Proceedings of the

18th European Lisp Symposium

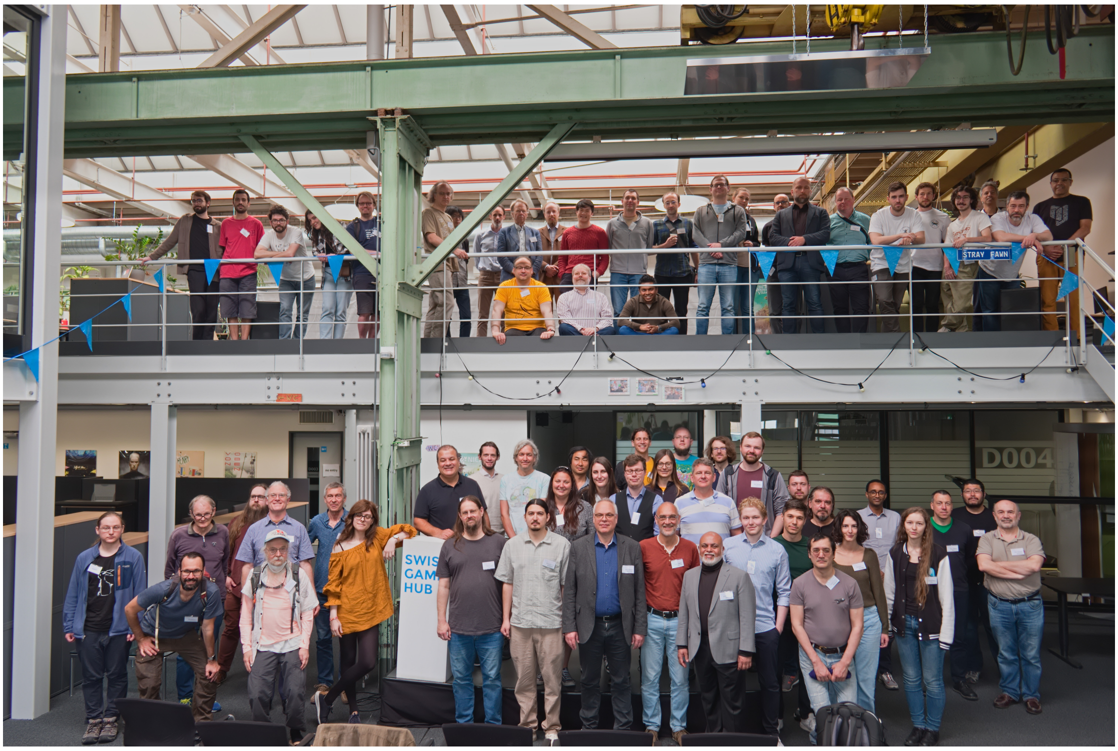
Zürich, Switzerland,
May 19 — 20, 2025

SWISS GAME HUB

SISCOG

ISBN-13: 978-2-9557474-9-0
ISSN: 2677-3465

Preface



Message from the Program Chair

It was an honor to be programme chair for ELS 2025, and I am deeply grateful to Yukari Hafner for organizing a great event, to all the authors for submitting interesting papers, and to all the attendees who make for a welcoming and enriching community. But I especially thank Didier for picking up the pieces where I failed to be up to the task ——- making this event a success despite me. The spirit of Lisp is alive and the tradition is perpetuated thanks to many wonderful people, and it is a blessing to have been among you all. May you keep channeling the lambda nature and illuminating the world.

Message from the Local Chair

I'm very happy to have finally had the chance to bring ELS to Zürich, and to have had the opportunity to show off the Swiss Game Hub to an international audience alongside. A big thanks to everyone at the Hub who assisted in making the conference happen, and of course to all the attendees for visiting.

Organization

Symposium Organizer

- Didier Verna, EPITA Research Laboratory, France

Programme Chair

- François-René Rideau, MuKn, USA

Local Chair

- Yukari Hafner, Shirakumo.org, Switzerland

Virtualization Team

Georgiy Tugai
Yukari Hafner

Programme Committee

Conrad Barski	USA
Marc Battyani	Enfabrica, USA
Dave Cooper	Genworks, USA
Ryan Culpepper	University of Massachusetts Boston, USA
Eitaro Fukamachi	Japan
Robert Goldman	SIFT, USA
Gavin Gray	Brown University, USA
Jason Hemann	Seton Hall University, USA
Kristopher Micinski	Syracuse University, USA
Marco Morazan	Seton Hall University, USA
Michael Raskin	LaBRI, France
Masatoshi Sano	Nayuta, Japan
Dimitris Vyzovitis	Mighty Gerbils

Sponsors

We gratefully acknowledge the support given to the 18thth European Lisp Symposium by the following sponsors:

The Swiss Game Hub
Erika-Mann-Strasse 11,
8050 Zürich
Switzerland
`www.swissgamehub.com`

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

Monday 19 May 2024

08:30–09:45		**Registration, Badges, Meet and Greet**
09:45–10:00		**Welcome Messages and Announcements**
10:00–11:15	Keynote	Project Oberon: a Late Appraisal Jürg Gutknecht
11:15–11:45		**Coffee break**
11:45–12:30	Experience Report	Growing Your Own Lispers Michal Herda, Wojciech Gac
12:30–14:00		**Lunch**
14:00–15:15	Keynote	Toward Safe, Flexible, and Efficient Software in Common Lisp Robert Smith
15:15–15:45		**Coffee break**
15:45–16:30	Research Paper	The Lisp in the Cellar Pierre-Evariste Dagand, Frédéric Peschanski
16:30–17:15	Research Paper	Programming with Useful Quantifiers Jim Newton
17:15–17:30		**Short break**
17:30–18:00		**Lightning Talks**

Tuesday 20 May 2025

08:30–09:30		**Registration, Badges, Meet and Greet**
09:30–09:45		**Announcements**
09:45–10:45	Keynote	Is Lisp Still Relevant in the Age of AI? Anurag Mendhekar
10:45–11:15		**Coffee break**
11:15–12:00	Research Paper	A Brief Perspective on Deep Learning Using Common Lisp Martin Atzmueller
12:00–13:30		**Lunch**
13:30–14:15	Experience Report	Porting the Steel Bank Common Lisp Compiler and Runtime to the Nintendo Switch Charles Zhang, Yukari Hafner
14:15–15:15	Round Table	Lisp and Artificial Intelligence Anurag Mendhekar, Martin Atzmueller, Vsevolod Domkin Gábor Melis, Dave Cooper
15:15–15:45		**Coffee break**
15:45–16:45		**Lightning Talks**
16:45–18:00		**Hackathon**

Contents

Invited Contributions

Project Oberon: A Late Appraisal

After ETH's success with Pascal in the 70's, programming languages like Modula-2, Oberon, and Lola in the 80s and 90s served the purpose of codesigning, pioneering personal workstations such as Lilith and Ceres and were also used in teaching generations of students.

Toward Safe, Flexible, and Efficient Software in Common Lisp

Common Lisp is renowned for its ability to express safe, flexible, or efficient code. However, these characteristics are often at odds with one another, especially in practical software co-development settings. Coalton is an embedded language within Common Lisp that leverages a Haskell-like type system to prove type safety of a program and performs a variety of type-based optimizations. Coalton also permits new abstractions that are difficult to express in ordinary Common Lisp. We discuss Coalton and its use at two commercial organizations.

Is Lisp Still Relevant in the New Age of AI

Lisp owes its existence and popularity to early AI research. At one time, the entire AI world revolved around Lisp, which provided an enormous amount of energy for the language's development and for pioneering technologies in compiler design, language innovation, and high-performance hardware (such as the Connection Machine). However, in today's AI landscape, Lisp is nowhere to be found. Instead, languages like Python—many of whose ideas are borrowed from Lisp—have become the mainstream tools for modern AI.

This raises a key question: What made Lisp so relevant during the first AI revolution but seemingly irrelevant in the second? Is there still a place for Lisp in this new AI era? If so, what should the Lisp community focus on to re-enable its relevance?

Monday, 19 May 2025

Growing Your Own Lispers — An Experience Report

Michał "phoe" Herda
mhr@keepit.com
Keepit

Wojciech S. Gac
wga@keepit.com
Keepit

Abstract

Choosing Lisp as a primary programming language is a bold step for a company to take. Basing one's success on a non-mainstream language with relatively low profile, even among CS students and graduates, can seem like skating on thin ice. Lispers themselves tend to flock around such companies — which, given their relative scarcity, is a fairly reliable mechanism for getting Lisp programmers in touch with Lisp employers. We found that the intersection of company culture, specificity of work, and compliance requirements creates a pressure to hire Lispers locally, but the local talent pool seems to have been exhausted.

This report tries to encapsulate the structure, execution, and general mood of a Common Lisp internship program launched at our company in late 2024. Our preliminary conclusions at the end of the program? Filter candidates for a mixture of technical acumen and good communication skills, provide plenty of programming tasks from the very beginning, be prepared to underestimate the speed at which interns solve problems, improvise locally, but plan globally.

CCS Concepts

• **Social and professional topics** → **Computing education**; • **Human-centered computing** → *Social content sharing*; • **Software and its engineering** → *General programming languages.*

1 About Keepit

Keepit is a cloud backup company. We started operation in 2012. We target multiple cloud platforms, among others, Microsoft 365, Google Workspace, Zendesk, and Salesforce. Over the years, the company has evolved greatly to accommodate the growing diversity of supported cloud solutions and the increasing sophistication of our clients' needs. Needless to say, a large part of this growth has been the restructuring and rethinking of our own approach to business problems.

Our basic cloud backup offering involves collecting daily snapshots from thousands of end users — our customers' employees — frequently referred to as *seats*; multiplied by hundreds of companies choosing our services, this combines to produce a huge amount of data. Add to this the ability to quickly browse and search across years of incremental backups, seamlessly access historical versions of files, archives, emails, etc. and you might begin to appreciate the complexity of the technical problems this entails.

Aside from data volume, another big challenge is flexibility and the ability to evolve. As customers internalize the fact that their data are taken care of and the integrity of these data is ensured, they start wanting to do things with them. It's good to be able to second-guess the customers' needs, but it's even better to let them tell us what they want and try to address their wishes. This won't work when the product structure is too rigid, nor when the technology chosen is too inflexible with respect to exploration and change.

Since its inception, Keepit has experimented a lot, trying to find our own formula for success. From having our own data centers, to building highly optimized software components, to mistrusting established "best practices", and following our own intuitions (and calculations). In those formative years, the company developed a utilitarian approach to software, aggressively rewriting and re-architecting parts of our system whenever need arose.

1.1 Architecture Overview

A brief tour of Keepit's current technical architecture is in order, to give the reader some idea of how we manage backups.

Going from the outside in, the first components we see are the so-called Cloud Connectors. They are written in a combination of a specialized DSL and pure C++. They target specific cloud services via their published APIs and perform a wide range of operations, including authorization, listing, and downloading of user data at scheduled intervals and restoring on demand. Next up is OS3 (Object Store v3). It is a home-grown file system/object store responsible for physically storing backup snapshots. As can be surmised, it is the third and most stable in a chain of rewrites and refinements. In many respects this is the part of our system that is the most hardware-conscious and low-level, hence it is also written in C++. At the time of writing, the production deployment environment for OS3 involves storage racks composed of large numbers of 7200 rpm HDD drives. The choice of older, but battle-tested technology has been driven largely by trying to find the sweet spot between cost, performance, and reliability. The trio of low-level systems written in C++ is topped by BSearch — our comprehensive search tool able to reach into backup snapshots and look for keywords, patterns and text. In other words, the user can search the past (or rather different versions of the past), not just the present.

The complete backup system is orchestrated by Buslog (short for Business Logic). Written in Common Lisp, it is the glue that keeps things together, handles scheduling, moving data around, data consistency checks, invoicing, logging, notifications, and much more. It is the primary target of the majority of API requests that our system accepts. And it is by far the single most complex part of the entire Keepit architecture. Its functionality is complemented by two other Lisp components — SImport and SReimport. Their

ELS'25, Zurich, Switzerland
© 2025 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.15424521

responsibilities focus primarily on importing all kinds of search-enabled metadata to BSearch, based on periodic difference analysis of incoming backup snapshots.

The system also features a proxy component (handling API call authentication and proper redirection), *transform caches* responsible for generating and serving documents, and e-discovery tools. Production deployments of the Keepit system include a web frontend application and a public API for customers willing to work on their own integrations. Figure 1 gives a visual overview of the system architecture. Figure 2 outlines the flow of data within our system.

