lisp data (LDN).
    -t output as TXT [default].
    -z preserve empty lines in TXT.
    ...
```

2.7 Processing text

To start, here is a query that splits the incoming `string` at every `"x"`, before it converts each new `string` to uppercase (`sup`). `splt` will trim off any white space by default.

```
$ echo 'a b c x def x 27'\
  | tqn '(splt _ :x) sup'
> A B C
  DEF
  27
```

```
abc
1
33
def
abcdefghi
7
```

Listing 2: Contents of `dat.txt`

Notice that the first expression receives the entire incoming `string` as its input. Whereas "bare" top-level `symbols` inside the implicit pipe operator are called on each individual item in the incoming `vector`; i.e. they are shorthand for the map operator.

Next we read from the `txt` file `dat.txt`. You can see the contents in listing 2. This query finds strings that contain the substring `"ghi"`, as well as all items that can be parsed as an integer by `int!?`:

```
$ tqn '[:ghi int!?]' dat.txt
> 1
  33
  abcdefghi
  7
```

We have seen the filter operator already. But note that now a `keyword` is used as shorthand for case insensitive substring search.[6] `strings` do the same, except then the case is required to match.

The syntax is the same as we saw when using `LQN` as a library. So we can use modifiers to require multiple substring matches:

```
$ tqn '[:+@abc :+@ghi]' dat.txt
> abcdefghi
```

```
animal,cat,angry
animal,yak,shaved
obj,pen,red
shape,ball,round
shape,box,square
```

Listing 3: Contents of `dat.csv`

[6]There is no explicit support for `regex` in `LQN` yet. But using existing libraries is trivial.

2.8 Transforming with Selectors

A frequent task when handling data structures like in listing 1 is iterating all items to perform some selection and/or transformation. `LQN` has several operators for this purpose:

- `{...}`: select keys from a `hash-table` into a new `hash-table`;
- `#{...}`: select keys from `vector` of a `hash-tables` into a new `vector` of `hash-tables`;
- `#[...]`: select keys from `vector` of a `hash-tables` into a new `vector`.

So to select the `id` and `msg` fields from all items we can do this:

```
$ jqn '#{ :id :msg }' dat.json
> [ { "id": "1", "msg": null },
    { "id": "2", "msg": "Hi" },
    { "id": "3", "msg": "Hello!" } ]
```

If you also want to transform some keys you can do this instead:

```
$ jqn '#{ (:id (+ 10 (int! _)))
          (:?@msg sup) }' dat.json
> [ { "id": 11 },
    { "id": 12, "msg": "HI" },
    { "id": 13, "msg": "HELLO!" } ]
```

Again we see that bare symbols are interpreted as a function with the current value as the only argument. Whereas expressions are evaluated as they are.

Notice that we have used the `?@` modifier to handle that the first item is missing a field. In all are three modifiers to augment the behaviour of selectors:

- `?@`: include if the key exists and its value is not `NIL`;
- `%@`: include only if our expression is not `NIL`;
- `-@`: drop this key.

Consider this query to see how the `%@` modifier only includes the `msg` key if the length of the `string` is greater than 3.

```
$ jqn '#{ :id
          (:%@msg (and (> (size? _) 3)
                       (sup _))) }
      ' dat.json
> [ { "id": "1" },
    { "id": "2" },
    { "id": "3", "msg": "HELLO!" } ]
```

Sometimes you want everything except some keys. The `-@` modifier combined with `_` allows us to discard or override keys. As such, the following will yield the same output as we just saw:

```
$ jqn '#{ _ :-@objs
            (:%@msg (and (> (size? _) 3)
                         (sup _))) }
      ' dat.json
```

Selectors, and all other operators can be nested. Here is one more example where we use nested selectors, and print the result as newline separated `JSON`.

```
$ jqn -tjm '#[ (:objs #[ (:obj sdwn)
                         (:?@is sup) ]) ]
```

```
          ' dat.json
> ["ball","ROUND"]
  ["box","EMPTY","yak","SHAVED","computer"]
  ["paper","bottle","EMPTY"]
```

LQN could use more utilities for handling csv files properly. But here is a more complex expression to illustrate how to group items by the first column of the csv file in listing 3; before printing the output as JSON.

```
$ tqn -j '#( (splt _ ",") )
          (?grp (@) (new$ :id (@ 1)
                          :is (@ 2)))
         ' dat.csv
> { "animal": [
      { "id": "cat", "is": "angry" },
      { "id": "yak", "is": "shaved" } ],
    "obj": [...],
    "shape": [...] }
```

2.9 Other Operators

There are several other operators we haven't covered. Here are a few compressed examples that demonstrate additional operators, in combination with what we have seen already:

- ?srch: search nested data with custom expressions. E.g. this will find all symbols in a lisp file and sort them as strings:

  ```
  $ lqn -t "(?srch symbolp) (uniq _)
            (sort _ #'string-lessp)" <file>
  ```

- ?txpr: search and replace nested data. The following will alter the id of any JSON object with at least two items.

  ```
  $ jqn '(?txpr (>= (size? (@ :items)) 2)
                {_ (:id (str! "new-" _))})
        ' [file]
  ```

- ?fld: reduce with a function or expression. This expression will parse and sum integers from the second column in a csv file:

  ```
  tqn '#((splt _ ",") (@ 1) int!?) [is?]
       (?fld 0 +)' [file]
  ```

- ?rec: repeat an expression while the/an expression is T. The following will iteratively add the next Fibonacci number to the end of the incoming vector, until it reaches a number larger than 50:

  ```
  $ echo '#(1 1)' |  lqn '
     (?rec (<= (@ -1) 50)
           (cat* _ (apply* + (tail _ 2))))'
  ```

LQN also has a number of other utilities for numbers, vectors, hash-tables, strings, symbols and lisp data. Such as: getting ranges, indices or keys; concatenating, compacting and combining; comparing; and checking and coercing types. The LQN documentation covers this in more detail.

3 NOTES ON PERFORMANCE

The current implementation of LQN loads all the input into appropriate data structures before executing any of the transforms. This simplifies the implementation, and makes it possible to do some things that would otherwise be difficult or impossible. However, it also has implications for performance; most notably it can increase memory usage. A possible way alleviate this is to adapt the various operators to support e.g. generators or streams in addition to vectors and hash-tables.

Depending on the environment there might be a noticeable delay when using LQN on the command-line, as SBCL first has to start and load the LQN library, before it can load the input data and execute the query. A possible way to reduced this delay considerably is to create an SBCL image (sb-ext:save-lisp-and-die) where LQN is already loaded. Then use the image to compile and execute the query.

Additionally it is noticeably slower to process data from a pipe on the command-line; compared to having LQN read the same data from a file, or from *standard-input*. We have been unable to find an explanation for this, but it is likely an issue in the current implementation; as opposed to an issue with the overall approach.

4 CONCLUSION & FURTHER WORK

lqn is a young experiment. It has not been tested in many different circumstances, or for different tasks. For this reason it is hard to know if the behaviour—such as defaults, and the order of arguments—of the operators and utilities are convenient. Obviously this will always depend on the use-case. But there is likely room for improvement that will become more apparent with further use.

There is also a discussion to be had about the syntax of several of the operators and modifiers. Particularly the selectors in section 2.8. Maybe the overall syntax can be cleaner, or easier to read?

The language is still missing native utilities for data processing. Such as calculating statistics (median, mean, variance), sorting, and aggregations. And while there are some ways to interact with other terminal commands, there is considerable room for improvement. The same applies to interacting with files and the file system. Both from the terminal and when using LQN as a library.

In a similar vein, it would be an interesting challenge to implement a more interactive command-line interface. Where the CLI interaction and syntax is closer to the LQN language than e.g. bash.

Finally we would like to note that we have already found LQN useful for performing code transformation in (CL) compilers for other DSLs. We did not anticipate this from the beginning, but it will probably shape the direction of further LQN development. In particular it would be helpful to improve utilities and operators when it comes to processing CL data; in particular data that represents source code. In all LQN is a useful little language, with potential for expansion in several dimensions.

5 ACKNOWLEDGEMENTS

Thanks to Jack Rusher, Robert Smith and Rainer Joswig for answering my questions; Zach Beane for making and maintaining Quicklisp; and thanks to the larger Lisp community for all their interesting work.

Grants4Companies: The Common Lisp PoC

Philipp Marek
Bundesrechenzentrum GmbH
Vienna, Austria
philipp.marek@brz.gv.at

Björn Lellmann
Bundesministerium für Finanzen
Vienna, Austria
bjoern.lellmann@bmf.gv.at

Markus Triska
Bundesministerium für Finanzen
Vienna, Austria
markus.triska@bmf.gv.at

ABSTRACT

The application *Grants4Companies* was recently introduced in the Austrian Federal Business Portal (*Unternehmensserviceportal*, *USP*). The productive application displays a list of business grants which apply to a business depending on the data available about this business in the systems of Austrian public administration. In this article we describe the underlying Proof of Concept implementation, used to experiment with and test new features. This PoC implementation is written in Common Lisp, interfaces with a Prolog-reasoner, and makes use of formalised grant descriptions based on S-expressions.

CCS CONCEPTS

• **Theory of computation** → *Automated reasoning*; • **Computing methodologies** → **Knowledge representation and reasoning**; • **Software and its engineering** → **Context specific languages**.

KEYWORDS

S-Expressions, Expert System, Applications

ACM Reference Format:
Philipp Marek, Björn Lellmann, and Markus Triska. 2024. Grants4Companies: The Common Lisp PoC. In *Proceedings of the 17th European Lisp Symposium (ELS'24)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.5281/zenodo.10992449

1 INTRODUCTION

Business grants offer vital funding opportunities for businesses and companies, and at the same time provide an important tool for supporting and steering economy. The efficiency of this tool, however, depends on whether it is easily possibly for businesses to find the suitable grants. In order to support businesses in this search in Austria we recently introduced the application *Grants4Companies* in the Austrian *Unternehmensserviceportal*[1] (*USP*, the *Business Service Portal*). The USP is the main portal for contact between businesses and public administration in Austria, currently serving over 120 integrated applications and more than 600.000 registered businesses.

The application *Grants4Companies* was introduced in November 2022 and offers registered businesses the possibility to evaluate a list of business grants based on the data available for the business from sources in Austrian public administration. For this purpose, formal eligibility criteria of a number of business grants are formalised in a logical language. The descriptions and eligibility criteria are based on the *Transparenzportal*[2], the official Austrian portal containing a wealth of data about Austrian funding possibilities for businesses as well as natural persons. Following the explicit consent for using the data available for the registered business, the formalised eligibility criteria are evaluated based on data from Austrian registers. The registers queried are currently the *Unternehmensregister für die Zwecke der Verwaltung*[3] (*Administrative business register*, a register collecting data about businesses from several sources in Austrian public administration) and the *Firmenbuch*[4] (Austrian company register). An extension to further registers is planned. The available grants are then presented to the business based on the evaluation of their criteria as "criteria are satisfied", "criteria are not satisfied" or "information is missing for sufficiently evaluating the criteria".

To quickly evaluate different strategies, ideas, and concepts, we implemented a Proof-of-Concept (POC) in Common Lisp in December 2019; over time we extended this POC to experiment with and test new features. In this article we describe the POC implementation and the underlying design choices. Differences between the POC and the productive software are noted as well. The POC also includes an interface to a logical reasoning engine for proving properties of the grants (or combinations thereof) which are independent of the particular businesses. This reasoning engine is implemented Prolog. Here we focus on the Lisp-specific parts of the POC and refer the reader to the companion paper [5] for the detailed description of the interface to the Prolog reasoning engine.

2 THE INITIAL PROOF-OF-CONCEPT

The POC is written in Common Lisp. Apart from personal preferences the reasons consisted of easier symbolic manipulation, quicker iteration cycles, and better performance. Based on that data point, choosing S-expressions as grant definition format (see Sec. 3) certainly looks like an easy and logical choice, and indeed was selected after careful deliberation. Apart from reading grant definitions, type-checking them, allowing evaluation against (fake) company data (the POC has no connection to the production register data bases), and converting the grant code back to natural language, the POC also acquired (limited) symbolic capabilities: *given a company, which data points (e.g., HQ location) need changing to match additional grants*? Other symbolic computations were made available by transpiling the grant code to Prolog and providing a Prolog REPL in the web interface; this allows queries like *Which grant totally includes another grant?*. For further discussion of this topic please see the already mentioned companion paper [5].

[1] https://www.usp.gv.at/en/index.html

[2] https://transparenzportal.gv.at/tdb/tp/startpage

[3] https://www.statistik.at/en/databases/business-register/administrative-business-register-abr

[4] https://www.justiz.gv.at/service/datenbanken/firmenbuch.36f.de.html

ELS'24, May 6–7 2024, Vienna, Austria

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.5281/zenodo.10992449

3 GRANT LANGUAGE AND SEMANTICS

There are a few thousand programming languages, even without counting the one-offs that are used by less than 10 people worldwide. Some share a bit of syntax, others are completely different. As basis for the formal language for specifying the grant conditions we needed something

- easy to read,
- unambiguous,
- future-proof (ie. backwards-compatible even for vastly changed situations),
- and easy to parse.

The first point means no XML; infix operators with their precedences (see the presentation) are worse off for the second. One format stood out as especially long lasting: S-Expressions. Participants of this ELS will already know, but the quick overview is:

- Evaluation from inner to outer, left to right within one form[5].
- No other precedence rules [6].
- Tokens are either atoms (numbers, strings, symbols), or
  - an opening parenthesis,
  - a list of (zero or more) tokens,
  - and a closing parenthesis.
- Comments are introduced with one or more semicolons (compatible with Common Lisp), but are not discarded but associated with the next form resp. the surrounding form. This allows to have human-readable explanations collected and used for explaining the evaluation of a grant.
- For ease of use (and compatibility with Common Lisp), symbols are defined to be case-insensitive; mixed case is not used. In strings and comments the case is kept, though.

S-Expressions have been cited for these features for a long time[1, 102], even in completely different fields (e.g. music [3, p.171]).

3.1 Concepts of the Formalisation Language

One point that we pondered for quite some time was the actual language used for the concepts of the formalisation language. We finally decided on a German/English mix:

- The specific data-query-functions, i.e., atomic concepts, that are derived from laws written in German were kept in German (`BETRIEBSSTANDORT-IN`, `ÖNACE-IN`, ...); the higher similarity to the original law proved to be helpful in translating to computer language.
- Typical programming "keywords" like `AND`, `OR`, `NOT`, used for constructing complex expressions, were taken from English - the higher familiarity with these (compared to `"UND"`, `"ODER"`, `"NICHT"`, which just remind us of Winword macros!) makes them a better match.

3.2 Packages and Local Definitions

Grant definitions are stored in a GIT repository (for Version Control, Historical, Reproducability, and Data Sharing reasons). A directory tree definition ensures that each funding agency has their own workspace (sub)directory; a scoping rule that reads all grants within a directory into the same package allows funding agencies to define their own higher-level functions (see the presentation for an example).

The package that gets created for each directory imports functions from an API package automatically; so the most-often used symbols can be referenced directly. In the future, some kind of marker (eg. a specific filename like `package.lisp`) might switch to another behaviour: defaulting to another, improved API package, or manually specifying a `DEFPACKAGE` form. This separation into multiple packages also allows to divide up some responsibilities: by having high-level constructs in an extra package, it becomes much easier to maintain that in some separate organization.

An issue that came up right from the beginning is having one concept in multiple different implementations. A clause "*Der Antragsteller muss ein KMU*[6] *sein*" is used in many grants; sadly there are three different definitions for this term, one from the federal government in Austria, one from the EU, and one from the FFG[7].

Making (optional) packages available solves this in a neat way - there are simply three functions, `GV.AT:IS-KMU`, `FFG:IS-KMU`, and `EU:IS-KMU`. This enables the use of different interpretations of the same natural language term depending on the source of the regulation. Of course, the person digitizing the grant needs to know which one to use, which is a separate can of worms.

3.3 Security

Of course, nothing is ever quite so simple. As the funding agency is the (only) one who knows exactly what they want, they're the logical choice for capturing the intent in computer code, and maintaining it later on - including putting a cut-off date on it, or invalidating it some other way. Formalisation of the grants currently still happens via a few people and not the funding agencies, as would ideally be the case. But that means that the "code" (which is "data" here as well) is then run on a different computer system: primarily the central evaluation platform in the USP, but also decentralized on some funding agency's machine (when testing a grant), or potentially at the *Statistics Austria* (when estimating the number of businesses that potentially match some newly-defined grant). So all kinds of concerns regarding security come up! To address these, the specification says that only exported symbols from defined API packages may be used - so it's not allowed (respectively possible) to write `(CL:WITH-OPEN-FILE (s "/etc/shadow") ...)` in a grant to hack the grant evaluation platform.

3.4 Example grant

An example of a grant definition is shown in Fig. 1.

4 EVALUATING GRANTS

As grant forms are *by definition* side-effect free, their evaluation is in principle straightforward: Evaluate the atomic concepts based on the available data, and recursively evaluate complex expressions according to the outermost operator. In the productive version the atomic concepts are evaluated based on the data available for the logged-in business from public administration, while the POC uses

[5] Common Lisp Hyperspec, 3.1.2.1.2.3 Function Forms

[6] "*KMU*" means "*Klein- und Mittelunternehmen*", ie. "*small and medium-sized companies*".

[7] "*FFG*" is the abbreviation for "*Österreichische Forschungsförderungsgesellschaft*" (in English "*Austrian Research Promotion Agency*"), see https://www.ffg.at/en.

```
(define-grant ("Umweltschutz- und Energieeffizienzförderung – Förderung sonstiger Energieeffizienzmaßnahmen Villach"
               (:href "https://transparenzportal.gv.at/tdb/tp/leistung/1052703.html")
               (:transparenzportal-ref-nr 1052703)
               (:Fördergebiet :Umwelt)
               (gültig-von "2019-01-01"))
  "Unter der Berücksichtigung der Verwendung erneuerbarer Energieträger sowie
   der Umsetzung der Intention der Umweltschutz- und Energieeffizienzrichtlinie im
   Bereich privater Haushalte fördert die Stadt Villach folgende Energieeffizienzmaßnahmen."
  ;; Voraussetzungen
  ;;
  ;; - Förderungswerber/innen können natürliche oder juristische Personen sein.
  ;;   Bei juristischen Personen hat die firmenmäßige bzw. statutenkonforme
  ;;   Unterfertigung des Antrages auf Gewährung einer Förderung durch den
  ;;   Vertretungsbefugten zu erfolgen.
  (AND
    (GV.AT:natürliche-oder-juristische-Person)
    ;; - Die Förderungswerber haben bei der Antragstellung zu erklären, dass
    ;;   für die beantragten Förderungen keine weiteren Förderungen von anderen Stellen
    ;;   beantragt wurden.
    ;; - Ein Förderungsansuchen muss spätestens innerhalb von 8 Monaten nach
    ;;   Umsetzung der Maßnahme/n bzw. Kaufdatum bei der Stadt Villach einlangen
    ;; - Die Förderung wird nur für die sach- und fachgerechten Umsetzung der
    ;;   Maßnahme (Einbau) im Stadtgebiet von Villach gewährt.
    (OR
      (Unternehmenssitz-in 20201)
      (Betriebsstandort-in 20201))))
```

Figure 1: Example grant, TPPNr#1052703

dummy data of made-up businesses instead. Apart from this, there are some finer points for taking into consideration.

4.1 Evaluation modes

Because there are two main use cases, the Proof-of-Concept in Common Lisp implements two evaluation modes.

Fully Recursive, Exhausting Evaluation. In this mode all the forms are evaluated and their intermediate results are kept; by reporting these values in the same tree structure as the grant, manual verification of the calculation can be performed (Fig. 2). This evaluation mode is used when displaying grant results for a single company.

Fast Evaluation using Shortcut Properties. As the grants forms are pure, we have a few degrees of freedom for manipulating them or otherwise speed up evaluation. We implemented a short-circuit evaluation which can quickly discard grant/company pairs, allowing for faster mass assessments: given a newly proposed grant, how many companies in Austria will be able to apply? For the future, further optimisation are possible: for commutative operations (like `"AND"`, `"OR"`, numerical addition via `"+"`), we can reorder the forms before compiling. In the typical case of a top-level `AND` we can look at the sub-forms, and move the one with the highest probability for a negative result to the front, benefitting the short-circuiting operation again. This reordering is not implemented yet, though[8].

4.2 Three-valued Logic

Of course not all data required to evaluate whether a company satisfies the formalised eligibility criteria of a grant is always available. While data like *location* of a company needs to be provided before it is officially recognised, e.g., the (Ö)NACE classification[9] is not complete. In particular, for a sizeable number of companies the ÖNACE-classification has not yet been assigned. In addition, a data source might be not available, eg. due to maintenance work.

So the evaluation allows a value of *unknown* as well; many operations then need to propagate that *unknown* upwards. Easy cases are `OR` with a `true` value or `AND` with a `false` value. To be precise, we use (so far quantifier-free) *strong Kleene-Logic* K_3, considered e.g. in [4]. This ensures that grants which have been evaluated for a company to `true` or `false` while some of their atomic components are evaluated to `unknown`, get evaluated to the same result when additional data becomes available and these components no longer return `unknown`. Range-based reasoning for numeric operations would also be possible, but is not implemented yet.

4.3 Extension to probabilities

A major difference between the POC and the production software in the USP is that the POC already got extended to experiment with a 12-bits+1 probability space, with `false` being at one end, `true` at the other, and the `unknown` space spanning the values in between, with the canonical 0.5 `unknown` value in the exact center of the value range. This probability space gets evaluated Bayes-compatibly - so an `AND` over three `unknowns` means 0.5 to the third power, or a probability of 0.125. Of course this makes potentially problematic assumptions about the independence of the sub-expressions of a complex expression. The analysis on the suitability of this approach

[8] See chapter 4.9 below.

[9] https://www.statistik.at/en/databases/classification-database

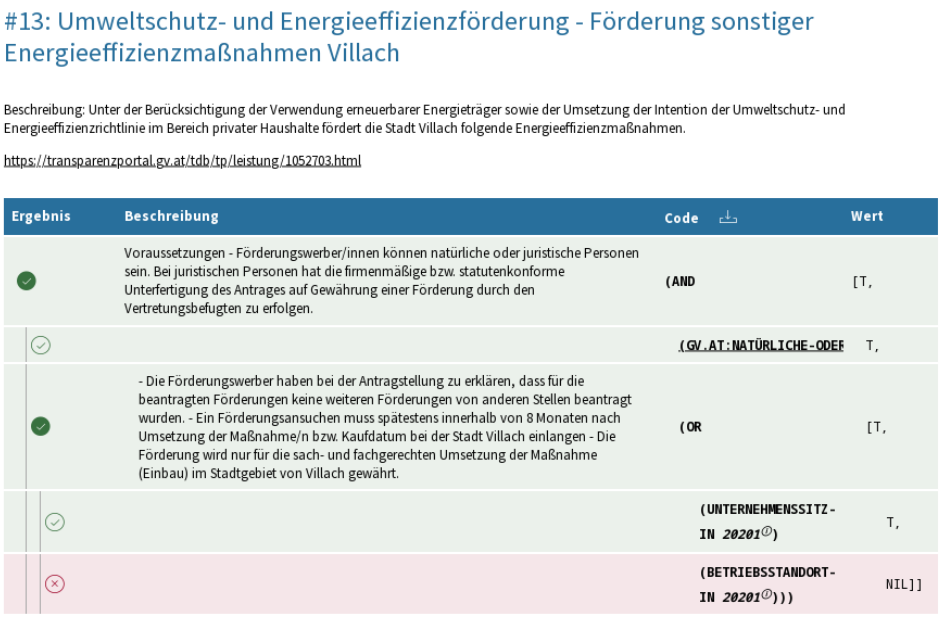

Figure 2: Evaluated example grant with results.

and potential alternative ones is still ongoing, but its implementation in the POC means it is possible to experiment with the approach. By explicitly avoiding saturation[10], the strong Kleene-Logic still applies – but having a range of unknowns means that the output category where it matters most can be sensibly sorted!

4.4 Numeric calculations in the POC

The POC includes a small set of date and numeric capabilities - like checking whether a date precedes another (used to find out how long a company exists), respectively mirroring the calculations in some of the natural language grant texts; as an example, during the pandemic the "*Härtefallfonds*" asked whether the income exceeds 80% of some social security limit. See Fig. 3 for an abbreviated example; for production use the part of the calculation that references a common concept (e.g., the "*sozialversicherungsrechtliche Höchstbeitragsgrundlage*", the *Social security maximum contribution base*) would be extracted in its own function.

4.5 Calculations for the Past

If a calculation must be run later on (to check its validity, an application coming in the next calendar year, etc.), some concepts need to know the *application date*. The previously mentioned "*sozialversicherungsrechtliche Höchstbeitragsgrundlage*", like many other law-mandated values like tax limits, changes over time - but the value that was valid at the date of application must be used (which could be a few years in the past), so the function that encapsulates that concept needs to take the application date into consideration.

[10] So that an AND over 12 or more unknowns will never becomes a `false`, etc.

4.6 Input/output type derivation and -checks

The POC implements the expected boolean operators (AND, OR, NOT) as well as the G4C-specific atoms (like fetching the company legal form, the place of the headquarter, etc.), and some numeric capabilities. That means that grant descriptions (forms) have different input and output types:

- AND, OR, NOT only accept boolean (resp. probability) values;
- the numeric operators expect numbers and return numbers;
- the output of data query functions (atomic propositions) depends on the specific atom.

By using a small set of hard-coded input and output types, the types of all forms in a grant can be fully derived and checked for consistency; also, the expected value type of questions (see below) can be automatically decided and the correct type of HTML input field[11] used in the questioning form. This is one area where having some extra support from the Common Lisp compiler[12] would be a great plus: a stable, documented function to get the compiler-derived types of (some) subforms and a list of type mismatches after compilation. Because some macros are being used[13], association to the source forms might become a challenge, though.

4.7 Interactively asking for Data

Not all data queried by grants is stored in government registers; other data (in particular personal information, like number of disabled employees) would skyrocket the costs if it was stored persistently; and some items cannot be known in advance (eg., clauses

[11] Like `<input type=text>`, `type=number`, `type=date`, or radiobuttons.
[12] not all of them, of course -just one (SBCL) would be enough!
[13] Most notably for AND and similar, so that intermediate results get stored and associated to the subform - a function would only receive input values!