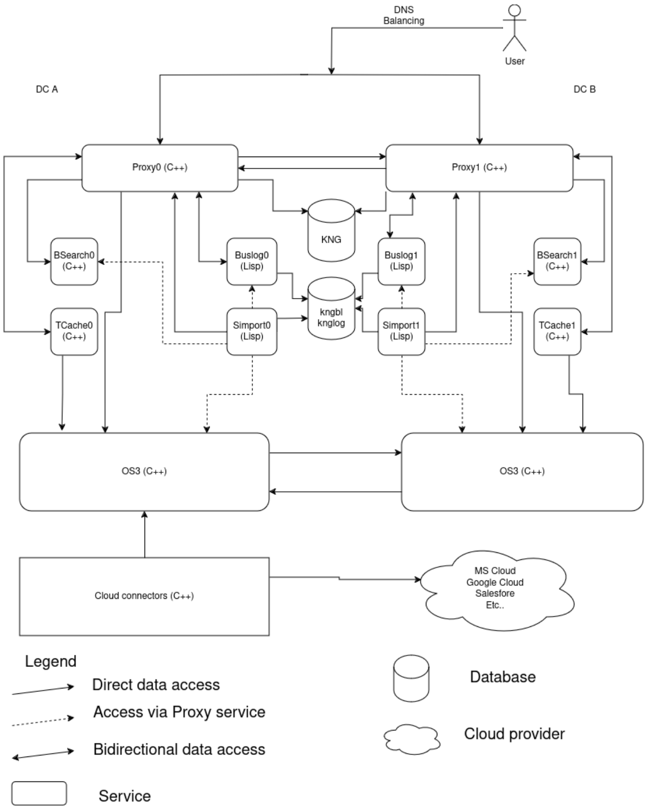


Figure 1: Keepit architecture

1.2 Why Lisp?

As the above architecture rundown demonstrates, backend components have been developed primarily in two programming languages — Common Lisp and C++. The rationale behind these choices is fairly simple. There was a clear need for a performant low-level language with proven track record of implementing high-speed applications running close to the metal. There was a similarly clear requirement for a versatile and high performance language for just about everything else, that would facilitate adaptation, change, and experimentation, while also promising stability.

In 2012, when crucial technological decisions were being made, the programming language landscape was not as rich as it is today.

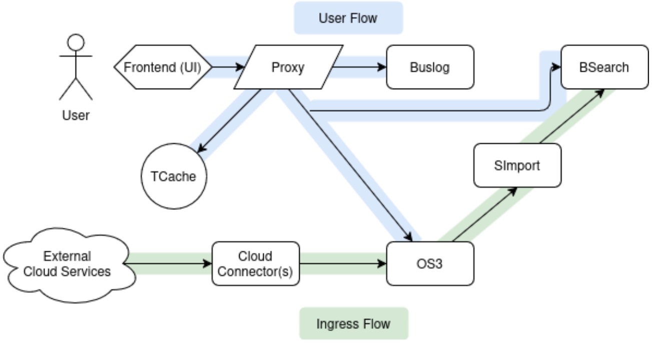


Figure 2: Data flow in the Keepit system

Moreover, there was another constraint on the selection process; namely, to be considered, a language had to have had at least 10 years of presence on the market until that point and would need to be maintainable over the coming 20 years. The argument being that 10 years is a sufficiently long period for a technology to more or less prove its usefulness, outlive the initial hype and ensure some amount of stability of the toolchain. In 2012 not many languages fit the bill and so it came to pass that C++ and Common Lisp became Keepit's choices.

To make the above argument more concrete, it might be interesting to ask a couple of "why not X?" questions. Java, for instance, had had quite enough exposure and testing by that time, but the performance characteristics of JVM were deemed too unpredictable. Go (2009) and Rust (2012) were too young to meet the stability requirements. Python had at least two major downsides at that time — relatively poor performance in general and the GIL (Global Interpreter Lock) making multithreaded programs inefficient in particular.

In retrospect, and taken on technical merits alone, Common Lisp seems to have completely redeemed the trust the technical leadership of Keepit had placed in it. The ability to connect to a system, inspect its state, alter its behavior or step through a piece of code in the middle of a production issue, helped fix many subtle bugs (some of them only materializing above certain thresholds of system load). In many areas of the system, initial code had been written in a more permissive, generic style, but Common Lisp's type system allowed us to incrementally tighten security in those areas as the system matured. The ability to compile into native code has enabled good and predictable performance for the components written in this language.

2 Into the Wild — The Internship

Around mid-2024, an idea started to form. We had a somewhat unsettling sensation that the two of us — the authors of this paper — were among the last Lispers in the region, available and willing to join the Keepit office in Kraków, Poland. With ambitious growth plans set up for the years to come, the company did not have a clear path in sight to scaling up the Lisp department. We started bouncing around the idea of "growing our own Lispers" as an alternative to finding existing ones. At first it was something of a loose thought, but quickly took on a more serious tone.

In one meeting with higher management the idea was presented, received enthusiastically, and subsequently greenlit for implementation. When laying out plans for the upcoming program, we needed to account for the time required to teach newcomers Common Lisp essentially from scratch. Typical internship programs take around 3 months to complete. We figured we would need additional 2 months to cover the essentials of Common Lisp and still be able to fit some time for work in the company codebase. Hence we arrived at the duration of 5 months.

Another vital question to address was the number of participants we wanted to recruit. We had to operate within two main constraints. On the one hand, we wanted to have sufficiently many people to make the whole program worthwhile. We estimated that a retention rate of 30-50% would be ambitious, but not unrealistic. On the other hand, we needed to account for the dynamics of our teaching process. With two program coordinators, we had a limited pool of energy and attention we could distribute among all participants. Starting with the assumption that we would need at least 5 new Lisp programmers within 6 months or so, and taking into account the constraints above, we settled on wanting to hire 10 interns for our program.

Not having clear ideas about what sort of response to expect, we decided to address the local student population. Official ads introducing the internship program were placed on several well-known job boards. Using contacts at Jagiellonian University, we set up an additional publicity event — a guest lecture presenting the technology stack behind Keepit, and our work opportunities for students and graduates. Around the same time, another department at Keepit (working in the connector-specific DSL area) came up with a similar idea and we opted for a joint presentation at the aforementioned lecture.

We then proceeded to conceptualize a rough outline of the upcoming internship. It was obvious that the first steps would need to cover language basics and tooling. Which, in turn, raised the shakespearean question: to Emacs or not to Emacs? Despite being wary of setting our future interns on too many parallel learning paths (especially, burdening them with ancient semiotic traditions, many of which are present in Emacs), we answered the above question in the affirmative, largely due to the fact that we felt less confident about alternatives, such as toolkits for Vim or VSCode. We were also keeping in mind the main goal of the program — to prepare people to join the Lisp department — and we already had a solid base of Emacs users there.

2.1 Recruitment

Once the idea was well established and put on solid institutional footing, we enlisted the help of our Talent Acquisition team. They set up a dedicated dashboard in Teamtailor — an applicant-tracking system that had been successfully used at Keepit before. A pipeline for processing prospective candidates was been created to accommodate relevant stages of the process, including resumé screening, interviews, work on our recruitment problem, and subsequent technical discussion.

We formulated a programming task for the candidates to solve, which would provide starting points for a technical discussion. We were primarily interested in two things: overall technical mindset (curiosity and tenacity rather than a solid body of knowledge) and good communication skills. Because we had received more than 70 resumés, we knew we could not realistically expect to talk to every candidate and so were rather strict with those we had any misgivings about. Broader engineering culture was also a concern, which is why we required the submitted solutions to be documented and to conform to some formal constraints that we set up.

Over the course of the next few weeks, we received around 30 solutions of varying quality and depth. The languages used ranged from Python and C++, through Java, Go and C#, with one or two people actually taking the trouble to attempt a solution in Common Lisp. We were not expecting flawless approaches at this point. In fact, we were hoping to gain some talking points for the in-person technical interviews scheduled for the next stage. Common programming problems, such as excessive complexity, disregarding edge cases, and possible input sizes were valuable to us, as they allowed us to further inspect and analyze the candidates.

In addition to us discussing the technicalities of submitted solutions, those interviews were meant to assess the candidates' command of spoken English and see how they behave in a moderately stressful situation. We were particularly interested in whether they would be able to hold a highly technical discussion with some amount of fluency and clarity. Regrettably, some otherwise promising candidates were rejected by us solely on the basis of how stress overpowered their communication skills. Our thinking in that respect was that being able to talk about programming is vitally important and a candidate demonstrating instability at such an early stage posed significant risk of having this kind of problems later as well. Contrary to many statements that describe Lisp as a language for "lone wolves", it is possible and expected to write it collaboratively — we were filtering people based on that.

After more than two weeks of having daily interviews, usually with several candidates on a single day, we had a preliminary ranking of candidates and we were beginning to make some early acceptance decisions. With outstanding candidates, we did not want to risk "losing" them by deliberating too long. Slowly but steadily our ten slots began to fill and eventually we had a full roster of people to whom we presented our offer and who accepted it.

One observation we were able to make at this stage was just how much energy it required. Having two or three candidates on a single day was mentally taxing and disrupted other things we might have planned. Four was exhausting beyond our previous beliefs. One time we had five, which we managed, but then deemed too much to sanely repeat.

2.2 Teaching

Once our new interns had arrived at the office in late November 2024, there came the time to test our assumptions and set to work getting them up to speed. In keeping with the bottom-up traditions of tool building, we decided to tackle the development environment first. Most people were somewhat familiar with mainstream offerings, such as VSCode and the JetBrains IDEs. So naturally there was some unlearning to do.

2.2.1 Emacs. The first order of business, after giving the interns their work machines, was to get them to install GNU Emacs and run the built-in tutorial. This was the first time most of them had seen, let alone used, Emacs. We gave them them plenty of time to get used to the unfamiliar keybindings, idiosyncratic terminology, and workflow. In the meantime, we offered our assistance with any questions they had, and held some live demo sessions to review and extend what they had learned.

Around the same time, we started thinking about a bulk-purchase of some quality introductory materials for Emacs learners. We surveyed the available resources and decided to contact the author of [6] with regard to getting some digital copies of the book, plus physical copies of [3] and [5][1]. Soon after, we made the purchases and invited the interns to devote additional time and energy to self-study.

Over the course of the next few days, we encouraged them to take the immersive approach and spend as much time inside Emacs as possible. There were many aspects in their GNU/Linux work environments that needed tweaking and configuring so we tried to make them use Emacs for that as well. We showed how learning a new set of keybindings can be useful in other contexts, e.g. in Unix shells and Readline-enabled programs.

All in all, what proved the most important and efficient in terms of knowledge transfer was direct assistance whenever people encountered problems, and leading by example. Also, sharing of configuration code snippets saved a lot of time with discovering Emacs' potential and saved the interns the frustration of searching disparate sources.

2.2.2 Common Lisp. After this brief prelude came time for the gist of our internship - the actual teaching of Common Lisp. Long before starting, we decided to pattern our exposition on [7]. Although it did feel slightly dated in some respects, we deemed the overall structure of the book very much relevant, particularly the beginner-friendly explanations of Lisp-specific notions of symbols, `cons` cells and CLOS. At the same time it has a much faster pace than, for example, [8], reflecting the fact that our interns were already in the middle of their CS education.

The added benefit, from our point of view, had been how [7] intersperses more theoretical, explanatory chapters with practical ones, often implementing some toy, yet functional project. In our opinion, the above neatly illustrated how you can start doing interesting things with Common Lisp with relatively modest theoretical foundations.

From the very beginning, we knew that synchronization between interns was going to be a problem. Most of them still had university classes they needed to attend, and their schedules were essentially incompatible with each other. We decided to make our video sessions recorded and available internally, which not only worked around this problem, but also provided our interns with material for later review that also constitutes our only internal Lisp teaching resource to date.

We also recognized the importance of visual aids to our more verbal explanations and hands-on examples. A graphical tablet, easier to record than whiteboard, was employed to draw conceptual diagrams and explain the various aspects of CL that caused confusion. The ad-hoc nature of these diagrams made plenty of room for humor. By presenting “serious matters” in a frequently jocular manner, we managed to dispel the initial feeling, among many of our interns, of dealing with something alien and incomprehensible. To our delight, merriment became a staple of our almost daily meetings and our sometimes quirky sense of humor got picked, to some extent, by others.

In December, another edition of Advent of Code ([1]) launched, and we already knew that some interns were willing to participate. So as a complement to our regular lectures, we organized optional AoC problem-solving sessions, where we would tackle one or two problems and construct solutions in a highly interactive manner. These sessions proved to be of great value, since they highlighted the areas where the understanding built through attending our regular lectures proved insufficient or where explaining things from a fresh angle made it easier for the interns to grasp. We used the opportunity to talk about more general Computer Science topics as well, discussing specific algorithms, questions of complexity, and demonstrating how the conversational nature of Common Lisp facilitated bottom-up construction of programs. It was around this time that we adopted the attitude that we would rather repeat something three times than zero times.

We used the course outline developed before the internship to track the progress of our curriculum — both for ourselves and to communicate the status to our colleagues in the People & Culture department who were responsible for the administrative aspects of our program. Figure 3 shows a finished summary of our teaching efforts, along with a rough initial estimate on when we would touch given topics. This estimate was mostly correct, with us moving some topics around the month boundaries, adapting to the needs of our interns.