```
;; Im letzten abgeschlossenen Wirtschaftsjahr darf das Einkommen
;; vor Steuern und Sozialversicherungsabgaben maximal
(<= (frage "Einkommen")
    (*
      ;; 80%
      0.8d0
     ;; der jährlichen sozialversicherungsrechtlichen Höchstbeitragsgrundlage
     ;; betragen (https://www.oesterreich.gv.at/lexicon/H/Seite.991498.html).
      (+ (* 12 5370)
         ;; Sonderzahlung
         10740)))
```

Figure 3: Numeric calculation.

that describe the application itself). To reduce the set of potentially applicable grants it makes sense to ask a (limited) number of questions regarding the most often used data items.

The POC includes a high-performance evaluation engine for the grants (see below for details); this allows to recalculate the applicable set of results for the (planned) set of about 3000 grants in the backend and send results back to the frontend, *interactively*. So when some question gets answered, a quick check with the back-end allows to shrink the useful set of questions immediately, reducing the cognitive load on the person using the interface.

As an example, see Fig. 4 for the form before a (too high) number is put into row 2 ("*Anzahl der Kinosäle*"); as soon as the number 100 was acknowledged (typically by pressing `Tab` to get to the next input field), the form data are sent to the POC, which recalculates all grants that contain this question and replies with an update regarding styles (colors) and availability of input boxes - see Fig. 5. As the given value is too high, the conditions of the grant the data is used in (here labelled "`48`") can not be fulfilled any more - so the label's background becomes red, and related inputs are immediately disabled, as there's no need to answer them any more; if there were more questions for other grants, the cursor would move to the next one (done automatically by the browser frontend).

Also, the list of questions is sorted by impact - the more grants' results a question influences, the sooner it is listed.

4.8 Limited Reasoning - "What If"?

Before a Prolog interface was implemented, the POC got (limited) exploration capabilities, giving simple "What If?" answers.

As an example, one computation checks whether a grant would fail to apply because of (parametrized) number of clauses in it (kind of deduplicated, in case they are used in multiple places in a grant, like three `AND` branches all concerned with the `ÖNACE` and some other stuff); this allows to check for things like "*What do I need to change to apply for other grants?*".

4.9 Performance

The Common Lisp POC, utilizing SBCL[14] on standard x86-64 hardware, compiles the grant forms to native code; for nested loops over multiple (test) companies and about 30 grants (including fairly complex ones, see the presentation) the evaluation time averages to 0.5µsec (about 1000 CPU cycles). For ~600000 companies in Austria a test cycle in a browser frontend therefore takes less than half a second, facilitating true interactive grant development.

5 DATA SOURCES

The current sources for data about the companies in the production environment are the "*Unternehmensregister für die Zwecke der Verwaltung*" and the "*Firmenbuch*", a public listing of companies. These two registers provide general data about the company, but in order to evaluate certain eligibility conditions other information about the company might be required. Querying additional registers providing this information is ongoing work.

To enable also the evaluation of conditions, for which no data is available from an official source, in the POC we already drafted the concept `FRAGE` (question), which asks data from the company to answer grant forms (deduplicating questions, and not asking for data that is irrelevant because it won't be used[15].

For the future, we're investigating to add other data sources as well; most of these will (for legal reasons) require an explicit consent from the company.

6 EXPERIENCE REPORT

Our experience matches documented history: Common Lisp is a viable programming language for rapid prototyping. Using it for production use still proves challenging - the strategic focus of most companies is still fixed on Java, changing the multi-man-year lore cannot be done in a day. While there are Prolog libraries for Common Lisp[16], and even one that allows converting Lisp data to Prolog syntax and forward the result to an implementation[17], none of them completely fulfilled the requirements:

- bidirectional communication,
- parsing the Prolog output to provide a highlighted/clickable display in the web UI,
- multiple parallel, independent sessions to concurrently test different analyses,
- ability to export the Prolog input data for use in a separated (offline), ISO-conformant Prolog system.

[14] https://sbcl.org

[15] For example, `(AND (`*<some clause that evaluates to false>*`) (FRAGE "..."))` doesn't need to be asked for this grant – though another grant might require the same data item and is not always rejected!

[16] See, e.g., [2], https://www.lispworks.com/documentation/lw445/KW-W/html/kwprolog-w-152.htm, https://github.com/nikodemus/screamer

[17] https://github.com/cl-model-languages/cl-prolog2

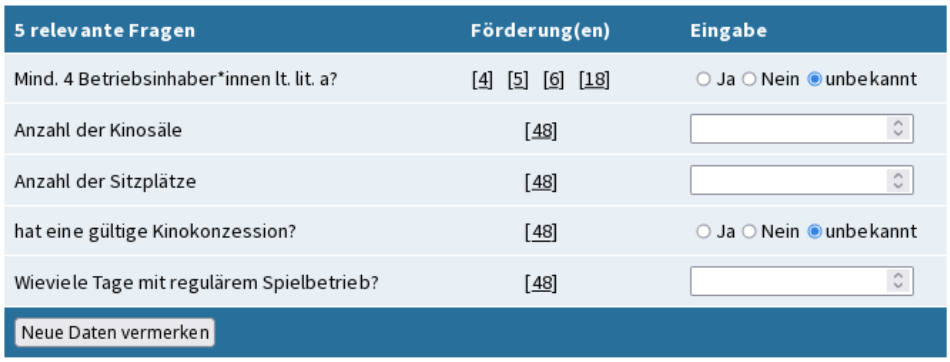

Figure 4: Interactive queries, before answering.

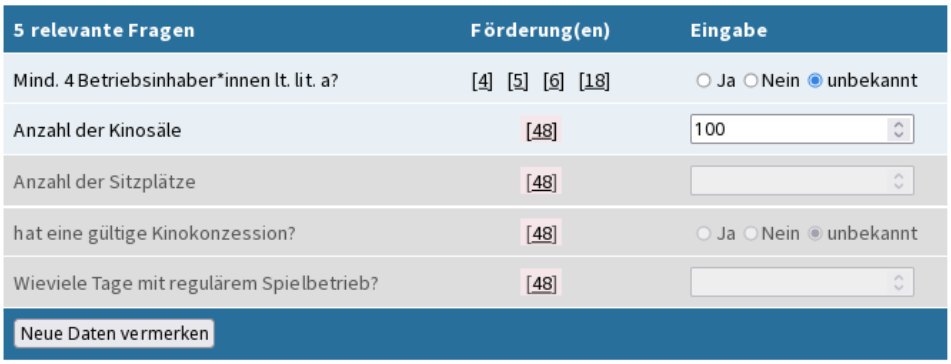

Figure 5: Interactive queries, after input.

So we ended up with our own implementation, using *Scryer Prolog*[18] as backend. The technical iterations proved to be easy; organisational/legistic changes (eg., having computer code on an equivalent legal basis as the grant texts) are hard, and still being worked upon. The bottleneck for broad usage is the translation from natural language to computer code; work takes place to run first translations via an Artificial Intelligence[19]. Of course, to get some real legal weight, a legal spokesperson would need to sign off the translated computer code; designing processes (re-translating the code to natural language for easier comparison again, having the sign-off directly via a GIT commit, etc.) is another required major step forward. Work continues...

7 CONCLUSION

By using plain text files with a reasonably simple syntax it is possible to translate written law into computer-readable data that's at the same time usable as computer code. By using data sources that are defined to contain valid and up-to-date data, a quick pre-selection (ie. not showing grants that are known not to apply to a company) can be provided to company owners. In the future, the fact-checking that currently is done manually could possibly be avoided - either by just providing the available data items in some secure form (a digitally signed JSON-blob), or by simply signing a statement that the company matches the requirements, obviating any need for further checks.

[18] https://www.scryer.pl

[19] Efforts driven by the Ministry of Finance under the umbrella *Law as Code*, though interest is found on the EU level as well, see https://joinup.ec.europa.eu/collection/better-legislation-smoother-implementation/news/new-course-law-code.

REFERENCES

[1] J. Belzer, A.G. Holzman, and A. Kent. 1978. *Encyclopedia of Computer Science and Technology: Volume 10 - Linear and Matrix Algebra to Microorganisms: Computer-Assisted Identification.* Taylor & Francis.

[2] Giuseppe Cattaneo and Vincenzo Loia. 1988. A Common-LISP implementation of an extended Prolog system. *SIGPLAN Notices* 23, 4 (1988), 87–102.

[3] Lounette M. Dyer. 1986. MUSE: An Integrated Software Environment for Computer Music Applications. In *Proceedings of the 1986 International Computer Music Conference, ICMC 1986, Den Haag, The Netherlands, October 20-24, 1986.* Michigan Publishing, 167–172. https://hdl.handle.net/2027/spo.bbp2372.1986.033

[4] Stephen Cole Kleene. 1952. *Introduction to Metamathematics.* North-Holland, Amsterdam.

[5] Björn Lellmann, Philipp Marek, and Markus Triska. 2024. Grants4Companies: Applying declarative methods for recommending and reasoning about business grants in the Austrian public administration (System description). In *Proceedings of FLOPS2024 (accepted).*

[6] William G. Wong. 1983. LISP for CP/M. *Microsystems* 4, 8 (1983), 30–43.

An Introduction to Array Programming in Petalisp

Marco Heisig
Sandoghdar Division
Max Planck Institute for the Science of Light
Erlangen, Germany
marco.heisig@mpl.mpg.de

Abstract

Petalisp is a purely functional array programming language embedded into Common Lisp. It provides simple yet powerful mechanisms for reordering, broadcasting, and combining arrays, as well as an operator for element-wise mapping of arbitrary Common Lisp functions over any number of arrays.

This introduction covers the process of writing high-performance array programs in Petalisp and showcases its main concepts and interfaces. It continues with a simple example of an iterative method and some benchmarks, and concludes with a tour of the Petalisp implementation and a discussion how it achieves high performance and a low memory footprint.

ACM Reference Format:
Marco Heisig. 2024. An Introduction to Array Programming in Petalisp. In *Proceedings of the 17th European Lisp Symposium (ELS'24).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.11062314

1 Introduction

At the 11th European Lisp Symposium in Marbella we proposed a lazy, functional array programming language with significant potential for automatic parallelization. We showed a working prototype and some promising benchmarks that place its performance somewhere above NumPy and below C++[2]. Our hope was that this prototype could be quickly extended to cover more sophisticated problems, and to actually reach the performance of C++. This endeavor turned out to be substantially harder than expected. A key challenge we had to overcome was that of choosing memory layouts with good spatial and temporal locality, and to fairly distribute work across multiple cores. Now, after six years of hard work, we can finally say we have overcome these problems. We proudly present the first production-quality version of Petalisp, and are looking forward to receiving community feedback.

Petalisp is free software. The full source code and many examples can be found at https://github.com/marcoheisig/Petalisp. It can be installed with Quicklisp by typing `(ql:quickload :petalisp)`.

2 Related Work

Array programming is a discipline with a long history. The first array programming language was Kenneth E. Iverson's APL[6], whose terse notation and productivity benefits inspired a multitude of derivatives. Many recent array programming languages, e.g., Repa[7] or Futhark[5], have shifted towards the functional programming paradigm, but sacrificed some amount of interactivity and dynamism on the way. Petalisp delivers all the benefits of purely functional programming while retaining the interactive nature of Common Lisp. A project with a similar goal is the APL compiler April[8], that also targets Common Lisp.

ELS'24, March 06–07, 2024, Vienna, Austria

https://doi.org/10.5281/zenodo.11062314

3 Concepts

There are many approaches to designing a programming language. The one extreme is the *big ball of mud* approach, where more and more potentially competing features are added over time. The other extreme is to have a minimal set of orthogonal features. Petalisp pursues the latter extreme: It features only one data structure — the lazy array, six ways to reorder arrays, one function for combining arrays, and one function for mapping Common Lisp functions over any number of arrays.

3.1 Lazy Arrays

All data manipulated by Petalisp is represented as lazy arrays, which are similar to regular Common Lisp arrays except that their contents cannot be accessed directly, and that they support a more general notion of an array shape. Where the shape of a regular array is defined by its list of dimensions, the shape of a lazy array is defined by a list of ranges. Each range is a set of integers x defined by an inclusive lower bound a, an exclusive upper bound b, and a step size s in the following way:

$$\begin{aligned} a \leq x < b \quad & (x \text{ is bounded by a and b}) \\ s | (x - a) \quad & (s \text{ divides } (x - a)) \\ a, b, s, x \ & \in \mathbb{Z} \end{aligned}$$

Formally, the shape of each lazy array is defined as the Cartesian product of its sequence of ranges. Informally, this means that lazy array have a numbering that doesn't necessarily start from zero, and that each axis can have holes in it as long as those holes are all regularly spaced.

Lazy array shapes have their own shorthand notation, which is a list consisting of tilde symbols and integers. Each tilde must be followed by one, two, or three integers, describing the size, start and end, or start, end, and step of the range in the corresponding axis, respectively. In this notation, a 2×3 array has a shape of `(~ 2 ~ 3)`, and a vector with step size two and four elements has the shape `(~ 0 7 2)`.

Lazy arrays can be created as copies of existing regular arrays or scalars with the `lazy-array` constructor. All Petalisp functions use this constructor to automatically convert their arguments to lazy arrays, so there is usually no need to call it explicitly. For

efficiency reasons there exists a second constructor called `lazy-index-components` for creating lazy arrays whose contents are fully described by a range and an axis. This second constructor is special because the resulting lazy arrays have a memory footprint of zero.

3.2 Evaluation

As mentioned in section 3.1, there is no way to access elements of a lazy array directly. Instead, a user has to convert lazy arrays into the equivalent regular arrays with an explicit function call. The main interface for doing so is the function `compute`. It receives any number of lazy arrays, moves them such that their shapes have a start of zero and a step size of one, and returns the equivalent regular arrays.

3.3 Lazy Map

There are only two mechanisms with which Petalisp communicates with its host language Common Lisp. The first mechanism is the conversion from Common Lisp arrays to Petalisp lazy arrays and vice versa. The second mechanism is that of mapping Common Lisp functions over lazy arrays to obtain new lazy arrays. The higher-order function for doing so is called `lazy` — a rare case of a function whose name is an adjective. The first argument to `lazy` must be a function f of k arguments, followed by k lazy arrays $a_0, \ldots, a_{k-1}$ that are broadcast to have the same shape. The result is a lazy array r of the same shape as the arguments, whose element at index I is defined as

$$r(I) := f\left(a_0(I), \ldots, a_{k-1}(I)\right).$$

Listing 1 illustrates the behavior of `lazy`. There is also another function for lazy mapping called `lazy-multiple-value` that can be used to map functions with multiple return values and gather each of those values in a separate lazy array.

```
(compute (lazy #'*))
 => 1

(compute (lazy #'+ 2 3))
 => 5

(compute (lazy #'+ 2 #(1 2 3 4 5)))
 => #(3 4 5 6 7)

(compute (lazy #'* #(2 3) #2A((1 2) (3 4))))
 => #2A((2 4) (9 12))
```

Listing 1: Examples for using the function `lazy`.

3.4 Lazy Reshape

True to the goal of being a minimalist programming language, Petalisp offers a single function, named `lazy-reshape`, for moving data. It can be used to select, reorder, or broadcast elements of a particular lazy array. All its operations can be described as the superposition of six elementary operations, which are shown in Figure 1.

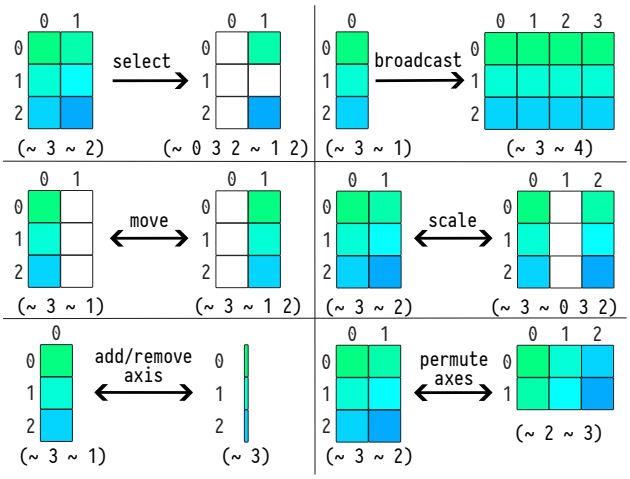

Figure 1: The six elementary reshape operations.

The first argument to lazy reshape is the lazy array that is being reshaped, and all the remaining arguments are so-called *modifiers* that are processed left-to-right and each describe a particular combination of elementary operations. One possible modifier is a shape, in which case the result is a lazy array of that shape and the modification is a combination of selecting, broadcasting, and moving of data. Another possible modifier is that of a transformation, which describes some combination of moving, scaling, permuting, adding, or removing of axes. Transformations can be created using a lambda-like syntax with the `transform` macro, or using the `make-transformation` constructor. Examples of the various kinds of modifiers and their effect are shown in Listing 2

```
(compute (lazy-reshape #(1 2 3 4) (~ 1 2)))
 => #(2)

(compute (lazy-reshape #(1 2 3 4) (~ 2 ~ 3)))
 => #2A((1 1 1) (2 2 2))

(compute (lazy-reshape #(1 2 3 4) (~ 4 ~ 2)))
 => #2A((1 1) (2 2) (3 3) (4 4))

(compute (lazy-reshape #2A((1 2) (3 4))
           (transform i j to j i)))
 => #2A((1 3) (2 4))

(compute (lazy-reshape #(1 2 3 4)
           (transform i to (- i))))
 => #(4 3 2 1)
```

Listing 2: Examples for using the function `lazy-reshape`.

3.5 Lazy Fuse

The final piece of functionality that makes up Petalisp is that of fusing multiple arrays into one. The function for doing so is called

`lazy-fuse`. It takes any number of non-overlapping lazy arrays, determines the shape that covers all these lazy arrays, and returns the array with that shape that contains all the data of the original arrays. An error is signaled in case any of the supplied lazy arrays overlap, or if they cannot be covered precisely with a single shape.

4 The Standard Library

4.1 Moving Data

Shapes and transformations aren't the only valid modifiers accepted by `lazy-reshape`. It also accepts modifiers that are functions that take a shape of the lazy array being mutated, and return any number of further modifiers as multiple values. These functions are called *reshapers*, and they are a generalization of NumPy's relative addressing with negative indices. Petalisp features three built-in functions for constructing reshapers: `peeler`, for removing some of the outer layers of a lazy array, `deflater`, for shifting a lazy array to have a start of zero and a step size of one, and `slicer`, for selecting a particular subset of a lazy array using relative indices.

4.2 Reducing

The function `lazy-reduce` combines the contents of k arrays with a function of $2k$ arguments and k return values. It is an improved version of the multiple value reduction we presented at the 12th European Lisp Symposium in Genova[3].

4.3 Sorting

The function `lazy-sort` constructs a sorting network that sorts the supplied lazy array along the first axis using some predicate and optional key.

4.4 Differentiating

The function `differentiator` can be applied to a list of lazy arrays and a list of gradients at those lazy arrays to return a function that computes the gradient of each input of any of those lazy arrays. This is achieved by using our type inference Typo to compute the derivatives of Common Lisp functions. This functionality can serve as the staring point for writing a machine learning toolkit in Petalisp.

5 Example: Jacobi's Method

Listing 3 shows an implementation of a simple numerical scheme in two dimensions. Although this code is purely functional and has a very high level of abstraction, our benchmark results in Figure 2 show that it has a multi-core performance that is close to hand-optimized C++ code, and even outperforms the popular machine learning framework JAX.

6 The Implementation

Each call to `compute` or any of the other evaluation functions entails a full run through our optimization and code generation pipeline. With a careful choice of algorithms, data structures, and caching schemes, we managed to squeeze the time to execute this entire pipeline to something on the order of a few hundred microseconds. Because of these extremely fast compilation times we can

```
(defun lazy-jacobi-2d (u)
 (with-lazy-arrays (u)
  (let ((p (lazy-reshape u (peeler 1 1))))
   (lazy-overwrite-and-harmonize u
    (lazy #'* 1/4
     (lazy #'+
      (lazy-reshape u (transform i j to (1+ i) j) p)
      (lazy-reshape u (transform i j to (1- i) j) p)
      (lazy-reshape u (transform i j to i (1+ j)) p)
      (lazy-reshape u (transform i j to i (1- j)) p)))))))
```

Listing 3: Jacobi's method on arrays of rank two.

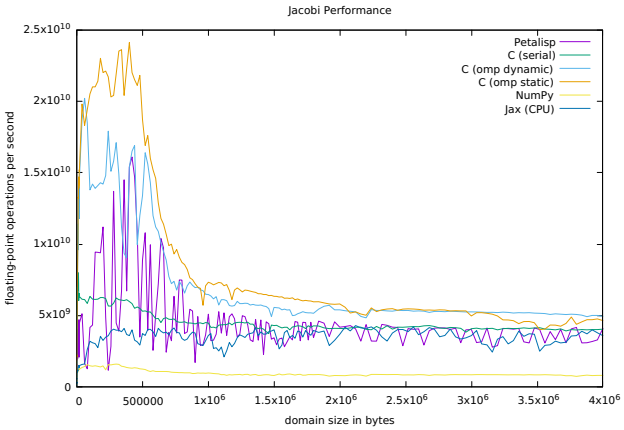

Figure 2: Benchmark results for various implementations of Jacobi's method.

pretend that Petalisp is an interactive language, yet receive all the performance advantages of static compilation.

6.1 Data flow graphs

Initially, each Petalisp program is represented as a data flow graph whose nodes are lazy arrays and whose edges are direct data dependencies. This graph is assembled by invoking Petalisp functions such as `lazy-reshape` or `lazy-fuse`. Several optimizations are already carried out during graph assembly: consecutive reshape operations are combined into one, nodes with no effect are discarded, and fusions of reshape operations that are equivalent to a single broadcasting reshape are represented as such.

The most important optimization at this stage is to narrow down the element types of all lazy arrays produced by lazy mapping, which is a prerequisite for choosing a memory-efficient representation during execution. To do so, we wrote a portable and extremely fast type inference library named Typo that is available at https://github.com/marcoheisig/Typo. Typo can derive the (approximate) return types of almost all standard Common Lisp functions, and it can rewrite calls to polymorphic functions with specialized arguments into calls to more specialized functions.

6.2 Kernels and Buffers

Once a data flow graph of lazy arrays is submitted for evaluation, it is converted to the Petalisp intermediate representation, which is a bipartite graph of kernels and buffers. Each kernel represents some number of nested loops whose body contains loads, stores, and function calls. Each buffer represents a virtual memory region. We developed an algorithm that ensures that most values are produced and consumed in the same kernel so that the size and number of necessary buffers is minimal.

Once a program is converted to this intermediate representation, it is subject to several optimizations: kernels and buffers are rotated in a way that maximizes memory locality, all shapes and iteration spaces are normalized to have a starting index of zero and a step size of one, and buffers that are involved in reduction-like patterns are eliminated and replaced by additional instructions in the adjacent kernels.

6.3 Partitioning

The next step in the optimization pipeline is to break up buffers and the kernels writing to them into shards of roughly equal computational cost, which is a prerequisite for scheduling them onto parallel hardware. We developed an iterative partitioning algorithm that minimizes the amount of synchronization and communication.

6.4 Scheduling

The partitioned intermediate representation is fed into our scheduling algorithm, which is a variant of Blelloch's parallel depth-first scheduler algorithm[1] with several tweaks that improve memory locality. Our custom scheduler has several advantages over general purpose schedulers: It has full knowledge about the origins of each load and the users of each store and the partitioning step has already ensured that all tasks have roughly the same size.

6.5 Allocation

At the end of the scheduling phase, each buffer shard is assigned a particular memory allocation in the following way: buffer shards of similar size are all grouped into one bin, and within each worker and each particular bin, a register-coloring algorithm is used to assign an allocation to each buffer shard while keeping the total number of allocations small.

6.6 Code Generation

When executing the schedule, each kernel is converted to an optimized Lisp function that is invoked on three arguments: The iteration space of a kernel shard, the memory corresponding to each buffer shard, and a vector of all functions that are called in the kernel. Because kernel compilation is rather costly, each kernel is first converted to a hash-consed minimal representation that can be used as a key for caching, and compilation only occurs when a kernel is invoked for the first time.

We already have code generators for turning kernels into Common Lisp code and for turning a subset of kernels into C++ or CUDA code. Right now, our strategy is to use C++ and GCC when possible, and Common Lisp code otherwise. In the future, we plan to make the C++ generator obsolete by using SIMD optimized Common Lisp instead. Doing so would build on our previous work on `sb-simd` that we presented at the 15th European Lisp Symposium in Porto[4].

7 Conclusions

We presented a data flow programming language that masquerades as a Common Lisp library for manipulating arrays. The language stands out by having an extremely simple set of core operations, a versatile standard library, and a mature implementation. A major achievement of our implementation is that it can already outperform optimized C++ code in certain cases — both in execution time and memory consumption. In the past, the manipulation of high-dimensional arrays in Common Lisp has been tedious and often inefficient. Petalisp addresses this issue thoroughly, and turns Common Lisp into an excellent tool for all sorts of massively parallel array programming tasks.

References

[1] Guy E. Blelloch, Phillip B. Gibbons, and Yossi Matias. Provably efficient scheduling for languages with fine-grained parallelism. *J. ACM*, 46(2):281–321, mar 1999. ISSN 0004-5411. doi: 10.1145/301970.301974. URL https://doi.org/10.1145/301970.301974.

[2] Marco Heisig. Petalisp: A common lisp library for data parallel programming. In *Proceedings of the 11th European Lisp Symposium on European Lisp Symposium*, ELS2018. European Lisp Scientific Activities Association, 2018. ISBN 9782955747421.

[3] Marco Heisig. Lazy, parallel multiple value reductions in Common Lisp. In *Proceedings of the 12th European Lisp Symposium*, European Lisp Symposium, 2019. ISBN 978-2-9557474-3-8. doi: 10.5281/zenodo.2642164. URL https://european-lisp-symposium.org/static/proceedings/2019.pdf.

[4] Marco Heisig and Harald Köstler. Closing the performance gap between lisp and c. In *Proceedings of the 15th European Lisp Symposium*, ELS2022. Zenodo, March 2022. doi: 10.5281/zenodo.6335627. URL https://doi.org/10.5281/zenodo.6335627.

[5] Troels Henriksen. *Design and Implementation of the Futhark Programming Language*. Universitetsparken 5, 2100 København, 11 2017.

[6] Kenneth E Iverson. *A Programming Language*. John Wiley & Sons, Nashville, TN, December 1962.

[7] Ben Lippmeier, Manuel Chakravarty, Gabriele Keller, and Simon Peyton Jones. Guiding parallel array fusion with indexed types. In *Proceedings of the 2012 Haskell Symposium*, Haskell '12, page 25–36, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315746. doi: 10.1145/2364506.2364511. URL https://doi.org/10.1145/2364506.2364511.

[8] Andrew Sengul. April: Apl compiling to common lisp. In *Proceedings of the 15th European Lisp Symposium*, European Lisp Symposium. Zenodo, March 2022. doi: 10.5281/zenodo.6381963. URL https://doi.org/10.5281/zenodo.6381963.

Adaptive Hashing

Faster Hash Functions with Fewer Collisions*

Gábor Melis
melisgl@google.com
Google DeepMind
London, United Kingdom

ABSTRACT

Hash tables are ubiquitous, and the choice of hash function, which maps a key to a bucket, is key for their performance. We argue that the predominant approach of fixing the hash function for the lifetime of the hash table is suboptimal and propose adapting it to the current set of keys. In the prevailing view, good hash functions spread the keys "randomly" and are fast to evaluate. General-purpose ones (e.g. Murmur) are designed to do both while remaining agnostic to the distribution of the keys, which limits their bucketing ability and wastes computation. When these shortcomings are recognised, the user of the hash table may specify a hash function more tailored to the expected key distribution, but doing so almost always introduces an unbounded risk in case their assumptions do not bear out in practice. At the other, fully key-aware end of the spectrum, Perfect Hashing algorithms can discover hash functions to bucket a given set of keys optimally, but they are costly to run and require the keys to be known and fixed ahead of time. Our main conceptual contribution is that adapting the hash table's hash function to the keys online is necessary for the best performance as adaptivity allows for better bucketing of keys *and* faster hash functions. We instantiate the idea of online adaptation with minimal overhead and no change to the hash table API. The experiments show that the adaptive approach marries the common-case performance of weak hash functions with the robustness of general-purpose ones.

CCS CONCEPTS

• **Theory of computation → Data structures design and analysis**; • **Information systems → Hashed file organization**.

KEYWORDS

Adaptive, data structure, hash function, hash table, Common Lisp

ACM Reference Format:
Gábor Melis. 2024. Adaptive Hashing: Faster Hash Functions with Fewer Collisions. In *Proceedings of The 24th European Lisp Symposium (ELS'24).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.5281/zenodo.10991322

*…Especially in Certain Situations

ELS'24, May 06–07, 2024, Vienna, Austria

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.5281/zenodo.10991322

1 INTRODUCTION

Hash tables [13, 20] map keys to values and are one of the most fundamental data structures. As such, their performance is of considerable interest. For example, Hentschel et al. [8] claimed in 2022 that "a typical complex database query (TPC-H) could spend 50% of its total cost in hash tables, while Google spends at least 2% of its total computational cost across all systems on C++ hash tables". At that scale, even small gains in this area can have huge impact. This work aims to provide a general framework to improve the performance of hash tables in practice. Theory mostly concerns itself with the unknown key distribution setting and the cost of key lookup abstracted as the number of key comparisons required. This approach is highly successful due to its wide applicability and being a reasonable model *asymptotically* for a certain class of hash functions. However, practice stubbornly happens in the non-asymptotic regime with particular key distributions (even concrete keys) on computers with memory caches. We propose a method to equip hash tables with the ability to change their hash function based on the keys being added to improve their overall performance by bucketing the keys more evenly, making the hash function faster or more cache-friendly. We are not aware of a cost model that incorporates these effects and is amenable to theoretical analysis, hence this work tilts heavily towards Experimental Algorithmics [14].

It is useful to contrast our method with hand-crafting hash functions for particular key distributions. We argue that hand-crafting is too rigid: it breaks badly if the distributional assumption is violated [9, 10]. It is also costly in terms of human labour, and designing hash functions with guarantees is hard. While selecting the hash function offline amortizes the design cost, it also attempts to solve a much harder problem than necessary in balancing the complexity and the quality of the hash function across all possible sets of keys.

A less rigid solution is to select a hash function *online* given the actual keys in the hash table. This pulls the hash function selection cost into the runtime realm, and we must be extremely cautious of introducing overhead lest any possible gains be wasted. Here, we propose hiding some of the selection cost in rehashing, when the hash table grows. In summary, our main contributions are:

- On the conceptual level, we argue that hash functions must be able to change during the lifetime of a hash table for best performance.
- We propose light-weight adaptation mechanism tailored to string and integer/pointer hashing.
- We empirically demonstrate performance gains in a real-life hash table implementation.[1]

[1]The proposed algorithms were implemented within SBCL [15], a high-quality Common Lisp. SBCL hash tables are reasonably fast given the constraints of the ANSI standard [19] but not close to the state of the art. We microoptimized the baseline SBCL v2.4.2 to make performance comparisons fairer.

Algorithm 1 Sketch of a possible adaptation mechanism implemented in *put* with low overhead. The unchanged parts of a usual *put* implementation are grayed out. The hash function may be adapted when there are too many keys in the same bucket, or when rehashing finds that the number of collisions is too high. Alternatively, the total cost of access (Definition 2) could be tracked, requiring an additional write to memory and also a performance penalty to key deletion.

```
1:  function put(key, value)
2:      bucket ← hash(key) mod m
3:      chain_length ← 0
4:      for k ← next key in bucket do
5:          if (hashes are not cached) or cached_hash(k) = h then
6:              if compare(key, k) then
7:                  value of k ← value
8:                  return
9:          chain_length ← chain_length + 1
10:     if chain_length too high then
11:         h ← safer_hash_function(h)
12:         bucket ← hash(key) mod m
13:     if bucket is full then
14:         increase hash table size
15:         adapt_and_rehash()
16:         if hash function was changed then
17:             bucket ← hash(key) mod m
18:     add (key, value) to bucket
```

To ground the exposition, Algorithm 1 sketches the implementation of the high-level part of a possible adaptation mechanism. In later sections, we flesh out this skeletal algorithm.

2 THE CASE FOR ADAPTIVE HASHING

In this section, we present the traditional cost model for lookup in hash tables, which is based on the number of comparisons, then characterize how much being key-agnostic costs in these terms.

We denote the number of buckets with m, the number of keys with n, the keys (assumed to be integers) with k, hash values with $h \in \mathbb{N}_0$, buckets with $b \in [0, m-1]$, and say that hash h falls into bucket b if $h \bmod m = b$. We often refer to vectors of certain sizes, e.g. of hashes, with notation like $h_{1:n}$.

Definition 1 (Bucket Count)**.** *For a given set of hash values $h_{1:n}$ and m buckets, we denote the bucket count vector with $c(h_{1:n}, m) \in (\mathbb{N}_0)^m$, where $c(h_{1:n}, m)_b = |\{i\colon i \in [1, n], h_i \bmod m = b\}|$ is the number of hashes falling into bucket b, for all $b \in [0, m-1]$.*

Next, we define the cost of hash values at a given number of buckets to be the expected number of comparisons one has to make to find the value associated with a key present in the hash table. For a single bucket with c_b hashes, this is $(c_b + 1)/2$ assuming a uniform distribution over the keys being looked up.

Definition 2 (Cost of Hashes)**.** *The cost of hashes $h_{1:n}$ with m buckets is $C(c) = n^{-1} \sum_{b=0}^{m-1} c_b(c_b + 1)/2$, using the shorthand $c = c(h_{1:n}, m)$.*

Note that this definition differs from *hash function quality* in the Dragon Book [1] only in the normalization.

We define perfect hashes as those that fill all buckets as equally as possibile. This generalizes the classic definition [7], which requires no collisions, to the space-restricted setting of $m < n$.

Definition 3 (Perfect Hash)**.** *We say the hashes $h_{1:n}$ are perfectly distributed in m buckets if $m - (n \bmod m)$ buckets have $\lfloor n/m \rfloor$ hashes in them, and $n \bmod m$ buckets have $\lfloor n/m \rfloor + 1$ hashes in them.*

Proposition 4. *[Perfect Hashes have Minimal Cost] Let $U(n, m)$ be the bucket count vector of any perfect hash of n keys and m buckets. Let $q = \lfloor n/m \rfloor$ and $r = n \bmod m$. Then,*

$$C(U(n, m)) = (m - r)\frac{q(q+1)}{2n} + r\frac{(q+1)(q+2)}{2n},$$

and this cost is minimal.

For the proof of this and other propositions, see Appendix A.

In the illustrative special case of an integer load factor n/m, we have that $n = qm$ (i.e. $r = 0$), the counts will be the same for all buckets, and $C(U(n, m)) = m\frac{q(q+1)}{2n} = \frac{q+1}{2}$.

Since perfect hashes have minimal cost, we define the regret of a hash the excess cost over that.

Definition 5 (Regret of Hashes)**.** *The regret of hashes $h_{1:n}$ is*

$$R(h_{1:n}, m) = C(c(h_{1:n}, m)) - C(U(n, m)).$$

Note that the classic cost model in Definition 2 is simple but clearly wrong if hashes of keys are cached (see Algorithm 1) because it costs one memory access to look up the cached hash for a key, but the cost of key comparison may be much higher. Still, with random hashes of many bits, this distinction becomes moot for regret because only one cached hash is likely to match, so there will be a single comparison for each lookup, and their contributions to $C(c(h_{1:n}, m))$ and $C(U(n, m))$ cancel out. Thus, with cached hashes, the regret can be interpreted to be in terms of memory access.

Hash functions strive to be indistinguishable with reasonable effort from a uniform hash [5], which assigns each key to a bucket with uniform probability, whose cost we consider next.

Proposition 6. *[Expected Cost of the Uniform Hash] Let P be a uniform distribution over functions that map keys to buckets. Then,*

$$\mathop{\mathbb{E}}_{\pi_{1:n} \sim P} C(c([\pi_1(k_1), \ldots \pi_n(k_n)], m)) = 1 + \frac{n-1}{2m},$$

where $\pi_{1:n}$ are n independent samples from P.

The uniform hash is optimal among hashes that are functions of a single key. However, its cost is clearly worse when compared to a perfect hash, which can be viewed as having knowledge of all keys. Next, we characterize its regret, assuming that $m \mid n$, for convenience.

Proposition 7. *[Expected Regret of the Uniform Hash] For all load factors $q \in \mathbb{N}$ ($n = qm$), the expected regret of the uniform hash is $0.5 + \frac{1}{m}$.*

So, the uniform hash needs about one extra comparison per two lookups compared to a perfect hash because it does not take all the keys into account. That's not good but not terrible either. It may be worth improving, especially if the hash function can be made faster at the same time.

Algorithm 2 Hashing strings with a *limit* on the number of characters taken into account. The algorithm moves inwards from the two ends of the string because those tend to be the most informative and because this scheme can be easily extended to reuse a previously computed hash with a lower limit. The function *add_char* performs one step of the FNV-1A algorithm.

```
1:  function hash_string(s, limit)
2:      h ← len(s)                          ▷ Initialize the hash to the length
3:      a, b ← 0, len(s) − 1
4:      n ← min(limit, l)
5:      while a < (n ≫ −1) do
6:          h ← add_char(h, s[a])
7:          h ← add_char(h, s[b])
8:          a, b ← a + 1, b − 1
9:      if n mod 2 = 1 then                 ▷ Add the odd middle character
10:         h ← add_char(h, s[a])
11:     return h
```

3 HALF AN EXAMPLE

In related work, Hentschel et al. [8] engage with one half of this problem: they select high-entropy parts of the key to feed to a general-purpose hash function. Their approach can speed up the hash function but cannot reduce the expected number of collisions. Crucially, once a hash function has been learned in an offline manner for a given key distribution, it remains fixed for the lifetime of the hash table. In a similar vein but adapting the hash function on the fly, we demonstrate significant speedups on string hashing even with slightly more collisions.

In particular, we hash only a subset of the data in compound keys, where the size of the subset is subject to a dynamically adjusted limit. In case of string keys, we limit the number of characters hashed. Hashing proceeds inwards alternating between taking a character from the beginning and the end of the string. The algorithm (FNV-1A [21]) is initialized with the length of the string to cheaply introduce some information about the truncated away characters into the hash. See Algorithm 2 for the code listing.

To detect overly severe truncation, we track the maximum chain length. That is, when a new key is being inserted whose hash is computed with truncation (e.g. it's a string longer than the current limit), we check the number of keys already in its bucket. If the probability of that many keys having collided with the uniform hash (without truncation) is less than 1%, we double the limit and rehash. Since hashes of strings are expensive to compute, they are cached (see Algorithm 1, Section 2, and Appendix C). By compromising the hash function's quality, we run the risk of having to perform more comparisons, which can be costly, thus, it is important to have a tight limit on the chain length, which we achieve by precomputing them for all possible power-of-2 bucket counts at load factor 1 and changing the current limit when the hash table is resized.

If after this rehash, the number of collisions is significantly higher than would be expected with the uniform hash, then we double the limit again and rehash. This procedure repeats until there are no more keys with truncated hashes[2] or the number of collisions falls near the expected level (see Appendix B).

[2]We use the highest bit in the hash to indicate truncation.

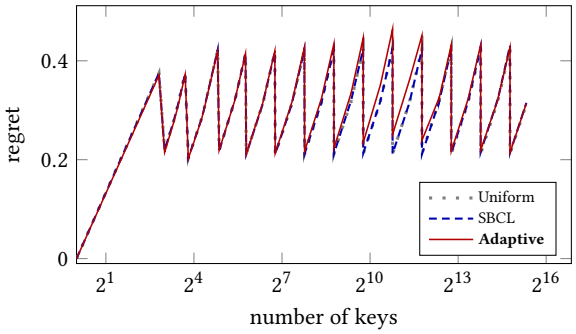

Figure 1: Regret (Definition 5) with string keys. Adaptive does not gain or significantly compromise on regret. Points where the truncation limit changes vary between runs.

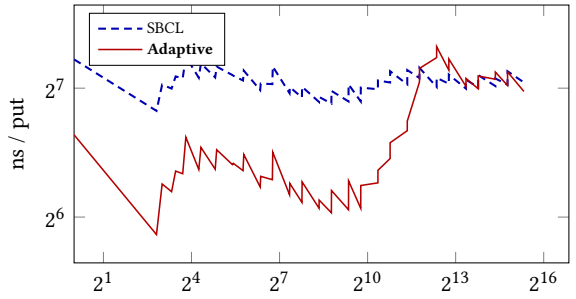

Figure 2: PUT timings in nanoseconds with string keys. Note the log scales. The plot shows the *average* time for inserting a new key when populating an empty hash table with a given number of keys.

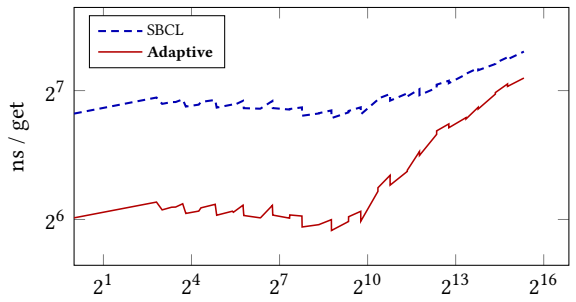

Figure 3: GET timings with string keys.

3.1 Experiments with String Keys

We implemented the adaptive hash algorithm by changing SBCL's standard `equal` hash tables[3]. Then, we collected all different strings present in the running Lisp, which gave us about 40 000 keys and measured the time it takes to populate hash tables from an empty state, averaging over same-sized random subsets of the keys. We partitioned the range of possible key counts into maximally large segments within which the hash table internal data structures are

[3]Common Lisp's `equal` is like Java's `.equals()`: it compares two objects by value.

not resized and measured performance with the lowest and highest possible key count in each segment. We also measured the time it takes to look up an existing key (GET), to look up a key not in the hash table (MISS), and to delete an existing key (DEL). All reported times are in nanoseconds per operation (e.g. insertions for populating the table, lookups for GET). For details of the experimental setup see Appendix F.

Figures 1 to 3 show our results. The jaggedness of the lines is the effect of hash table resizing. At smaller sizes, the gains are considerable and similar to those in Hentschel et al. [8]. With the current implementation, the performance of the adaptive method eventually falls back to the baseline (unmodified SBCL) because the max-chain-length check in Algorithm 1 gets triggered by strings that share a long common prefix and suffix, pushing the truncation limit beyond the length of most keys.

Note that a more advanced implementation could reduce the overhead of rehashing by starting Algorithm 2 from the hash value produced with a lower *limit* and only hashing the characters beyond that. While this would help PUT results a bit, GET would not benefit. The more fundamental problem is that max-chain-length is a really loose indicator of the cost, and we should track the regret instead.

3.2 Experiments with List Keys

The solution used in the string case also works for lists with a small modification, which is another common type of key in `equal` hash tables. Since lists are not random-access, we only consider their prefixes (unlike Algorithm 2, which also includes suffixes of strings). At least since the year 2000, stock SBCL has truncated list keys to length 4 to avoid stack exhaustion in case its recursive hashing algorithm is invoked on a circular key. This value might have been chosen empirically to maximize performance, or user code has adapted to this limitation even if the root cause of the issue remained unrecognised, or both. Regardless, we found that a default limit of 4 worked best. From this default, the limit is increased as collisions warrant, as in the string case. So, the practical gains for the adaptive method may be limited to cases where crucial bits in keys are put unknowingly beyond length 4. Curiously, there are two such cases in SBCL's own test suite (`arith-combinations.pure.lisp` and `save4.test.sh`), which experience a disastrous number of collisions and are sped up by 60% when the adaptation mechanism increases the truncation length.

4 INTEGER AND POINTER KEYS

Section 2 indicates that it may be possible to improve the regret by adapting the hash function to the keys on the fly, but whether there is a practical implementation of adaptation – which covers a non-trivial set of workloads and is lightweight enough to benefit overall performance – remains to be demonstrated.

In the previous section, we showed an example of how to reduce the complexity of the hash without increasing the number of collisions too much. In this section, we instantiate the general idea of adaptive hashing on the problem of hashing integer and pointer keys [11]. In this case, we will consider reducing the number of collisions at the same time as speeding up the hash function, requiring more than a passing familiarity with the key distribution.

4.1 Perfect Hashing on Arithmetic Sequences

First, we consider the idealized case of adding keys to a power-of-2 hash table from an arithmetic sequence of integers in order. Let $a_i = a_0 + id$ for all $i \in [1, \dots]$, where a_0 is the offset and d is difference between successive elements. a perfect hash here is $h_i = \lfloor a_i/d \rfloor = h_0 + i$, but this requires division by an arbitrary constant d, which is slow on current hardware.

Since the number of buckets m is a power of 2, as long as d is odd, any finite progression in a will be perfectly distributed modulo m because d and m are coprime. Let s be the largest integer such that $2^s \mid d$. Then $h_i = \lfloor a_i/2^s \rfloor$ is an arithmetic sequence with odd increment $d/2^s$, thus perfectly distributed. So, if we know s, we can use the perfect hash function $k \longrightarrow k \gg s$ with a single arithmetic shift. Less regular than arithmetic sequences are the addresses of sequentially allocated objects, which we consider next.

4.2 Page-Based Memory Allocators

If keys are memory addresses (e.g. pointers to objects), then we may be able to take advantage of how the allocator works. We consider the case of page-based allocators, which first allocate contiguous memory ranges called pages from the OS. From these pages, they are then able to allocate objects much more quickly. To decrease contention, pages are often assigned to individual threads. A thread may have multiple pages assigned to it, in which case it may choose between pages based on the allocation size. In particular, TCMalloc allocates pages of 8KB by default and has allocations of roughly the same size within the same page. SBCL, a Common Lisp implementation with a moving garbage collector (GC), has two 32KB pages per thread: one for conses (whose size is two machine words), and another for all other objects[4].

Under such allocators, if the hash table keys are of the same size and are allocated in a tight loop, we can expect to have roughly arithmetic progressions in terms of addresses. But only roughly because with TCMalloc there may be holes (that belonged to previously freed objects) on the page, which may be filled in a more irregular pattern, and when a page is full, the new page may be anywhere in memory. With SBCL, pages have no holes because allocation within a page is simply a pointer bump[5] and because the GC compacts. When there is not enough room left on the current page or GC happens, the allocator gets a new page, but the addresses of subsequent pages are much less regular than the address within pages. A further complication with SBCL is that there are no separate pages for objects of different sizes, so if objects of non-constant sizes are allocated between subsequent keys, that throws regularity off and may reduce the density of addresses of keys within the page.

In summary, to the extent that addresses of keys are distributed like elements of pure arithmetic progressions, their hashes can be improved. We leverage the following properties:

- **denseness**: many keys are allocated on the same page,
- **alignment**: the power-of-2 alignment of keys is constant (especially within a single page).

[4]This is a simplification. Some platforms also have immobile space, arenas, and "large" objects are handled specially.

[5]The address of the next object is the address of the previously allocated object plus its size aligned to a double word boundary.

Algorithm 3 Detecting common low bits in integer keys (e.g. pointers from page-based allocators) $k_1, \ldots k_n$. This is to find the largest power-of-2 factor of the common difference in an arithmetic progression regardless of the offset caused by the first term. The symbols $\vee$, $\oplus$, $\neg$ denote the bitwise OR, XOR and NOT operations. Note that *count_leading_zero_bits* is often a single assembly instruction such as LZCNT on x86.

```
1: function count_common_prefix_bits(k_1, ..., k_n)
2:     mask ← 0                          ▷ Changed bits detected so far.
3:     for i ← 2 to n do
4:         mask ← mask ∨ (k_1 ⊕ k_i)
5:     return count_leading_zero_bits(¬mask)
```

4.3 Detecting Common Power-of-2 Factors

We need to detect s in the factor 2^s common to all keys fast and safely. Fast because this will be done at runtime, and safely because using an overestimation of s in e.g. the Arithmetic hash $k \to k \gg s$ can discard valuable bits and lead to a disastrous number of collisions, which must then be detected and corrected by the another change to the hash function (see Algorithm 1 and Algorithm 4).

Algorithm 3 is designed to fulfil these requirements. It is extremely light, performing about 2 bitwise assembly instructions per key, over only a subset of keys to limit memory access. Also, it detects the common factor without assuming that the sequence is arithmetic, which makes it applicable in more circumstances.

As to safety, if we have n keys, then the probability of a bit appearing constant by chance is 2^{1-n} (assuming that bits are Binomial(0.5) in the hash values). Thus, in practice, we can detect constant low bits with high probability with as few as 8-16 keys. We use the detected shift s in the following three hash functions.

4.4 The Arithmetic Hash

The Arithmetic hash function is $k \to k \gg s$, where $s \in \mathbb{N}_0$. As discussed in Section 4.1, this is a perfect hash function for arithmetic progressions.

4.5 The Pointer-Mix Hash

The Arithmetic hash function can easily have high cost if, for example, pointers on multiple pages come from the same smallish subset on each page.

Our next hash function, Pointer-Mix, combines the Arithmetic hash $k \gg s$ with a general purpose hash of the page address $k \gg \text{PB}$, where PB is the base 2 logarithm of the allocation page size in bytes. The Pointer-Mix hash function is $k \to k \gg s \oplus \text{safe}(k \gg \text{PB})$, where $\oplus$ is the bitwise XOR operation, and safe() is a general purpose hash function such as Murmur3.

Next, we characterize its regret in the setting discussed in Section 4.2, when keys are allocated in a tight loop but do not fit on a single page. The keys are pointers to objects distributed uniformly between multiple pages and within pages form subsets of values of arithmetic sequences of the same increment.

Proposition 8. *[Expected Cost of Pointer-Mix] Let $k_{1:n}$ be integer keys, and $\mathcal{P} = \{k_i \gg \text{PB} : i \in [1, n]\}$ the set of pages (the high bits of keys). Let the keys be distributed over the pages uniformly, $n = |\mathcal{P}|u$, where u is the number of keys on the same page ($u = |\{i\colon k_i \gg \text{PB} = p\}|$ for all pages $p \in \mathcal{P}$). We assume that all u keys on the same page form random subsets of arithmetic progressions with page specific offsets but the same increments. Then, the expected cost of the Pointer-Mix hash function is*

$$1 + \frac{n - u \min(1, \frac{2^{\text{PB}-s}}{m})}{2m}.$$

See Appendix A for the proof.

This cost is upper bounded by that of the uniform hash (Proposition 6), $1+\frac{n-1}{2m}$, and we can see that with a few densely packed pages, we can get reasonable improvements, which diminish quickly with more pages and sparsity. Meanwhile, at the single-page extreme ($u = n \leqslant m$), Pointer-Mix is a perfect hash.

4.6 The Pointer-Shift Hash

We have seen that as the number of pages grows, Pointer-Mix quickly falls back to the performance level of the uniform hash. It is also slow: it includes a general purpose hash.

In practice, we found that the Pointer-Shift hash $k \to k \gg s' + k \gg \text{PB}$ often outperforms Pointer-Mix. Furthermore, it also behaves rather similarly to the Arithmetic hash if s' and PB are not close in value. To avoid degenerating to $k \to 2k \gg \text{PB}$ at $s = \text{PB}$, s' is set to a large value in this case to zero out the first term without introducing a slow conditional.

4.7 The Constant Hash

The hash functions discussed up to now are adaptive as they all involve the shift s detected from the keys. A different kind of adaptation, based the number of keys but ignoring their values is also possible. We mirror the common practice of starting with an array plus linear search and switching to a hash table above a predefined, small number of keys but hide it behind the hash table API and utilize it when the comparison function is extremely light as is the case with integer / pointer hashing. This may also be viewed as as a *Constant* hash with a specialized implementation or a hash table with only one bucket.

4.8 Other Hashes

Stock SBCL comes with the *Prefuzz* hash function, which was designed by hand and performs well empirically in many situations. Naturally, Prefuzz takes advantage of the memory allocation patterns to make common use-cases fast (e.g. `symbol` keys, frequently used in the compiler), but it does so at the expense of extreme penalties to others.

The *Murmur3* mixer function is a fast, widely used, general-purpose, non-cryptographic hash function with strong mixing properties. Its bucket distribution is very close that of the uniform hash.

4.9 Adapting the Hash Function

We implemented the above hash functions in SBCL and modified its hash table implementation to perform adaptation with pointers and integers. So, we use Common Lisp's `eq` hash tables, whose comparison function is based on object identity (i.e. it compares the address of non-immediate objects, or the value of immediate objects such as integers that fit into a machine word) similarly to the == operator in Java.

Algorithm 4 Adapting the hash function in `eq` (i.e. object identity based) hash tables at rehash. Note that we count collisions with Prefuzz only at larger sizes; otherwise it adapts only through the max-chain-length mechanism (see Algorithm 1) to reduce the overhead. We refer to this algorithm as Co+PS>Pr>Mu when comparing minor variations.

Require: Integer/pointer keys $k_i (i \in [1, \ldots, n])$, doubled number of buckets m, current hash function h.

```
1:  procedure adapt_and_rehash_eq
2:      if h = constant_hash then
3:          if m = 64 then
4:              s ← count_common_prefix_bits(k_{1:10})
5:              h ← pointer_shift
6:      if h = pointer_shift then
7:          n_collisions ← rehash(m, h, count_collisions = True)
8:          if n_collisions is too many then
9:              h ← prefuzz
10:     if h = prefuzz then
11:         if m < 2048 then
12:             rehash(m, h)
13:         else
14:             n_collisions ← rehash(m, h, count_collisions = True)
15:             if n_collisions is too many then
16:                 h ← murmur3
17:     if h = murmur3 then
18:         rehash(m, h)
```

The adaptation mechanism for `eq` hash tables (Algorithm 4) fleshes out Algorithm 1 and works as follows. Hash tables are initialized with the Constant hash, which remains in effect until the number of keys exceeds 32. At that point, *count_common_prefix_bits* determines the shift s to use, and we switch to the Pointer-Shift hash function. Whenever rehash finds that there are significantly more collisions than would be expected with a uniform hash (see Appendix B), the hash table switches to Prefuzz.

We also need to fall back on Murmur3 in case Prefuzz produces too many collisions. However, because Prefuzz is somewhat robust and considerably faster than Murmur3, we count collisions only at larger sizes to reduce the adaptation overhead[6] and otherwise rely on the max-chain-length mechanism from Algorithm 1 to fall back on the next safer hash function (in the order they appear in *adapt_and_rehash_eq* in Algorithm 4) if the key being inserted falls into a bucket with at least 14 other keys[7]. Using the tighther limit from Section 3 would be too costly for `eq` hashing. See Appendix D for the more details.

In summary, we have two ways of detecting when the current hash function is likely suboptimal: tracking collisions at rehash and chain length at insertion. We use collision tracking to catch gradual degradations of performance, and max-chain-length to catch catastrophic failures in a single bucket. One can construct lower and upper bounds on the average cost of lookup based on the collision count and max-chain-length. Neither of these mechanisms are perfect, but they work quite well in tandem to inform the adaptation logic about the cost of lookup.

[6]The overhead is that of incrementing a single counter if the bucket in `index-vector` is not zero, which is a hard to predict branch for the CPU.

[7]With a uniform hash, the probability of the maximum chain length being at most 14 is higher than 99% for all possible hash table sizes.

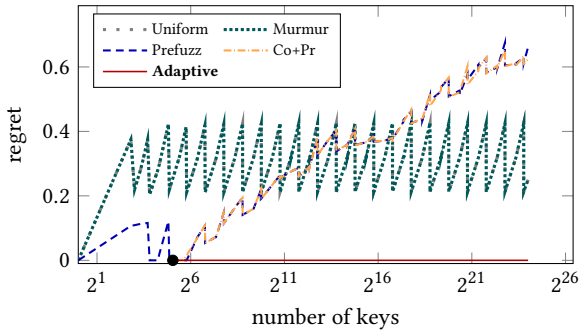

Figure 4: Regret with `FIXNUM :PROG 1`. Murmur closely tracks Uniform. Prefuzz is aggressively optimized for small sizes. Adaptive (Algorithm 4) is a perfect hash here. Both Co+Pr (Constant followed by Prefuzz) and Adaptive use the Constant hash until the fixed switch point at 32 keys (black dot).

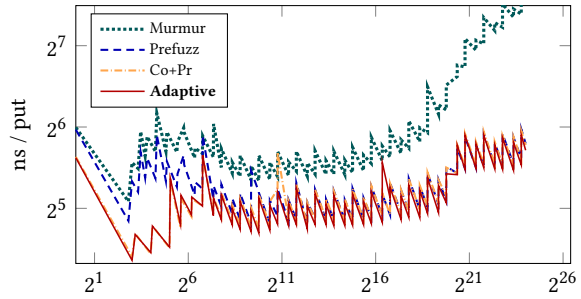

Figure 5: PUT timings with `FIXNUM :PROG 1`. Prefuzz outperforms Murmur even at large sizes despite higher regret because it's friendlier to the cache (its collisions are between subsequent elements of the progression), and its combination with Constant is even faster. Thus, despite being a perfect hash, Adaptive can improve on them only marginally.

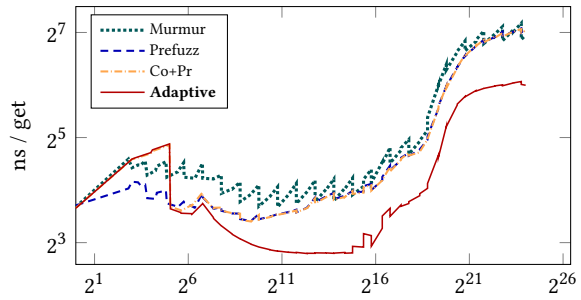

Figure 6: GET timings with `FIXNUM :PROG 1`. Keys are queried in random order so regret matters more here than with PUT, but the cache-friendliness of Prefuzz still keeps it ahead of Murmur. As expected, Adaptive can finally benefit from its zero regret after the Constant hash phase.

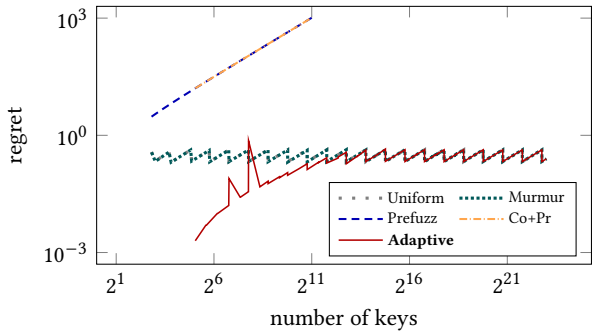

Figure 7: Regret with `FLOAT :PROG 1`. To be able to plot the catastrophic failure of Prefuzz (and of Co+Pr, consequently), we use log scale for regret on this graph. Single floats are especially problematic for Prefuzz because they can have many constant low bits. Adaptive detects these constant low bits and does better than Uniform until variation in the floating point exponents makes its original estimate of the number of constant low bits invalid, and the resulting gradual increase in collisions makes it switch to Prefuzz at rehash time. This is a spectacularly bad idea in this scenario, and the high number of collisions causes an immediate switch to Murmur. The switch times vary by hash table because the key sets are generated starting from random offsets.

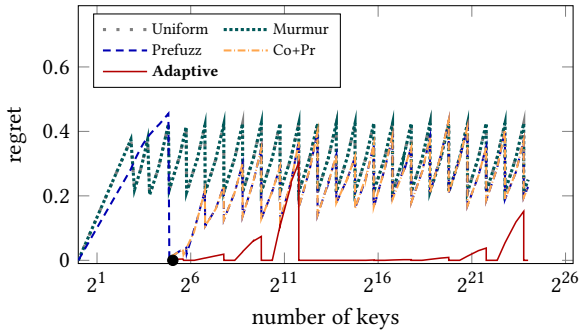

Figure 8: Regret with `FIXNUM :PROG 12`. Murmur closely tracks Uniform, but Prefuzz is better across almost the whole range. Arithmetic (Section 4.4) would be a perfect hash here, but Adaptive, which uses Pointer-Shift (Section 4.6), is not quite perfect due to the interference of its $k \gg$ PB term.

Both the collision count and max chain length are proxies for the cost of lookup as defined in Definition 2, which in turn is a proxy for performance that ignores important factors such as cache effects. Thus, even if adaptation can be driven by proxy statistics, there is no way around actually measuring performance directly.

4.10 Microbenchmarks

We conducted experiments on SBCL's `eq` hash tables with various hash functions, hash table sizes, integer and pointer keys. Our experimental methodology is the same as in Section 3.1 except for how keys are generated, which we briefly describe below (see Appendix F for the details). In the experiments, Co+Pr starts with the Constant hash and switches to Prefuzz above 32 keys. Adaptive behaves as described in Algorithm 4.

- (`FIXNUM :PROG 1`) The first experiment is with `fixnums`[8] following an arithmetic progression with increment 1 (denoted by `:PROG 1`). As Figure 4 shows, the regrets of Murmur and Uniform are very close, as expected. Prefuzz seems to be aggressively optimized for small hash table sizes, and Adaptive is a perfect hash in this simple scenario. However, differences in regret do not predict actual performance well, which is most visible in Figure 5, where insertion is much faster with Prefuzz than with Murmur even where the latter has much smaller regret. This is because the collisions with Prefuzz are between keys close in insertion order, which benefits the CPU's cache. The same effect is present in lookups (Figure 6), although to a lesser degree because the benchmark looks up keys in random order, so some random access to memory is inevitable. See Appendix I for the full set of results.
- (`FLOAT :PROG 1`) Similar to the previous `fixnum` case, we also tested `single-float` keys. As the regret curves in Figure 7 show, Prefuzz suffers a catastrophic failure, placing most keys in the same bucket, but Adaptive takes advantage of the many constant low bits in the keys, eventually falling back to Prefuzz and then to Murmur. See Appendix J for the full set of results.
- (`FIXNUM :PROG 12`) Next, we tested arithmetic progressions with increment 12. This is intended to test whether the shift detection in Algorithm 3 works. The regret curves in Figure 8 and the operation timings tell a similar story as with `:PROG 1` except that Adaptive is no longer a perfect hash due to Pointer-Shift's $k \gg$ PB term. See Appendix K for more.
- (`FIXNUM :RND 6`) Like `:PROG 6`, but keys follow a random progression: 0–5 values are skipped randomly between subsequent keys. This is intended to approximate populating a hash table with non-uniformly sized keys or values being allocated in a tight loop. Results in Appendix L shows that Prefuzz is better than Murmur, and the large gains made by the Constant hash persist, but with less structure and more noise in the key sets, Prefuzz is becoming harder to beat.
- (`CONS :RND 6`) Similar to `FIXNUM :RND 6`, but we allocate real `cons` objects. See Appendix M for the results.
- (`SYMBOL :EXISTING`) Finally, we explore the case where keys are not allocated a tight loop that populates the hash table by using the set of existing `symbols` from Lisp as keys. Appendix N shows that despite the scarcity of structure in the key distribution, Prefuzz maintains a small advantage over Murmur. As expected, Co+Pr and Adaptive follow in suite, most of their advantage being in the Constant hash phase.

In all results presented, the Co+Pr and Adaptive hash functions, which both start out with the Constant hash, gain a lot of performance on insertion but lose on lookups. This is still an overall win based just on the numbers presented here except in very lookup-heavy workloads. However, an even larger unquantified benefit is in

[8] A fixnum is a signed integer in Lisp that fits into a machine word. It's similar to an `int64` on x86-64.

the reduced memory usage and garbage collection times due to the Constant hash having a specialized single vector implementation.

In summary, a general-purpose hash such as Murmur is a safe but suboptimal choice for many common situations in `eq` hash tables. SBCL's own Prefuzz hash is hand-crafted for these common situations, on which it is difficult to beat. Still, because it is non-adaptive, it sacrifices performance in other cases and has terrible worst-case behaviour. Our adaptive approach combines the worst-case safety of Murmur with the common case performance of Prefuzz. The adaptive approach even manages to slightly outperform Prefuzz because having the reliable safety net of the fallback mechanism allows it to be more aggressive in catering to the common case.

4.11 Macrobenchmarks

Microbenchmarks are useful indicators of the highest achievable throughput, but their results do not necessarily carry over to more complex workloads, where factors such as code size and complexity gain importance. To investigate this issue, we conducted experiments where hash table operations constitute only a small fraction of the workload. In particular, we measured the times to

(1) compile and load a set of libraries;
(2) run the tests of the same set of libraries;
(3) run each test file in SBCL's `tests/` directory.

Appendix G has the detailed results; here, we only provide a summary. Among the three benchmarking suites, the first one is the heaviest on `eq` and `equal` hash table operations. With SBCL's statistical profiler, we estimated that about 1.7% of the total runtime was spent in small `eq` hash tables (that is, within Constant hash's range of 0–32 keys) and 1.3% in larger ones, while operations on `equal` hash tables took 1%. The relative speedup was 8% in the large `eq` case and 50% for `equal`. The gain in the small `eq` case is harder to pin down because Constant hash's significantly reduced garbage collection cost; we estimate it to lie in the 8%–14% range.

In the second suite, small/large `eq` and `equal` hash table operations constituted 0.55%, 0.25% and 0.3% of the baseline result. Relative gains were as previously except for the large `eq` case, possibly because our benchmarking methodology is not able to detect differences so small, or because the increased code size is not worth it in code paths so cold.

Finally, we timed SBCL tests on a per-file basis and found no major performance regressions. Overall, we observed a 0.7% gain due to adaptive `eq` hashing, with adaptive `equal` hashing contributing another 1.5%, which came almost exclusively from the two tests with longer list keys (see Section 3.2).

In summary, by surviving the difficult transition from hot-path microbenchmarking to the cooler workloads reported in this section, adaptive hashing emerges as a method of practical relevance.

5 RELATED WORKS

Our method takes inspiration from Perfect Hashing, which selects a hash function for a given set of keys (known as static hashing). As its name implies, Dynamic Perfect Hashing [3, 6] allows the set of keys to change but still guarantees worst-case constant lookup time. However, it requires more memory than plain hash tables, so it is not a drop-in replacement for them. Cuckoo hashing also has worst-case constant lookup time but with a lower memory footprint. In a sequential implementation, it requires 1.5 hash function evaluations and memory accesses on average per lookup, which is about what the uniform hash has at load factor 1 (see Proposition 6).

Probably Dance [16], Rabbit Hashing [17] use the maximum probe length to decide when to grow the hash table or reseed the hash function but, unlike our adaptive method, they do not change its functional form, nor do they select the hash function to fit a given set of keys (see Algorithm 3).

VIP hashing [12] moves the more frequently accessed keys earlier in the collision chains to reduce the average lookup cost. Since adapting to the key access distribution is performed online, this method also needs to take extreme care to minimize overhead. In contrast to our work, they perform online adaptation of the hash table internal storage layout but leave the hash function constant.

Hentschel et al. [8] propose a method to learn the hash function offline from samples of the key distribution by using the most informative parts of compound keys. Our case study on string keys in Section 3 can be seen as an online version of their method, with the expensive learning phase removed.

In the taxonomy of Chi and Zhu [4], adaptive hashing falls under data-dependent hashing although they assume that the training is performed offline. The adaptive hashing method can also be viewed as a more robust, key-aware version of user-defined hash functions, which are also adapted offline to a particular key distribution.

There is a history of handcrafted hash functions that perform well in common cases, but exhibit spectacular failures in others. As we have seen, Prefuzz in SBCL is one such example. Java used to hash only about 1/8 of the characters in long strings [9]. This hardcoded limit made hashing faster, but as it could lead to lots of collisions without no fallback mechanism to save it, from JDK 1.2 on, all characters are hashed.

6 CONCLUSION

We have laid out the case for adaptive hash functions, which reside between key-agnostic and perfect hashes. Our primary contribution lies in reconceptualizing hash tables as inherently adaptive data structures, which can marry the theoretical guarantees of universal hashing with the common-case performance of weak hash functions. To support this viewpoint, we implemented this approach in a real-life system and demonstrated improved performance as well as robustness on string and integer/pointer hashing by capturing real-life key patterns and providing efficient search algorithms. The design space opened up by the adaptive hashing framework is large, and the adaptation mechanisms investigated in this work hardly cover a substantial or particularly imaginative part of it, leaving ample room for further developments. In particular, the max-chain-length mechanism can form the basis of a defense against denial-of-service collision attacks without constraining the choice of hash function [2].

Finally, the source code of the SBCL changes and the benchmarking code to reproduce the experimental results are open-sourced and available at https://github.com/melisgl/sbcl/tree/adaptive-hash.

ACKNOWLEDGMENTS

We thank Christophe Rhodes, Miloš Stanojević, Andrew Senior, Paul-Virak Khuong, and the reviewers for their valuable comments.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006. ISBN 0321486811.

[2] Jean-Philippe Aumasson and Daniel J Bernstein. SipHash: a fast short-input PRF. In *International Conference on Cryptology in India*, pages 489–508. Springer, 2012.

[3] Djamal Belazzougui, Fabiano C Botelho, and Martin Dietzfelbinger. Hash, displace, and compress. In *European Symposium on Algorithms*, pages 682–693. Springer, 2009.

[4] Lianhua Chi and Xingquan Zhu. Hashing techniques: A survey and taxonomy. *ACM Computing Surveys (Csur)*, 50(1):1–36, 2017.

[5] William Collins. *Data Structures and the Java Collections Framework*. McGraw-Hill Science/Engineering/Math, 2004. ISBN 0073022659.

[6] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. Dynamic perfect hashing: Upper and lower bounds. *SIAM Journal on Computing*, 23(4):738–761, 1994.

[7] Michael L Fredman, János Komlós, and Endre Szemerédi. Storing a sparse table with 0 (1) worst case access time. *Journal of the ACM (JACM)*, 31(3):538–544, 1984.

[8] Brian Hentschel, Utku Sirin, and Stratos Idreos. Entropy-learned hashing: Constant time hashing with controllable uniformity. In *Proceedings of the 2022 International Conference on Management of Data*, pages 1640–1654, 2022.

[9] JDK bug database. JDK-4045622: java.lang.String.hashCode spec incorrectly describes the hash algorithm. https://archive.fo/LB0wY, 1997. Accessed: 2024-04-14.

[10] JDK bug database. JDK-4669519: Hashmap.get() in JDK 1.4 performs very poorly for some hashcodes. https://bugs.java.com/bugdatabase/view_bug?bug_id=4669519, 2023. Accessed: 2024-04-14.

[11] Bob Jenkins. Integer hashing. https://web.archive.org/web/20070210182431/http://burtleburtle.net/bob/hash/integer.html, 2007.

[12] Aarati Kakaraparthy, Jignesh M Patel, Brian P Kroth, and Kwanghyun Park. VIP hashing – Adapting to skew in popularity of data on the fly (extended version). *arXiv preprint arXiv:2206.12380*, 2022.

[13] Hans Peter Luhn. A new method of recording and searching information. *American Documentation*, 4(1):14–16, 1953.

[14] Bernard ME Moret. Towards a discipline of experimental algorithmics. In *Data structures, near neighbor searches, and methodology*, pages 197–213. Citeseer, 1999.

[15] William Newman. SBCL: Steel Bank Common Lisp, 1999. URL https://sbcl.org.

[16] Probably Dance. I wrote the fastest hash table, 2017. URL https://probablydance.com/2017/02/26/i-wrote-the-fastest-hashtable/. Accessed: 2024-04-14.

[17] Rabbit Hashing. Rabbit hashing, 2015. URL https://github.com/tjizep/rabbit. Accessed: 2024-04-14.

[18] Kyle Siegrist. Probability, mathematical statistics, stochastic processes. https://www.randomservices.org/random/urn/Birthday.html, 1997.

[19] Guy Steele. *Common LISP: The language*. Elsevier, 1990.

[20] Wikipedia contributors. Hash table — Wikipedia, The Free Encyclopedia, 2024. URL https://en.wikipedia.org/w/index.php?title=Hash_table&oldid=1207615479. [Online; accessed 3-March-2024].

[21] Wikipedia contributors. Fowler–Noll–Vo hash function — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Fowler%E2%80%93Noll%E2%80%93Vo_hash_function&oldid=1215467221, 2024. [Online; accessed 14-April-2024].

A PROOFS

We restate the propositions from Section 2 and Section 4.5 and provide proofs.

Proposition 6. *[Expected Cost of the Uniform Hash] Let P be a uniform distribution over functions that map keys to buckets. Then,*

$$\mathop{\mathbb{E}}_{\pi_{1:n}\sim P} C(c([\pi_1(k_1),\dots\pi_n(k_n)],m)) = 1+\frac{n-1}{2m},$$

where $\pi_{1:n}$ are n independent samples from P.

Proof. Because we sample a hash function independently for each key, $\pi_i(k_i)$ are independent and distributed uniformly over the key space. Writing the expected number of comparisons for a lookup as a sum over the ith key added to the hash table, we get

$$\frac{\sum_{i=0}^{n-1}\left(1+\frac{i}{m}\right)}{n} = \frac{n+\frac{n(n-1)}{2m}}{n} = 1+\frac{n-1}{2m}. \qquad \square$$

Proposition 4. *[Perfect Hashes have Minimal Cost] Let $U(n,m)$ be the bucket count vector of any perfect hash of n keys and m buckets. Let $q=\lfloor n/m\rfloor$ and $r = n \bmod m$. Then,*

$$C(U(n,m)) = (m-r)\frac{q(q+1)}{2n} + r\frac{(q+1)(q+2)}{2n},$$

and this cost is minimal.

Proof. A set of hash values is either perfect or non-perfect. Since all perfect hashes have the same cost $C(U(n,m))$, if we could construct a perfect hash with lower cost from any non-perfect hash, it would follow that $C(U(n,m))$ is minimal.

Next, we show one such construction. For any non-perfect set of hash values with bucket counts c, there are always two buckets i and j such that $c_i > c_j + 1$ (else it would be a perfect hash due to bucket counts having to sum to n). By moving one hash from bucket i to j, we get a new set of hash values with bucket counts c' whose cost is lower because $2n(C(c)-C(c')) = c_i(c_i+1)+c_j(c_j+1)-(c_i-1)c_i-(c_j+1)(c_j+2) = c_i - c_j - 1 > 0$. $\square$

Proposition 7. *[Expected Regret of the Uniform Hash] For all load factors $q \in \mathbb{N}$ ($n = qm$), the expected regret of the uniform hash is $0.5+\frac{1}{m}$.*

Proof. Let $\pi_{1:qm}(k_{1:qm}) = [\pi_1(k_1),\dots,\pi_{qm}(k_{qm})]$. Then,

$$\begin{aligned}
&\mathop{\mathbb{E}}_{\pi_{1:qm}} R(\pi_{1:qm}(k_{1:qm}),m)\\
&\quad= \mathop{\mathbb{E}}_{\pi_{1:qm}} C\big(c(\pi_{1:qm}(k_{1:qm}),m)\big) - C\big(U(n,m)\big)\\
&\quad= 1+\frac{qm-1}{2m}-\frac{q+1}{2}\\
&\quad= 0.5+\frac{1}{m}.
\end{aligned} \qquad \square$$

Proposition 8. *[Expected Cost of Pointer-Mix] Let $k_{1:n}$ be integer keys, and $\mathcal{P} = \{k_i \gg \mathrm{PB} : i \in [1,n]\}$ the set of pages (the high bits of keys). Let the keys be distributed over the pages uniformly, $n = |\mathcal{P}|u$, where u is the number of keys on the same page ($u = |\{i\colon k_i \gg \mathrm{PB} = p\}|$ for all pages $p \in \mathcal{P}$). We assume that all u keys on the same page form random subsets of arithmetic progressions with page specific offsets but the same increments. Then, the expected cost of the Pointer-Mix hash function is*

$$1+\frac{n-u\min(1,\frac{2^{\mathrm{PB}-s}}{m})}{2m}.$$

Proof. First, we look at the case where there can be no collisions between keys on the same page. This is true if the keys form an arithmetic progression and are not just random subsets. Due to the subset assumption, it is also true if the hash table is large enough to hold a page worth of keys (shifted by s): $log_2(m) \geqslant \mathrm{PB}-s$.

The cost decomposes as the sum of the number of hashes in the same bucket as keys are added one by one. Thus, when there are $(p-1)$ previous pages' worth of keys already in the hash table, all u keys on the next page will contribute the same amount $1+(p-1)\frac{u}{m}$ to the cost because there are no collisions between them.

$$C = n^{-1}\sum_{p=1}^{|\mathcal{P}|} u\left(1+(p-1)\frac{u}{m}\right) = 1+\frac{n-u}{2m}.$$

Second, if keys (shifted by s) on the same page may collide randomly modulo m, then we can expect to get only $um^{-1}2^{\mathrm{PB}-s}$ guaranteed no colliding keys. Updating our formula, we get that

$$C = 1 + \frac{n - u\min(1, \frac{2^{\mathrm{PB}-s}}{m})}{2m}. \qquad \square$$

B THE EXPECTED NUMBER OF COLLISIONS

Here, we derive computationally cheap upper bounds on the expected number of collisions with the uniform hash for testing whether the observed number of collisions in Algorithm 1 is too many.

Given n keys, m buckets, and hash values $h_{1:n}$, let c and u be the number of collisions and unused (empty) buckets, respectively. The number of used buckets $m - u$ is equal to the number of non-colliding keys $n - c$, so $c = u + n - m$ and it suffices to bound u to bound c. Appealing to the birthday problem [18], the expected number of unused buckets is

$$\mathbb{E}\, u = m\left(1 - \frac{1}{m}\right)^n.$$

Holding the load factor $f = n/m$ constant, the proportion of unused buckets is monotonically increasing in m and

$$\lim_{m\to\infty} \mathbb{E}\,\frac{u}{m} = \lim_{m\to\infty} \mathbb{E}\left(\left(1 - \frac{1}{m}\right)^m\right)^{n/m} = \exp\left(-\frac{n}{m}\right) = \exp(-f),$$

using the product limit formula of the exponential function.

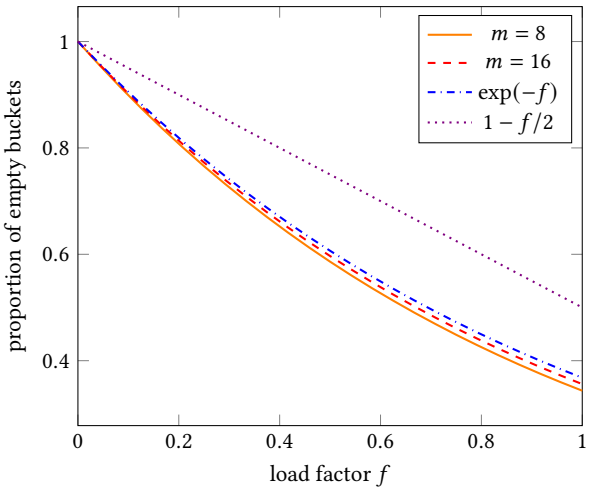

Figure 9: Expected proportion of empty buckets with the uniform hash given a load factor with the two smallest possible hash table sizes ($m = 8$ and $m = 16$), the tight upper bound $\exp(-f)$, and the loose but cheap upper bound $1 - f/2$ used for small sizes.

Based on this upper bound, for small eq and all equal hash tables, we use the test $m\exp(-f) < 0.9u$ (where $u = m + c - n$) to indicate whether the expected number of unused buckets (hence collisions) is too high.

For eq hash tables when $m < 2048$, we use a faster to compute and looser upper bound. In SBCL's hash table implementation, the load factor cannot exceed 1, so we only need to consider the $[0, 1]$ interval. In this interval, $\exp(-f) \leqslant 1 - f/2$. Then, from $\mathbb{E}\,\frac{u}{m} \leqslant \exp(-f)$, we have that

$$\begin{aligned} \mathbb{E}\,\frac{u}{m} &\leqslant 1 - f/2 \\ \mathbb{E}\, u &\leqslant m - n/2 \\ \mathbb{E}\, c + m - n &\leqslant m - n/2 \\ \mathbb{E}\, c &\leqslant n/2. \end{aligned}$$

Since this upper bound is already quite loose at most load factors, we use $c > (n \gg 1)$ to test for too many collisions at small hash table sizes.

C SBCL HASH TABLES

SBCL hash tables are technically separate-chaining, meaning that there are explicit chains of keys which fall into the same bucket. Ironically for a Lisp, these chains are not lists: for performance reasons, they are represented by two arrays of indices, called the `index-vector` and the `next-vector`), which are only resized when the hash table grows. The main pieces fit together as follows:

- The vector `pairs` holds alternating keys and values in a stable order, which is important for iteration (e.g. `maphash`). The first key is at index 2.
- `index-vector` is a power-of-2-sized array of indices, that maps a bucket to the index of the first key–value pair in `pairs` or zero if it's empty.
- `next-vector` maps the index of a pair to the index of the next pair in the collision chain or zero at chain end. It also chains empty slots in `pairs` together.
- For all but the lightest hash functions (standard eq and eql tables), the hash values of all keys in the hash table are cached in `hash-vector`. At lookup, the cached hash is compared to the hash of the key being looked up, and if they are different, then we know without invoking the potentially expensive comparison function that they cannot match (Algorithm 1).

An important operation is *rehashing*. When the number of buckets increases, keys are reassigned ("rehashed") to buckets based on their hash values, which are taken from `hash-vector` if there is one. Rehashing iterates over `pairs`, rewriting `index-vector` and `next-vector`.

Each standard hash table type (`eq`, `eql`, `equal`, `equalp`) have separate accessors (GET, PUT, DEL), which are invoked through an indirect call, and the two lighter ones have the hash function and comparison function inlined. There is no SIMD, and SBCL does not devirtualize calls.

D IMPLEMENTATION DETAILS

`Eq` hash tables are initialized with the Constant hash and a `pairs` vector is allocated for up to 8 key–value pairs. At this time, there is no `index-vector` and `next-vector`. The `pairs` vector is doubled in size as more keys are added, but no rehashing is necessary as there are no chains yet. Once the number of keys exceeds 32, *count_common_prefix_bits* (see Algorithm 3) is invoked with 16 keys to guess the shift s, we switch to the normal SBCL hash table

implementation, set the hash function to *pointer_shift* and rehash (Algorithm 4). Switching to the new hash function is implemented as changing the set of accessors.

We added new hash table accessors for Murmur and added a new slot for the detected shift s to the hash table structure. Instead of adding separate accessors for Pointer-Shift, which actually had the best performance in microbenchmarks, we made Pointer-Shift and Prefuzz share accessors, and to the inlined hash function we added an $if\ s = 0$ that dispatches to Prefuzz. This is to reduce pressure on the instruction cache, which is an important consideration in macrobenchmarks. To minimize the performance penalty on rehashing, we lift this dispatch out of the rehashing loop in the same way that dispatches to different accessors are.

For `equal` hash tables, we pack the truncation limit and the current max-chain-length into a single machine integer and store it in the same slot that we used for s in the `eq` case.

E BENCHMARKING ENVIRONMENT

All reported results are from a single Intel Core i7-1185G7 laptop running Linux with the `performance` scaling governor. Turbo boost and CPU idle states were disabled.