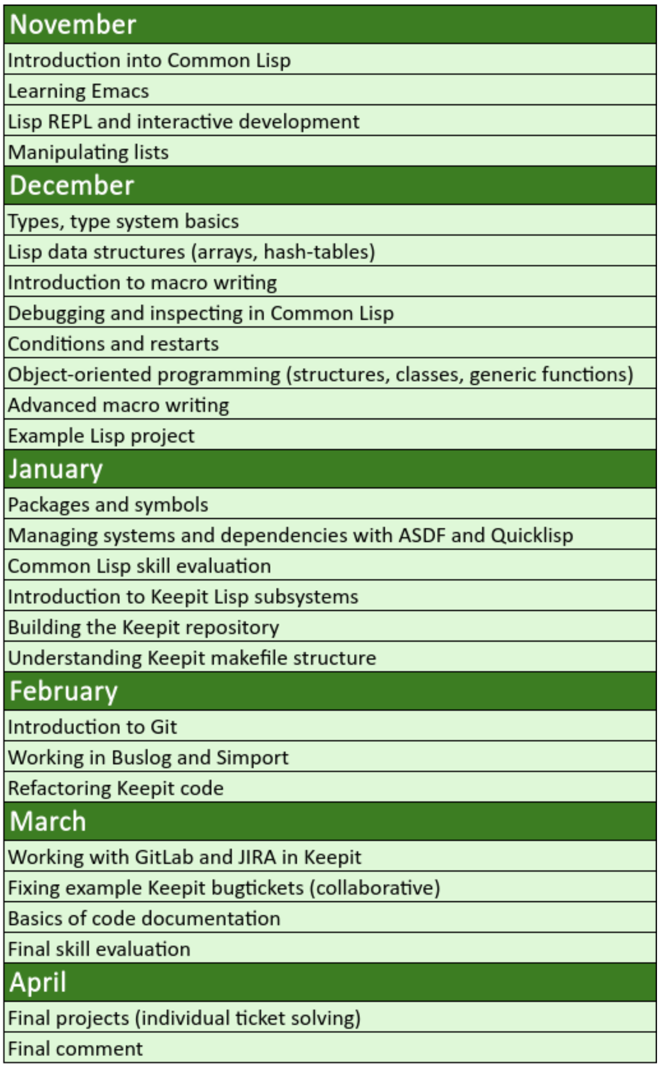
2.3 Evaluation

After approximately two months, we had managed to cover every topic that we considered the core of the program. Around that time we began thinking about ways to evaluate our interns' progress. On the one hand, this was a semi-formal requirement from our colleagues coordinating the internship, on the other, we wanted to see how well they would be able to apply the knowledge they had accumulated. There were two major projects aimed at getting the interns deeply involved in some more complex programming.

Firstly, as an outgrowth of a fantasy-themed demonstration of CLOS, we presented our interns with code for elements of a role-playing game engine, implementing mechanics of hitting monsters with weapons. This allowed us to express different class hierarchies for weapons and monsters, as well as means of showing method combinations, chaining generic functions, and working with their recursive calls. Tasked with adding various changes to this codebase and with freeform experimentation, people started extending and developing their custom versions of this engine. In our observations, having a concrete narrative in their heads made it easier for them to think about the purpose of having class hierarchies and advanced method combination mechanics. At times, we were astonished by the extent to which they allowed their imaginations to lead them toward deeper understanding of CLOS and CL itself.

Secondly, one of us conceived of a programming variant of an Easter egg hunt. A Lisp image spiked with riddles, hints, hidden messages and codes was been prepared, tested, and deployed in a compiled form to be shared with the interns. Our intention with

[1] One of the authors apologizes for a self-insert.



November
Introduction into Common Lisp
Learning Emacs
Lisp REPL and interactive development
Manipulating lists
December
Types, type system basics
Lisp data structures (arrays, hash-tables)
Introduction to macro writing
Debugging and inspecting in Common Lisp
Conditions and restarts
Object-oriented programming (structures, classes, generic functions)
Advanced macro writing
Example Lisp project
January
Packages and symbols
Managing systems and dependencies with ASDF and Quicklisp
Common Lisp skill evaluation
Introduction to Keepit Lisp subsystems
Building the Keepit repository
Understanding Keepit makefile structure
February
Introduction to Git
Working in Buslog and Simport
Refactoring Keepit code
March
Working with GitLab and JIRA in Keepit
Fixing example Keepit bugtickets (collaborative)
Basics of code documentation
Final skill evaluation
April
Final projects (individual ticket solving)
Final comment

Figure 3: Common Lisp curriculum

it was to emulate a connection to a running Lisp process and introspecting it via the REPL, debugger, and inspector only, without access to the source code comprising the egg hunt itself.

We encouraged our interns to cooperate to some extent, sharing insights and methods, but not to spoil each others’ fun going through the riddles. The idea proved to be very popular and got the interns visibly engaged and excited about this method of teaching. The source code of the Keepit Egg Hunt has been made public and is available in a Git repository at [2].

Having successfully completed this part of the internship, we deemed them ready to reach for more advanced resources and recommended that they start reading [4]. We ran some additional sessions explaining and applying the creation, usage, and pitfalls of macros. It took a while for the fundamental nature of macros to sink in, but ultimately we seem to have succeeded.

Our goals with the different parts of the evaluation process were to verify different qualities of our interns. The CLOS role-playing game project was meant to check their creativity in terms of extending existing code with new ideas, as well as their willingness to question the assumptions made by us for how monsters are allowed to be hit and how weapons are supposed to work. The egg hunt was meant to verify their knowledge of Lisp process internals and develop their intuition of working with image-based programming, including the dangers of destroying some data and needing to possibly restart the process in such cases.

Additionally, we kept an eye on initiatives taken by the interns themselves. Some examples are writing programs with more complicated logic (e.g. chess engines), modifying FOSS Lisp programs like simple games, or making contributions and bugfixes to libraries used throughout the Lisp ecosystem. This approach has proven valuable to us later, when we were selecting candidates for employment in Keepit.

2.4 Diving into the System

At this stage, our students were ready to get introduced into the architecture of our system and subsequently to our codebase. We started with a fairly detailed drawing session with all the elements described above arranged into a logical diagram with data flow and dependencies pictured. Once they had a grasp of “what” and “why”, we moved on to “how”.

Our presentation of how the code is organized proceeded outside-in. We took the entry points of our Lisp components and systematically went through them, commenting on the purpose of various subcomponents, and frequently jumping into definitions. This way we could quickly and concretely address many questions arising along the way, but also provide yet another example of using Emacs and Slime as a jack-of-all-trades, apt for writing Lisp code, but also for exploring existing structures, with example uses of breakpoints, trace facilities, crossreferencing, and browsing large projects.

We then tried to demonstrate a typical workflow, including working with Jira on task formulation and discovery, and using Git and GitLab to manage changes and merge requests. In order to get the interns more familiar with the approach of branching off, doing work, creating merge requests and merging, we created a dedicated Markdown file to collect their feedback and requests regarding the internship itself. We asked them to contribute to that file using the workflow above to exercise that aspect of working with files.

To top off this path to becoming more self-reliant and independent programmers, we devised some real-world but simple tasks for the interns to work on in teams. We provided a simple Lisp program for splitting people into three groups and reviewed it together for the sake of transparency and learning. Using it, we divided them into 3 teams randomly assigned to those tasks. We spent some time explaining the tasks and fleshing out their descriptions. Next, we gave them a couple of days to self-organize, research their respective topics, and divide the work.

Over the course of the next two weeks, they worked diligently on the tasks. We observed they changed their sitting arrangements to make collaboration easier. Seeing them take initiative, try various approaches, and have constructive discussions by the whiteboard was a heartwarming sight after all the time spent teaching them.

Afterwards, we moved on to smaller and more refined tasks, requiring more insight and self-reliance with regards to exploring the code and finding information, as well as contacting other people within the company for hints and support with merging. Those tasks were from our backlog of so-called "patch pool" tickets, from which we selected small fixes that could be developed independently. Gradually, our interns moved from necessary collaboration into solo work, with the tasks also becoming more and more specific.

2.5 Wrap-Up And Hiring

By the last month of the internship, we had collected enough information to be able to make our decisions with regards to hiring. We came to a conclusion that — however unlikely we consider it to be — all ten of our interns were viable for being hired as a team, each of them displaying a different combination of qualities, simple technical prowess being only one of them. Our opinion was consulted with the rest of our Lisp team and found further support.

After bringing our concerns up with our company management, it became obvious to us that our current business constraints do not permit us to hire all ten of them, even taking their part-time availability into account. Based on this, we made a decision to pick six of them to be offered contracts for Junior Common Lisp Developers right away, while keeping contact information for the remaining four to use as soon as new Lisp positions open up in the future.

We wrapped the internship by holding individual performance reviews with each of the interns. We made it explicit that the internship result was positive for each of them and the hiring decision was made as a result of company-wide business constraints that our Lisp department in Keepit was not able to negotiate around.

At the time of finalizing this paper, the contracts are currently being signed. There is no data for long-term effects and career paths for the interns, as the internship is currently being concluded; this shall remain to be discovered in future studies.

3 Lessons Learned

Five months spent in close proximity with a group of ten bright and eager young people has been a curious experience for the authors. Constantly demanding and taxing on the one hand, energizing and rewarding on the other. We saw almost daily the joy of discovery and amazement at the brilliance and simplicity of Lisp, which served to renew our own enthusiasm, but also instilled in us a heightened sense of responsibility for others. It was with a touch of nostalgia that we saw the conclusion of our program, but at the same time we were happy for our interns having become more experienced and mature in the process.

As a company, we were able to draw some conclusions from all the above. First and foremost, when faced with a shortage of programmers available on the market, sometimes the best a company can do is "grow your own". The process may not be easy nor quick, but when done right it can bring value to an employer in the form of junior programmers who already know the language and the basics of the company codebase.

Having limited time and a lot of material to cover is challenging in that one needs to find the right formula of timesharing and sequencing. Multiple topics such as tooling, language structure and ecosystem, existing codebase, and established company practices need to be taught in parallel. But the perils of too much context switching and logically unsound sequencing of topics can cause more confusion than comprehension.

One particular problem made evident by our approach is that learning Emacs is a major factor that counts towards a Lisp programmer's productivity. Even a person with good knowledge of Lisp and advanced interpersonal skills may be crippled by their lack of efficiently moving around Emacs — an observation we made during the internship based on the individual progress of some interns.

Another piece of feedback that came from our interns was that we should have begun assigning practical tasks to them from the very beginning, rather than focusing on the theory and exercises from [7]. It was noted that the hands-on exercises, such as Advent of Code, the role-playing game, and Egg Hunt were of great value to them.

Yet another issue we have noticed was not enough work for our interns, especially toward the end of the program. Procuring tasks ad-hoc proved difficult and messy. The interns regularly exceeded our expectations regarding time spent solving problems. Preparing a long enough backlog of tasks ahead of time (potentially recruiting the help of our Lisp colleagues) could have alleviated this problem.

4 Conclusions

There's a well known quote by Marvin Minsky: "Anyone could learn Lisp in one day, except that if they already knew Fortran, it would take three days.". It describes aptly (if a little indirectly) the situation with our interns. Having little programming experience meant having to "unlearn" fewer things before approaching Common Lisp with an open mind. In our view, that "freshness"[2] really served our interns well in the longer run.

The recruitment process for our internship started amidst a so-called "employer's market" phenomenon in IT. Companies could afford to be picky about candidates with many talented and motivated people competing for a single opening. One can't help but think that such circumstances make young people in need of professional experience more eager to express interest in less orthodox offerings. Such was the story behind this one.

References

[1] Advent of Code. URL https://adventofcode.com/.

[2] Keepit Egg Hunt. URL https://github.com/phoe/keepit-egg-hunt/.

[3] Vsevolod Domkin. *Programming Algorithms in Lisp: Writing Efficient Programs with Examples in ANSI Common Lisp.* Apress, 2021. ISBN 9781484264287. doi: 10.1007/978-1-4842-6428-7. URL http://dx.doi.org/10.1007/978-1-4842-6428-7.

[4] Paul Graham. *On Lisp.* Prentice Hall, New York, NY, 1993. URL http://www.paulgraham.com/onlisptext.html.

[5] 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.

[6] Mickey Petersen. *Mastering Emacs.* 2024. URL https://www.masteringemacs.org/.

[7] P. Seibel. *Practical Common Lisp.* Books for Professionals by Professionals. Apress, 2006. ISBN 9781430200178. URL https://books.google.pl/books?id=dVJvr8PtCMUC.

[8] David S. Touretzky. *Common LISP: A Gentle Introduction to Symbolic Computation.* Dover Publications, 1990. URL http://www.cs.cmu.edu/~dst/LispBook/index.html.

[2] fresh, adj. 1. (of an object yielded by a function) having been newly-allocated by that function.

The Lisp in the Cellar

Dependent Types that Live Upstairs

Pierre-Évariste Dagand
CNRS – IRIF – Université Paris Cité
Paris, France
dagand@irif.fr

Frédéric Peschanski
Sorbonne Université – LIP6
Paris, France
frederic.peschanski@lip6.fr

ABSTRACT

Dependent types provide a way for programmers to write code that computes types. Type-level computations in turn may depend on values, allowing various powerful programming patterns. Moreover, even though types exhibit a rich dynamic semantics, type-checking remains a purely compile-time operation.

The Deputy system presented in this paper is a Clojure-hosted dependently-typed programming language, featuring inductive datatypes. It serves as an experimental vehicle to explore the implications of the Lisp-based REPL-driven interactive development workflow, not only while programming but also during type checking. The system is thus developed as a Clojure library, which means that the host language is still available while "programming" at the type-level.

CCS CONCEPTS

• **Theory of computation** → **Type theory**; **Constructive mathematics**; • **Software and its engineering** → **Data types and structures**.