```
echo performance > /sys/firmware/acpi/platform_profile
echo 1 > /sys/devices/system/cpu/intel_pstate/no_turbo
cpupower -c 3 idle-set -D 0
```

The benchmarking process was run at maximum priority (`nice -n -20`), pinned to a single CPU (`taskset -c 3`).

F MICROBENCHMARKING METHODOLOGY

We plot the estimated time it takes to do a particular operation vs the number of keys (between 1 and at most 2^{24}) in the hash table for different key types and allocation patterns. All hash table variants compared in this paper allocate memory only when the insertion of a new key requires growing the internal data structures.[9] To reduce the computational burden of the benchmarking process, we list ranges of key counts within which no resizing takes place and only measure performance at the minimum and maximum key count in each range, assuming that linear interpolation is a reasonable approximation in between.

At each such key count, we then estimate the average time it takes to perform a hash table operation (e.g. PUT). First, a set of keys is generated for the given type (e.g. `FIXNUM`) and allocation pattern (e.g. `:RND 6`). The hash table is allocated in an empty state with comparison function `equal` for string keys or `eq` in all other cases. Then, we measure the average time it takes to perform a given operation for keys in the key set:

- PUT: Inserting one key when populating the hash table with all keys in the key set. Keys are inserted in the order they were generated.
- GET: Looking up a key in the key set. Keys are looked up in random order.
- MISS: Looking up a key not in the key set. Keys are looked up in random order.

[9] Moreover, they do so in an identical and deterministic pattern, so we exclude the time spent in garbage collection from the measurements. The exception to this pattern is the Constant hash, whose specialized implementation has a smaller than usual memory usage.

Table 1: Estimated means and relative standard errors of real (wall clock) and CPU times in seconds to *compile and load* a set of libraries.

	Real time	±RSE%	CPU time	±RSE%
Pr	24.068	0.02%	24.037	0.02%
Mu	24.152	0.01%	24.117	0.01%
Co+Pr	23.976	0.03%	23.945	0.02%
Co+Mu	23.979	0.03%	23.943	0.03%
Co+Pr>Mu	23.988	0.02%	23.955	0.02%
Co+PS>Pr>Mu	23.951	0.02%	23.918	0.01%
Equal*	23.824	0.02%	23.792	0.02%

Table 2: Estimated times to *test* a set of libraries.

	Real time	±RSE%	CPU time	±RSE%
Pr	25.512	0.03%	24.493	0.02%
Mu	25.632	0.05%	24.632	0.04%
Co+Pr	25.367	0.03%	24.352	0.02%
Co+Mu	25.360	0.04%	24.342	0.03%
Co+Pr>Mu	25.385	0.04%	24.367	0.03%
Co+PS>Pr>Mu	25.372	0.03%	24.374	0.02%
Equal*	25.257	0.03%	24.256	0.02%

- DEL: Deleting a key in the key set. Keys are deleted in random order.

The above steps are performed in the order listed. That is, first the hash table is populated, and PUT is measured, followed by GET then MISS. Finally, DEL timings are taken. At the end of the DEL phase, the hash table is once again empty.

These average times over key sets have a low relative standard deviation of about 0%–2%. To reduce the variance further, we take multiple such measurements and report their average. In particular, we take at least 3 measurements, then continue until the total number operations performed exceeds 5 000 000.

Finally, when the number of keys is less than 100, timing granularity is a limiting factor, so we allocate a number of hash tables (plus the two key sets for each, the second being for MISS) and measure the total time it takes to e.g. populate them. The number of hash tables is chosen such that the total number of keys is at least 100. This is a low enough number that the total memory footprint of the allocated hash tables stays below 32KB, the CPU's L1 cache size in our benchmarking environment.

SBCL had a 16GB heap.

G MACROBENCHMARK RESULTS

Here, we describe our macrobenchmarking experiments in Section 4.11 in more detail. In the first suite, 16 libaries were compiled and loaded with `(asdf:load-system <library> :force t)`. In the second, the tests of the same libraries were run with `(asdf:test-system <library>)`. In the third, SBCL tests were run with file-by-file with `tests/run-tests.sh`.

We compared the following configurations (following a naming convention like in Algorithm 4):

Table 3: Estimated times to *run SBCL tests*.

	Real time	±RSE%	CPU time	±RSE%
Pr	582.164	0.02%	452.344	0.03%
Mu	582.858	0.02%	453.069	0.02%
Co+Pr	580.071	0.02%	450.294	0.03%
Co+Mu	579.448	0.02%	449.644	0.03%
Co+Pr>Mu	578.909	0.02%	449.035	0.02%
Co+PS>Pr>Mu	578.207	0.02%	448.392	0.03%
Equal*	568.890	0.02%	439.091	0.02%

(1) **Pr**: unchanged baseline version with the Prefuzz `eq` hash and the non-adaptive `equal` hash
(2) **Mu**: the `eq` hash was changed to Murmur3
(3) **Co+Pr**: Constant hash followed by Prefuzz
(4) **Co+Mu**: Constant hash followed by Murmur3
(5) **Co+Pr>Mu**: Constant hash followed by Prefuzz with fallback to Murmur3
(6) **Co+PS>Pr>Mu**: Constant hash followed by Pointer-Shift with fallback to Prefuzz then to Murmur3 (called Adaptive in Section 4.10).
(7) **Equal***: Like the previous, but the `equal` hash is also adaptive (see Section 3).

Note that since Pr and Co+Pr lack an eventual fallback to Murmur, they have an easy-to-trigger unbounded worst case, so it may be more pertinent to think of Mu as the baseline.

In the benchmarking environment described in Appendix E, all individual benchmarks in the three benchmark suites (e.g. running the tests of a single library or a single SBCL test file) were run 10 times on each configuration with their runs interleaved to reduce the effect of correlated noise. We estimated the mean time a benchmark took on each configuration and computed the standard error of this estimated mean. We report the total estimated mean times for each benchmark suite along with their relative standard errors (RSE) in Tables 1 to 3.

H RESULTS FOR STRINGS

Keys are sampled from the set of all (about 40 000) strings in the running Lisp. This includes names of symbols, packages, docstrings, etc. The 10th and 90th percentiles of the distribution of the string lengths is 7 and 39. The MISS key set consists of random strings with length sampled uniformly from the [4, 44] interval.

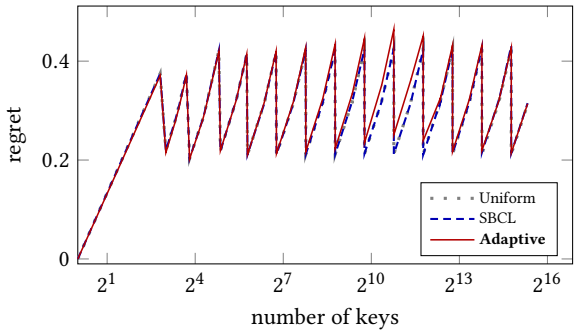

Figure 1: Regret (Definition 5) with string keys. Adaptive does not gain or significantly compromise on regret. Points where the truncation limit changes vary between runs.

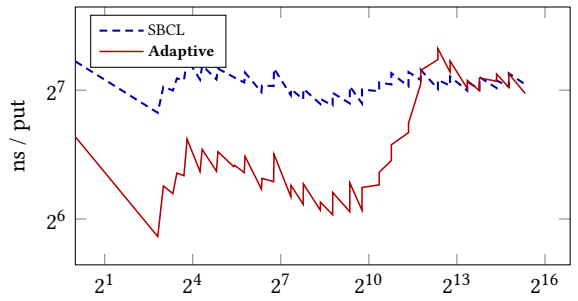

Figure 2: PUT timings in nanoseconds with string keys. Note the log scales. The plot shows the *average* time for inserting a new key when populating an empty hash table with a given number of keys.

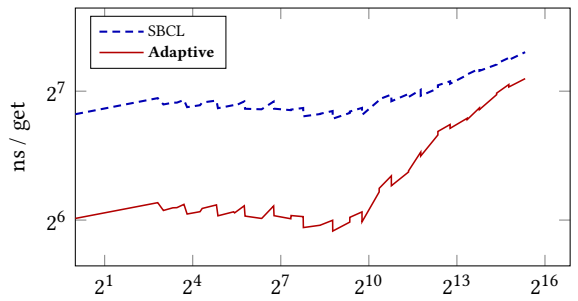

Figure 3: GET timings with string keys.

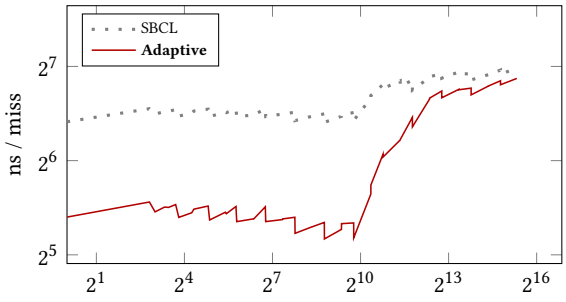

Figure 10: MISS timings with string keys

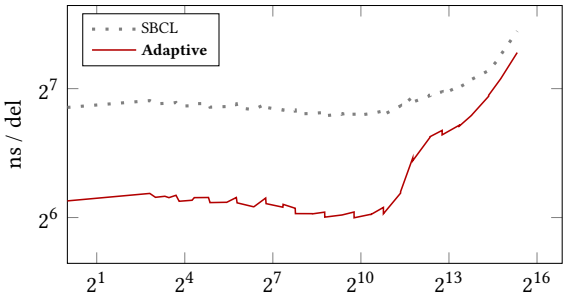

Figure 11: DEL timings with string keys

I RESULTS FOR `FIXNUM :PROG 1`

Keys form an arithmetic progression with difference 1 starting from a large random offset and are used in that order for PUT. We also generate a set of keys not in the hash table for MISS, by using another, suitable offset (so that the two sets are disjunct). For GET, MISS, and DEL, keys are presented in random order.

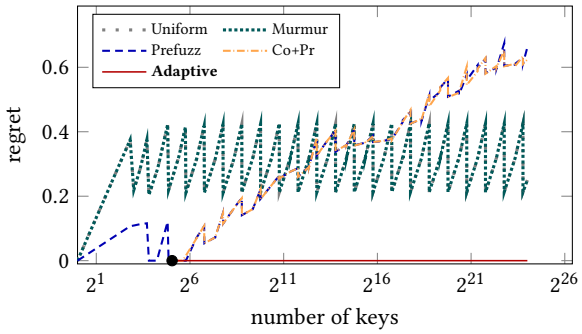

Figure 4: Regret with `FIXNUM :PROG 1`. Murmur closely tracks Uniform. Prefuzz is aggressively optimized for small sizes. Adaptive (Algorithm 4) is a perfect hash here. Both Co+Pr (Constant followed by Prefuzz) and Adaptive use the Constant hash until the fixed switch point at 32 keys (black dot).

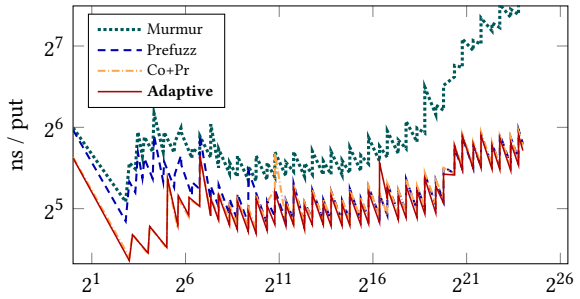

Figure 5: PUT timings with `FIXNUM :PROG 1`. Prefuzz outperforms Murmur even at large sizes despite higher regret because it's friendlier to the cache (its collisions are between subsequent elements of the progression), and its combination with Constant is even faster. Thus, despite being a perfect hash, Adaptive can improve on them only marginally.

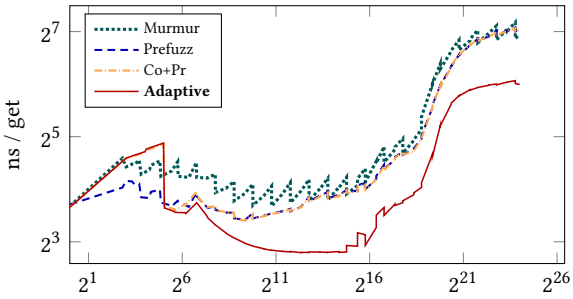

Figure 6: GET timings with `FIXNUM :PROG 1`. Keys are queried in random order so regret matters more here than with PUT, but the cache-friendliness of Prefuzz still keeps it ahead of Murmur. As expected, Adaptive can finally benefit from its zero regret after the Constant hash phase.

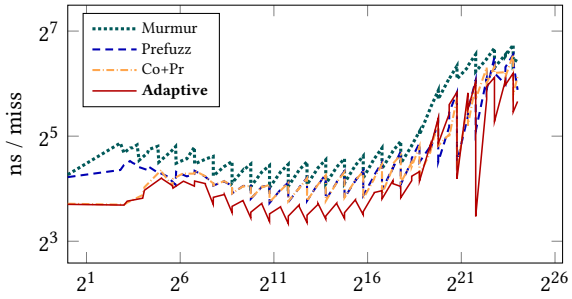

Figure 12: MISS timings with `FIXNUM :PROG 1`

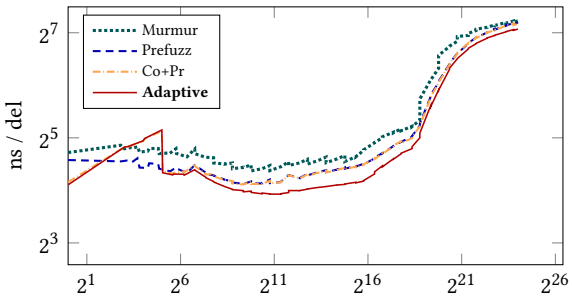

Figure 13: DEL timings with `FIXNUM :PROG 1`

J RESULTS FOR FLOAT :PROG 1

Keys are generated as in Appendix I, but the `fixnum` values are converted to `single-float`.

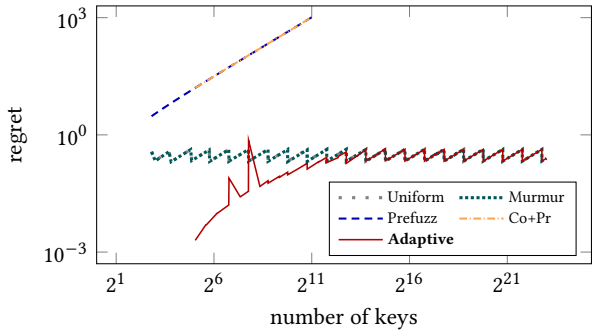

Figure 7: Regret with FLOAT :PROG 1. To be able to plot the catastrophic failure of Prefuzz (and of Co+Pr, consequently), we use log scale for regret on this graph. Single floats are especially problematic for Prefuzz because they can have many constant low bits. Adaptive detects these constant low bits and does better than Uniform until variation in the floating point exponents makes its original estimate of the number of constant low bits invalid, and the resulting gradual increase in collisions makes it switch to Prefuzz at rehash time. This is a spectacularly bad idea in this scenario, and the high number of collisions causes an immediate switch to Murmur. The switch times vary by hash table because the key sets are generated starting from random offsets.

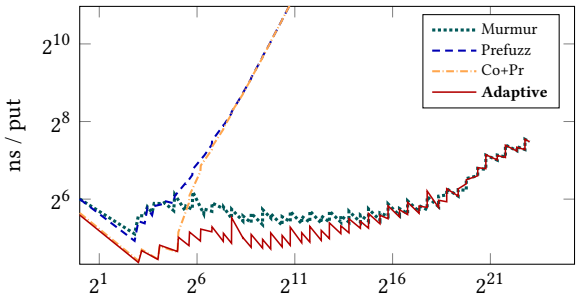

Figure 14: PUT timings with FLOAT :PROG 1

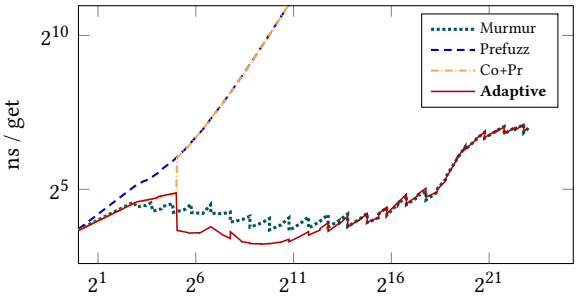

Figure 15: GET timings with FLOAT :PROG 1

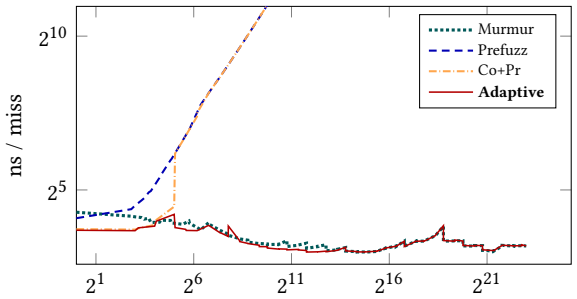

Figure 16: MISS timings with FLOAT :PROG 1

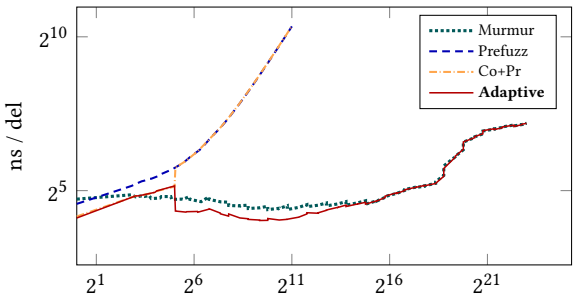

Figure 17: DEL timings with FLOAT :PROG 1

K RESULTS FOR `FIXNUM :PROG 12`

Same as `FIXNUM :PROG 1`, but with a difference of 12.

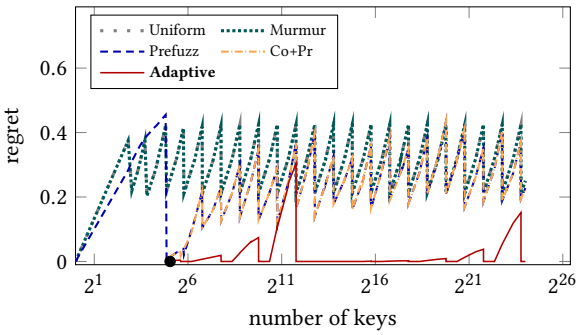

Figure 8: Regret with `FIXNUM :PROG 12`. Murmur closely tracks Uniform, but Prefuzz is better across almost the whole range. Arithmetic (Section 4.4) would be a perfect hash here, but Adaptive, which uses Pointer-Shift (Section 4.6), is not quite perfect due to the interference of its $k \gg$ PB term.

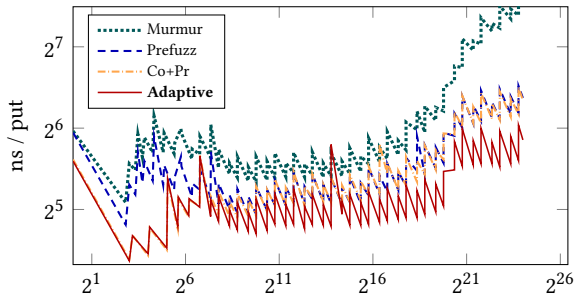

Figure 18: PUT timings with `FIXNUM :PROG 12`. Prefuzz outperforms Murmur due to its speed, lower regret and cache-friendliness. Adaptive is able to benefit from its advantage in regret only at larger sizes.

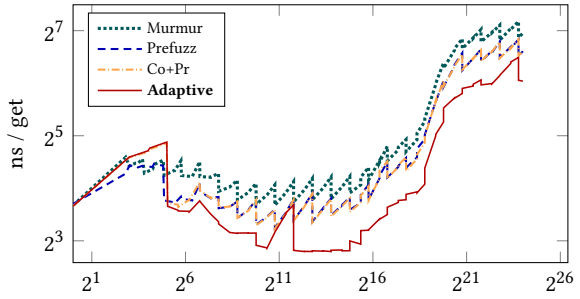

Figure 19: GET timings with `FIXNUM :PROG 12`. Compared to PUT, differences in regret translate more clearly to lookup performance because keys are queried in random order.

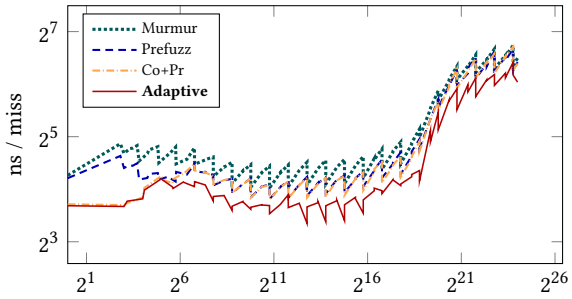

Figure 20: MISS timings with `FIXNUM :PROG 12`

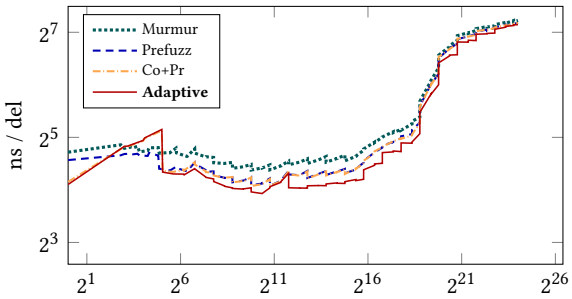

Figure 21: DEL timings with `FIXNUM :PROG 12`

L RESULTS FOR `FIXNUM :RND 6`

Similar to `:PROG 6`, but the difference between successive keys is sampled uniformly from the $[0, 5]$ interval.

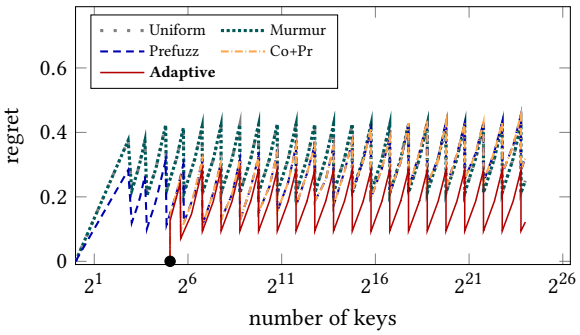

Figure 22: Regret with `FIXNUM :RND 6`. Prefuzz does better than Murmur initially but gradually loses its edge. Adaptive keeps its edge.

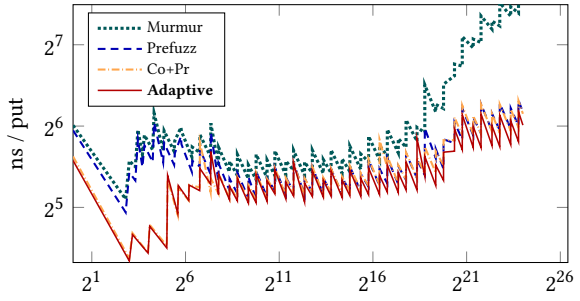

Figure 23: PUT timings with `FIXNUM :RND 6`. Once again, the advantage of Prefuzz over Murmur grows with size because it is friendlier to the cache. Adaptive manages to translate some of its lead in regret into outright speed.

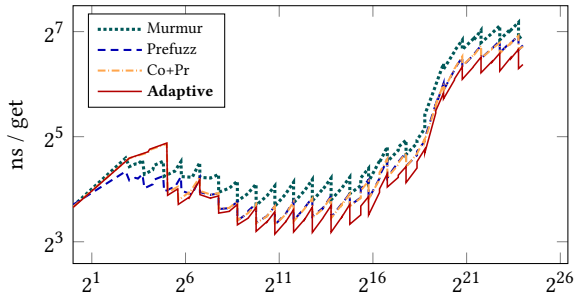

Figure 24: GET timings with `FIXNUM :RND 6`. Prefuzz is considerably ahead of Murmur at all sizes. Adaptive is better than Prefuzz at larger sizes.

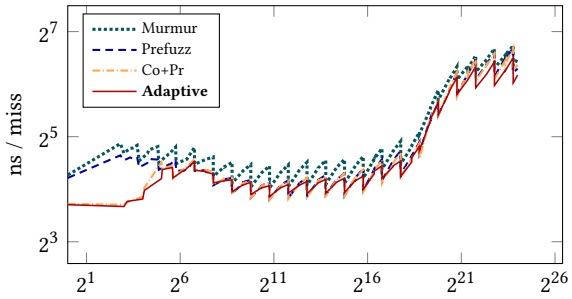

Figure 25: MISS timings with `FIXNUM :RND 6`

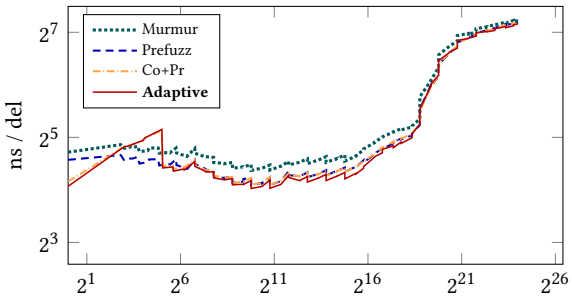

Figure 26: DEL timings with `FIXNUM :RND 6`

M RESULTS FOR CONS :RND 6

Like the FIXNUM :RND 6, but keys are cons objects with a random number of conses allocated between them. There is no explicit random offset is here; we rely on the addresses assigned by the memory allocator. To prevent the garbage collector from compacting memory regions holding these objects in memory, the skipped over conses are kept alive.

In all experiments, there is some unexplained weirdness at large sizes, which affects all hashes except Murmur. Even the regret of Adaptive is affected: it improves to an unexpected level but does so very erratically.

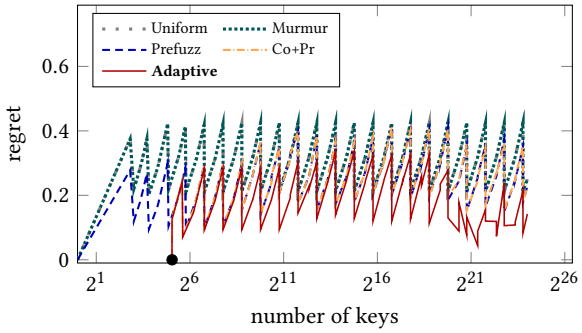

Figure 27: Regret with CONS :RND 6. Murmur closely tracks Uniform. Prefuzz works well at small sizes. Adaptive is a bit better still.

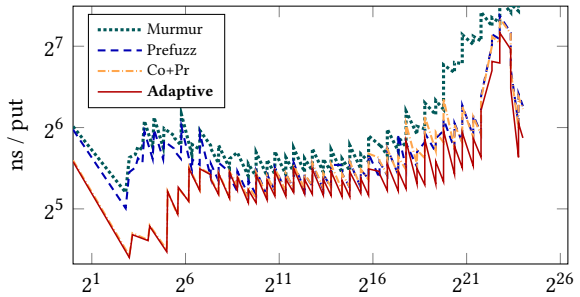

Figure 28: PUT timings with CONS :RND 6. Prefuzz outperforms Murmur even at large sizes despite its higher regret because its collisions are between subsequent elements of the progression, which is friendly to the cache.

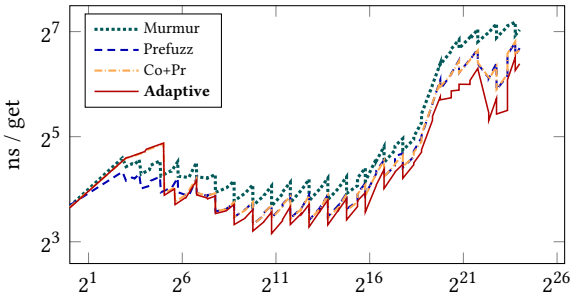

Figure 29: GET timings with CONS :RND 6

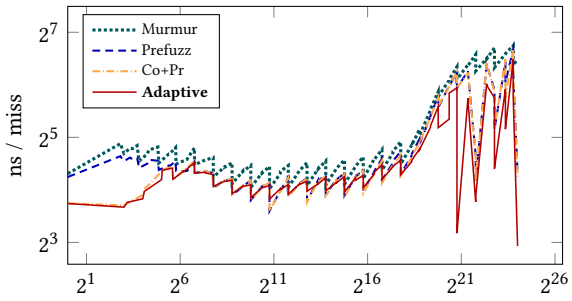

Figure 30: MISS timings with CONS :RND 6

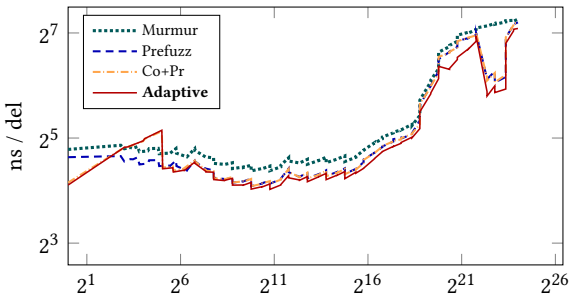

Figure 31: DEL timings with CONS :RND 6

N RESULTS FOR SYMBOL :EXISTING

We list all symbols in the running Lisp system and take a random subset of the desired size. These symbols happen to be packed tightly in not too many pages, and their addresses, if sorted, would approximately follow an arithmetic progression, which is a great fit for Pointer-Shift (Section 4.6, Algorithm 4). But the keys are randomly selected and presented in random order even for PUT, so this effect can only be seen at close to maximal sizes.

The MISS key set is generated with `(gensym)`.

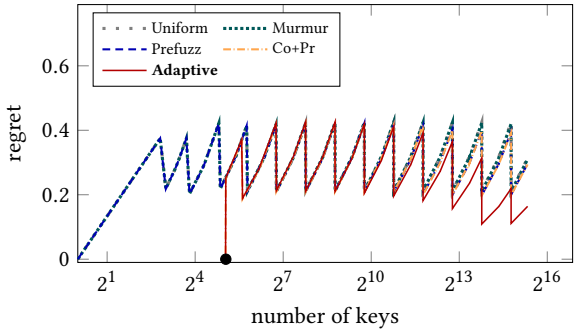

Figure 32: Regret with SYMBOL :EXISTING. All hashes closely track Uniform. Adaptive, which leaves memory addresses most intact, takes advantage of the nature of the data at close to maximal sizes.

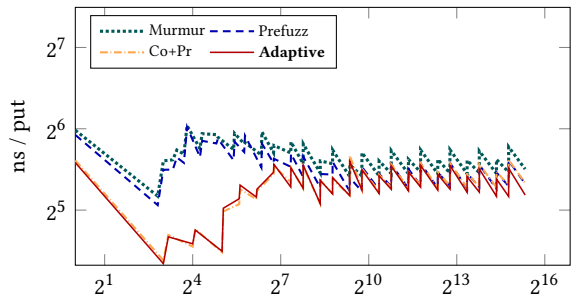

Figure 33: PUT timings with SYMBOL :EXISTING. Prefuzz outperforms Murmur because it is faster to compute. The advantage of Co+Pr and Adaptive over Prefuzz comes almost exclusively from the initial Constant hash phase, with Adaptive enjoying a small edge due to its better regret at the very end.

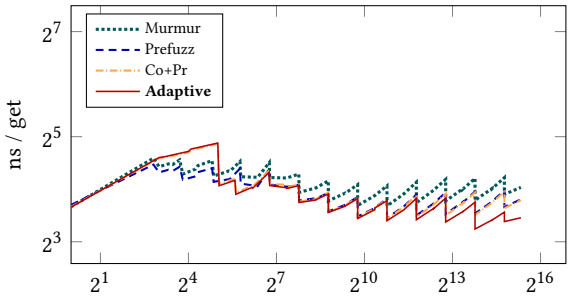

Figure 34: GET timings with SYMBOL :EXISTING.

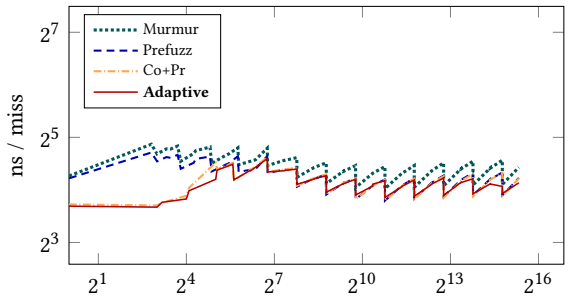

Figure 35: MISS timings with SYMBOL :EXISTING

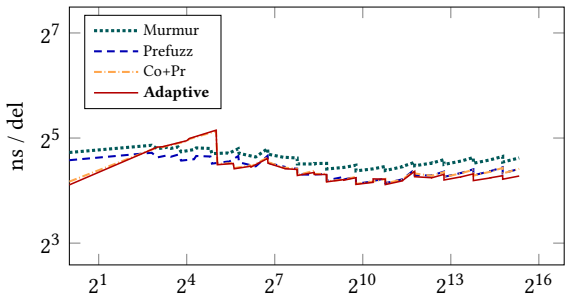

Figure 36: DEL timings with SYMBOL :EXISTING

Tuesday, 7 May 2024

Period Information Extraction: A DSL Solution to a Domain Problem

Arthur Evensen
RavenPack
Marbella, Spain (remote)
arthur@evensen.space

ABSTRACT

PEREL, a lisp-like proprietary domain-specific language (DSL), was developed for the period detection system at RavenPack to enable non-developer team members with domain expertise to directly manage the period detection rules. This solution was developed in response to previously encountered workflow problems, namely slow feedback cycles, developer bottlenecks, and the split between domain experts and implementors. This paper provides a high-level overview of the period detection system, details the pain points that led to PEREL, and outlines the development cycle of the project. Subsequent sections then delve into the DSL itself, first as a language, then exploring key implementation details, before turning to the supportive tooling. Finally, we present more examples.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**.

KEYWORDS

Common Lisp, DSL, period detection, workflow optimization

ACM Reference Format:
Arthur Evensen. 2024. Period Information Extraction: A DSL Solution to a Domain Problem. In *Proceedings of the 17th European Lisp Symposium (ELS'24)*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.5281/zenodo.11008823

1 CONTEXT

RavenPack converts unstructured text to structured data, primarily oriented towards the financial industry. We have a period detection system as part of our document analysis, which identifies the start and end times of events. Each *period* has three parts: a *type*, a *pattern*, and an *extractor*.

For example, "the nineteenth week of 2024" could be interpreted as a period spanning [2024-05-06, 2024-05-13) and categorized as a `$period-week` type. The types can be referenced in the patterns and form part of the means of abstraction. The pattern is what to match to count as a detection, e.g. `$numeric-ordinal %literal-week of $period-absolute-year` – in this pattern, `%literal-week` is the result of pattern-matching in another part of the system, while `$period-absolute-year` is another period detection as discussed in Section 4.2.3. The extractor is code to execute to compute the *period spec* for the period detection, pulling data from the text match and calculating the start and end specifications. In other words, the extractor defines the semantics of the particular period detection. For this example:

Listing 1: Example period extractor

```
(let ((n (extract $numeric-ordinal)))
  (add
    (extract $period-absolute-year)
    start [= [week n]]
    end [+ [weeks 1]]))
```

In Listing 1, `n` gets bound to the number extracted from the `$numeric-ordinal`. Afterwards, it extracts the period spec from the `$period-absolute-year`'s extractor and (non-destructively) modifies it by `add`ing (appending) `[= [week n]]` to the end of its `start`, and `[+ [weeks 1]]` to its `end`. Finally, the resultant period spec struct is returned, and might in turn be used by other periods/extractors or evaluated to produce a time interval.

2 PROBLEM

Originally, non-developer team members (*editors*) noticed missing/poor detections, made the decisions for the desired behaviour, and submitted requests for developers to implement. This process worked, as the corpus of 1.2K periods attests. However, it made for a slow feedback cycle where developer availability bottlenecked improvements, and split the reasoning based on the text examples – the domain reasoning – from the implementors of the periods. Most developers spend only a small fraction of time looking at the raw documents, compared with what the editors do, and might not be aware of usage/description peculiarities for the text of the periods.

Inspired by the then-recent success of shifting another (simpler) system area into direct editor control, the editor manager asked whether something similar would be possible for the period detections. Two initial dead-end solutions were considered: the extractors were too varied/numerous for a simple choice selection system to pair patterns and extractors, and the extractors were too arbitrary for a slot-in system to leave them mostly pre-written. Reflecting on the arbitrariness of the extractors and their individual limited size and simplicity, we realized we were looking for a way to write arbitrary rules for a particular domain – which is what a programming language *is* (teleologically). Hence, a solution would be isomorphic to a DSL. After taking rough stock of what capabilities a DSL would need, we said we could make one with supportive tooling to hand over the periods to editorial control and that it should address the identified pain points.

ELS'24, May 6–7 2024, Vienna, Austria

ACM ISBN 978-2-9557474-8-3
https://doi.org/10.5281/zenodo.11008823

3 DEVELOPMENT CYCLE

3.1 Timeline

The project started with an in-depth analysis of existing extractors and concurrent drafting of a DSL spec in spring 2022 by two developers. During the analysis, it was found that some aspects of existing extractors were unnecessarily low-level: for example, `extract` (as shown in Listing 1) in PEREL replaces three different prior approaches to accessing underlying data. Also, various common list manipulations (on the `start` and `end` of period specs) were generalized into means of combination. After the initial pass to look for similarities in extractors, a second pass to look for differences was done, focused on edge cases like extractors of unusually big size. This latter pass resulted in an additional mean of combination and a couple utilities, but the inspected edge cases were primarily found to be of dubious quality, reinforcing the decision to make the DSL as simple as possible.

Once drafted, the DSL spec was handed over to the test user. Despite the lack of any tooling, the test user went from initial confusion to productive in a month's time with about a week of spread effort. Observing that – as a medium – a programming language is in part a user interface[3], and borrowing from Design Thinking principles regarding lo-fi prototyping[6], the initial editor-written extractors were made through the lo-fi approach of pretence. That is, by simulating hypothetical UI/compiler responses in chat. Unfortunately, the lo-fi stage led to less constructive critique than anticipated: The goal had been to rapidly iterate on the spec during the lo-fi stage with minimal cost sunk into any particular version of it, but it was only later with more experience and growing confidence in the DSL programming that the test user started contributing wishful thinking and suggestions.

The test user succeeding in writing extractors without any tooling and minimal experience validated the overall feasibility, though, as supportive tooling should both ease on-boarding and extractor writing. Hence, the DSL was implemented in summer 2022: Given the minimal transpilation implementation as discussed in Section 5, the initial implementing presented no major problems. The main UI saw development next – with iterative development, so the test user could use it in rudimentary state from early on – followed by DB work, sync processes, QA processes, etc. The editor's experiences writing extractors occasionally led to updates to the spec/implementation, and to uncovering bugs/errata. In general, the project has taken about two calendar years, with approximately one year of development effort, occasionally interrupted by other priorities for months at a time. The more interesting part of the UI is a CodeMirror[4] editor, which we return to in Section 6.

3.2 LLM Considerations

Given the rise in popularity and capability of LLMs last year, we detoured from other period detection and PEREL work for a week in summer 2023 to investigate possible inclusions of LLMs into detections or workflows. Initial ideas included whether it might be possible to use an LLM to detect periods and the involved time intervals, hint at period detections, or write the extractors given a DSL specification.

Hence, the better part of a week was spent doing qualitative experiments by trying out prompts to see how reliably/well ChatGPT[1] might detect periods. These experiments were done using an informal/unmeasured mixture of the gpt-3.5-turbo and gpt-4 models, and consisted of trying various prompts while providing ChatGPT with sample paragraphs containing period-related text. The prompts tasked the model with extracting all period-related data and providing a JSON response.

While both models would technically accomplish the given task, none of the prompts attempted successfully made it enumerate all period information in the sample texts. gpt-3.5-turbo often returned results that were invalid JSON, e.g. by inserting ellipses. gpt-4 gave properly formatted results more of the time, though the lack of full enumeration was present for it too. Overall, the models missed period detections, reported false positives (e.g. identifying "ca.16%" as a period), and sometimes returned "reverse periods" with start timestamps occurring after their end timestamps.

However, the models did detect some periods missed by our existing set. Having made that observation, but attempted enough samples where the LLM models performed worse than the existing systems, we decided that the most salient use while retaining control over period detection might be to add a future report using an LLM to trawl documents for undetected periods. Editors could then consider the various findings of the LLM and decide whether, and how, to implement the suggested detections. Such a report remains as future work, though.

4 DSL: PEREL

4.1 Overview

The DSL has been named "PEREL" (PERiod Extraction Language). PEREL has been designed to be as minimal and simple as possible while still covering (as discussed above in Section 3) *almost* all functionality necessary to write the set of "legacy" (non-PEREL) extractors. Given that the target users don't have prior programming expertise and hence preconceived notions of syntax, and since staying with s-exps meant easy use of the Lisp reader as a parser, we decided to embrace a minimal, lisp-like design.

Here is an overview of changes compared to Common Lisp:

(1) **Locked-down permissions**: Users can't add new operators. Users can't access arbitrary Lisp functionality. Every input atom is validated for 'legality' prior to testing the code (we return to this in Section 5). Users are restrained to PEREL. This means a limited vocabulary of about 40 operators, a dozen of which are simple mathematical ones that everyone is presumably familiar with from school.

(2) **Magic symbols rather than keywords**: Since all extractors are read into the perel package and the DSL has a limited vocabulary, it's possible to determine symbol semantics by context. This allowed the removal of keywords, so users can simply treat everything as 'magic text' that carries out the specified rules: One less concept to wrestle with (and since explaining keywords would mean explaining symbols, two concepts less). Hence, PEREL accepts `(spec start [= [day 3]])` rather than `(spec :start [:= [:day 3]])`.

(3) **Square bracket list construction**: Given the above observation that keyword use of a symbol was determinable by context, it was also determinable whether an element of a list would require evaluation or not. Hence, backquote

notation became unnecessary as well, making for the introduction of square bracket list construction syntax: `[1 2 3]` ⇒ `(perel-list 1 2 3)` ⇒ `(list 1 2 3)`. (`perel-list` is an intermediary for validation & debugging during transpilation.)

(4) **Assorted minor changes**: Some symbols have been "renamed", e.g. PEREL `let` ⇒ Lisp `let*`, and similarly for `true` ⇒ `t`, `false` ⇒ `nil`. `error` in PEREL become calls to Lisp `error` with a `perel-error` error type. These simplifications aim to minimize necessary context to avoid questions like "Why is it called `t` and `nil`?"

In short, two categories of changes: 1) Restricting the language to deal with – and to only deal with – the domain in question and hence to work to add/modify extractors, 2) syntactical simplifications to prune away concepts and historic baggage irrelevant to the task at hand, to minimize the necessary cognitive load/learning. It's the restriction part that changes the overall semantics. Besides the above, legal extractors *must* return a period spec – assuming it returns rather than (intentionally) errors out.

4.2 Analysis

We present an analysis of PEREL per the framework introduced in chapter one of *Structure and Interpretation of Computer Programs*[2].

4.2.1 What are the primitives? The overall system which PEREL interfaces with presents a set of "primitive attributes" (from PEREL's point-of-view) which deal with numbers and month names and so on, which are what the period patterns match against. In addition, PEREL has numbers, words, strings, lists, and a set of built-in operations (e.g. `let`, `add`, `extract`). `spec` serves as the foundational operation, by construing period specs directly.

4.2.2 What are the means of combination? The (sub)set of built-in operations that construe specs from specs and/or parts of specs: These allow simple modifications of existing specs, enabling users to write extractors without having to rebuild `start` and `end` definitions from the ground up. `add`, as shown in Listing 1, is one of these operations.

4.2.3 What are the means of abstraction? Each period is assigned a period type, making it available for use in other patterns/extractors as a black-box abstraction. In Listing 1, we saw an extractor that abstracted across all `$period-absolute-year` ones (that happen to match in the given pattern). So, the categorization into period types (together with `extract`) permits the use of these higher-level building blocks.

4.3 In Summary

The constraint of dealing only with the period domain as encapsulated by PEREL, and the reliance on other parts of the overall system in preparing primitive data, etc., means that the mental mode of dealing with PEREL is rather different from that of dealing with Lisp. The constrained capacities force engagement with the domain representation on its own terms. The test user has on multiple occasions emphasized that the difficulty (once he got his head around PEREL and the mechanical execution of code) usually doesn't lie in figuring out what code to write, but in finding good candidate rules for dealing with the period text/pattern to start with.

5 IMPLEMENTATION

5.1 Overview

The periods live in a DB: Because of the legacy 1.2K periods, the period definitions table has separate `perel_code` and `lisp_code` columns. A definition will either have `perel_code` or `lisp_code` present. This means that when loading periods, we wrap the extractors up in lambdas as per Listing 2:

Listing 2: Lambda wrapping of extractor code

```
(defun generate-$period-extractor (perel-code lisp-code)
  (let ((code (if perel-code
                  (transpile-perel-code perel-code)
                  (with-standard-io-syntax
                    (let (...)
                      (read-from-string lisp-code))))))
    (compile nil `(lambda (...)
                    (let (...)
                      ,code)))))
```

Where this is the main entry into the transpiler:

Listing 3: Entry into the transpiler

```
(defun transpile-perel-code (perel-code)
  (let* ((perel-form (perel-code-string->perel-code-sexp
                      perel-code))
         (_ (validate-legal-symbols perel-form))
         (lisp-form (perel-form->lisp-form perel-form)))
    (apply #'validate-form-types lisp-form)
    lisp-form))
```

As per Listing 3, the overall transpilation strategy consists of reading, validation pass one, transpiling, validation pass two, and returning. Validation works by throwing an error if there is any.

5.2 Reading

The parsing of the PEREL code string into an s-exp is done with a customized readtable:

(1) Nil out various default macro characters that are of no concern in PEREL: `(#\# #\: #\| #\' #\` #\, #\.)`. This protects against malicious input – and, more importantly, typos.
(2) Square brackets read a delimited list and cons `'perel-list` onto its head.

Beyond that, since a PEREL extractor is a single form, the reader step throws an error if it returns before having read the entire input stream, i.e. if the input consists of multiple top-level forms.

5.3 Validation Pass One

The first validation pass goes through all atoms in the extractor and checks that they're 'legal'. An atom is legal if it is a number, string, or an expected symbol. The various `def-perel-fun` forms (an example is below) register related symbols, while the majority of the other legal symbols come from the underlying data for the pattern-matching part. Further, there is some special handling of `let` forms to track user-introduced variables to treat those as legal too.

5.4 Validation Pass Two

The later pass is more involved, and performs rudimentary static type checking. All PEREL operators have predefined type information, as seen below in Listing 4:

Listing 4: Sample definitions with type information

```
(defparameter *fallthrough-operations*
  '(((<    boolean)  . (&rest (numbers number)))
    ...))

(def-perel-fun (add spec)
    ((spec spec)
     &key
     ((start  nil start-supplied-p) op-specs)
     ((end    nil end-supplied-p) op-specs)
     ((anchor nil anchor-supplied-p) op-specs)
     (props spec-props))
  ...)
```

Syntactically, the function names and each parameter get wrapped up in an extra list whose second element is the type specification. In the case of multiple possible types, the type specification looks like e.g. `(U null number)` (with the `U` standing for union).

Anyhow, the validation descends each s-exp and checks that each operator receives permissible types. If not, it signals an error that is hopefully friendlier to the end-user than the runtime errors that might have arisen otherwise.

5.5 The Transpilation Itself

The `def-perel-fun` form macroexpansion generates transformer methods for each operation, which are used in the transpilation step. The transformer methods check the argument forms (arity and keywords), transform the argument forms, and return a normal Lisp s-exp equivalent of the original PEREL s-exp (which e.g. takes care of turning magic symbols into keywords). Suffice it to say the implementation involves some fun with backquotes.

There is a handcoded method for `let` forms due to the syntax involved. The more interesting handcoded method is the one for `perel-list` forms, which checks the list for the presence of markers of the different semantic lists we deal with in PEREL. This is necessary to correctly transform them since not all the lists – unfortunately, given prior constraints – are plists, but also let us add certain user conveniences as syntactic sugar: for example `[- [days 3]]` gets transformed into `(list :+ (list :days (- 3)))`.

6 TOOLING

6.1 Overview

As part of the project, periods have received supportive tooling – indeed, the tooling took longer than the DSL design and implementation itself. Numerous parts of the tooling setup would have been necessary to support any solution to hand over control to the editors: moving periods into a DB, displays, UIs, reports, sync mechanisms across pipeline stages… To integrate with existing systems used by editors, the UIs for dealing with period detection all run in the web browser.

6.2 Migration into a DB

To make it possible for editors to manipulate periods directly, they had to be stored in a DB. This necessitated the migration of the pre-existing periods from the codebase into the DB, too. This was straightforwards once a schema design was ready, as the periods were defined in macro forms: writing a new macroexpansion for it (for use as a script) and replacing the old one migrated them into the DB by recompiling the relevant source.

However, the old macro forms had comments in them detailing which document and sample text each period had been created to target. This is useful context to have when considering their quality and investigating potential issues, albeit not all the comments had the same format or all of the relevant information. Some periods also lacked comments entirely.

To preserve as much of this context as possible and establish a canonical context for editor use, we wrote yet another macroexpansion for the period definition forms and coupled its use with a modified readtable which had its `#\;` reader macro replaced by a function which read the rest of the line as a string and wrapped it up into a marked list. The macroexpansion function processed these with an iteratively developed set of regexes to extract as much context as possible. Checking these contextual documents for the relevant period detection, along with searching through a general batch of documents to look for new contexts to treat as canonical in case of a miss (or lack of a context), allowed the preservation of the majority of the contexts and contributed to a useful setup overall.

6.3 CodeMirror Editor

As mentioned earlier, the input for the extractors is a CodeMirror editor. CodeMirror is a JavaScript library to create code editors for the web and supports many common editor features. Language packages for CodeMirror can be written by creating a Lezer[5] grammar, where Lezer is a parser generator made for use with CodeMirror. CodeMirror was chosen based on word-of-mouth recommendations by another Lisp programmer, combined with good documentation and how making an editor using it could be done in a predominantly declarative manner. We only had two occasions where we broke the offered declarative paradigm when writing the editor and the PEREL Lezer package:

(1) It required some imperative code to make the code editor aggressively autoindent all lines on all input.
(2) After initial struggles trying to differentiate the tokenization of numbers and symbols using a pure Lezer grammar, the tokenization of those two node types was split off into `ExternalTokenizers` – a feature Lezer offers up precisely in case a grammar description proves awkward.

The extractor editor, shown above in Figure 1, features autocompletion (with descriptions of the operators as reminders to the users), autoformatting, syntax highlighting (though the figure doesn't show it due to the lack of strings, numbers or comments), bracket closing, and bracket highlighting (for well-formed input due to the grammar). The autoformatting was the main goal, to improve the detection of 'the same' extractor across history rather than hide sameness behind minor cosmetic code-as-string changes (since a period is a type, pattern and code tuple and hence each

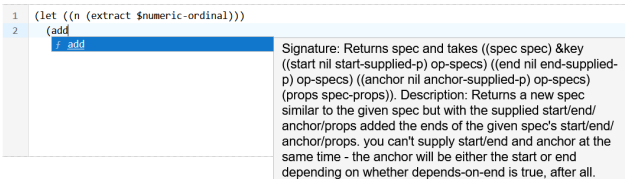

Figure 1: The extractor editor, with visible autocompletion

such combination gets assigned a unique id). The autoformatting also ensures improved readability for the extractors: Despite seeing some well-formatted examples early on and receiving hints as to how to format for better readability, the test user tended to produce indentationally flat extractors prior to the introduction of the CodeMirror editor. The other features than the autoformatting are directly for user convenience.

6.4 Validation and Quality Assurance

Period additions used to be validated in an ad-hoc manner by developers during implementation and implicitly through presence/absence as part of automation-assisted validation work by the editorial team focused on overall event detections. In addition, editors would check up on the behaviour of the periods they requested. To mimic the previous ad-hoc checks by developers, the UI for writing new periods forces them to be locally tested against the text sample they're created against, so editors not only *can*, but *have to*, verify that they are locally well-behaved. In fact, the automated checks per period addition are generally broader than the ad-hoc predecessors. However, the handover of periods to editor control, combined with the observation that the creation of new periods is not purely additive (as outlined in Section 4.2.3), meant that the previous validation approach was deemed no longer sufficient.

The early plan to improve the validation for period detections was to create analogous automation to the validations done for overall event detections. However, this was eventually scrapped for essentially duplicating existing processes and workflows. Instead, it was decided to integrate into the existing process by broadening the work done by the existing validation automation, as this would lead to better visibility and readily slot into existing editor workflows. As a consequence, overall validation for period detections is no longer primarily local, but systematically evaluates the broad impact of changes to the set of periods.

7 EXAMPLES

Having given a high-level overview of PEREL, we turn now to multiple examples and their discussion to give a more concrete view. To format the listings in this section, each contains first a pattern as a list, followed by the extractor code. As the type assigned to each period is not pertinent to the extractor discussion, we skip the type discussion. To clarify, the work on PEREL has also not involved any direct changes to the pattern-matching system.

Listing 5: An extractor from primitives

```
(past %several %literal-weeks)
(spec
  start [- [weeks 3] depend-on-ends true]
  end [this week])
```

Listing 5 shows an extractor construed from primitives, as discussed in Section 4.2.1. Periods constructed in this way form the base layer that other, more high-level, periods can be built atop. The particular extractor uses the publication timestamp of the document it matches in to find the start of the current week, and treats that as the end-point. For the start, it moves back three weeks from the end-point. The editor in question decided that three weeks serves as a practical heuristic for text like "the past several weeks".

Listing 6: The simplest possible extractors

```
(%all %literal-day $period-relative-week)
(extract $period-relative-week)
```

Listing 6 shows the simplest possible extractors, where the longer pattern provides no further time information. These cases allow "passing up" the time information from the shorter pattern represented by the `$period-relative-week` used in the above, to the longer pattern that wraps it. This allows better pattern-matching in other parts of the system that make use of the periods. The particular extractor uses `extract` to return the period spec from the underlying `$period-relative-week` – in other words, these cases rely purely on the means of abstraction.

Listing 7: Simple combination of extractors

```
($period-absolute-weekday $period-day-part
 at $period-absolute-hour)
(combine
 (extract $period-absolute-weekday)
 (extract $period-absolute-hour))
```

Listing 7 shows an extractor made using the `combine` operator, which is one of the means of combination (and, implementation-wise, wraps a pre-existing function to present a simplified interface to the user). `combine` works by, in order, appending the starts of the period specs into a single start, and similarly for the ends. It's a very convenient way to write extractors where each underlying period has distinct size granularity. However, it can lead to undesired behaviour when used too broadly, e.g. in case of overlap in the time units, or if underlying periods have extractors that rely on the rounding behaviour involved in conversion to a time interval rather than having both an explicit start and end. The test user initially made broad use of `combine`, before experience with such cases resulted in treating it as a less immediate option: more precise manipulations are possible using the other means of combination. The test user also observed that an undue amount of legacy extractors using `combine` were assigned to the ambiguous `$period-combinations` type, making the latter difficult to use as building blocks for further periods.

Listing 8: An extractor using multiple means of combination

```
(%early in %the morning of $period-absolute-day)
(overwrite
 (add (extract $period-absolute-day)
      start [= [hour 5]])
 end [+ [hours 3]])
```

Listing 8 serves as an example of a more complex extractor. It takes the period spec from the underlying `$period-absolute-day`, adds `[= [hour 5]]` to the end of its start, hence ensuring a starting time of 5 am, and then it uses `overwrite` to take the period spec returned by the `add` and overwrites its end with the new specification of `[+ [hours 3]]`. The overall result is to constrain the extracted

time interval to [5 am, 8 am) on the given `$period-absolute-day` – again, the particular interpretation of "early in the morning" is a heuristic introduced by the editor.

Listing 9: An extractor with a conditional