KEYWORDS

dependent types, REPL, symbolic debugging

1 INTRODUCTION

Dynamic versus static typing is perhaps the longest unresolved programming debate, revived almost daily on the *Hacker news*[1]. Static typing helps to find (some) bugs early, undoubtedly. But, the other way around, types often get in the way of just thinking about solving computation problems. Consider for example *data-oriented programming* [22] that involves complex pipelines along which semi-structured data pass through chains of transformations. With classical algebraic datatypes, the specification of such systems ofen leads to significant redundancy (trying to exactly capture the specifics of each step) or loss of accuracy (merging multiple, distinct representations in a coarser one). Compilation chains provide another similar situation, with the implementation of multiple compilation passes, each one characterized by an input and an output intermediate language representation. Of course, more powerful type systems have been proposed to improve the situation: polymorphic variants [12], refinement types [10], set-theoretic types [4], *etc.* to name but a few. However, if the comfort improves, we still face the situation that types remain mostly static and overall distinct from *actual* computations

From a totally different perspective, *dependent types* [16] put into question the overly *static* nature of classical type systems. Pioneered by Epigram [18], languages based on dependent-type theory provide an integrated development environment *assisting* programmers in writing code that compute *types*. In data transformation pipelines or compilation chains, each node of a data-transforming pipeline can be specified at the type level by a transformation *performed* at the type-level. Most interestingly, type-level computations may *depend* on "normal" value-level computations, *i.e.* taking the output of a data-transforming node to adapt the *type* of the next node in the chain. A typical down-to-earth example is that of type-checking `format` calls by first parsing the format strings, and then checking the argument types. One can also think about modern web frameworks hosted in typed languages, performing an extraction of the database schema at boot time so as to validate the type safety of database queries [11] ahead of any execution. Dependent types natively support these patterns, and much more [21]. Moreover, even though the system is dynamic, type-checking is still performed purely at compile-time, which makes the approach quite different from runtime validation frameworks such as `clojure.spec` [2] or Malli [3].

The goal of the Deputy project[4] is to support an integrated experience of *programming with dependent-types* from *within* a Lisp host – actually the Clojure programming language – in a typical *domain-specific language* approach. The choice of Clojure is mostly pragmatic: the essence of dependently-typed programming being purely functional. Another less fundamental but still enjoyable characteristic of Clojure is its support for vectors and maps beyond lists and S-expressions. The flip side of the coin is that computing with dependent types can be quite demanding performance-wise, involving non-trivial memory usage patterns[5].

Technically-speaking, Deputy is a Lisp remake of the work of Chapman et al. [5], which describes an implementation of inductive families, the dependently-typed generalization of algebraic datatypes [3]. Drawing on the work of Benke et al. [1] and Morris et al. [20], this work devised a tiny yet powerful presentation of inductive types and inductive families. A fascinating property of this implementation was its reflexive nature: the formalization of inductive types was carried (and in fact implemented!) in terms of

[1] Actually, Lispers know that the debate is somewhat pointless, considering e.g. Typed Racket, Typed Clojure or Coalton for instance.

[2] https://clojure.org/guides/spec

[3] https://github.com/metosin/malli

[4] https://gitlab.com/fredokun/deputy

[5] cf. https://github.com/AndrasKovacs/smalltt

ELS'25, May 19–20 2025, Zurich, Switzerland
© 2025 Copyright held by the owner/author(s).
https://doi.org/10.5281/zenodo.15424968

itself: the *grammar* of inductive types was bootstrapped, encoded as an inductive type similar to any other inductive definition. To achieve this feat, the entire type theory was designed to natively support a (restricted) form of homoiconicity, heavily inspired by Lisp representation of lists as cons-cells. As a consequence, native and reflected terms are syntactically identical and the bootstrap goes unnoticed for the user. The present work reiterates this bootstrap process[6], taking full advantage of the host language and programming environment.

The presentations being done, we now provide the outline of the remainder of the paper. In Section 2, we illustrate one of the main *tangible* features of the Deputy system: the fact that the user of the system can interact in a typical REPL-based workflow with the main and most complex component of the sytem: its type-checker. In Section 3, we introduce dependent types through the Curry-Howard correspondance adopting dependent types as a means of specifying our programs. We next develop statically-typed counterparts to several idiomatic Clojure idioms (Section 4), namely keywords and maps. This enables us to transfer the burden of validating basic schemas from run-time to compile-time ; We ultimately extend our type theory with inductive types (Section 5), thus giving ourselves the ability to program with recursive (well-founded) data-structures. As a consequence, recursive schemas can thus be checked at compile-time too.

We do expect some readers to remain unmoved by the possibility of embedding (yet) another type checker in their favorite Lisp. Having ourselves endured quite a few hours pondering over inscrutably complex type errors, we sympathize with this stance: aside from the academic exercise, why subject oneself to the ordeal of arguing with a type-checker? The general take-away of the present work remains that dependent types offer a language as powerful as constructive mathematics to specify computation of software artifacts. With great power …

2 AN APPETIZER: TYPE-LEVEL SYMBOLIC DEBUGGING

The primary goals of the Clojure implementation are two-fold. First, we want to exploit the full potential of the homoiconoicity requirements by having the object language (our dependently-typed language, Deputy), live as an embedded language into its host language. This is the easy part. Second, and it is a little bit more tricky, we want to explore the implications of the Lisp-based REPL-driven interactive development workflow *during type-checking*. While typed programs do not fail (in the sense that they are immune to runtime type errors), type-level programs can (and do!) fail. Indeed, the dynamic nature of dependent types make them quite hard to keep track of in one's head, and type errors may be tricky to fix. Programmers need assistance during type-checking, to the point where one would contemplate having a type-level debugger. However, we knew from experience that rolling our own interactive tooling would be prohibitively expensive, inducing a significant risk that our experiment would grind into a halt. Instead, we reasoned that we could take advantage of the tooling already provided by our host language, namely the REPL and associated debugging environment. This, we believe, may be the most innovative aspect of our implementation: the fact that the type-checker is completely reflected into the host.

Indeed, dependent types are "nothing but" abstraction of programs, with the type-checker playing the role of an interpreter, reducing the pair of a program *applied to its type* to `true` only if the program is provably *not going wrong* over any concrete value inhabiting that type. In the field of abstract interpretation, a similar observation lead to "abstract debugging" [2, 13, 19]: rather than debug a program with a single concrete, run-time value (such as an integer), one can just as well debug the abstract program (obtained by abstract interpretation of the concrete one) using an abstract, symbolic representation of a potentially-infinite set of values (such as open-ended intervals of integers).

Hence, a Deputy program has two distinct execution phases: In the first phase, its type checker runs the program symbolically (and in an open context) by referencing top-level typing annotations. When this phase succeeds, the program can then be considered *well-defined*, permitting Deputy to progress to the second phase of execution. In this second phase it directly runs the actual program using concrete values. With either phase, when encountering an execution error, Deputy is capable of launching a debugging session. Currently, it uses the the debugger included with the Cider mode of Emacs [7].

Similarly, in case of an error during type-checking, we can take advantage of debugging breakpoints in the implementation of the typing rules. For example, the following

```
(defterm [wrong [f (=> :unit :unit)]
                [x :int]
                :unit]
  ;; WRONG: `x` is an `:int`
  ;;        cannot be an argument to `f`
  (f x))
```

attempts to define a function called `wrong` that takes two arguments, `f` of type (⇒ :unit :unit) and `x` of type :int, outputs a result of type :unit, and simply applies `f` on `x`. Since the type :unit is distinct from the type :int, this definition is ill-typed[8]. In particular, the issue will be detected by the type conversion rule

```
(defmethod type-check-impl
  :computation
  type-check-impl-computation
  [ctx vtype term]
  (ok> (type-synth-impl ctx term) :as synth-vtype
       [:ko> (#dbg "Cannot␣synthesize␣type.")
         {:term (a/unparse term)}]
       (if (n/alpha-equiv vtype synth-vtype)
         true
         (#dbg [:ko "Term␣`term`␣is␣not␣of␣type␣`vtype`."
           {:term (a/unparse term)
             :synth-type (a/unparse synth-vtype)
             :vtype (a/unparse vtype)}]))))
```

[6] Also helped by a talented student: thank you, Teo Bernier !

[7] https://docs.cider.mx/cider

[8] This can be fixed by either changing the type of `f` to (⇒ :int :unit), or changing the type of `x` to :unit, depending on the context in which this definition is actually used.

Being instrumented with judiciously-placed #dbg breapoints (a very convenient debugging aid provided by Cider), the type-checker gets suspended at the origin of the type error we intend to debug, bringing Cider into the state depicted in Figure 1.

From there, one can query the stack trace (using the command “s” in Cider) of the type-checker, which implicitly represent the collection of typing rules involved so far:

```
        (...)
        REPL:  131  deputy.typing/eval15258/type-check-impl-computation
MultiFn.java:  239  clojure.lang.MultiFn/invoke
  typing.clj:   64  deputy.typing/type-check
  typing.clj:   52  deputy.typing/type-check
  typing.clj:  220  deputy.typing/eval1422/type-synth-impl-application
MultiFn.java:  234  clojure.lang.MultiFn/invoke
        REPL:  127  deputy.typing/eval15258/type-check-impl-computation
       (...)
```

or we can query the local variables (using the command “l” in Cider) involved in that particular typing rule:

```
Class: clojure.lang.PersistentArrayMap
Count: 6

--- Contents:
  type-check-impl-computation = #function[(...)]
  ctx = {f {:node :pi, :domain :unit,
            :codomain {:node :bind, :name _, :body :unit}}, x :int}
  vtype = :unit
  term = {:node :free-var, :name x}
  res15256 = :int
  synth-vtype = :int
```

More conveniently, we can also use local evaluation (using the command “e” in Cider) to call arbitrary functions defined in the Deputy library. For example, calling `(debug-ctx ctx)` will display a pretty-printed representation of the `ctx` variable (Figure 2). Of course, most of the required machinery is made available thanks to Cider, Clojure itself and the underlying runtime (in general, the JVM but JS in another possibility through Clojurescript). The contribution on our part is the architecture of the type-checker that provide several entry points for reflecting its behaviour.

3 DEPENDENT TYPE THEORY

In the following, we offer a piecemeal presentation of the syntax and semantics of Deputy's kernel: a simple dependently-typed language. We start off with some rudimentary syntactic constructs in order to establish conceptual foundations of the language. We then incrementally build up support for functions (Section 3.1) and pairs (Section 3.2).

Unlike their simply-typed cousins, dependently-typed languages provide a single, unified language for describing programs as well as their types. There is therefore a single syntactic category encompassing both terms *and* types. It is only type-checking that decides whether an expression is meant to denote a program (in which case, we conventionally use the letter t to denote such an expression) or as the type of of a program (in which case, we conventionally use the letter T to denote such an expression). Both belong to the same syntactic category of “terms and types”.

Because open term reduction (*i.e.*, reducing a program with free variables whose type is provided by a non-empty context) plays a key role during type-checking, we define two syntactic categories, distinguishing canonical terms (representing either values or terms stuck on a variable) from computational terms (representing either a computation stuck on an open variable, or a suspended computation t carrying a type annotation). A short extract of the syntax is given below (cf. Appendix A for the complete syntax).

$$\begin{array}{rlr}
T, t & ::= & \text{(canonical types \& terms)} \\
 & \mid \texttt{:type} & \text{(type of types)} \\
 & \mid e & \text{(stuck term)} \\
 & \mid \ldots & \\
e & ::= & \text{(computational terms)} \\
 & \mid (\text{the } T\ t) & \text{(type annotation)} \\
 & \mid x & \text{(variable)} \\
 & \mid \ldots &
\end{array}$$

Over such a tiny fragment, the reduction relation, written $t \rightsquigarrow t'$ to denote the fact that a term t reduces to a term t', is almost trivial: a :type is already a value, thus fully reduced; an open variable x is already neutral, thus fully reduced. Only the type annotation hides some potential reductions, we thus have:

$$(\text{the } T\ t) \rightsquigarrow t$$

If we forbid type annotations from appearing within computational terms, the resulting syntactic category corresponds exactly to values (denoted by the letter v) in an open term reduction system: it is either a canonical term or a neutral term, *i.e.* a computation stuck on some open variable (denoted by the letter n).

The design of the type system can be thought off as the identification of some invariant over terms so as to ensure that

(1) a well-typed computational term e always reduces to a neutral term n (*i.e.*, all redexes have been eliminated) ;
(2) a well-typed canonical term t always reduces a value v, formally: $t \rightsquigarrow^* v$.

Together with the fact that reduction should preserve typing (*i.e.*, if t has type T and $t \rightsquigarrow t'$, then we expect that t' also has type T), this means that a type-checked term can be *trusted*. For example, we know in advance that a Boolean expression in an empty context will consistently return either *true* or *false*. This means that this expression can be promoted and safely get involved in the characterization of the type of programs: type-checking is decidable and terminating.