```
(%the remainder of $period-digits-year)
(let ((y (extract $period-digits-year)))
  (if (later-than-p year y)
      (error "Unable to compute a precise period")
      (spec
       start [this day]
       end [= [year y month 12 day 31] inclusive true])))
```

As a final example, Listing 9 shows an extractor using a conditional to check whether the document was published later than the mentioned year. If so, due to the lack of information necessary to do a sensible extraction, it throws an error. `later-than-p` allows similar checks for a number of different time units, e.g. month, possibly multiple at a time. The availability of throwing an error means that contextually ambiguous detections, like the example, can be handled meaningfully.

As can be observed, PEREL programming is "programming in the small", matching the initial observations of the limited size of existing extractors as mentioned in Section 2. Effort has been put into creating a stratified design so that the extractors can encapsulate the intended period rules directly, without having to resort to manipulations of lower-level details. Hence, while the `start` and `end` of a period spec are implemented as lists representing operations to calculate a timestamp, and there are accessors that return these lists (e.g. `extract-end`), PEREL intentionally does not provide a mechanism for low-level manipulations of lists. For example, it's not possible to delete the last operation in the `start` of a spec, since that would require breaking the black-box abstraction approach towards making use of existing periods. In short, as the examples highlight, PEREL's semantic level is that of period specs.

8 REFLECTION

We have detailed the conception of the project from initial pain points through full-fledged system, where the core realization that led to settling on a DSL as a solution was that given a domain and the need to express arbitrary (combinations of) rules in that domain, any solution would be isomorphic to a DSL anyhow. PEREL is fairly minimalistic as a DSL, predominantly keeping simplified Lisp syntax and having a small transpilation step down to Common Lisp. Hence, it's a semantics-oriented DSL, with the language constraints enforcing a qualitative change in the approach to writing extractors: Unlike general purpose programming, where the focus is on the expression of arbitrary processes and operations, the focus in PEREL is on specifying the sensible time extraction for the given pattern. The tight integration with the existing system also frames the language, as discussed in Section 4.2.3.

While it was not an explicit design goal, the attempt to make PEREL as simple as possible has led to a declarative DSL. Given that *Concepts, Techniques, and Models of Computer Programming*[8] claims that the declarative programming paradigm is the simplest programming paradigm[7], this seems circumstantial evidence in favour of having succeeded at a simple design. The behind-the-scenes details of how the PEREL operators produce their outputs are hidden from the users (except in the case of stray errors). The PEREL spec does not discuss execution models, PEREL does not offer any iteration constructs, and PEREL does not have mutation.

The development of the CodeMirror editor took less time than originally anticipated, partly for representing a greenfield development task, and partly for the documentation quality and the largely declarative design of the CodeMirror library. In contrast, the DB setup (involving both schema creation, integration with existing sync processes, and ensuring coherent behaviour across multiple tables for the addition/modification/deletion of periods) and the UI development took considerably more tweaking than initially estimated. The problem surface area represented by the existing systems and workflows for the latter (as well as the more imperative approaches) presumably impacted this, as both the CodeMirror editor, DB setup, and UI development required on-the-fly learning by the developer.

As observed, the test user quickly became productive when given the DSL and eventually grew confident enough to provide constructive critique and feedback on the language and systems. During the development cycle, even with the system in a more rudimentary state for most of it, the test user created almost 400 periods – a sizeable chunk given the legacy corpus size of 1.2K. Indeed, all PEREL listings in this paper were written by the test user. It will be interesting to see how the system and its use develop now that it's launched (internally). The process of engaging deeply with the period detection system as necessitated by the work surrounding PEREL has also led to the identification of various possible overall improvements orthogonal to the direct addition of new periods.

REFERENCES

[1] 2023. *ChatGPT*. https://openai.com/chatgpt
[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. 1996. *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press.
[3] Matthew Butterick. 2016-2024. Beautiful Racket: Appendix - Why Lisp? Why Racket? https://beautifulracket.com/appendix/why-lop-why-racket.html. Accessed: 2024-02-25.
[4] Marijn Haverbeke and Contributors. 2023. *CodeMirror: Extensible Code Editor*. https://codemirror.net/
[5] Marijn Haverbeke and Contributors. 2023. *Lezer: The Lezer Parser System*. https://lezer.codemirror.net/
[6] Ralf Martin Steinert and Federico Lozano. 2015. TMM4220 - Innovation by Design Thinking. Department of Mechanical and Industrial Engineering, Norwegian University of Science and Technology. https://www.ntnu.edu/studies/courses/TMM4220
[7] Peter Van Roy. 2024. *The principal programming paradigms*. https://www.info.ucl.ac.be/~pvr/paradigmsDIAGRAMeng201.pdf
[8] Peter Van Roy and Seif Haridi. 2004. *Concepts, Techniques, and Models of Computer Programming* (1st ed.). MIT Press.

The Quickref Cohort

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

ABSTRACT

The internal architecture of Declt, our reference manual generator for Common Lisp libraries, is currently evolving towards a three-stage pipeline in which the information gathered for documentation purposes is first reified into a formalized set of object-oriented data structures. A side-effect of this evolution is the ability to dump that information for purposes other than documentation. We demonstrate this ability applied to the complete Quicklisp ecosystem. The resulting "cohort" includes more than half a million programmatic definitions, and can be used to gain insight into the morphology of Common Lisp software.

CCS CONCEPTS

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

KEYWORDS

Information Extraction, Software Analysis, Morphological Statistics

ACM Reference Format:
Didier Verna. 2024. The Quickref Cohort. In *Proceedings of the 17th European Lisp Symposium (ELS'24).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.10947962

1 INTRODUCTION

> *Cohort: a group of individuals having a statistical factor (such as age or class membership) in common in a demographic study.*
> *– The Meriam-Webster Dictionary[a], definition 2.b.*

[a]https://www.merriam-webster.com/dictionary/cohort

1.1 Context

Declt is a reference manual generator for Common Lisp libraries. The project started in 2010, leading to a first stable release in 2013 [2]. Four years later, the Quickref project was born [1, 4–6] (at the time, Declt was at version 2.3 [3]). Quickref runs Declt over the whole Quicklisp[1] repository and offers a website[2], currently aggregating more than two thousand reference manuals for Common Lisp libraries.

[1]https://www.quicklisp.org/
[2]https://quickref.common-lisp.net

ELS'24, May 6–7 2024, Vienna, Austria

ACM ISBN 978-2-9557474-8-3
https://doi.org/10.5281/zenodo.10947962

Declt runs by loading an ASDF system into memory and introspecting its contents. Because it is unrealistic to load the complete set of Quicklisp libraries into a single Lisp environment, Quicklisp runs Declt as a separate process for each library. The unfortunate consequence is that the information gathered by Declt is not directly available to the Quickref instance. Under those conditions, it remains easy to build a library index (by sorting the listing of the generated reference manuals directory), but it is for instance less straightforward to build an author index, as the author information, extracted from each ASDF system, needs to survive each and every Declt run.

Originally, Declt was designed to generate reference manuals in GNU Texinfo[3], an intermediate format suitable for software documentation, which can in turn be converted into a number of user-readable ones such as HTML, PDF, *etc.* Hence its name: Documentation Extractor from Common Lisp to Texinfo…

Over the years, there has been some pressure to extend Declt's rendering capabilities to other output formats (including HTML *without* the Texinfo intermediary). This led to an architecture overhaul, which is ongoing.

1.2 The Declt Pipeline

The goal is to implement Declt as a three-stage pipeline, as depicted in Figure 1. Declt's historical entry point, the `declt` function, triggers the whole pipeline, but for a more advanced usage, each stage of the pipeline is meant to be accessible separately and directly via its own entry point function.

(1) The first stage of the pipeline is called the *assessment* stage. At this stage, Declt loads the library and introspects the Lisp environment in order to extract the pertinent information. This information is stored in a so-called *report*.
(2) The second stage of the pipeline is called the *assembly* stage. At this stage, Declt organizes the information provided by a report in a specific way. The result is called a *script*. A script begins to look like a properly organized reference manual, but is still independent from the final output format.
(3) Finally, the third stage of the pipeline is called the *typesetting* stage. At this stage, Declt renders a script to a file by typesetting its contents in a specific documentation format.

In 2022, we released version 4.0b1 of Declt, marking the achievement of stage 1 of the pipeline [7]. Declt now provides a function called `assess`, which takes an ASDF system name as argument, loads the corresponding library, introspects it, and creates the report. The rest of the pipeline, which is not yet implemented, is wrapped in a temporary function called `declt-1`, going directly from a report to a Texinfo file.

[3]https://www.gnu.org/software/texinfo/

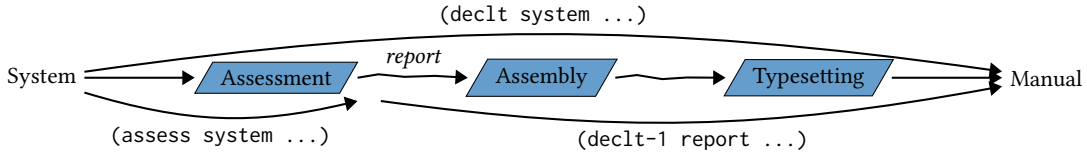

Figure 1: The Declt Pipeline

The first direct benefit of this evolution is the ability for Quickref to build an author index file in a much simpler and robust way. Instead of calling the global `declt` function, Quickref now triggers Declt in two steps. First, it calls the `assess` function to get a handle on the generated report, and then continues with `declt-1`. In the meantime however, the library's contact information is extracted from the report and dumped into a specific file. Once Quickref has finished processing the full set of Quicklisp libraries, it loads back all the contact information for all the libraries to create the index.

The funny thing is that once this was implemented, it quickly occurred to us that Declt reports, now in a stable format, could be fully dumped into files and used for all sorts of purposes other than documentation. In fact, it is relatively easy to "hijack" the Quickref infrastructure in order to dump Declt reports for the whole set of Quicklisp libraries, effectively creating a *cohort* of programmatic definitions.

In the following sections, we describe a preliminary cohort implementation, which currently contains more than half a million entities, and is already publicly accessible. Additionally, we show how such a cohort can be used to gain insight into the current shape of Lisp software.

2 DECLT REPORTS

A Declt report is a data structure containing general information about a library (authors, license, copyright, *etc.*), and a flat list of the discovered ASDF components and programmatic definitions (packages, variables, functions, classes, *etc.*).

2.1 Definitions

Definitions are themselves reified in an object-oriented fashion which is described in the Declt User Manual[4]. An excerpt of the definitions hierarchy is given in Figure 2.

For documentation purposes, the information provided by each kind of definition is as exhaustive as introspection permits. Most of them point back to the original Lisp object, can access the object's docstring if any, *etc.* On top of that, the assessment stage finalizes a library's definitions list by constructing an extensive set of cross-references (definitions pointing to definitions) that will eventually lead to internal hyperlinks in the generated reference manual.

For example, a generic function definition contains a list of method definitions (not raw method objects; pointers to the corresponding method definitions), a reference to it's method combination definition, but also a reference to a setf expander using this function for access, and a list of (short form) setf expanders using this function for update, if applicable.

[4]https://www.lrde.epita.fr/~didier/software/lisp/declt/bibliography/

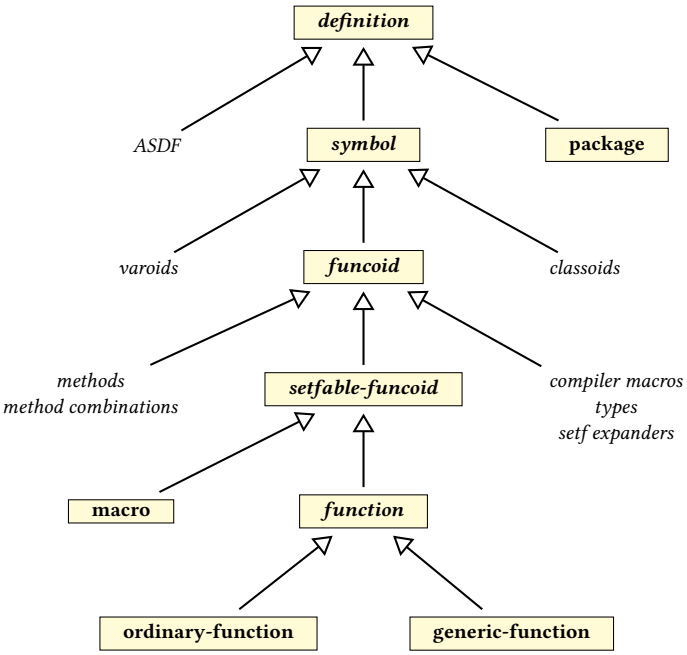

Figure 2: Definitions Hierarchy Excerpt

2.2 Dumping

As mentioned before, Declt reports were originally used by Quickref only to dump library author information, so as to build an author index afterwards. When the idea of a full cohort emerged, we decided to evaluate the potential usefulness of the idea by first creating a quick cohort prototype.

To this aim, the current prototype only dumps an incomplete and simplified version of Declt reports, that is, without performing true serialization. Pointers to the original Lisp objects can of course *not* be preserved in the dump. Cross-references between definitions are not currently preserved either, and only a few interesting attributes of each definition kind are retained, with some amount of pre-processing for subsequent statistical analysis.

Figure 3 provides an excerpt from the dump of Declt's own report. The contents should be mostly self-explanatory. Programmatic definitions start by a keyword denoting the definition kind, and name. Docstrings are replaced by their length, and cross-references by their number.

Such a simple dump already provides enough information to perform all sorts of interesting morphological studies on the 2000+

```
("net.didierverna.declt"
  (:CONTACTS 1)
  ...
  (:SYSTEM "net.didierverna.declt.assess"
   :DOCSTRING 44 :DEPENDENCIES 2 :CHILDREN 2
   :DEFSYSTEM-DEPENDENCIES 0)
  ...
  (:PACKAGE "NET.DIDIERVERNA.DECLT.ASSESS"
   :DOCSTRING 39
   :EXTERNAL-SYMBOLS 169 :INTERNAL-SYMBOLS 119
   :USE-LIST 2 :USED-BY-LIST 1)
  ...
  (:CLASS "GENERIC-FUNCTION-DEFINITION"
   :DOCSTRING 154
   :DIRECT-SUPERCLASSES 1 :DIRECT-SUBCLASSES 1
   :DIRECT-METHODS 11
   :DIRECT-SLOTS 3)
  ...
  (:GENERIC-FUNCTION "DOCUMENT"
   :DOCSTRING 45 :MEHTODS 39)
  ...)
```

Figure 3: Declt Dump Excerpt

libraries available in Quicklisp, as will be exemplified in the next section.

3 QUICKREF COHORT ANALYSIS

The current (beta) version of Quickref dumps Declt reports, as shown in the previous section, for every Quicklisp library. The resulting cohort (containing more than half a million programmatic definitions) is available for download from the website[5]. In order to demonstrate its potential usefulness, Quickref also performs a number of example statistical computations on the cohort, and generates subsequent plots, also visible on the website. Some of them are reproduced below.

3.1 Symbols Morphology

Figure 4 presents the histogram of symbol names lengths in Quicklisp, showing a peak at 11 characters, but also going as far as 135 characters for a single symbol name. Two other plots, not included in this article but visible on the website, show that most composed symbols have a cardinality (the number of `com-po-nents`) of 1, 2, or 3. The longest symbol appears to have 13 components. Most symbol components are 4 characters long, although one symbol (with a cardinality of 2) has a 126 characters long component. In fact, it is the very same symbol that is 135 characters long in total.

3.2 Documentation Shape

Another interesting area of investigation is the current state of Lisp documentation. Figure 5 shows the percentage of documented definitions per kind. For most types of programmatic entities, only 20 to 40% get a docstring. Slightly above this range are method combinations: half of them seem documented. On the other hand,

[5] https://quickref.common-lisp.net/cohort/

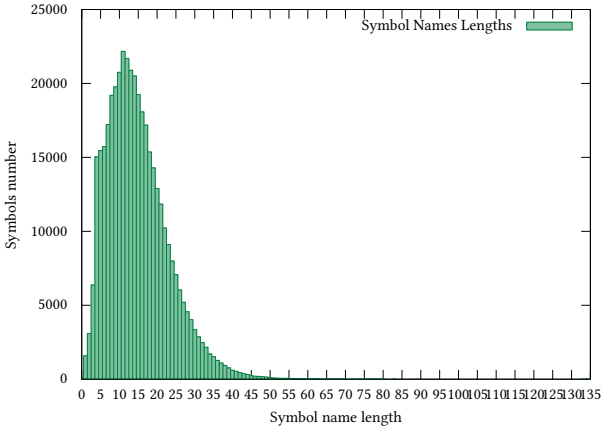

Figure 4: Symbol Names Lengths Histogram

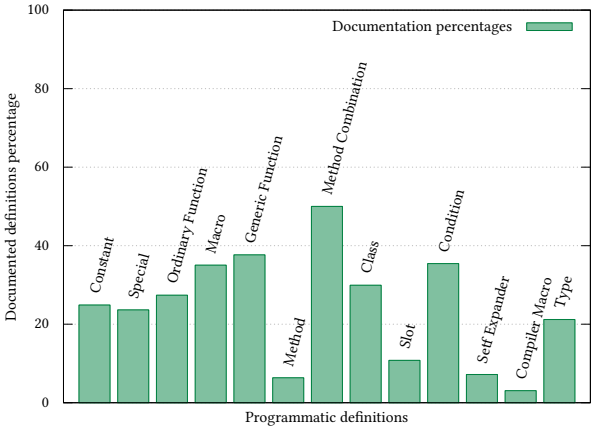

Figure 5: Documentation Percentages

Lisp programmers seem to disregard the documentation capabilities of methods, slots, setf expanders and compiler macros.

3.3 Classoid Profiles

As a final example of cohort analysis, Figure 6 presents the average number of direct slots, methods, parents, and children for structures, classes, and conditions. The most striking element in this plot is the average number of direct methods on classes, a little more than 6, which is much higher than on structures or conditions. It also seems that the multiple inheritance capability of classes and conditions is not used extensively, as the average number of parents remains only slightly above 1 (of course, it is *exactly* 1 for structures). Finally, we can see that the average number of direct slots is significantly higher in structures than in classes (and even more so in conditions), probably because of slot inheritance. Indeed, we can also see, by looking at the average number of children, that subclassing is more frequent than "substructuring".

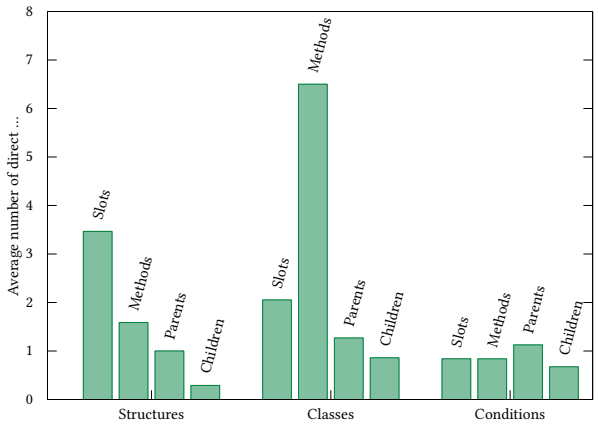

Figure 6: Aggregative Data Structure Averages

4 PERSPECTIVES

The Quickref cohort is currently a proof of concept, but we hope that the existence of a free database of more that half a million programmatic (and ASDF) entities will trigger some interest. Section 3 provided a glimpse at what can be done with it in terms of statistical analysis, but we're eager to hear about other potential use cases.

The cohort is essentially a collection of Declt reports, presented one way or another. Because of that, it makes sense to equip Declt itself with some cohort manipulation ability. For example, it could be interesting for a Lisp programmer to analyze their own (and only their own) library / libraries in a way similar to what was described in Section 3. We definitely are interested in doing so. We plan on extending Declt along these lines in a near future. In such a case, Declt could even manipulate actual reports (Lisp objects) rather than their dumped form.

In order to make the whole Quickref cohort truly usable, the next step is to stabilize the format used for dumping Declt reports. Contrary to the current format illustrated in section 2.2, Declt reports should be preserved as much as possible in order to *not* impose any limit on potential applications. In particular, no precomputation should be performed prior to dumping and cross-references between definitions should be preserved.

On the other hand, some parts of the reports need not (in fact, should not) be preserved in the dump. We want the ability to manipulate reports *without* the corresponding libraries being loaded in memory. This means that the actual Lisp objects corresponding to each definition (whether programmatic or ASDF) should be excluded from the dump.

All in all, it seems that what we are talking about here is some kind of serialization, a topic on which we currently have no experience. Consequently, we're eager to get some advice on that matter.

REFERENCES

[1] Antoine Hacquard and Didier Verna. A corpus processing and analysis pipeline for Quickref. In *14th European Lisp Symposium*, pages 27–35, Online, May 2021. ISBN 9782955747452. doi: 10.5281/zenodo.4714443.

[2] Didier Verna. Declt 1.0 is out. https://www.didierverna.net/blog/index.php?post/2013/08/24/Declt-1.0-is-out, August 2013. Blog entry.

[3] Didier Verna. Declt 2.3 "Robert April" is out. https://www.didierverna.net/blog/index.php?post/2017/10/16/Declt-2.2-Christopher-Pike-is-out, October 2017. Blog entry.

[4] Didier Verna. Announcing Quickref: a global documentation project for Common Lisp. https://www.didierverna.net/blog/index.php?post/2017/12/13/Announcing-Quickref%3A-a-global-documentation-project-for-Common-Lisp, December 2017. Blog entry.

[5] Didier Verna. Parallelizing Quickref. In *12th European Lisp Symposium*, pages 89–96, Genova, Italy, April 2019. ISBN 9782955747438. doi: 10.5281/zenodo.2632534.

[6] 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, TEX Users Group, September 2019.

[7] Didier Verna. Declt 4.0 beta 1 "William Riker" is released. https://www.didierverna.net/blog/index.php?post/2022/05/10/Declt-4.0-beta-1-William-Riker-is-released, May 2022. Blog entry.

py4cl2-cffi: Using CPython's C API to call Python callables from Common Lisp

Shubhamkar Ayare
shubhamkar.ayare@gmail.com

ABSTRACT

Common Lisp is an ANSI-standardized programming language with excellent implementations and several features that make it suitable for both prototyping and long-term projects. However, the library count of Common Lisp (about 2,000 in Ultralisp) vastly lags mainstream programming languages like Python (about 500,000 in PyPI). In recent years, the Lisp library py4cl used a subprocess and stream-based interprocess-communication (IPC) approach to call Python from Common Lisp. Another library py4cl2 built upon py4cl's IPC approach and imported information about Python callables' function signature through Python's introspection facilities and also enabled a switchable type mapping. However, interprocess communication can be very slow. Furthermore, both py4cl and py4cl2 primarily rely on eval and exec, which is generally frowned upon. In this paper, we present a Lisp library, py4cl2-cffi, that provides a CFFI bridge to Python libraries through the excellently documented C-API of CPython and compare its performance against other existing approaches that bring Python libraries to other languages.

CCS CONCEPTS

• **Software and its engineering** → Interoperability.

KEYWORDS

Common Lisp, Python, CPython, Foreign function interface

ACM Reference Format:
Shubhamkar Ayare. 2024. py4cl2-cffi: Using CPython's C API to call Python callables from Common Lisp. In *Proceedings of the 17th European Lisp Symposium (ELS'24).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.5281/zenodo.10997435

1 INTRODUCTION

Lisps are known for their homoiconicity and metaprogramming facilities. The ANSI 1994 standard of Common Lisp[20] packs local lexical binding with global dynamic binding, a numeric tower, a package system for namespacing, the Common Lisp Object System with multiple dispatch, a condition system that allows restarting programs without unwinding the stack[16], and an extensive iteration facility. In the years since, several additional facilities have been standardized through de facto standard libraries[14]. These include multithreading, Meta-Object Protocol, a C foreign function interface, and more. Excellent implementations such as SBCL generate native assembly code (including SIMD) and also provide facilities for compile-time type checking.

In addition to these, Common Lisp has extensive support for interactive programming allowing for redefining constants, variables, functions, macros, packages, and classes at runtime. This has been primarily available through Emacs[22], but efforts are underway to port this to Intellij/Jetbrains[10] as well as VS Code[15].

Overall, these facilities make Common Lisp implementations excellent tools for both exploratory research and long-term projects. However, the 500,000+ Python libraries in PyPI vastly outnumber the 2000+ Lisp libraries in Ultralisp[6]. In modern times, this can make Lisp less suitable for exploratory projects as one may be left to reinvent the Python library in Lisp. The current project is an attempt at providing facilities to call CPython callables from Common Lisp, so that the extensive ecosystem of CPython becomes useable through Common Lisp implementations. This is achieved by embedding CPython in Common Lisp using the former's excellently documented C-API[5].

1.1 Previous Work

Burgled batteries[11] aimed at a deep integration between Common Lisp and CPython, so that a Lisp user of burgled-batteries could have transparent access to Python objects. However, a deep integration might not be the best approach to the problem[12]. The CPython and Common Lisp environments can disagree on signal handling, timeouts, floating point traps, or other low-level details. They also have high-level disagreements, for example, how to resolve a class' inheritance tree. py4cl[9] and py4cl2[7] avoid these issues by keeping the Python and Lisp processes separate and calling Python through stream-based interprocess communication.

While burgled-batteries aimed to interface with the CPython implementation of Python, CLPython[8] aims at providing a complete Python implementation written in Common Lisp itself. This means that Python code can be compiled to native Lisp, potentially side-stepping many of the issues discussed above. On the other hand, CLPython does not have inherent access to C extensions written for CPython such as Numpy.

1.2 Our Contribution

The interprocess-communication approach of py4cl and py4cl2 has a high per-Python-call overhead. The C-API approach of burgled-batteries and py4cl2-cffi can minimize this overhead. But in contrast to burgled-batteries, py4cl2-cffi does not aim to enable subclassing and inheriting Python classes which may raise issues of resolving inheritance trees. We only aim to conveniently call Python callables (see section on Type Correspondence). This enables us to achieve several tasks marked as "To Do" on burgled-batteries in an equivalent amount of code (about 3,000 LOC).

ELS'24, May 6–7 2024, Vienna, Austria

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.5281/zenodo.10997435

We hope that describing the implementation of py4cl2-cffi through this document would allow other developers (including those of CLPython) to interface with CPython according to their own requirements and also enable better maintenance of py4cl2-cffi itself over the longer run.

2 A BRIEF INTRODUCTION TO CPYTHON'S PYTHON/C API

CPython provides an extensive Python/C API for extending and embedding Python [5]. The API is made available in a C program through the following two lines of code:

```
#define PY_SSIZE_T_CLEAN
#include <Python.h>
```

Most Python/C API functions take in one or more arguments and return an object which is an instance of the C structure `PyObject`. Each such object has a type and a reference count associated with it. The reference count is the number of places with a strong reference to the object. The object is deallocated once its reference count becomes zero. Strong reference may be contrasted with borrowed reference. By holding a strong reference, the calling code shows ownership of the object and indicates that the object should not be deallocated. Once the calling code has no use of the `PyObject` instance, it must decrement the reference count of the object, giving up its ownership of the object. On the other hand, by holding a borrowed reference, the calling code shows no ownership and indicates that the object may be deallocated while the calling code is still in progress. Most Python/C API functions return a new (strong) reference to the returned `PyObject` instance . However, some return a borrowed reference. Some functions even steal the reference of their arguments, that is, they take over the ownership of the object. Such functions are especially convenient when populating tuples or lists. The Python/C API expects reference counts for embedded Python to be managed through the `Py_IncRef` and `Py_DecRef` C functions.

The API relies on a considerable use of C macros available only during the compilation of the C code. On the other hand, Common Lisp's CFFI relies on C functions and variables available in a foreign library loaded dynamically. For this reason, several C macros are wrapped in equivalent C functions in py4cl-utils.c[1]. This file also contains other utility functions aimed at obtaining compile-time information from the C environment.

In the rest of this paper, the term `PyObject` will denote either the C structure `PyObject` or an instance of that structure. Such a `PyObject` instance is essentially just a `cffi:foreign-pointer` in Lisp. However, we also implement a wrapper structure `pyobject-wrapper` in Lisp. An instance of `pyobject-wrapper` stores a `cffi:foreign-pointer` pointing to the corresponding `PyObject`.

The Python GIL

The Python interpreter is thread unsafe and expects that a global lock — named the Global Interpreter Lock (GIL) — must be held before doing even the simplest operations[4]. These include operations to manage the reference counts of objects. We use the C functions `PyGILState_Ensure` and `PyGILState_Release` for this purpose. To operate well with the often-multithreaded interactive environment of SLIME, we attempt to release the GIL lock as soon as possible during `(pystart)` itself. Subsequently, a Lisp expression using the Python/C API can be compiled or evaluated in any Emacs buffer. However, it must first acquire the GIL, then perform what it actually wants to perform with the embedded Python, and finally release the GIL. Functions exported by the py4cl2-cffi package use the `pyforeign-funcall` macro implemented as a wrapper around `cffi:foreign-funcall`. Amongst other things, `pyforeign-funcall` acquires and releases the GIL. Thus, users of py4cl2-cffi need not worry about acquiring and releasing the GIL while using the py4cl2-cffi interface.

3 USING PYTHON3-CONFIG TO AUTOMATE CONFIGURATION

Python comes with a convenient program `python(3)-config` that supplies build options for embedding Python. py4cl2-cffi makes use of python(3)-config present in the environment to obtain the list of includes, ldflags as well as shared libraries and their locations. The system and package `py4cl2-cffi/config`[2] obtain this information by running python3-config with `uiop:run-program` and stores it in the following Lisp variables

- *python-ldflags*
- *python-ignore-ldflags*
- *python-includes*
- *python-compile-command*

This allows py4cl2-cffi to be simply dropped in a Python virtual or conda environment. It will pick up the libraries for embedding the appropriate Python automatically, thus easing user configuration.

4 CALLING PYTHON CALLABLES

py4cl[9] and py4cl2[7] primarily rely on eval and exec without bothering with the Python internals. In contrast, py4cl2-cffi uses the internals in an attempt to achieve better performance. The process of calling a Python callable from Lisp can be described in the following steps, implemented by the Lisp function `pycall` in the source code:

(1) retrieve the `PyObject` corresponding to the Python-callable
(2) `pythonize` the Lisp arguments: convert them to `PyObject`
(3) call the Python callable with the pythonized arguments
(4) `lispify` the Python return values: convert `PyObject` to Lisp objects

4.1 Retrieving the PyObject corresponding to the python-callable

For convenience, one needs a way to retrieve `PyObject` instances bound to a name given as strings. py4cl2-cffi provides the functions `pyvalue` and `pyvalue*` to perform exactly this. Given a Python name as a Lisp string, `pyvalue*` returns a `cffi:foreign-pointer` pointing to the `PyObject` instance, while `pyvalue` lispifies the `PyObject` instance.

[1] The source code for py4cl2-cffi is available at https://github.com/digikar99/py4cl2-cffi.

[2] py4cl2 loads after py4cl2-cffi/config

```
(defvar *py-type-lispifier-table*
        (make-hash-table :test #'equal))
(defmacro define-lispifier
    (name (pyobject-var) &body body)
  (declare (type string name))
  `(setf (gethash ,name *py-type-lispifier-table*)
         (lambda (,pyobject-var) ,@body)))
(define-lispifier "int" (o)
  (pyforeign-funcall "PyLong_AsLong"
                     :pointer o
                     :long))
(define-lispifier "numpy.ndarray" (o)
  ...)
```

Listing 1: The define-lispifier macro and some examples illustrating the extensible functionality to convert `PyObject` instances to Lisp objects.

4.2 `pythonize`-ing the Lisp arguments

The Python/C API provides several functions to create and interact with `PyObject` instances using built-in C data types such as long, const char*, float, and double. In `pythonizers.lisp`, use them extensively to provide a generic function `pythonize` with several implemented methods. Each takes in a Lisp object and returns a `cffi:foreign-pointer` to a corresponding `PyObject` instance.

Object Handles for Unknown Lisp Objects. When there is no specialized method for the Lisp object, the default method of `pythonize` is called. This takes in a Lisp object, creates a unique handle (an integer) corresponding to the Lisp object, wraps the handle in an instance of the Python class `UnknownLispObject`, and returns a `cffi:foreign-pointer` pointing to this object. This mapping is maintained in a `cl:hash-table`.

4.3 `lispify`-ing the Python return values

The return value of a Python callable is a `cffi:foreign-pointer` to a `PyObject` instance. It is a new (strong) reference. The function `lispify` in `lispifiers.lisp` takes the pointer to `PyObject` as input and returns a Lisp value corresponding to the `PyObject`. While `pythonize` is implemented as a generic function, `lispify` is implemented manually as a function dispatching from the Python type to a lambda function using a `cl:hash-table`. The Python type is itself retrieved from the `PyObject` as a string using the helper functions in `py4cl-utils.c`. Several lispifiers are provided for the standard types, including integers, floats, strings, tuples, lists, dicts. Users can implement additional lispifiers using the exported `define-lispifier` macro.

Lisp Objects from Unknown Lisp Objects. If the `PyObject` is an instance of `UnknownLispObject`, then the Lisp object is retrieved from the integer handle stored in the `UnknownLispObject` instance.

Unknown Python Objects. When there is no lispifier for a given Python type, then the pointer to the `PyObject` is stored in an instance of `pyobject-wrapper`. Pythonize-ing such a `pyobject-wrapper` instance involves merely returning the pointer stored in the instance's slots.

4.4 Calling the Python callable with pythonized arguments

The Python/C API provides several C functions [2] to call `PyObject`. We rely on the most general `PyObject_Call`, which takes in a pointer to a Python callable, a tuple of positional arguments, and an optional dictionary of keyword arguments. It returns a pointer to a `PyObject` instance if no errors occur during the execution of the Python callable. This is a new (strong) reference. In case of an error, `PyObject_Call` returns a NULL pointer.

Python error handling. The Python/C API function `PyErr_Occurred` is used to check if an error has occurred. It returns a NULL pointer if no error has occurred but a pointer to the exception type if an error has occurred. In case of an error, py4cl2-cffi retrieves the error information using `PyErr_Fetch`, and raises a Lisp error of class `pyerror` with the Python error and traceback. Proceeding to run more Python code without checking for the error may lead to mysterious failures [3]. Thus, it becomes important to check for errors after calling a Python callable. In py4cl2-cffi, the previously mentioned `pyforeign-funcall` macro also performs error checking before returning.

5 TYPE CORRESPONDENCE

Type conversions involved while calling a Python callable are performed according to a correspondence summarized by Table 1. A few peculiarities are raised by the overloaded semantics of `cl:nil`. In Common Lisp, `cl:nil` is the boolean false value, the null object, as well as the empty list. Python makes a distinction between `False`, `None`, as well as the empty tuple `()`. Thus, to enable the Lisp user to disambiguate these cases of Python's values -

(1) Python's False value is mapped to the cl:nil Lisp value (equivalently the (eql nil) or the null Lisp type).
(2) Python's `None` value (equivalently, objects of Python type `NoneType`), are mapped to a Lisp constant `+py-none+` within the `py4cl2-cffi` package.
(3) Python tuples are mapped to Lisp lists. This leads to the natural mapping that empty tuples would be mapped to Lisp's `nil`. However, to avoid ambiguity with the above cases, we map the empty tuple to `+py-empty-tuple+` within the `py4cl2-cffi` package.

Amongst other tasks performed to initialize the embedded Python interpreter, `(py4cl2-cffi:pystart)` also binds the Lisp constants `+py-none+` and `+py-empty-tuple+` to the `pyobject-wrapper` instances corresponding to the Python values None and the empty tuple respectively.

5.1 Customizing the type correspondence

Different users — or even different applications by the same user — may want different type correspondences. A set of custom pythonizers may be installed using the Lisp variable `*pythonizers*` or the macro `with-pythonizers`. A custom pythonizer maps from a Lisp type to a Python callable. Similarly, a set of custom lispifiers may be installed using the Lisp variable `*lispifiers*` or the macro `with-lispifiers`. These map from a Lisp type to a Lisp function. Listing 2 provides an example usage.

Lisp type	Python type (value)
(eql nil)	(False)
(eql +py-none+)	NoneType (None)
integer	int
rational	fractions.Fraction
float	float
complex	complex
(or string (and symbol (not null))	str
hash-table	dict
(or list (eql +py-empty-tuple+))	tuple
vector	list
array	numpy.ndarray
function	LispCallbackObject
*	UnknownLispObject
pyobject-wrapper	*

Table 1: A summary of the correspondence between Lisp and Python types. Pythonize-ing Lisp values of types given in the right column gives `PyObject` instances of types given by the left column. Lispify-ing the `PyObject` instances of Python types given in the left column gives Lisp objects of Lisp types given by the right column.