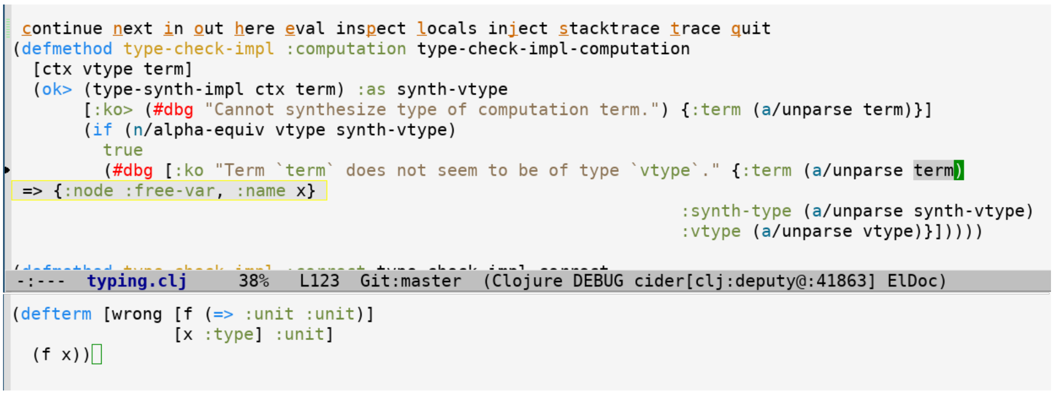
Mechanistically, type-checking can thus be understood as the process of analyzing a term t and making sure that it will always handle every interaction available to an environment specified by a typed context Γ and producing outputs following the specification set by a type T. We thus define the type-checking judgment

$$\boxed{\Gamma \vdash T \ni t}$$

to mean that the type T accepts the term t in context Γ, where the typing context associates a type to every free variable of t:

$$\begin{array}{rlr}
\Gamma & ::= & \text{(contexts)} \\
 & \mid \epsilon & \text{(empty context)} \\
 & \mid \Gamma, x : T & \text{(variable declaration)}
\end{array}$$

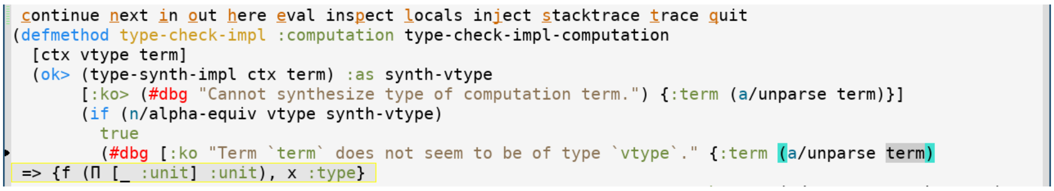
On paper, the type-checking judgment is presented as a relation. However, it admits a straightforward algorithmic reading: it can in fact be implemented as a function taking a context, a type and



```
 continue next in out here eval inspect locals inject stacktrace trace quit
(defmethod type-check-impl :computation type-check-impl-computation
  [ctx vtype term]
  (ok> (type-synth-impl ctx term) :as synth-vtype
       [:ko> (#dbg "Cannot synthesize type of computation term.") {:term (a/unparse term)}]
       (if (n/alpha-equiv vtype synth-vtype)
         true
         (#dbg [:ko "Term `term` does not seem to be of type `vtype`." {:term (a/unparse term)
 => {:node :free-var, :name x}
                                                                   :synth-type (a/unparse synth-vtype)
                                                                   :vtype (a/unparse vtype)}]))))

(defmethod type-check-impl :correct type-check-impl-correct
-:---  typing.clj     38%    L123  Git:master  (Clojure DEBUG cider[clj:deputy@:41863] ElDoc)
(defterm [wrong [f (=> :unit :unit)]
                [x :type] :unit]
  (f x))
```

Figure 1: (Symbolic) debugging



```
 continue next in out here eval inspect locals inject stacktrace trace quit
(defmethod type-check-impl :computation type-check-impl-computation
  [ctx vtype term]
  (ok> (type-synth-impl ctx term) :as synth-vtype
       [:ko> (#dbg "Cannot synthesize type of computation term.") {:term (a/unparse term)}]
       (if (n/alpha-equiv vtype synth-vtype)
         true
         (#dbg [:ko "Term `term` does not seem to be of type `vtype`." {:term (a/unparse term)
 => {f (Π [_ :unit] :unit), x :type}
```

Figure 2: Accessing the context by *e*valuation in the debugger

a term as arguments, returning either a success or a (very, almost too) detailed error trace detailing why the term is ill-typed.

In the actual implementation, type-checking is delegated to a multimethod definition:

```
(defmulti type-check-impl #'type-check-dispatch-fn)
```

for which we implemented an *ad-hoc* dispatch function for resolving the multimethod calls.

As usual with trust, our intuition suffers from a bootstrapping problem: if types are specifications against which we validate or reject the validity of terms, how do we assert the validity of types themselves? For simplicity, we adopt the following inference rule

$$\frac{}{\Gamma \vdash \text{:type} \ni \text{:type}}\ \textsc{Type-in-Type}$$

stating that the type of all types (denoted by :type) is itself well-typed (and its type is :type itself). Recall that inference rules reads in a bottom-up manner: here, we conclude that the judgment $\Gamma \vdash$:type $\ni$:type is valid without any further assumption as there are no premises judgments above the inference rule. In the implementation, this is implemented as follows:

```
(defmethod type-check-impl [:type :type]
  type-check-type-in-type [_ _ _]
  true)
```

From this assumption, it is easy to check that a given type T is valid in a context Γ: we must have $\Gamma \vdash$:type $\ni T$. Unfortunately for Mathematicians, the rule Type-in-Type [15] leads to an inconsistent logic [14]. However, it is not the kind of "logical exploit" that one ends up writing by accident[9]. Solution exists [24] to sidestep the problem entirely but they are costly to implement. Trusting the benevolence of our users, we stick with Type-in-Type.

Type-checking means that types have to be explicitly provided by programmers. To alleviate this burden, we adopt a bidirectional type system [6, 9], whereby we exploit the fact that we can systematically *synthesize* the type of computational terms through a type synthesis judgment $\boxed{\Gamma \vdash e \in T}$ which reads as "in context Γ, we can deduce that the term t has type T". Algorithmically, this means that this judgment can be implemented as a function taking a context and a term, producing a type. The implementation relies on the following multimethod:

```
(defmulti type-synth-impl
  (fn [_ term] (:node term)))
```

The dispatch is simply done on the syntactic category of the expression. Note that Deputy does not provide a generic type inference mechanism, for the very simple reason that dependent types are too powerful: the problem is indecidable. Instead, we rely on type synthesis as a restricted form of local type inference with a clear, syntax-directed information flow.

For example, upon a variable, we simply have to lookup the context to figure out its type:

$$\frac{}{\Gamma_0, x : T, \Gamma_1 \vdash x \in T}$$

This translates as follows in the implementation:

[9] In Agda, this consists in about 40 lines of carefully-crafted definition, maintained as part of the official test suite [8]. We expect a similar effort to port this result to Deputy.

```
(defmethod type-synth-impl :free-var
  type-synth-impl-free-var [ctx term]
  (get ctx (:name term)
       [:ko "Type␣Synthesis␣Error␣(...)"]))
```

Similarly, an annotated term carries along its type, so we just need to check the validity of the provided type (which cannot be trusted *a priori*, since it was written by the user) and then check that the term is admitted at the given type:

$$\frac{\Gamma \vdash \text{:type} \ni T \qquad \Gamma \vdash T \ni t}{\Gamma \vdash (\text{the}\ T\ t) \in T}$$

where this inference rule reads as "assuming that the judgments $\Gamma \vdash \text{:type} \ni T$ and $\Gamma \vdash T \ni t$ hold, we can conclude that the judgment $\Gamma \vdash (\text{the}\ T\ t) \in T$ holds".

Conversely, type checking can switch to type synthesis, upon a change of syntactic category (when a computational term is embedded into a canonical term). In this case, type synthesis returns a type T_0 while type checking proposes another type T_1. We must therefore also check that these two types denote the same object:

$$\frac{\Gamma \vdash e \in T_0 \qquad \Gamma \vdash T_0 \equiv T_1 : \text{:type}}{\Gamma \vdash e \ni T_1}$$

To decide equality of types, we crucially rely on the invariant that types always reduce to values: in principle, we just have to fully evaluate them and compare the resulting values. In practice, various tricks are used to improve upon the inefficiency of this approach, preserving as much sharing as possible.

3.1 Dependent functions

At the term level, the functional fragment of our dependently-typed programming language consists of run-of-the-mill λ-abstraction and function application, whereas to specify functions, we introduce the Π type constructor:

$$\begin{array}{lll}
T, t ::= \ldots & & \\
\quad \mid (\Pi\ [x\ T_0]\ T_1) & & \text{(Pi type)} \\
\quad \mid (\lambda\ [x]\ t) & & \text{(abstraction)} \\
e ::= \ldots & & \\
\quad \mid (e\ t) & & \text{(application)}
\end{array}$$

Unlike their simply-typed cousins, the co-domain of dependent functions *depends* on the value passed as argument. This is visible in the syntax of Pi types as well as their typing rule:

$$\frac{\Gamma \vdash \text{:type} \ni T_0 \qquad \Gamma, x : T_0 \vdash \text{:type} \ni T_1}{\Gamma \vdash \text{:type} \ni (\Pi\ [x\ T_0]\ T_1)}$$

$$\frac{\Gamma, x : T_0 \vdash T_1 \ni t}{\Gamma \vdash (\Pi\ [x\ T_0]\ T_1) \ni (\lambda\ [x]\ t)}$$

$$\frac{\Gamma \vdash e \in (\Pi\ [x\ T_0]\ T_1) \qquad \Gamma \vdash T_1 \ni t}{\Gamma \vdash (e\ t) \in T_1[x \mapsto t]}$$

If the co-domain of a Pi type does not depend on the domain value, we recover the usual, simply-typed function space:

$$(\Rightarrow T_0\ T) \triangleq (\Pi\ [\_\ T_0]\ T)$$

where the variable _ is not bound in T.

Also, to simplify notations, we adopt a sequential notation that implicitly desugar to right-nested chains of Pi types :

$$(\Rightarrow T_0\ T_1\ \ldots\ T) \triangleq (\Rightarrow T_0\ (\Rightarrow T_1\ \ldots\ T))$$
$$(\Pi\ [x_0\ T_0][x_1\ T_1] \ldots T) \triangleq (\Pi\ [x_0\ T_0]\ (\Pi\ [x_1\ T_1]\ \ldots\ T))$$

To reflect dependent functions at the Clojure level, we provide a `defterm` macro that lets us introduce typed definitions with the syntactic form

```
(defterm [f [x_1 T_1]
            [x_2 T_2]
            ...
            T]
  t)
```

which results in, first, checking that the proposed type is valid:

$$\Gamma \vdash \text{:type} \ni (\Pi\ [x_1\ T_1][x_1\ T_2] \ldots T)$$

and, then, checking the the proposed term `t` is indeed of the proposed type:

$$\Gamma \vdash (\Pi\ [x_1\ T_1][x_2\ T_2] \ldots T) \ni t$$

If the type-checker is able to check both declarations, then `f` is defined as `t` in the context. Consider the following (somewhat silly) Clojure function:

```
(defn f [x] (if (true? x) 42 (not x)))
```

If we would like to type such function, the return type would be problematic: is it an integer? is it a Boolean? In Deputy, the following type adequately capture the intent of the function:

```
(Π [x :bool] (if (true? x) :int :bool))
```

We thus gain the ability to write functions that compute types from values. The archetypical use of this feature are format strings, where one must first parse the format string to determine the type of the arguments. This parsing function is thus a simply-typed function, of type $(\Rightarrow \text{:string :type})$

```
(defterm [format-type [s :string]
                      :type]
  (...))
```

which we then use to assign a type to the arguments of the `clojure.core` operator:

```
(defterm [format [fmt :string]
                 [args (format-type fmt)]
                 :string]
  (...))
```

3.2 Dependent pairs & unit

After functional values, the second and third fundamental datastructures of a Lisp are probably the cons cell $[t_0\ t_1]$ and its nullary counterpart, nil. Interestingly, they correspond exactly to two fundamental type constructors: the Sigma type and unit type, respectively:

$$\begin{array}{lr}
T, t ::= \ldots & \\
\quad \mid \mathsf{:unit} & \text{(unit type)} \\
\quad \mid \mathsf{nil} & \text{(unit value)} \\
\quad \mid (\Sigma\ [x\ T_0]\ T_1) & \text{(Sigma type)} \\
\quad \mid [t_0\ t_1] & \text{(pair)} \\
e ::= \ldots & \\
\quad \mid (\pi_0\ e) & \text{(first projection)} \\
\quad \mid (\pi_1\ e) & \text{(second projection)}
\end{array}$$

$$\begin{array}{l}
(\pi_0\ [t_0\ t_1]) \rightsquigarrow t_0 \\
(\pi_1\ [t_0\ t_1]) \rightsquigarrow t_1
\end{array}$$

In particular, Sigma types are a key tool to model structured data: in a pair $[t_0\ t_1]$, the first component t_0 determines the *type* of the second component t_1. Formally, the type-checking rules for Sigma types mirror the ones for Pi types:

$$\frac{}{\Gamma \vdash \mathsf{:type} \ni \mathsf{:unit}} \qquad \frac{}{\Gamma \vdash \mathsf{:unit} \ni \mathsf{nil}}$$

$$\frac{\Gamma \vdash \mathsf{:type} \ni T_0 \qquad \Gamma, x : T_0 \vdash \mathsf{:type} \ni T_1}{\Gamma \vdash \mathsf{:type} \ni (\Sigma\ [x\ T_0]\ T_1)}$$

$$\frac{\Gamma \vdash T_0 \ni t_0 \qquad \Gamma \vdash T_1[x \mapsto t_0] \ni t_1}{\Gamma \vdash (\Sigma\ [x\ T_0]\ T_1) \ni [t_0\ t_1]}$$

$$\frac{\Gamma \vdash e \in (\Sigma\ [x\ t_0]\ T)}{\Gamma \vdash (\pi_0\ e) \in T_0} \qquad \frac{\Gamma \vdash e \in (\Sigma\ [x\ T_0]\ T_1)}{\Gamma \vdash (\pi_1\ e) \in T_1[x \mapsto (\pi_0\ e)]}$$

with a notational reduction whenever the dependency is not exploited

$$\begin{array}{c}
(*\ T_0\ T) \triangleq (\Sigma\ [\_\ T_0]\ T) \\
(*\ T_0\ T_1\ \ldots\ T) \triangleq (*\ T_0\ (*\ T_1\ \ldots\ T))
\end{array}$$

in which case Sigma types simply boil down to the Cartesian products of simply-typed programmers.