```
; A convenience function
(defun pyprint (object)
  (pycall "print" object)
  (pycall "sys.stdout.flush")
  (values))

(pyprint #(1 2 3)) ; prints [1, 2, 3] ; the default object
(with-pythonizers ((vector "tuple"))
  (pyprint #(1 2 3))
  (pyprint 5))
; (1, 2, 3) ; coerced to tuple by the pythonizer
; 5         ; pythonizer uncalled for non-VECTOR
5

(raw-pyeval "[1, 2, 3]") ;=> #(1 2 3) ; the default lispified object
(with-lispifiers ((vector (lambda (x) (coerce (print x) 'list))))
  (print (raw-pyeval "[1,2,3]"))
  (print (raw-pyeval "5")))
; #(1 2 3) ; default lispified object
; (1 2 3)  ; coerced to LIST by the lispifier
; 5        ; lispifier uncalled for non-VECTOR
5
```

Listing 2: An example usage of `with-pythonizers` and `with-lispifiers`. Both macros take a list of overriding lispifiers as the first argument. Each element of this list of two elements: the first of which is a Lisp type, while the second is a Python function or a Lisp function respectively.

5.2 with-remote-objects

Sometimes, converting a Python object to Lisp can be prohibitively expensive. Such cases arise particularly while performing operations on large datasets. For these cases, py4cl[9] provided macros so that the code in the body of these macros returns an integer handle to Lisp without converting the Python object to Lisp. The `with-remote-objects` macro in py4cl2-cffi maintains this functionality but returns a `pyobject-wrapper` instance instead. `pyvalue` and `pycall` usually try to lispify the `PyObject` and return a `pyobject-wrapper` only when a lispifier is not found. Instead, when executed inside the body of `with-remote-objects`, they always return a `pyobject-wrapper`.

5.3 passing array data by reference

In contrast to other data types, the data in specialized arrays in Common Lisp can be passed by reference to Python callables. Numpy's C API[1] provides functions such as `PyArray_NewFromDescr` that take in a pointer to a block of memory (along with other arguments) and return a pointer to PyObject denoting the numpy array. This can be coupled with Common Lisp CFFI's `cffi:with-pointer-to-vector-data` that pins Lisp vectors and obtains a C pointer to the vector's data. Thus, Lisp array data can be passed by reference, that is, without copying their data. However, the data in the output arrays of Python callables has to be copied over to Lisp arrays. Most Numpy functions take in an `out` keyword argument corresponding to the output arrays, sidestepping this problem. For algorithms with superlinear computational complexity, too, copying data should not be a limitation. For convenience, function wrappers are defined in a helper file `py4cl-utils.c` that convert Lisp array element types to and from the enumerated Numpy array element types.

6 MEMORY MANAGEMENT

As discussed in section 2, several functions in the Python's C-API create new (strong) references to the `PyObject` instances. In other cases, developers may themselves need to create the new reference using the C function `Py_IncRef`. The `PyObject` instance is deallocated by Python only when its reference count reaches zero. Thus, possessing a strong reference prevents the object from being deallocated while the object is being used. Once the object is no longer needed, its reference count must be decreased using `Py_DecRef`.

The Common Lisp ANSI Standard leaves memory management up to the individual implementations. However, trivial-garbage[18] provides a portability layer over the different implementations.

To recall, there are two ways in which `PyObject` instances are converted to Lisp objects:

(1) Python to Lisp correspondence is known
(2) Python to Lisp correspondence is unknown

Both cases start out with a `cffi:foreign-pointer` to the instance of `PyObject` in C. This pointer was the return value of the C function `PyObject_Call` or equivalents. In the first case, the internals of the `PyObject` instance are used to construct a corresponding Lisp object using a lispifier. The `PyObject` instance is no longer needed, thus, its reference count can be decremented immediately. This case is handled by the `with-pygc` macro. Just before the toplevel `with-pygc` form exits, it decrements (or increments in case of stolen references) the reference counts. This reference count management is performed by running `(pygc)` only during the exit of the toplevel `with-pygc`. This also means that if users are passing around `cffi:foreign-pointer`, they should wrap their code in a `(with-pygc ...)` form. Ignoring this, the code could result in segmentation faults in the best case, and continue with unexpected results in the worst case.

For instance, at the time of writing, the below code returns an unexpected value "str". `pycall*` returns a `cffi:foreign-pointer` to the `PyObject` instance of the string `"1"`. This pointer is initially

bound to `ptr`. However, before the value of `ptr` returns from the `let`-form, a call to `(pygc)` takes place which decrements the reference count of the `PyObject` instance to zero. This frees up the `PyObject` pointed to by the `ptr`. The outermost `(pycall "str" ...)` continues to use this freed up object, leading to the unexpected value "str".

```
(pycall "str"
        (let ((ptr (pycall* "str" 1)))
          -- some code that results in a call to (pygc) --
          ptr))
;=> "str"
```

But wrapping it inside `(with-pygc ...)` avoids prematurely freeing up the `PyObject` `"1"` and maintains it until the toplevel `(with-pygc ...)` escapes. This leads to the expected result "1".

```
(with-pygc
  (pycall "str"
          (let ((ptr (pycall* "str" 1)))
            ;; (pycall* ...) returns a cffi:foreign-pointer
            -- some code that results in a call to (pygc) --
            ptr)))
;=> "1"
```

Note that the above wrapping is required only if one passes around foreign pointers. Avoiding the requirement of wrapping requires being able to finalize a `cffi:foreign-pointer`. This way, as soon as a `cffi:foreign-pointer` is no longer accessible in the Lisp environment, the reference count of the corresponding `PyObject` can be incremented or decremented. But this is not recommended[19].

The second case, when Python to Lisp correspondence is unknown, is actually easier. In this case, an instance of `pyobject-wrapper` is created with the pointer to `PyObject` as one of its slots. The `pyobject-wrapper` instance is finalized using `tg:finalize` so that when it is garbage collected by the Lisp environment, it results in a call to the C function `Py_DecRef`. Note that one needs to hold the GIL while decrementing the reference count, and that the finalizer may be called in a different thread altogether.

Beyond the reference count management of `PyObject` instances, one also needs to clear the `cl:hash-table` used for maintaining the integer handles to Unknown Lisp Objects. This task is also performed by `(pygc)`.

7 HANDLING PYTHON'S STDOUT & STDERR

We expect one of the primary attractions of Common Lisp to be the interactive Emacs/SLIME[3] development environment [22]. Left in its default state, the embedded Python has its stdout pointing to the file descriptor 1 of the Lisp process. This is reasonable in the absence of multithreading as the file descriptor 1 may also be pointing to the Lisp REPL's display itself. However, in multithreaded environments, SLIME has a dedicated thread[4] for the REPL and file descriptor 1 may not point to the REPL's display.

In py4cl[9], Python's stdout (that is, `sys.stdout`) is set to Python's `io.StringIO`. The output is, thus, collected into a string during the evaluation or execution of a Python expression or statement. After the evaluation or execution is complete, the output is then sent to Lisp, where it is then sent to `cl:*standard-output*`. This allows

```
(cl:with-output-to-string (*standard-output*)
  ...)
```

to behave as expected and capture the Python output emitted inside the `cl:with-output-to-string` form. On the other hand, this has the disadvantage that the Python's output is unavailable to the Lisp user before the complete evaluation or execution of the Python expression or statements. See Listing 3 for an example. If a piece of code takes too long or loops infinitely, it is convenient to know its intermediate output even if the execution is incomplete.

```
(raw-pyexec "
import time

for i in range(5):
  print(i)
  sys.stdout.flush()
  time.sleep(1)")
```

Listing 3: Example code demonstrating the need for capturing Python output asynchronously. See the text for an accompanying explanation.

Thus, in py4cl2-cffi, we redirect the embedded Python's stdout to a named pipe. We read the contents of the named pipe from a separate Lisp thread and print its contents to the stream indicated by the initial value of `cl:*standard-output*` of the thread calling `(pystart)`. This allows the Python's output to be available to the Lisp user even if the processing is incomplete on the Python side. However, on certain implementations including SBCL `cl:with-output-to-string` establishes local bindings through a `cl:let`. This locally modified binding of `cl:*standard-output*` may not be visible from the separate Python output thread. Thus, this approach requires a dedicated functionality to obtain the Python output as a string.

with-python-output

The functionality to obtain Python output as a string is implemented as a macro `with-python-output`. This enables one to obtain the Python output as Lisp strings.

```
(with-python-output
  (pycall "print" "hello" :end ""))
;=> "hello"
```

However, this facility currently depends on a system of semaphores and locks interoperating correctly, and might be more complicated and prone to errors than is necessary. Listing 4 summarizes the algorithms followed by the two threads involved in output processing of stdout.

with-python-error-output

Similar arguments hold for the Python error messages sent to stderr (that is, `sys.stderr`). In this case, the equivalent functionality is provided through the `with-python-error-output macro`.

[3]We use the term 'Emacs/SLIME development environment' as a blanket term for other Common Lisp IDEs that enable interactive development, and hope that the solutions employed by py4cl2-cffi for Emacs/SLIME would also be trivially applicable to them.
[4]See the sub-section 3.1.3 on User-interface Conventions -> Multithreading of the SLIME User Manual version 2.24 [23].

```
Shared resources:
  A counter: in-with-python-count # initialized to 0
  A recursive lock: count-lock    # accompanying the counter
  Two semaphores: start-semaphore, end-semaphore
  An input stream: py-stream
    # an input stream from a named pipe to which
    # python writes its output
  Two output stream:
    default-output-stream, with-python-stream

python-output-thread:
  # The thread that reads python output from the named pipe
  # and prints the output to the lisp
  loop:
    if in-with-python-count == 0:
      # We are outside all of the with-python-output form
      peek(py-stream) # wait for input
      if in-with-python-count == 0:
        # Check if we are still outside all of the
        # with-python-output forms
        char = read_char(py-stream)
        write_char(char, default-output-stream)
    else:
      without-python-gil:
        wait(start-semaphore)
      loop while listen(py-stream):
        char := read_char(py-stream)
        write_char(char, with-python-stream)
      signal(end-semaphore)
      without-python-gil:
        with-recursive-lock-held(count-lock):
          decf(in-with-python-count)

Thread executing (with-python-output &body body):
  # Thread in which the (with-python-output ...) form runs
  with-python-stream := make_string_output_stream()
  without-python-gil:
    with-recursive-lock-held(count-lock):
      incf(in-with-python-count)
  -- body --
  signal(start-semaphore)
  without-python-gil:
    wait(end-semaphore)
  OUTPUT := get_output_stream_string(with-python-stream)
```

Listing 4: Algorithms used by python-output-thread and the thread executing with-python-output. The OUTPUT is the output of the form (with-python-output ...) executed by the second thread.

8 PYTHON CALLABLES AS LISP FUNCTIONS, PYTHON MODULES AS LISP PACKAGES

py4cl[9] provides an `import-function` macro to define a Lisp function to call a Python callable. It also provides an `import-module` macro to define a Lisp package corresponding to a Python module. Such a Lisp package contains symbols fbound to Lisp functions that call Python callables in the corresponding module. This functionality is enabled by the `inspect` module of Python that ships with its standard library. `import-module` of py4cl is renamed to `defpymodule` in py4cl2[7] and py4cl2-cffi to signify that a certain something — a Lisp package — is being defined. py4cl2 and py4cl2-cffi use further introspection facilities of Python and perform three additions (i) import parameter list of the functions (ii) define packages recursively to access submodules (iii) add helper code to reimport functions or modules when the Python process is restarted (in py4cl2) or embedded Python's state is cleaned (in py4cl2-cffi).

8.1 Importing parameter lists of functions

The `inspect.signature` Python function takes in a Python callable and returns a `inspect.Signature` object containing the parameter list of the callable. Processing the `inspect.Signature` object allows preparing the parameter list for the Lisp function. There are several considerations, however:

(1) Python callables can take in a variable number of positional arguments and a variable number of keyword arguments at the same time. In contrast, Lisp functions only allow either a variable number of positional arguments or a variable number of keyword arguments but not both. Thus, a Lisp function calling such a Python callable has its parameter list given by the default `(&rest args)`.
(2) In some cases, such as when the Python callable is a built-in function defined in C, they may not have a signature. In this case too, the Lisp function calling the Python callable has the parameter list given by the default `(&rest args)`.
(3) Python is case-sensitive, while the Lisp reader is case insensitive by default. Thus, it is possible for a Python function to have arguments which map to the same Lisp name.

Many functions in the Python package numpy are also subject to the second consideration above. These functions are instances of numpy.ufunc. But, user convenience demands that the Lisp functions calling these functions should have their parameter lists more specific than the default (&rest args). To handle this case in general, py4cl2 and py4cl2-cffi provide a generic function `%get-arg-list` which specializes on its first argument. This first argument is the name of the type of the Python callable converted to a Lisp keyword. This allows for the implementation of callable-type-specific logic to derive the parameter list of the function.

8.2 Defining packages recursively to access submodules

The `defpymodule` macro provided by py4cl2 and py4cl2-cffi take in an optional argument `import-submodules`. If this argument is non-NIL, `defpymodule` calls the helper function `defpysubmodules`. This helper function uses the `pkgutil` Python module along with some inspection and recursion to define further submodules if required, as in the case of Python packages[5].

8.3 Helper code for reimport

Experience suggests that Python is restarted more often than Lisp. Thus, Lisp code bridging to Python should consider that the Python interpreter might be in a fresh state and the Python function or module may not have been imported yet.

[5]Python packages are a collection of Python modules. Each Python module is a distinct Python file.

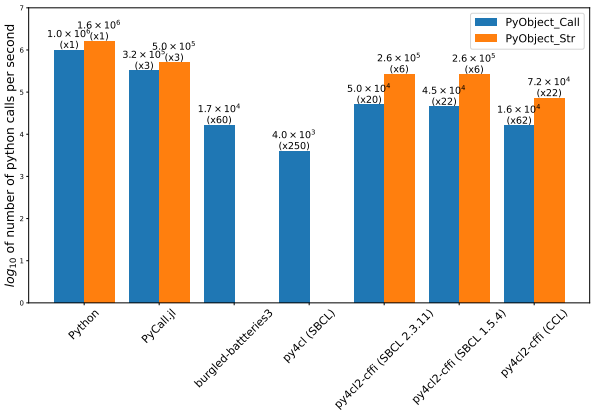

Figure 1: Performance comparison of calling CPython natively against several libraries and platforms. Numbers inside brackets indicate the degree of slowness compared to native CPython — thus, x3 means 3 times as slow as using CPython directly. x20 means 20 times as slow.

9 PERFORMANCE

The prime reason for developing py4cl2-cffi has been performance. Two main ways to call Python callable have been considered. The one involving the Python/C API function `PyObject_Call` is the most generic and allows calling any Python callable. The particular situation we test is using the Python callable `str` to convert an integer to a string. However, this particular task can also be performed by `PyObject_Str`. As Figure 1 shows, this can be substantially faster than calling `PyObject_Call`.

The CPU frequency was locked to 0.8GHz using the cpufreq-set of cpufreq-utils. burgled-batteries3 used SBCL 1.5.4 and python3.6. The performance of py4cl2-cffi was measured for both SBCL 1.5.4 and python3.6 as well as SBCL 2.3.11 and python3.10. The others used SBCL 2.3.11 or CCL 1.12.2 and python3.10. PyCall.jl[17] used Julia 1.10.

The overall code for measuring the performance is in the `perf-compare/` directory in the project root[6]. Listing 5 demonstrates loading this performance data using pandas and using it to plot Figure 1 using matplotlib.

10 LIMITATIONS AND FUTURE WORK

Even though py4cl2-cffi achieves a number of tasks marked as "To Do" in burgled-batteries[11], it still has a number of limitations which we discuss below.

10.1 Limited to Unix-based OS

The Continuous Integration set up for py4cl2-cffi using Github Actions tests py4cl2-cffi on SBCL, CCL, and ECL, thus providing a reasonable amount of portability across the Lisp implementations. On the other hand, this portability is limited to Unix-based operating systems, and particularly Linux and MacOS. Porting py4cl2-cffi to other operating systems requires more work. Alternatively, OS independent facilities need to be used.

[6]See https://github.com/digikar99/py4cl2-cffi/tree/master/perf-compare.

```
(defpackage #:python-call-performance
  (:use :cl)
  (:local-nicknames (#:py #:py4cl2-cffi)
                    (#:a  #:alexandria)))
(in-package #:python-call-performance)

(py:defpymodule "matplotlib.pyplot" nil :lisp-package "PLT")
(float-features:with-float-traps-masked t
  (py:raw-pyexec "
import numpy as np
import pandas as pd

def read_csv_file(filename):
  with open(filename) as f:
    return pd.read_csv(f)
"))

(defvar *performance-data*
  (py:pycall "read_csv_file"
             (namestring
              (merge-pathnames "perf-compare/perf.csv"
                               (asdf:system-source-file
                                (asdf:find-system "py4cl2-cffi"))))))

(defun plot-performances (data)
  (let* ((data-labels
          (py:pycall "tuple" (py:pyref data "Platform or Library")))
         (object-call-values
          (py:pycall "tuple" (py:pyref data "PyObject_Call")))
         (str-call-values
          (py:pycall "tuple" (py:pyref data "PyObject_Str")))
         (xloc
          (a:iota (length data-labels))))
    (float-features:with-float-traps-masked t
      (flet ((0.2- (x) (- x 0.2))
             (0.2+ (x) (+ x 0.2))
             (log10 (n) (log n 10))
             (value-to-labels (values)
               (mapcar (lambda (value)
                         (when (and (floatp value)
                                    (float-features:float-nan-p value))
                           (setq value 1))
                         (multiple-value-bind (int decimal-part)
                             (floor (log value 10))
                           (format nil "$~,1F \\times 10^{~D}$~%(x~D)"
                                   (expt 10 decimal-part)
                                   int
                                   (floor (first values) value))))
                       values)))
        (plt:figure :figsize (list 960/80 720/80) :layout "constrained")
        (plt:ylim 0 7)
        (plt:bar-label
         :container (plt:bar :x (mapcar #'0.2- xloc)
                             :height (mapcar #'log10 object-call-values)
                             :width 0.4
                             :label "PyObject_Call")
         :labels (value-to-labels object-call-values)
         :fontsize 14)
        (plt:bar-label
         :container (plt:bar :x (mapcar #'0.2+ xloc)
                             :height (mapcar #'log10 str-call-values)
                             :width 0.4
                             :label "PyObject_Str")
         :labels (value-to-labels str-call-values)
         :fontsize 14)
        (plt:xticks xloc data-labels :rotation 45
                    :fontsize 16)
        (plt:ylabel "$\log_{10}$ of number of python calls per second"
                    :fontsize 18)
        (plt:legend :loc "upper right" :fontsize 16)
        (plt:show)))))

(plot-performances *performance-data*)
```

Listing 5: A demonstration of using py4cl2-cffi to load data using pandas and plotting it using matplotlib.

10.2 A difference of case (in)sensitivity

Common Lisp reader is case insensitive by default, while Python is case sensitive. This causes name translations to jump through a few hoops, making the integration less than ideal. While the Common Lisp reader can be made case sensitive using the `:modern` readtable provided by named-readtables[21], there is also a limitation of tooling. In particular, for both functions below, SLIME displays the lowercase `(foo &key a b)` for both of them, making the two indistinguishable.

```
(defun foo (&key a b) (+ a b))
(defun |foo| (&key |a| b) (+ |a| b))
```

It remains to be seen how other Common Lisp IDEs such as Intellij[10] and Alive on VS Code[15] behave in this regard.

10.3 Performance

From Figure 1, we note that even though py4cl2-cffi might be 10 times faster than py4cl, given the performance of PyCall.jl there is significant scope of improvement. Going by the steps outlined in section 4, several optimizations are visible

(1) If we can assume that the pointer to a Python callable will remain unchanged, the pointer can be fetched from the name at compile time.
(2) If the type of Lisp or Python arguments is known, then a dynamic dispatch on pythonizers or lispifiers can be avoided.

However, beyond these, one may need to look into the particularities to see where the performance disparities between py4cl2-cffi and PyCall.jl stem from.

10.4 Floating point traps and signal handling

The author of burgled-batteries[12] highlighted that Lisp environments and CPython environments can differ on low-level details such as floating point traps and signal handling. The author of py4cl2-cffi has certainly run into issues concerning floating point errors, but the float-features compatibility library[13] has been helpful in sidestepping these issues. Issues about signal handling remain to be encountered. Both these issues require a fuller treatment for more robust Python calls.

10.5 Python ecosystem can inherently oppose multithreading

Python packages (including matplotlib) expect their callables to be called from a single "main" thread in their default setting. This is at odds with Lisp IDEs which are often multithreaded. Thus, one may need to consider unithreaded approaches to calling Python callables. Currently, an experimental system and package `py4cl2-cffi/single-threaded` is provided for this purpose. More testing is required.

11 CONCLUSION

We hope that describing the approach of py4cl2-cffi can help other lispers adopt it for their own use cases. Working towards the limitations outlined here should help make Common Lisp easier for exploratory research even in modern times.

ACKNOWLEDGMENTS

I'd like to express my gratitude to Ben Dudson for py4cl which has served me well and provided a base for both py4cl2 and py4cl2-cffi. The tests for py4cl2-cffi are still based upon the extensive tests for py4cl written by Ben Dudson. py4cl was itself inspired by the cl4py project by Marco Heisig. py4cl2-cffi has received contributions github users 'enometh' and 'jcguu95' as well as from Paul Landes and Robert Brown. Finally, I'd like to thank Marco Heisig (once again!) and Robert Smith for useful guidance on citations and footnotes in the context of this manuscript and the ELS.

REFERENCES

[1] NumPy C-API; NumPy v1.26 Manual — numpy.org. https://numpy.org/doc/stable/reference/c-api/index.html. [Accessed 06-02-2024].
[2] Call Protocol — docs.python.org. https://docs.python.org/3/c-api/call.html#c.PyObject_Call, . [Accessed 16-02-2024].
[3] Exception Handling — docs.python.org. https://docs.python.org/3/c-api/exceptions.html, . [Accessed 16-02-2024].
[4] Initialization, Finalization, and Threads — docs.python.org. https://docs.python.org/3/c-api/init.html, . [Accessed 18-02-2024].
[5] Python/C API Reference Manual — docs.python.org. https://docs.python.org/3/c-api/index.html, . [Accessed 11-01-2024].
[6] Alexander Artemenko and et al. GitHub - ultralisp/ultralisp: The software behind a ultralisp.org common lisp repository — github.com. https://github.com/ultralisp/ultralisp. [Accessed 17-04-2024].
[7] Shubhamkar Ayare, Ben Dudson, Jin, Robert P. Goldman, and Github user - death. GitHub - digikar99/py4cl2: Call python from Common Lisp — github.com. https://github.com/digikar99/py4cl2. [Accessed 08-01-2024].
[8] Willem Broekema. CLPython - an implementation of Python in Common Lisp — clpython.common-lisp.dev. https://clpython.common-lisp.dev/. [Accessed 04-02-2024].
[9] Ben Dudson, Shubhamkar Ayare, Jason Ruchti, and Github User - akanouras. GitHub - bendudson/py4cl: Call python from Common Lisp — github.com. https://github.com/bendudson/py4cl. [Accessed 08-01-2024].
[10] Enerccio and Peter Vanusanik. GitHub - Enerccio/SLT: SLT is an IDE Plugin for Itellij/Jetbrains IDE lineup implementing support for Common Lisp via Slime/Swank and supported lisp interpret. — github.com. https://github.com/Enerccio/SLT/. [Accessed 11-01-2024].
[11] Github user - pinterface. GitHub - pinterface/burgled-batteries: A bridge between Python and Lisp (FFI bindings, etc.) — github.com. https://github.com/pinterface/burgled-batteries, . [Accessed 11-01-2024].
[12] Github user - pinterface. Is this project abandoned? - pinterface/burgled-batteries — github.com. https://github.com/pinterface/burgled-batteries/issues/15, . [Accessed 11-01-2024].
[13] Yukari Hafner. GitHub - shinmera/float-features: Portability library for ieee float features that are not covered by the cl standard. — github.com. https://github.com/Shinmera/float-features. [Accessed 17-02-2024].
[14] Yukari Hafner, Tarn W. Burton, S.M Mukarram Nainar, Marco Antoniotti, Karsten Poeck, Michael "phoe" Herda, and Paul M. Rodriguez. Common Lisp Portability Library Status — portability.cl. https://portability.cl/. [Accessed 11-01-2024].
[15] Rich Heller. Alive - the average lisp vscode environment. https://marketplace.visualstudio.com/items?itemName=rheller.alive. [Accessed 11-01-2024].
[16] Michał "phoe" Herda. *The Common Lisp Condition System: Beyond Exception Handling with Control Flow Mechanisms*. Apress, 2020. ISBN 9781484261347. doi: 10.1007/978-1-4842-6134-7. URL http://dx.doi.org/10.1007/978-1-4842-6134-7.
[17] Steven G. Johnson and et al. GitHub - JuliaPy/PyCall.jl: Package to call Python functions from the Julia language — github.com. https://github.com/JuliaPy/PyCall.jl. [Accessed 11-01-2024].
[18] Luis Oliveira and et al. Trivial Garbage — trivial-garbage.common-lisp.dev. https://trivial-garbage.common-lisp.dev/. [Accessed 14-01-2024].
[19] Paul Khuong. Finalizing foreign pointers just late enough - Paul Khuong mostly on Lisp — pvk.ca. https://pvk.ca/Blog/Lisp/finalizing_foreign_pointers_just_late_enough.html. [Accessed 17-04-2024].
[20] Kent Pitman. Common Lisp HyperSpec (TM) — clhs.lisp.se. http://clhs.lisp.se/, 1996. [Accessed 11-01-2024].
[21] Tobias C. Rittweiler and Gábor Melis. GitHub - melisgl/named-readtables — github.com. https://github.com/melisgl/named-readtables. [Accessed 17-02-2024].
[22] SLIME Contributors. GitHub - slime/slime: The Superior Lisp Interaction Mode for Emacs — github.com. https://github.com/slime/slime. [Accessed 11-01-2024].
[23] SLIME Contributors. Slime user manual - version 2.24. https://slime.common-lisp.dev/doc/slime.pdf, 2020. [Accessed 16-02-2024].

Qlot, a project-local library installer

Eitaro Fukamachi
e.fukamachi@mailfence.com
Tokyo, Japan

ABSTRACT

Although Quicklisp stands as the predominant Common Lisp library manager, the absence of library versions poses a significant challenge for developers managing applications. A common solution involves downloading libraries individually, yet this approach introduces another obstacle: ensuring consistent dependencies across various environments, particularly in collaborative development settings.

Qlot is a tool that installs libraries that are not registered or different versions of Quicklisp, making it easy to reproduce the same set of libraries in all environments.

The main purpose of this paper is to explain why Qlot is important, its usage, its internal design, and how it differs from other tools and methods.

CCS CONCEPTS

• **Software and its engineering** → *Software maintenance tools.*

KEYWORDS

Common Lisp, Quicklisp, dependency manager

ACM Reference Format:
Eitaro Fukamachi. 2024. Qlot, a project-local library installer. In *Proceedings of the 17th European Lisp Symposium (ELS'24).* ACM, New York, NY, USA, 5 pages. https://doi.org/10.5281/zenodo.10949389

1 INTRODUCTION

1.1 Background of Quicklisp age

Quicklisp[1] is a de facto library installer for Common Lisp and a central registry server for distributing Common Lisp libraries. However, it lacks a function considered essential in similar products in other languages – "versioning". For example, Python's PyPI[2] and Ruby's RubyGems[3] manage versions for each library, allowing the installation of a specific version. Unlike others, Quicklisp has versions of "dists" — a distribution set of registered projects — updated every few months. When changing the version of dist, all installed libraries can be changed.

It is not because of the negligence of Quicklisp's author. It is due to a historical background of Common Lisp: a lot of Common Lisp libraries were published before Quicklisp became available. There were many useful but inactive products when Quicklisp was made. If Quicklisp had abandoned them at its Genesis, it would not be used as it is today. Therefore, Quicklisp could not obligate authors of those libraries to follow certain rules of release processes and metadata for better management. Nonetheless, Quicklisp had to distribute them without any community consensus.

1.2 Problems

Quicklisp only has a date of dists as a version, like a snapshot. Such a situation leads to difficulties for application developers. Supposing a bug is found in a dependent library and the fixed version has been released at its upstream git repository. Until it is available in the next update of Quicklisp, it has to be manually downloaded to a place where ASDF[4] can find it, such as in `~/common-lisp`.

Although it is commonly used in many projects, it has two problems.

First, it is a user-wide configuration shared across multiple projects on the same machine. If one project requires upgrading a library, all projects in the machine are affected.

Second, ensuring that all environments running the product use the same set of dependencies is difficult. Very few projects run in a single environment. A project developed by multiple developers has to run on all collaborator's machines. Web applications have a production environment where the application will finally be deployed. In the case of distributed software, it must run on more than one machine. Same with CI environments.

There is a myth that the Common Lisp specifications remains standardized and functional even after several years have passed. However, this is only true if it does not rely on any external libraries. To verify this, one can easily confirm by trying to run modern code against an old Quicklisp distribution.

1.3 git submodules as a partial solution

Using `git submodules`[5] can be an option to manage dependent libraries. It adds another git repository to the project as a subdirectory. Given that many Common Lisp projects are hosted on GitHub and GitLab[1], this seems like a simpler way to do it than introducing a new tool.

However, dependencies must be managed manually, including dependencies of dependencies. If only direct dependencies are managed as submodules, the compatibility of indirect ones will not be guaranteed. If using Quicklisp, it is necessary to keep the version of its dist the same. If there are libraries with the same name but a different version in the ASDF load path, such as in `~/common-lisp`, it will unintentionally be loaded.

It is tiresome to list all dependencies, find out their provenances and add them as git submodules. Consider when updating a dependent library and its dependencies have been changed. It requires to do the same steps again.

ELS'24, May 6–7 2024, Vienna, Austria

https://doi.org/10.5281/zenodo.10949389

[1]Currently 4,632 projects are registered in the latest Quicklisp dist "2023-10-21", of which 4,083 (88%) are hosted on GitHub, and 220 (4.7%) are on GitLab (including gitlab.common-lisp.net).

`git submodules` seems simple at first glance, but there are many things to be concerned about as above.

1.4 Qlot as another solution

Qlot is a tool to solve the problems mentioned in section 1.2. Qlot manages assets installed by Quicklisp for each project, not for each user.

In addition, Qlot also exports information about installed libraries and their respective versions to a file. This feature simplifies the sharing of library configurations across different environments, allowing users to effortlessly rebuild the same set of libraries on any system with just a single command.

2 GETTING STARTED

2.1 Prerequisites

Qlot requires SBCL and OpenSSL. This does not mean that users' projects are limited to SBCL since Qlot is an external tool and is not necessary while running the main programs.

`curl` or `wget` are optional requirements, needed only to use Qlot's Automatic Installer. `git` is also needed only to install from git repositories.

```
# Linux
sudo apt install -y libssl-dev git
# macOS
brew install openssl git
```

2.2 Installation

The easiest way to install Qlot is to use the Automatic Installer. Here is a command to fetch and run it at once:

```
curl -L https://qlot.tech/installer | sh
```

It installs Qlot at `~/.qlot` and places a shell command to `~/.qlot/bin`. Be sure the directory path is in `PATH` of the running shell. It is not necessary if `XDG_BIN_HOME` is set, as it will be copied there.

```
qlot --help
Usage: qlot COMMAND [ARGS..]

COMMANDS:
    init     Initialize a project to start using Qlot.
    install  Install libraries to './.qlot'.
    update   Update specific libraries and rewrite
         their versions in 'qlfile.lock'.
    add      Add a new library to qlfile and trigger '
         qlot install'.
    remove   Remove specific projects from 'qlfile' and
          trigger 'qlot install'.
    check    Verify if dependencies are satisfied.
    exec     Invoke the following shell-command with
         the project local Quicklisp.
    bundle   Bundle project dependencies to './.bundle-
         libs'.

GLOBAL OPTIONS:
    --dir <directory>
        Directory to run the Qlot command
    --no-color
        Don't colorize the output

TOPLEVEL OPTIONS:
    --version
        Show the Qlot version
    --help
        Show help

Run 'qlot COMMAND --help' for more information on a
     subcommand.
```

The REPL interface has been added since v1.5.0, though it is still experimental. This paper explains based on the shell interface.

2.3 Start using Qlot

To start using Qlot, execute the following command:

```
qlot init
```

It creates `qlfile` and adds `.qlot` to `.gitignore` if it is a git repository, not to track the directory by git. `qlfile` is a file contains details of additional dependencies. Its initial content is empty.

It can take `--dist` option to add a custom Quicklisp dist, such as Ultralisp[6]:

```
qlot init --dist https://dist.ultralisp.org
```

Finally, run `qlot install` to set up a project-local Quicklisp.

```
qlot install
```

This command will install the latest Quicklisp dist in the `.qlot/` directory and write versions of libraries to `qlfile.lock`.

It is also used to synchronize up dependencies when different programmers updates `qlfile.lock`. See section 2.6 "Version lock and upgrade" for this topic.

2.4 Invoke REPL

To use the project-local Quicklisp, run `qlot exec` followed by a command to start Lisp.

```
qlot exec sbcl
CL-USER> ql:*quicklisp-home*
/path/to/project/.qlot/
```

The command following `qlot exec` has to be one of `sbcl`, `ecl`, `abcl`, `clasp`, `clisp`, `alisp`, or `ros`. If a different Lisp implementation is desired, it is mandatory to start the Lisp interpreter and load `.qlot/setup.lisp`.

For example, if Clasp was not supported by `qlot exec`. Only then Clasp can be started with Qlot by doing `clasp --load .qlot/setup.lisp`.

To run on Emacs/SLIME, add the `qlot exec` command in a list of `slime-lisp-implementations`:

```
(setq slime-lisp-implementations
    '((sbcl ("sbcl") :coding-system utf-8-unix)
      (qlot ("qlot" "exec" "sbcl") :coding-system utf
          -8-unix)))
```

In this way, it can be treated the same way as specifying a Lisp implementation. Use `M-- M-x slime RET qlot RET` to invoke a new SLIME buffer.

2.5 Dependency management

Qlot not only sets up the project-local Quicklisp that will not be affected by the global one but also enables the installation of libraries from git repositories and allows the specification of particular library versions.

To install a new library of a specific version, use `qlot add`:

```
qlot add mito
```

It adds the version of Mito included in the latest Quicklisp dist. This is not so useful if a newer version is available and not included in the latest dist yet. To install from git, you can use the `--upstream` option. It identifies the upstream repository URL of the library from Quickdocs API[7] and downloads the latest version. This requires `git` to be installed.

```
qlot add mito --upstream
```

Qlot also accepts the format `<username>/<repository>` to install from GitHub, which is useful to use a forked one.

```
qlot add fukamachi/mito
qlot add fukamachi/mito --branch next
```

To install from a git repository hosted other than GitHub is a bit verbose, but can be done as follows:

```
qlot add git iterate https://gitlab.common-lisp.net/
     iterate/iterate
qlot add git iterate https://gitlab.common-lisp.net/
     iterate/iterate \
  --branch release
```

To delete a dependency, use `qlot remove`:

```
qlot remove mito
```

2.6 Version lock and upgrade

The version information at the time `qlot install` runs is recorded in `qlfile.lock` placed in the project's root directory.

For example, the contents of `qlfile.lock` look like this:

```
("quicklisp" .
 (:class qlot/source/dist:source-dist
  :initargs (:distribution "https://beta.quicklisp..."
       :%version :latest)
  :version "2023-10-21"))
("mito" .
 (:class qlot/source/ql:source-ql-upstream
  :initargs nil
  :version "ql-upstream-53250af300b18c..."
  :remote-url "https://github.com/fukamachi/mito.git"))
```

`qlfile` and `qlfile.lock` are a pair of files that are needed to reproduce the environment. When these files are changed, `qlot install` applies the changes to `.qlot/`. When a different collaborator edits those files, do not forget to synchronize each machine's environment with this command. Even when there are newer versions at that time, Qlot always installs the same versions.

To upgrade to the latest one, run `qlot update` like:

```
qlot update mito
```

This command updates `qlfile.lock` if upgraded.

2.7 Loading local forks

In cases where developers want to use a locally cloned project modified slightly, Quicklisp's `local-projects` mechanism is handy. In a project-local Quicklisp, libraries at `.qlot/local-projects`, including symbolic links, can also be loaded.

```
ln -s ~/projects/mito .qlot/local-projects
```

2.8 Minimize footprint

In some projects, Quicklisp client should not be included in an execution process for reasons such as saving memory. `qlot bundle` outputs only the source code of the dependent libraries under the `.bundle-libs/` directory.

```
qlot bundle
```

Load `.bundle-libs/bundle.lisp` instead of `.qlot/setup.lisp` to load them without using Quicklisp and Qlot.

```
sbcl --load .bundle-libs/bundle.lisp
```

See also the documentation of Quicklisp about `ql:bundle-systems` [8] for the details since this feature is built on top of it.

3 A BIT OF INTERNAL DETAILS

3.1 Libraries as Quicklisp dists

Installation of dependent libraries is the core feature of Qlot. Quicklisp has a mechanism called "local-projects" that allows to load libraries that are not registered in the dist. When a library is placed under the local-projects directory in Quicklisp's home, it is prioritized over libraries registered in Quicklisp.[9]

However, Qlot does not use this feature; instead, Quicklisp uses "dist" to manage external libraries. Quicklisp has a feature to install unofficial dists, such as Ultralisp, which provides a different set of libraries. Qlot is "abusing" it.

```
CL-USER> (ql-dist:all-dists)
(#<QL-DIST:DIST mito git-53250af300b18cfe956cbe26...>
 #<QL-DIST:DIST quicklisp 2023-10-21>)
```

The advantage of this design is that functions for inspecting library information in Quicklisp, such as `ql:provided-systems` and `ql-dist:required-systems`, are accessible even within the project-local Quicklisp.

```
CL-USER> (ql:provided-systems
          (ql-dist:find-release "mito"))
(#<QL-DIST:SYSTEM lack-middleware-mito / mito-ref...>
 #<QL-DIST:SYSTEM mito-core / mito-ref-53250af300...>
 #<QL-DIST:SYSTEM mito-migration / mito-ref-53250...>
 #<QL-DIST:SYSTEM mito-test / mito-ref-53250af300...>
 #<QL-DIST:SYSTEM mito / mito-ref-53250af300b18cf...>)

CL-USER> (ql-dist:required-systems
          (ql-dist:find-system "mito"))
("cl-reexport" "lack-middleware-mito"
 "mito-core" "mito-migration")
```

Other notable examples that check dependencies are `ql:who-depends-on` and `ql-dist:dependency-tree`. Similar functions are provided by ASDF, but Quicklisp ones are much faster because they calculate dependencies based on pre-generated metadata.

3.2 HTTPS support

Quicklisp client is written in pure Common Lisp without relying on any external libraries to avoid affecting the user's application. Because of this policy, HTTPS support has been left due to the difficulty of implementing the HTTPS protocol solely in Common Lisp, though the server side of Quicklisp allows HTTPS access.

Thanks to extensibility of Quicklisp, Qlot adds it without modifying the code by using `local-init` and `ql-http:*fetch-scheme-functions*`.

`ql-http:*fetch-scheme-functions*` is an association list holding pairs of supported HTTP schemes and functions to fetch. The default list has only one for HTTP. Qlot adds a pair for HTTPS in `local-init` that loads all Lisp files in `local-init/` directory in the Quicklisp home.

```
CL-USER> ql-http:*fetch-scheme-functions*
(("https" . QLOT/LOCAL-INIT/HTTPS::RUN-FETCH)
 ("http" . QLOT/LOCAL-INIT/HTTPS::RUN-FETCH)
 ("http" . QL-HTTP:HTTP-FETCH))
```

Since this function downloads files in a different process, it does not affect the user's application in the main process, even though it uses external libraries.

4 TECHNICAL CHALLENGES

4.1 How to determine dependencies

As mentioned in section 3.1, Qlot sets up each library as a Quicklisp's dist and requires finding their dependencies. Simply thinking, it seems that using ASDF's `asdf:component-sideway-dependencies` would suffice. However, using ASDF's inspection functionalities proves difficulties due to its requirement for loading system definitions in the running Lisp image. This is because ASDF is an in-image build tool, unlike equivalents for other languages.[10]

For example, ASDF offers an option `:defsystem-depends-on`, enabling the specification of dependencies for the system definition itself. It is necessary to load the dependencies before reading the system definition, but the problem is where to load them. Although it should be installed from a project-local Quicklisp in the philosophy of Qlot, it does not exist yet.

Thus, it requires identifying dependencies by code-walking their ASDF system definitions. In most cases, it can be resolved simply by combining `:depends-on` and `:defsystem-depends-on`. However, in some cases, it may present challenges.

First, ASDF can include any Lisp code in ASD files. Some applications take advantage of this permissiveness of ASDF and dynamically load external libraries in their ASD files. It leads to the same problem `:defsystem-depends-on` has.

Second, ASDF's package-inferred-system is an ASDF extension that supports one-package-per-file style.[11] If `:class :package-inferred-system` is specified in the main system definition, all Lisp files in the project can be a sub-system with `defpackage` at the top of files. Qlot attempts to read all Lisp files that are possibly sub-system definitions, but it has the same problem as the ASD file since it can also contain any Lisp code.

Third, Qlot can not handle ASDF extensions other than package-inferred-system. ASDF allows to make a custom system class that inherits `asdf:system` by specifying `:class` to `defsystem`. Qlot ignores it because Qlot can not load any libraries while looking up dependencies due to the abovementioned problem. If the ASDF extension manipulates dependent libraries or inherits `package-inferred-system`, it will not work as expected.

4.2 REPL interface

Working with a REPL is one of the best parts of Quicklisp. Users can install libraries without leaving the REPL. On the other hand, Qlot provides command-line interface as the primary one, and its REPL interface is still experimental.

This is due to a design difference between Qlot and Quicklisp. Quicklisp runs in the main Lisp process, as does ASDF. While this has advantages in terms of use, seamless interaction, and speed, it also pollutes the user's process, as discussed in 3.2.

Qlot uses Dexador[12], an HTTP client, to communicate over HTTPS. If Qlot is run in the main process like Quicklisp, there is a possibility that its version conflicts with a user application's dependencies. Conversely, Qlot runs mainly in a separate process.

Of course, this model has its drawbacks. The most significant one is that the Common Lisp debugger cannot be used on errors, such as handling by condition class, and restarts.

Presently, the primary focus of Qlot development is on exchanging information between processes about errors to show useful messages and to provide restarts as its interface. This will make the usability in the REPL closer to Quicklisp.

5 COMPARED TO OTHER TOOLS

5.1 Quick-Patch

Quick-Patch[13] is a simple tool intended to replace git submodules. It is similar in the way that it complements Quicklisp but does not separate environments for each project like Qlot. Hence, it presents a similar issue to that git submodules.

5.2 CLPM

CLPM[14] is a similar tool developed after Qlot was out, but it has a different design philosophy and has a "develop-from-scratch" approach.

It was originally developed to replace the distribution service of Quicklisp with CLPI (Common Lisp Project Index)[15], and CLPM is the name of its client. It can keep multiple ASDF settings and switch them for each environment. In that sense, it is similar to Python's virtualenv[16].

In the past, CLPM had unique features, such as loading local libraries and downloading via HTTPS, but these are now implemented in Qlot. CLPM also imported a project-local version-locking mechanism from Qlot and suspended the original goal which was to substitute Quicklisp dist. Now, they are almost the same in functionality.

The only differences are usability and learning cost, which cannot be ignored in team development. Qlot enhances them by adopting a well-known Quicklisp client instead of a dedicated one.

5.3 ocicl

ocicl[17] is a tool to replace the Quicklisp registry in the same spirit as CLPM and CLPI. It packages each project in Open Container Initiative (OCI)[18] manner. ocicl already has its registry system built on GitHub Container Registry that instantly updates on changes of projects, whereas CLPI is still under development.

However, from a client perspective, it doesn't offer as many features. Similar to Quick-Patch, it downloads and only places libraries in a certain directory and does not have a mechanism to isolate the project from others. There is no capability to install libraries from VCS, and certainly no functionality to specify branches.

6 CONCLUSION

We have seen the problems that Qlot is trying to solve, its design, its usage, and how it differs from other tools.

Qlot has undergone significant refinement in response to user feedback, addressing various issues and enhancing usability. Initially requiring Roswell[19], it has since switched to work directly on SBCL, streamlining installation steps. Additionally, parallel execution has been introduced to greatly reduce runtime. The decision-making process for adopting Qlot involved the entire team, prioritizing user experience and considering even minor inconveniences.

The basic REPL interface has been incorporated in version 1.5.0, and users are encouraged to try it out and provide feedback.

REFERENCES

[1] Zach Beane. Quicklisp. https://www.quicklisp.org
[2] PyPI - The Python Package Index. https://pypi.org
[3] RubyGems. https://rubygems.org
[4] ASDF - Another System Definition Facility. https://asdf.common-lisp.dev
[5] Git Tools - Submodules. https://git-scm.com/book/en/v2/Git-Tools-Submodules
[6] Alexander Artemenko. Ultralisp - A fast-moving Common Lisp software distribution. https://ultralisp.org
[7] Quickdocs. https://quickdocs.org
[8] Zach Beane. Quicklisp library bundles. https://www.quicklisp.org/beta/bundles.html
[9] Zach Beane. The Quicklisp local-projects mechanism. http://blog.quicklisp.org/2018/01/the-quicklisp-local-projects-mechanism.html
[10] François-René Rideau and Robert P. Goldman. Evolving ASDF: More Cooperation, Less Coordination. ILC '10: Proceedings of the 2010 international conference on Lisp. Pages 29–42. https://doi.org/10.1145/1869643.1869648
[11] ASDF - The package-inferred-system extension. https://asdf.common-lisp.dev/asdf/The-package_002dinferred_002dsystem-extension.html
[12] Dexador - A fast HTTP client for Common Lisp. https://github.com/fukamachi/dexador
[13] Arnold Noronha. Quick-Patch. https://github.com/tdrhq/quick-patch
[14] 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.471646
[15] Eric Timmons. Common Lisp Project Index. https://gitlab.common-lisp.net/clpm/clpi
[16] virtualenv - Virtual Python Environment builder. https://virtualenv.pypa.io
[17] Anthony Green. An OCI-based ASDF system distribution and management tool for Common Lisp. https://github.com/ocicl/ocicl
[18] OCI - Open Container Initiative. https://opencontainers.org
[19] Roswell - Common Lisp environment setup Utility. https://github.com/roswell/roswell

Murmel & JMurmel

Robert Mayer
Research Industrial Systems Engineering (RISE)
Vienna, Austria

Thomas Östreicher
Research Industrial Systems Engineering (RISE)
Vienna, Austria

ABSTRACT

In this paper we will introduce Murmel, a Lisp dialect based on a subset of Common Lisp, and JMurmel, a Murmel implementation. JMurmel can be used as a standalone command line program as well as embedded in a Java application or Webpage.

CCS CONCEPTS

• **Software and its engineering → General programming languages**.

KEYWORDS

Lisp interpreter, Lisp compiler, JVM, embeddable

ACM Reference Format:
Robert Mayer and Thomas Östreicher. 2024. Murmel & JMurmel. In *Proceedings of the 17th European Lisp Symposium (ELS'24).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.5281/zenodo.10997870

1 INTRODUCTION

JMurmel [3] started in part out of boredom during the Corona pandemic, and in part out of a desire to discover the "essence of Lisp", i.e. what is minimally needed to write Lisp programs. Implementation of JMurmel started 2020-ish as a 200 line program that was able to run simple Lisp forms. Now in 2024 JMurmel has grown to approximately 13000 lines of Java and 3200 lines of Murmel (and an additional 3000+ lines of tests written in Murmel that mostly run on CL-implementations as well).

While JMurmel at this time is a hobby project and not an industrial strength Lisp it still can be used for various things, not the least for experiments. If there is a need for an in-application scripting language, then JMurmel may be a better choice than rolling your own.

The name "Murmel" is a pun based on the German words "Murmel/murmeln" – "murmeln" translated to English is "mumble". Also "Murmel" is the name of a small but pretty badass animal that lives high up in the Alps.

2 JMURMEL FEATURES

JMurmel features an interpreter and a compiler, a REPL with a trace facility (`trace` and `untrace` function calls), full tail call support (JMurmel's interpreter processes all tailcalls with a loop inside `eval`, JMurmel's compiler transforms tailrecursive self-calls into loops, other tailcalls are invoked through a trampoline), lexical closures, multiple return values, conditions, a macro facility, backquote expansion including nested backquotes, JSR223 support, support for JFR (Java Flight Recorder)/ JMC (Java Mission Control), turtle- and bitmap-graphics and garbage collection c/o JVM.

JMurmel implements Murmel's datatypes (including conditions) mostly one-to-one as Java datatypes, enabling easy embedding. JMurmel itself is written in very simple Java, and can also be run inside a Webbrowser using CheerPJ [1]; see e.g. the "Murmel Online Repl" [4].

For experimentation purposes JMurmel provides command line flags[1] to disable certain language features such as lexical closures, Java-FFI, some special forms, and so on. E.g. when invoked with

```
java -jar jmurmel.jar --XX-dyn --min+ --XX-oldlambda
```

then JMurmel behaves quite similar to the original Lisp from 1958.

3 COMPARISON MURMEL VS. COMMON LISP

Murmel is inspired by Common Lisp. That said, Murmel is somewhat close to but not really a "subset of Common Lisp as specified by ANSI [5]".

Major differences include:

- Murmel is a Lisp-1
- special variables work differently: Murmel doesn't have dynamic (special) variables but dynamic bindings via `(let dynamic (. . . .` CL's `defparameter` or `defvar` create special variables, re-binding them will be dynamic; Murmel's `define` creates global variables that by default will be shadowed by local variables or parameters unless rebound using `(let dynamic (. . .`
- vararg lambda lists are specified as a dotted list (CL has `&rest` and `&body`)
- the reader macro `#!` is used in addition to `#|` for multi-line comments, making hash-bang shellscript work right out of the box
- Murmel does not have a package system (yet), i.e. there are no `defpackage` or `in-package` forms
- the default library is not automatically available, i.e. programs need to `(require "mlib")` to use library functions. A "core" language is provided by the executable Jarfile, a "default library" is provided and can be require'd, or replaced by another library.
- Murmel supports fewer functions out of the box compared to Common Lisp: running the REPL command `:env` in JMurmel will list 182 symbols, and after loading the default library `:env` will list a total of 293 symbols as well as 67 macros which can be listed using the REPL command `:macros`
- Murmel does not support CLOS
- floating point by default is `double-float`

[1] These commandline flags currently only apply to the interpreter.

ELS'24, May 6–7 2024, Vienna, Austria

https://doi.org/10.5281/zenodo.10997870

- math functions such as `+` return a `double-float` regardless of their arguments
- the numeric tower is tiny: no `bignum`, `ratio`, `complex`, Murmel only has `fixnum` and `double-float`
- REPL variables are prefixed by `@`, e.g. `@*`
- `format`'s syntax currently is different, e.g. Murmel: `%s` vs. CL: `~A`
- `load` and `require` are performed at compile time
- some functions have different parameters because Murmel doesn't have keyword arguments

Extensions:

- `letrec`: similar to Scheme's `letrec`
- named `let, let*, letrec`: similar to Scheme's loop construct, e.g.

```
(let loop ((n 0))
  (print n)
  (when (< n 3) (loop (1+ n))))
```

- hash-tables: Murmel's `make-hash-table` accepts additional values for the parameter `key`. Common Lisp supports `eq, eql, equal` and `equalp`, Murmel adds `t, compare-eql` and `compare-equal`. `t` indicates that keys are to be compared using their natural equality[2], `compare-eql` and `compare-equal` indicate that hash table iteration will be ordered by key. Murmel's surface representation also includes hashtable literals using the `#H(. . .` reader macro.
- Murmel's default library contains some support for lazy sequences using the concept of "generators" which are somewhat similar to Java's iterators or Scheme's SRFI-158.

See the file "Murmel-vs-CL.md" in the Github repo [2] for more details.

4 COMPARISON WITH OTHER LISPS

This section highlights some selected differences between Murmel/ JMurmel and other Lisp dialects/ Common Lisp implementations.

The subsections "Murmel vs. Clojure" and "Murmel vs. Scheme" address language differences between Murmel and Clojure and Scheme respectively, while the subsections "JMurmel vs. ABCL", "JMurmel vs ECL" and "JMurmel vs. SBCL" address implementation differences.

4.1 Murmel vs. Clojure

Clojure is an opinionated Lisp with some syntactic differences compared to more traditional Lisps such as using `[` vs. `(` in some places, different names for special forms, immutable data types and more, while Murmel tries to stay closer to (a subset of) Common Lisp. Clojure's syntactic changes may make writing code more convenient while Murmel's more traditional syntax may make writing code walkers more straightforward.

Unlike Clojure's JVM integration, Murmel's optional[3] JVM integration is not at the syntactic level, instead there are the two (optional) functions `jmethod` and `jproxy` to invoke Java code from Murmel and vice versa. Clojure's approach makes using Java libraries easier; all classes on the JVM's classpath can be used from Clojure code using Clojure syntax; in Murmel one would probable write wrapper functions or macros using jmethod and/ or jproxy (which may turn out to be too cumbersome for large scale use).

Common Lisp code samples from books or from the internet often are valid Murmel as is or can be ported to Murmel with only few changes; porting Common Lisp to Clojure might take more effort.

4.2 Murmel vs. Scheme

While Murmel and Scheme share some similarities such as the shared namespace for variables and functions there are differences as well:

Several special forms and primitive functions differ in their names, e.g. `set-cdr!` (Scheme) vs. `rplacd` (Murmel).

Scheme has continuations, Murmel does not. Murmel has non-local returns using `catch/ throw`, Scheme does not support these special forms out of the box, however non-local returns (and more) can be implemented using Scheme's more powerful continuations.

Murmel has Common-Lisp-style multiple return values, Scheme only added a limited form of multiple return values in more recent revisions, and Scheme implementations have subtle differences in their support for multiple return values, e.g. whether or not `map` accepts a function that returns more than a single value.

Murmel has Common-Lisp-style macros, Scheme has hygienic macros (and some Scheme implementations have non-hygienic macros as well).

4.3 JMurmel vs. ABCL

ABCL is a conforming Common Lisp implementation, JMurmel is not.

Running a "Hello, World!" program from the command line takes approximately 300ms with JMurmel (or approx. 400ms for loading Murmel's default library followed by "Hello, World!") while ABCL can take a second or two.

ABCL comes as an approx. 10 MB Jar-file while Murmel currently is 1/20th the size at approx. 560 kB (450 kB Jarfile + 113 kB library in source form). A significant part of the size difference probably is due to ABCL's richer standard library and the different compilation strategy.

JMurmel's smaller feature set that can easily reduced even more may be an advantage for embedding in another application, e.g. in an application for users with less programming skills where less features means less training required, or in a multi-user server application where each user gets their own sandboxed Lisp.

ABCL directly compiles Lisp to JVM bytecode, JMurmel transpiles Lisp to Java and then uses the JVM's builtin Java compiler to generate classfiles.

[2] Actually this is the platform leaking through: "natural equality" really is Java's `equal` method

[3] `jmethod` and `jproxy` are not a required part of Murmel. The Murmel language reference lists these two function in a separate chapter as "JMurmel specific extensions". JMurmel allows to disable `jmethod` and `jproxy` via a commandline switch when used as a standalone application, and programmatically when used as a library embedded in a Java program.

ABCL uses it's own datatypes, e.g. 1 would be internally represented as an object of class `org.armedbear.lisp.LispInteger` while JMurmel uses standard Java datatypes where possible, e.g. 1 is represented as an object of class `java.lang.Long`.

4.4 JMurmel vs. ECL

ECL is a conforming Common Lisp implementation, Murmel is not.

Execution speed of compiled ECL and JMurmel programs are roughly comparable.

ECL can produce standalone executables by transpiling Lisp to C and then compiling and linking the final program using the platform tools. JMurmel as well as Jarfiles created by JMurmel need a JVM runtime, using an ahead-of-time compiler such as GraalVM-native as an extra step it would be possible to create single-file-executables with JMurmel, though.

4.5 JMurmel vs. SBCL

SBCL is a conforming Common Lisp implementation, JMurmel is not.

SBCL runs unoptimized Common Lisp code 5 times (2-10x) faster than compiled JMurmel (after the usual JVM warmup). With SBCL most simple actions such as compiling and running a "fizzbuzz" program are instantenuous, while doing the same with JMurmel produces a noticeable delay. However, extremely GC-heavy programs such as the GC benchmark "gc-latency-experiment" will put SBCL at a disadvantage as JMurmel uses the JVM's set of highly optimized garbage collectors (the same would be true for ABCL, though).

5 FIELDS OF APPLICATION

Main fields of application of JMurmel probably will be where an industrial strength Lisp such as ABCL, ECL or SBCL is not supported (e.g. in a browser) or the full feature set of a Common Lisp is not required and may even be a disadvantage, e.g. when used as a scripting language embedded in another application JMurmel's configurability makes it possible to tailor the feature set to the actual needs of the application.

With the caveat that Murmel/ JMurmel are a hobby project and work in progress possible fields of application of JMurmel include:

- experimenting, teaching: JMurmel's feature set can be stripped down to pure Lambda calculus so even cons-cells/ car/ cdr can be re-implemented in Lambda calculus using JMurmel. JMurmel's support for turtle graphics may make examples more fun.
- Web-applications: using CheerpJ an Online REPL that doesn't need a supporting backend service was implemented with approx. 20 lines of Javascript code. Calling Javascript from Murmel and thereby changing the DOM was not tested yet but should be possible as well.
- Online-publications: JMurmel+CheerpJ [1] could be used to include runnable (and editable) Lisp samples in a Webpage
- embedded scripting languages for Java applications; if desired then sandboxing a JMurmel runtime should be fairly straightforward as e.g. JFFI can be disabled using a feature flag passed when creating an instance of JMurmel. Callbacks from Murmel into the host application still are possible even with JFFI disabled by inserting selected host functions in JMurmel's environment allowing for a high level of control of what a script can or cannot do.
- simple "hashbang-scripts"
- standalone applications: JMurmel's JFFI provides access to Java's large ecosystem of libraries

6 BENCHMARKS

The JMurmel github repo [2] contains a benchmark which consists of 15 programs mostly from [6] and is written in mostly unoptimized Common Lisp. Unfortunately this precludes comparing JMurmel with Clojure or Scheme implementations as porting the programs to Clojure or Scheme was not done. Instead JMurmel was compared to Common Lisp implementations with different runtime strategies: ABCL uses the JVM as does JMurmel, ECL uses a C-compiler as a backend, SBCL has a native compiler.

This benchmark can be run by loading the toplevel source file "samples.murmel-mlib/benchmark/all.lisp". Running the program produces the following results for JMurmel 1.4.6, SBCL 2.3.4, ABCL 1.9.1 and ECL (relative to SBCL 2.3.4):

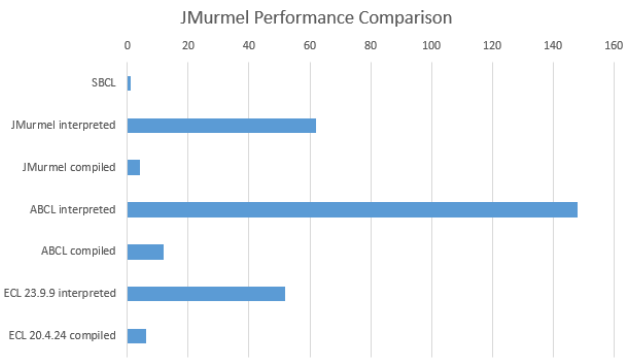

Figure 1: Performance Comparison

SBCL	1
JMurmel interpreted	62
JMurmel compiled	4.2
ABCL interpreted	148
ABCL compiled	11.8
ECL 23.9.9 interpreted	52
ECL 20.4.24 compiled	6

The numbers above should be interpreted as x times slower than SBCL.

The benchmarks were run on a Windows 10 Laptop with 32 GB RAM, an i5-1135G7@2.40GHz, and Turbo mode was disabled. The JVM was Temurin build 1.8.0_372-b07, JVM options were

`-XX:+UseParallelGC -Xmx1G`.

ECL was run under Windows/WSL/Debian-11.8, the other Lisp systems were run directly under Windows.

6.1 Interpretation of the results

In interpreter mode JMurmel is comparable to ABCL, in compiler mode JMurmel is comparable to ECL, at least according to JMurmel's

benchmark suite that is based on the benchmarks in "Performance and Evaluation of Lisp Systems" [6].

SBCL is significantly faster than the other implementations including JMurmel. It is estimated that SBCL's type derivation plays a significant role in this.

The final numbers should not be taken too seriously and at best give a coarse overview. JMurmel being a lot less mature that the competitors may have bugs such missing checks hat may give JMurmel an unfair advantage. Things like Garbage Collector settings, JDK versions etc. can have a rather big impact. Also the code for the benchmark harness is only a simple loop using `time` for measurement.

7 LESSONS LEARNED

A "hosted Lisp" as opposed to "implementing a Lisp in Lisp" has both advantages and disadvantages.

Especially on the JVM one gets a lot of things "for free", most notably a world class garbage collector, but also things like lexical closures are trivial to implement using the host's `lambda` language feature. The JVM provides memory safety to a very high level, and a Lisp running on the JVM will have good memory safety with little or no additional effort. Last but not least porting to other platforms becomes a non-issue, JMurmel runs on all platforms that support one of the Java 8 through 22.

Disadvantages include (usually) no control over the development of the host platform, limitations such as the need for tailcall-trampolines if the host does not support native tailcalls.

The Common Lisp specification leaves a considerable amount of freedom to implementers, and the authors of the CL specification seemed to have considerable foresight e.g. in specifying the Condition system in a way that CLOS is not a requirement but conditions can be implemented using platform facilities without CLOS.

Implementing a "Library" separate from the "core language" makes it easier to end up with a well-defined language that doesn't expose any implementation internals as is sometimes the case with Lisps that are implemented in Lisp, especially if there is no package system that allows to hide internals.

8 NEXT STEPS

Continuing development of Murmel/ JMurmel will likely address the following features:

8.1 Language:

- `format`
- `macrolet` (already implemented in not-yet-released JMurmel 1.4.7)
- support for `setf-functions`
- `eval-when`
- `defstruct`
- a package system
- probably case sensitive symbols
- a better numeric tower, at least bignums/ BigIntegers

8.2 Runtime:

- a debugger/ stepper
- support for larger programs – currently JMurmel translates Lisp into one named Java class + lots of anonymous Java classes. Global symbols and literals are currently emitted as members of this single named class, and the JVM limits the number of entries a Java class can have.
- support for using the JVM's Thread support, maybe add `(cas place)` functions similar to what SBCL supports
- `(declaim (optimize (inline. . .`
- some type support at compile time
- maybe a new "sea of nodes" based optimizing compiler
- maybe an implementation of Murmel with larger parts written in Murmel and a smaller runtime in e.g. C (or C#, or Go), or maybe an implementation of Murmel on top of SBCL.

ACKNOWLEDGMENTS

Thanks to Thomas Östreicher for help in preparing this paper and many fruitful discussions.

REFERENCES

[1] Cheerpj. URL https://labs.leaningtech.com/cheerpj3.
[2] Jmurmel github repo. URL https://github.com/mayerrobert/jmurmel.
[3] Jmurmel landing page. URL https://jmurmel.github.io.
[4] Murmel online repl. URL https://jmurmel.github.io/repl.
[5] Kent Pitman et al. Common lisp hyperspec, 1996. URL http://www.ai.mit.edu/projects/iiip/doc/CommonLISP/HyperSpec/FrontMatter/.
[6] Richard P. Gabriel. *Performance and evaluation of LISP systems.* Massachusetts Institute of Technology, USA, 1985. ISBN 0262070936.