As it turns out, Lisp programmers have been programming with Sigma types since the dawn of the Unix epoch. First, in terms of notations, right-nested cons cells

```
[1 [:string ["foo" [(format "%d-%s") nil]]]]]
```

are readily written in a tuple-like manner

```
[1 :string "foo" (format "%d-%s")]
```

Second, tuples naturally lend themselves to a left-to-right refinement of types, whereby the value stored in the tuple influences the type of values coming next in the tuple. For example, the previous tuple can be assigned the type

```
(Σ [_ :int][T :type][_ T][_ (=> :int T :string)] :unit)
```

that also accepts the following tuple

```
(defterm [example-tuple (Σ [_ :int]
                           [T :type]
                           [_ T]
                           [ _ (=> :int T :string)]
                           :unit)]
  [2 :int 42 (format "%d-%d")])
```

Through these two examples, we notice that the type constructors :string and :int are used as tags indicating the type of subsequent values in the tuple. In the dependently-typed programming community, this device is termed a "telescope" [17], whereby each Sigma type lets us provide a more *focused* type to the subsequent values.

4 PRESENTATION LAYER

We now extend the previous Section with *affordances*: means for programmers to translate their intents into the source code. None of the following definitions extend the expressivity of the overall type theory. In fact, most these constructs desugars into the core type theory.

4.1 Keywords

First, we extend the type theory with keywords. We also hardcode a dedicated type to represent sequence of keywords since, at this stage, our type theory does not support a notion of recursive structure. In Section 6, we address this redundancy. The syntax ought to be familiar to Clojure programmers:

$$\begin{array}{lr}
T, t ::= \ldots & \\
\quad \mid \mathsf{keyword} & \text{(keyword type)} \\
\quad \mid \mathsf{:k} & \text{(keyword)} \\
\quad \mid \mathsf{keywords} & \text{(keywords type)} \\
\quad \mid \mathsf{nil} & \text{(empty list of keywords)} \\
\quad \mid [t_0\ t_1] & \text{(cons cell of keywords)}
\end{array}$$

The typing rules are as expected:

$$\frac{\mathsf{:k}\ \text{any non-reserved keyword}}{\Gamma \vdash \mathsf{keyword} \ni \mathsf{:k}}$$

$$\frac{}{\Gamma \vdash \mathsf{keywords} \ni \mathsf{nil}} \qquad \frac{\Gamma \vdash \mathsf{keyword} \ni l \qquad \Gamma \vdash \mathsf{keywords} \ni ls}{\Gamma \vdash \mathsf{keywords} \ni [l\ ls]}$$

As earlier, our representation of lists of keywords is based on cons-cells but we can rely on some syntactic sugar (Section 3.2) to absorb the typical tuple notation. For example, translating Sharvit [22], we can define a sequence of authors as:

```
(defterm [authors :keywords]
  [:author/moore.alan
   :author/gibbons.dave])
```

4.2 Enumerations

Upon a statically-identified sequence of keywords, we define *struct types*. Struct types are the tag-less counterparts to Clojure's maps.

This requires introducing some dedicated syntax:

$$\begin{array}{lr}
T, t ::= \ldots & \\
\quad \mid (\text{enum } t) & \text{(finite type)} \\
\quad \mid 0 & \text{(index zero)} \\
\quad \mid (\text{suc } t) & \text{(successor index)} \\
\ \ e ::= \ldots & \\
\quad \mid (\text{struct } e \text{ :as } x \text{ :return } T) & \text{(finite function type)} \\
\quad \mid (\text{switch } e \text{ :as } x \text{ :return } T \text{ :with } t) & \text{(finite function application)}
\end{array}$$

In a struct type, the keywords only need to exist at compile-time (either as values or symbolically) whereas, in a map, the keywords are carried along at run-time. As a consequence, indexing into a struct type does not require a keyword either: the position of the data associated with the key is statically-known and amounts to a number, here encoded as a Peano number (0 and (suc)). In fact, indexing into a struct amounts to identifying a position in a C-like enumeration:

$$\frac{}{\Gamma \vdash (\text{enum } [v_0\ v_1]) \ni 0} \qquad \frac{\Gamma \vdash (\text{enum } v_1) \ni t}{\Gamma \vdash (\text{enum } [v_0\ v_1]) \ni (\text{suc } t)}$$

Note that neither the index nor the enumeration type are concerned with the actual keywords: only the *number* of keywords in the enumeration play a role here. However, one can extend the type-system to accept keywords as indices, whenever the keyword is found in In effect, the keyword :k would desugar to `(.indexOf v :k)` during its type-checking against the type (enum v).

The struct types amount to an association list of each keyword in the enumeration to a dedicated type:

$$\frac{\Gamma \vdash \text{keywords} \ni e \qquad \Gamma, x : (\text{enum } e) \vdash \text{:type} \ni T}{\Gamma \vdash (\text{struct } e \text{ :as } x \text{ :return } T) \in \text{:type}}$$

Given a struct type, we can project out the value at a specific index through a C-like `switch` statement:

$$\frac{\begin{array}{c}\Gamma \vdash e \in (\text{enum } v) \\ \Gamma, x : (\text{enum } v) \vdash \text{:type} \ni T \qquad \Gamma \vdash (\text{struct } v \text{ :as } x \text{ :return } T) \ni t\end{array}}{\Gamma \vdash (\text{switch } e \text{ :as } x \text{ :return } T \text{ :with } t) \in T[x \mapsto e]}$$

A dependently-typed counterpart to Sharvit [22], the `bookItem` object could be defined thus:

```
(defterm [bookItem :type]
  (struct [:id :rackId :isLent]
     :as x
     :return (switch x :as _ :return :type :with
                [(enum catalog-bookItem-id)
                 (enum catalog-racks)
                 bool])))
```

Assuming an index `book-item-1` in `(enum catalog-bookItem-id)` and `rack-17` in `(enum catalog-racks)`, we can populate an example of such bookItem:

```
(defterm [example-bookItem bookItem]
  [book-item-1 rack-17 true])
```

Dually, we determine whether the book was lent or not by dereferencing the third index:

```
(defterm [bookItem-get-isLent
             [b bookItem]
             bool]
  (switch (suc (suc 0)) as x
          return (switch x as _ return :type with
                 [(enum catalog-bookItem-id)
                  (enum catalog-racks)
                   bool]) with b))
```

Note, once again, keywords are absent at run-time: it is at compile-time that the return type of the `switch` is symbolically evaluated to confirm that the type denoted by the expression

```
(switch (suc (suc 0)) as _ return :type with
                 [(enum catalog-bookItem-id)
                  (enum catalog-racks)
                   bool])
```

indeed corresponds to the declared result of `bookItem-get-isLent`, namely `bool`.

5 INDUCTIVE TYPES

At this stage, our language of types does not support recursive definitions, hence cutting ourselves off from the ability to program with natural numbers, lists, trees, *etc.* We now extend our type theory to support inductive definitions.

We do so by extending our syntax with a grammar to describe type schemas. Indeed, to ensure logical coherence, dependent type theory does not allow us to manipulate types directly (for example, one cannot do a case analysis on types). To circumvent this limitation, we must first delineate the set of types we are interested in through a syntactic device. This is the role of the :desc type, whose constructors must be understood as "codes" to describe types:

$$\begin{array}{lr}
T, t ::= \ldots & \\
\quad \mid \text{:desc} & \text{(schema descriptions)} \\
\quad \mid [\text{:K } T] & \text{(type constant)} \\
\quad \mid \text{:X} & \text{(recursive argument)} \\
\quad \mid [\text{:prod } t_0\ t_1] & \text{(binary product)} \\
\quad \mid \ldots & \text{(cf. Appendix A)}
\end{array}$$

We relegate the definition of their formal static and dynamic semantics to the Appendix A. These codes are then *interpreted* into types through a schema interpretation function

$$\begin{array}{lr}
e ::= \ldots & \\
\quad \mid [\![t]\!]\ T & \text{(schema interpretation)}
\end{array}$$

$$\frac{\Gamma \vdash \text{:desc} \ni D \qquad \Gamma \vdash \text{:type} \ni X}{\Gamma \vdash [\![D]\!]\ X \in \text{:type}}$$

Through careful meta-theoretical analysis [7, 20], we know that taking the least fixpoint of such schemas keeps us in the realm of total functional programming [23]: despite the existence of recursive data-structure, our programs always terminate and, as a consequence, the logical consistency of our system is preserved (*i.e.*, the type-checker cannot get caught into an infinite loop or suddenly accept invalid judgements). One should therefore merrily add a generic fixpoint at the type-level and a generic datatype constructor at the term level:

$$\begin{aligned} T, t &::= \ldots \\ &\mid (\mu\ t_0\ t_1) && \text{(inductive fixpoint)} \\ &\mid [\mathrm{con}\ t_0\ t_1] && \text{(constructor)} \end{aligned}$$

We exploit the affordance of our presentation layer to expose a choice of constructor names by piggy-backing on the struct types:

$$\frac{\Gamma \vdash \mathrm{keywords} \ni l \qquad \Gamma \vdash (\mathrm{struct}\ l\ \mathrm{:as}\ \_\ \mathrm{:return}\ \mathrm{:desc}) \ni Ds}{\Gamma \vdash \mathrm{:type} \ni (\mu\ l\ Ds)}$$

$$\frac{\Gamma \vdash [\![[\mathtt{:struct}\ l\ Ds]]\!]\ (\mu\ l\ D) \ni [c\ xs]}{\Gamma \vdash (\mu\ l\ Ds) \ni [\mathrm{con}\ c\ xs]}$$

In such a system, the type of lists of (homogeneous) elements of type `A` could be manually coded as follows. First, we define the sequence of constructors:

```
(defterm [list-constr :labels]
  [:lnil :lcons])
```

For each of these, we associate a corresponding code. For the empty list constructor (tagged `:lnil`), there is no argument. For the cons-cell (tagged `:lcons`), there is a product of an argument of type `A` and a recursive argument:

```
(defterm [list-codes
          [A :type]
          (struct list-constr :desc)]
 [[:K :unit]
  [:prod [:K A] [:prod :X [:K :unit]]]])
```

Upon taking the fixpoint, this recursive argument itself becomes an inhabitant of the type list:

```
(defterm [list [A :type] :type]
 [:mu list-constr list-codes])
```

Obviously, we would not subject our users to writing such low-level codes. Instead, we offer the comfortable experience of a sum-of-product definition, thanks to a dedicated macro:

```
(defdata List "Inductive␣lists" [A :type]
 (lnil) (lcons [hd A] [tl <rec>]))
```

6 MORE FUN

Bootstrap. In Chapman et al. [5], it was observed that the type :desc is itself an inductive type and it could, modulo a definitional trick, be defined within itself. We could not resist the temptation to repeat this process in our setting: we have therefore bootstrapped the definition of description onto themselves, following the instructions exposed by the authors. Through a similar but careful process, we can represent the type keywords as a mere `List :keyword`.

Aside from the reductionist thrill of distilling our type theory to the bare minimal, this also brings out some practical advantages. For the grammar of descriptions (constructors of :desc) to have the same representation as defined inductive types (constructors of $(\mu\ l\ s)$ cs for some keywords *ls* and associated codes *cs*) means that any generic program over inductive types immediately applies specifically to the grammar of descriptions.

One such example is *case analysis*: one can write a program[10] that takes any pair of labels `ls` together with their associated descriptions `cs` and computes a generic case analysis principle for the resulting fixpoint, by recursively enumerating the type of arguments and results of each constructor case.

Typed data-oriented programming. To exercise our system[11], we attempted to give a dependently-typed presentation of the Chapter 3, entitled "Manipulate the whole system data with generic functions" of the book “Data-oriented programming” by Sharvit [22]. Following our examples in Section 4, we were easily[12] able to assign dependently-typed schemas to the various entities introduced in Chapter 3. However, the grand challenge would be to support the idiom

```
_.get(catalogData, ["booksByIsbn", "978-1779501127"]
```

used in the book to chain projections (first accessing field `booksByIsbn` from `catalogData` followed with accessing field `978-1779501127` from the result extracted by the previous projection). In our setting, these projections have to be resolved at compile-time, thus making sure that subsequent projections are indeed allowed. To model this recursive telescope of dependent types, one need to step beyond inductive types, calling for inductive *families* that would allow us to encode the evolution of types along the sequence of projections. Unfortunately, indexed families are intimately tied with the notion of (propositional) equality, which we had carefully avoided thus far for the significant complexity burden it adds to a type theory.

7 CONCLUSION

Dependent types and Lisp are, we believe, perfect complements. The level of interactivity provided by Lisp systems in general, and by Clojure/Cider in particular provides some support to program with dependent-types. There is still much room for improvement in the system. One feature that is for now missing is the possibility to erase types in the internal representation of Deputy definitions. For example, a Deputy function is not directly compiled as a Clojure function, which means that even closed terms have to be interpreted through normalization. We also intend to work on other performance-related issues in the implementation. The management and presentation of (dependent) type error is another major concern while developing system that perform complex type-level computations. For this, we think that working at the Emacs/Cider level would provide the best angle of attack to address this issue.

ACKNOWLEDGMENTS

The authors would like to thank Téo Bernier, a very fine and talented ex-student (now a talented research engineer) who took an important part of the develoment of the system a few years back.

[10] Highly non-trivial one! See deputy/stdlib/generics.clj in the associated project.

[11] See test/deputy/data_oriented_deputy.clj in the associated project.

[12] However, it was frustratingly verbose: syntactic support to lighten the definition of finite maps ought to be within reach.

REFERENCES

[1] Marcin Benke, Peter Dybjer, and Patrik Jansson. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.*, 10(4):265–289, 2003.

[2] François Bourdoncle. Abstract debugging of higher-order imperative languages. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 46–55. ACM, 1993. doi: 10.1145/155090.155095. URL https://doi.org/10.1145/155090.155095.

[3] Rod M. Burstall, David B. MacQueen, and Donald Sannella. HOPE: an experimental applicative language. In *Proceedings of the 1980 LISP Conference, Stanford, California, USA, August 25-27, 1980*, pages 136–143. ACM, 1980. doi: 10.1145/800087.802799. URL https://doi.org/10.1145/800087.802799.

[4] Giuseppe Castagna. Covariance and controvariance: a fresh look at an old issue (a primer in advanced type systems for learning functional programmers). *CoRR*, abs/1809.01427, 2018. URL http://arxiv.org/abs/1809.01427.

[5] James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 3–14. ACM, 2010. doi: 10.1145/1863543.1863547. URL https://doi.org/10.1145/1863543.1863547.

[6] Thierry Coquand. An algorithm for type-checking dependent types. *Sci. Comput. Program.*, 26(1-3):167–177, 1996. doi: 10.1016/0167-6423(95)00021-6. URL https://doi.org/10.1016/0167-6423(95)00021-6.

[7] Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988. doi: 10.1007/3-540-52335-9_47. URL https://doi.org/10.1007/3-540-52335-9_47.

[8] Agda development team. Hurkens paradox. URL https://github.com/agda/agda/blob/master/test/Succeed/Hurkens.agda.

[9] Jana Dunfield and Neel Krishnaswami. Bidirectional typing. *ACM Comput. Surv.*, 54(5):98:1–98:38, 2022. doi: 10.1145/3450952. URL https://doi.org/10.1145/3450952.

[10] Timothy S. Freeman and Frank Pfenning. Refinement types for ML. In David S. Wise, editor, *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*, pages 268–277. ACM, 1991. doi: 10.1145/113445.113468. URL https://doi.org/10.1145/113445.113468.

[11] Andreas Garnaes. Type-safe graphql with ocaml, 2017. URL https://andreas.github.io/2017/11/29/type-safe-graphql-with-ocaml-part-1/.

[12] Jacques Garrigue and Didier Rémy. Semi-explicit first-class polymorphism for ML. *Inf. Comput.*, 155(1-2):134–169, 1999. doi: 10.1006/INCO.1999.2830. URL https://doi.org/10.1006/inco.1999.2830.

[13] Karoliine Holter, Juhan Oskar Hennoste, Simmo Saan, Patrick Lam, and Vesal Vojdani. Abstract debugging with gobpie. In Elisa Gonzalez Boix and Christophe Scholliers, editors, *Proceedings of the 2nd ACM International Workshop on Future Debugging Techniques, DEBT 2024, Vienna, Austria, 19 September 2024*, pages 32–33. ACM, 2024. doi: 10.1145/3678720.3685320. URL https://doi.org/10.1145/3678720.3685320.

[14] Antonius J. C. Hurkens. A simplification of girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Typed Lambda Calculi and Applications, Second International Conference on Typed Lambda Calculi and Applications, TLCA '95, Edinburgh, UK, April 10-12, 1995, Proceedings*, volume 902 of *Lecture Notes in Computer Science*, pages 266–278. Springer, 1995. doi: 10.1007/BFB0014058. URL https://doi.org/10.1007/BFb0014058.

[15] Per Martin-Löf. An intuitionistic theory of types. unpublished preprint, 1971.

[16] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in proof theory*. Bibliopolis, 1984. ISBN 978-88-7088-228-5.

[17] Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, UK, 2000. URL https://hdl.handle.net/1842/374.

[18] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004. doi: 10.1017/S0956796803004829. URL https://doi.org/10.1017/S0956796803004829.

[19] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Easing maintenance of academic static analyzers. *Int. J. Softw. Tools Technol. Transf.*, 26(6):673–686, 2024. doi: 10.1007/S10009-024-00770-1. URL https://doi.org/10.1007/s10009-024-00770-1.

[20] Peter Morris, Thorsten Altenkirch, and Neil Ghani. A universe of strictly positive families. *Int. J. Found. Comput. Sci.*, 20(1):83–107, 2009. doi: 10.1142/S0129054109006462. URL https://doi.org/10.1142/S0129054109006462.

[21] Nicolas Oury and Wouter Swierstra. The power of pi. In James Hook and Peter Thiemann, editors, *Proceeding of the 13th ACM SIGPLAN international conference on Functional programming, ICFP 2008, Victoria, BC, Canada, September 20-28, 2008*, pages 39–50. ACM, 2008. doi: 10.1145/1411204.1411213. URL https://doi.org/10.1145/1411204.1411213.

[22] Yehonathan Sharvit. *Data-Oriented Programming: Reduce software complexity*. Manning, 2022. ISBN 9781638356783.

[23] D. A. Turner. Total functional programming. *J. Univers. Comput. Sci.*, 10(7):751–768, 2004. doi: 10.3217/JUCS-010-07-0751. URL https://doi.org/10.3217/jucs-010-07-0751.

[24] A.N. Whitehead and B. Russell. *Principia Mathematica*, volume 1-3. Cambridge University Press, 1910–1913.

A THE TYPE THEORY OF DEPUTY

Syntax:

$T, t ::=$	(canonical types & terms)
$\mid \texttt{:type}$	(type of types)
$\mid e$	(stuck term)
$\mid \texttt{(}\Pi\ \texttt{[}x\ T_0\texttt{]}\ T_1\texttt{)}$	(Pi type)
$\mid \texttt{(}\lambda\ \texttt{[}x\texttt{]}\ t\texttt{)}$	(abstraction)
$\mid \texttt{:unit}$	(unit type)
$\mid \texttt{nil}$	(unit value)
$\mid \texttt{(}\Sigma\ \texttt{[}x\ T_0\texttt{]}\ T_1\texttt{)}$	(Sigma type)
$\mid \texttt{[}t_0\ t_1\texttt{]}$	(pair)
$\mid \texttt{keyword}$	(keyword type)
$\mid \texttt{:k}$	(keyword)
$\mid \texttt{keywords}$	(keywords type)
$\mid \texttt{nil}$	(empty list of keywords)
$\mid \texttt{[}t_0\ t_1\texttt{]}$	(cons cell of keywords)
$\mid \texttt{(enum}\ t\texttt{)}$	(finite type)
$\mid \texttt{0}$	(index zero)
$\mid \texttt{(suc}\ t\texttt{)}$	(successor index)
$\mid \texttt{:desc}$	(schema descriptions)
$\mid \texttt{[:K}\ T\texttt{]}$	(type constant)
$\mid \texttt{:X}$	(recursive argument)
$\mid \texttt{[:struct}\ e\ t\texttt{]}$	(finite sum)
$\mid \texttt{[:sigma}\ T\ t\texttt{]}$	(large sum)
$\mid \texttt{[:prod}\ t_0\ t_1\texttt{]}$	(binary product)
$\mid \texttt{[:pi}\ T\ t\texttt{]}$	(large product)
$\mid \texttt{(}\mu\ t_0\ t_1\texttt{)}$	(inductive fixpoint)
$\mid \texttt{[con}\ t_0\ t_1\texttt{]}$	(constructor)
$e ::=$	(computational terms)
$\mid \texttt{(the}\ T\ t\texttt{)}$	(type annotation)
$\mid x$	(variable)
$\mid \texttt{(}e\ t\texttt{)}$	(application)
$\mid \texttt{(}\pi_0\ e\texttt{)}$	(first projection)
$\mid \texttt{(}\pi_1\ e\texttt{)}$	(second projection)
$\mid \texttt{(struct}\ e\ \texttt{:as}\ x\ \texttt{:return}\ T\texttt{)}$	(finite function type)
$\mid \texttt{(switch}\ e\ \texttt{:as}\ x\ \texttt{:return}\ T\ \texttt{:with}\ t\texttt{)}$	(finite function application)
$\mid [\![t]\!]\ T$	(schema interpretation)
$\Gamma ::=$	(contexts)
$\mid \epsilon$	(empty context)
$\mid \Gamma, x : T$	(variable declaration)

Small-step reduction:

$$
\begin{aligned}
(\text{the } T\ t) &\rightsquigarrow t\\
((\lambda\ [x]\ t)\ t_0) &\rightsquigarrow t[x \mapsto t_0]\\[1em]
(\pi_0\ [t_0\ t_1]) &\rightsquigarrow t_0\\
(\pi_1\ [t_0\ t_1]) &\rightsquigarrow t_1\\[1em]
(\text{struct nil :as } x \text{ :return } T) &\rightsquigarrow \text{:unit}\\
(\text{struct } [t_0\ t_1] \text{ :as } x \text{ :return } T) &\rightsquigarrow (*\ (T[x \mapsto 0])\\
&\qquad (\text{struct } t_1 \text{ :as } x \text{ :return}(T[x \mapsto (\text{suc } x)])))\\
(\text{switch } 0 \text{ :as } x \text{ :return } T \text{ :with } [t_0\ t_1]) &\rightsquigarrow t_0\\
(\text{switch } (\text{suc } t) \text{ :as } x \text{ :return } T \text{ :with } [t_0\ t_1]) &\rightsquigarrow (\text{switch } t \text{ :as } x \text{ :return}(T[x \mapsto (\text{suc } x)]) \text{ :with } t_1)\\[1em]
\llbracket [\texttt{:K}\ T] \rrbracket\ X &\rightsquigarrow T\\
\llbracket \texttt{:X} \rrbracket\ X &\rightsquigarrow X\\
\llbracket [\texttt{:struct}\ l\ Ds] \rrbracket\ X &\rightsquigarrow (\Sigma\ [e\ (\text{enum } l)]\ \ \llbracket (\text{switch } e \text{ :as \_ :return :desc :with } Ds) \rrbracket\ X)\\
\llbracket [\texttt{:sigma}\ S\ D] \rrbracket\ X &\rightsquigarrow (\Sigma\ [s\ S]\ \ \llbracket (D\ s) \rrbracket\ X)\\
\llbracket [\texttt{:prod}\ D_1\ D_2] \rrbracket\ X &\rightsquigarrow (*\ (\llbracket D_1 \rrbracket\ X)\ (\llbracket D_2 \rrbracket\ X))\\
\llbracket [\texttt{:pi}\ S\ D] \rrbracket\ X &\rightsquigarrow (\Pi\ [s\ S]\ (\llbracket D\ s \rrbracket\ X))
\end{aligned}
$$

$$\boxed{\Gamma \vdash T \ni t}$$

"Type T admits term t in context Γ"

$$\frac{}{\Gamma \vdash \text{:type} \ni \text{:type}}\ \textsc{Type-in-Type} \qquad \frac{\Gamma \vdash e \in T_0 \qquad \Gamma \vdash T_0 \equiv T_1 : \text{:type}}{\Gamma \vdash e \ni T_1}$$

$$\frac{\Gamma \vdash \text{:type} \ni T_0 \qquad \Gamma, x : T_0 \vdash \text{:type} \ni T_1}{\Gamma \vdash \text{:type} \ni (\Pi\ [x\ T_0]\ T_1)} \qquad \frac{\Gamma, x : T_0 \vdash T_1 \ni t}{\Gamma \vdash (\Pi\ [x\ T_0]\ T_1) \ni (\lambda\ [x]\ t)}$$

$$\frac{}{\Gamma \vdash \text{:type} \ni \text{:unit}} \qquad \frac{}{\Gamma \vdash \text{:unit} \ni \text{nil}} \qquad \frac{\Gamma \vdash \text{:type} \ni T_0 \qquad \Gamma, x : T_0 \vdash \text{:type} \ni T_1}{\Gamma \vdash \text{:type} \ni (\Sigma\ [x\ T_0]\ T_1)} \qquad \frac{\Gamma \vdash T_0 \ni t_0 \qquad \Gamma \vdash T_1[x \mapsto t_0] \ni t_1}{\Gamma \vdash (\Sigma\ [x\ T_0]\ T_1) \ni [t_0\ t_1]}$$

$$\frac{}{\Gamma \vdash \text{:type} \ni \text{keyword}} \qquad \frac{\text{:k any non-reserved keyword}}{\Gamma \vdash \text{keyword} \ni \text{:k}} \qquad \frac{}{\Gamma \vdash \text{:type} \ni \text{keywords}} \qquad \frac{}{\Gamma \vdash \text{keywords} \ni \text{nil}}$$

$$\frac{\Gamma \vdash \text{keyword} \ni l \qquad \Gamma \vdash \text{keywords} \ni ls}{\Gamma \vdash \text{keywords} \ni [l\ ls]}$$

$$\frac{\Gamma \vdash \text{keywords} \ni t}{\Gamma \vdash \text{:type} \ni (\text{enum}\ t)} \qquad \frac{}{\Gamma \vdash (\text{enum}\ [v_0\ v_1]) \ni 0} \qquad \frac{\Gamma \vdash (\text{enum}\ v_1) \ni t}{\Gamma \vdash (\text{enum}\ [v_0\ v_1]) \ni (\text{suc}\ t)}$$

$$\frac{}{\Gamma \vdash \text{:type} \ni \text{:desc}} \qquad \frac{\Gamma \vdash \text{:type} \ni T}{\Gamma \vdash \text{:desc} \ni [\text{:K}\ T]} \qquad \frac{}{\Gamma \vdash \text{:desc} \ni \text{:X}} \qquad \frac{\begin{array}{c}\Gamma \vdash \text{keywords} \ni l \\ \Gamma \vdash (\text{struct}\ l\ \text{:as}\ \_\ \text{:return :desc}) \ni Ds\end{array}}{\Gamma \vdash \text{:desc} \ni [\text{:struct}\ l\ Ds]} \qquad \frac{\begin{array}{c}\Gamma \vdash \text{:desc} \ni D_1 \\ \Gamma \vdash \text{:desc} \ni D_2\end{array}}{\Gamma \vdash \text{:desc} \ni [\text{:prod}\ D_1\ D_2]}$$

$$\frac{\begin{array}{c}\Gamma \vdash \text{:type} \ni S \\ \Gamma \vdash (\Rightarrow S\ \text{:desc}) \ni D\end{array}}{\Gamma \vdash \text{:desc} \ni [\text{:sigma}\ S\ D]} \qquad \frac{\begin{array}{c}\Gamma \vdash \text{:type} \ni S \\ \Gamma \vdash (\Rightarrow S\ \text{:desc}) \ni D\end{array}}{\Gamma \vdash \text{:desc} \ni [\text{:pi}\ S\ D]} \qquad \frac{\Gamma \vdash \text{keywords} \ni l \qquad \Gamma \vdash (\text{struct}\ l\ \text{:as}\ \_\ \text{:return :desc}) \ni Ds}{\Gamma \vdash \text{:type} \ni (\mu\ l\ Ds)}$$

$$\frac{\Gamma \vdash [\![[\text{:struct}\ l\ Ds]]\!]\ (\mu\ l\ D) \ni [c\ xs]}{\Gamma \vdash (\mu\ l\ Ds) \ni [\text{con}\ c\ xs]}$$

$$\boxed{\Gamma \vdash e \in T}$$

"Term t synthesizes type T in context Γ"

$$\frac{}{\Gamma_0, x : T, \Gamma_1 \vdash x \in T} \qquad \frac{\Gamma \vdash \text{:type} \ni T \qquad \Gamma \vdash T \ni t}{\Gamma \vdash (\text{the}\ T\ t) \in T}$$

$$\frac{\Gamma \vdash e \in (\Pi\ [x\ T_0]\ T_1) \qquad \Gamma \vdash T_1 \ni t}{\Gamma \vdash (e\ t) \in T_1[x \mapsto t]}$$

$$\frac{\Gamma \vdash e \in (\Sigma\ [x\ t_0]\ T)}{\Gamma \vdash (\pi_0\ e) \in T_0} \qquad \frac{\Gamma \vdash e \in (\Sigma\ [x\ T_0]\ T_1)}{\Gamma \vdash (\pi_1\ e) \in T_1[x \mapsto (\pi_0\ e)]}$$

$$\frac{\Gamma \vdash \text{keywords} \ni e \qquad \Gamma, x : (\text{enum}\ e) \vdash \text{:type} \ni T}{\Gamma \vdash (\text{struct}\ e\ \text{:as}\ x\ \text{:return}\ T) \in \text{:type}}$$

$$\frac{\Gamma \vdash e \in (\text{enum}\ v) \qquad \Gamma, x : (\text{enum}\ v) \vdash \text{:type} \ni T \qquad \Gamma \vdash (\text{struct}\ v\ \text{:as}\ x\ \text{:return}\ T) \ni t}{\Gamma \vdash (\text{switch}\ e\ \text{:as}\ x\ \text{:return}\ T\ \text{:with}\ t) \in T[x \mapsto e]}$$

$$\frac{\Gamma \vdash \text{:desc} \ni D \qquad \Gamma \vdash \text{:type} \ni X}{\Gamma \vdash [\![D]\!]\ X \in \text{:type}}$$

Programming with Useful Quantifiers

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

ABSTRACT

Many programming languages have functions which implement universal and existential quantifiers. Typically these are implemented as higher order functions which accept a Boolean predicate and some sort of iterable and return a Boolean value indicating whether a predicate is validated over the iterable or rather, whether a counterexample or witness was encountered. These quantifiers are elegant to use and allow application code to be written which closely resembles axiomatic algebraic specifications. A disadvantage is seen when unexpected results occur, *e.g.*, when attempting to debug user defined predicates. These quantifiers do not have easy ways of returning the witness or counterexample as such would typically violate the expected Boolean return value, and thus modify the program flow or in some cases cause compilation errors. Users are usually forced to refactor code to avoid the elegant quantifiers in order to debug their code. We present implementations of universal and existential quantifiers in Clojure, Common Lisp, Python, and Scala, which conceptually extend the Boolean type with decorations which are useful not only for debugging but for mathematical purposes.

CCS CONCEPTS

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

KEYWORDS

common lisp,clojure,scala,python,quantifiers

1 OVERVIEW

In this article we present *Heavy Booleans*, and their implementation in Clojure [8], Common Lisp [1, 12], Python [13], and Scala [9, 10]. These data structures are values which on the one hand implement Boolean (*true*/*false*) semantics, and on the other hand provide reflective behavior [6]; *i.e.*, giving access to the witness of an existential quantifier, $\exists a \in M, p(a)$, or counterexample of a universal quantifier [2], $\forall a \in M, p(a)$.

Heavy Booleans convey information about the counterexample when universal quantification fails. If $\forall a \in M, p(a)$ returns *false*, the user should be able to ask which counterexample value of M made p *false*. Dually, if $\exists a \in M, p(a)$ returns *true*, the user asks which value of M satisfied p.

When we designed the system, we had several goals in mind. The heavy Boolean values should (as much as possible) behave like Boolean values in Boolean contexts, but should contain reflective data for debugging or for providing mathematical semantics. Furthermore, and perhaps most importantly, these Heavy Boolean values should compose with each other through the Boolean operations of conjunction, disjunction, and negation, and through concentric use of multiple quantifiers of uniform or mixed types without loosing critical information.

These ambitious goals provide implementation challenges depending on the programming language, as different languages handle Boolean values differently in primitives such as `and`, `or`, `not`, `if`, *etc.*. In the Scala and Python languages, we have added behavior to the Heavy Boolean objects themselves so that they behave in a reasonable way with the built-in equivalents of `if`, `and`, and `or`. However, in the Clojure and Common Lisp languages, we are not able to augment the behavior of `false` and instead we have created heavy-Boolean-friendly variants of `if`, `and`, and `or`.

On the other hand, using the meta-programming capabilities of the lisp dialects (Common Lisp and Clojure) we were able to create a more expressive `exists` and `forall` than in the non-lisp languages. In the Scala and Python versions of `exists` and `forall`, the user is required to provide redundant information. This redundant information is obviated in the lisp dialects thanks to their meta-programming layers.

2 CONTRIBUTIONS

Our contributions in this article are libraries for Clojure, Common Lisp, Python, and Scala implementing the Heavy Boolean protocols.

- Clojure: https://github.com/jimka2001/heavybool/clojure
- Common Lisp: https://github.com/jimka2001/heavybool/common-lisp
- Python: https://github.com/jimka2001/heavybool/python
- Scala: https://github.com/jimka2001/heavybool/scala

3 QUANTIFIERS IN VARIOUS LANGUAGES

All programming languages which we are concerned with have built-in predicate functions which implement universal (1) and existential (2) quantifiers which satisfy DeMorgan's law (3). This law allows any proposition stated in terms of a universal quantifier to be refactored into an equivalent proposition stated in terms of an existential quantifier and vice versa.

ELS'25, May 19–20 2025, Zurich

https://doi.org/10.5281/zenodo.14775258

$$\textbf{Universal Quantifier:} \quad \forall a \in M, p(a) \tag{1}$$

$$\textbf{Existential Quantifier:} \quad \exists a \in M, p(a) \tag{2}$$

$$\textbf{DeMorgan's Law:} \quad \exists a \in M, p(a) \iff \neg\forall a \in M, \neg p(a) \tag{3}$$

In Sections 3.1 through 3.4, we look at the syntax of quantifiers and DeMorgan's law in four programming languages.

3.1 Clojure

In Clojure the existential quantifier, $\exists a \in M, p(a)$, is implemented by the built-in `some` function as on lines 3 and 2 of Listing 1. The universal quantifier, $\forall a \in M, p(a)$, is expressed by the built-in `every?` function as on lines 7 and 6 of Listing 1. Furthermore, by DeMorgan's law, Equation (3), lines 10, 11, and 12 are equivalent.

Listing 1: Clojure: Existential and Universal Quantifiers

```
1  ;; existential quantifiers
2  (some p M)
3  (some (fn [a] (p a)) M)
4
5  ;; universal quantifiers
6  (every? p M)
7  (every? (fn [a] (p a)) M)
8
9  ;; equivalent expressions: DeMorgan's Law
10 (some p M)
11 (not (every? (fn [a] (not (p a))) M))
12 (not (every? (complement p) M))
```

3.2 Common Lisp

In Common Lisp the existential quantifier, (2), is implemented by the built-in `some` function as on lines 3 and 2 in Listing 2. The universal quantifier, (1), is expressed by the built-in `every` function as on lines 7 and 6 in the same listing. Furthermore, DeMorgan's law, Equation (3), is verified in that the following expressions on lines 10, 11, and 12 are logically equivalent.

Listing 2: Common Lisp: Existential & Universal Quantifiers

```
1  ;; existential quantifiers
2  (some #'p M)
3  (some #'(lambda (a) (p a)) M)
4
5  ;; universal quantifiers
6  (every #'p M)
7  (every #'(lambda (a) (p a)) M)
8
9  ;; equivalent expressions: DeMorgan's Law
10 (some #'p M)
11 (not (every #'(lambda (a) (not (p a))) M))
12 (not (every (complement #'p) M))
```

3.3 Python

In Python, the built-in functions `all` and `any` implement the universal and existential quantifiers respectively. An example of the existential quantifier, (2), is shown on line 2 of Listing 3. An example of the universal quantifier, (1), is shown in the same listing on line 5. The `all` and `any` functions are related by DeMorgan's Law in that lines 8 and 9 are equivalent.

Listing 3: Python: Existential and Universal Quantifiers

```
1  # existential quantifier
2  any(p(a) for a in M)
3
4  # universal quantifier
5  all(p(a) for a in M)
6
7  # equivalent expressions: DeMorgan's Law
8  all(p(a) for a in M)
9  not any(not p(a) for a in M)
```

3.4 Scala

In Scala the existential quantifier is implemented as a built-in method `exists` defined on the `Seq` class (and on other classes), while the universal quantifier is implemented as the built-in method `forall`. The existential quantifier, can be written as on lines 3 and 2 in Listing 4. The universal quantifier can be written as on lines 7 and 6. As in the three previous examples, DeMorgan's law, Equation (3), is expressed once again (lines 11 and 10).

Listing 4: Scala: Existential and Universal Quantifiers

```
1  // equivalent existential quantifiers
2  M.exists(p)
3  M.exists{(a) => p(a)}
4
5  // equivalent universal quantifiers
6  M.forall(p)
7  M.forall{(a) => p(a)}
8
9  // equivalent expressions: DeMorgan's Law
10 M.exists(p)
11 !M.forall{(a) => !p(a)}
```

4 ASSOCIATIVITY EXAMPLE

In Section 3 we examined the simple use of quantifiers. In each case the quantifier returns a Boolean value which lacks reflective capacity [6]. In this section we look at examples where an unadorned Boolean value does not suffice. In each case we first look at some standard techniques for working around this limitation, and in Section 6 we introduce the *Heavy Boolean* as a generic solution.

In this section we consider a pedagogical mathematical example from abstract algebra. A talented researcher, Helen Wheels[1], is writing some code to detect whether a given binary operation, $\circ$, is associative on a given finite set, M. The proposition to verify is stated succinctly, mathematically as

$$\forall a \in M, \forall b \in M, \forall c \in M \; (a \circ b) \circ c = a \circ (b \circ c). \tag{4}$$

For example if $M = \{1, 2, 3\}$ and the operation is subtraction, although other solutions are possible, it would suffice that Helen

[1] https://www.youtube.com/watch?v=RSWQ0tkxtdk, in real life Hellen Wheels is a talented comedian who inspires others to excel beyond what some people see as a handicap.

find the pair $(1, 1, 2)$ because

$$-2 = (1 - 1) - 2 \neq 1 - (1 - 2) = 2\,.$$

4.1 Technique 1: Using Universal Quantifiers

Helen, being a formidable, polyglot programmer, implements Equation (4) in several programming languages as follows:

Listing 5: Clojure: Implementation of Eq (4)

```
(every? (fn [x]
   (every? (fn [y]
      (every? (fn [z] (= (op (op x y) z)
                         (op x (op y z))))
              M))
      M))
   M)
```

Listing 6: Common Lisp: Implementation of Eq (4)

```
(every #'(lambda (x)
  (every #'(lambda (y)
    (every #'(lambda (z)
               (equal (op (op x y) z)
                      (op x (op y z))))
          M))
  M)
```

Listing 7: Python: Implementation of Eq (4)

```
all(op(op(x,y), z) == op(x, op(y,z))
    for x in M
    for y in M
    for z in M)
```

Listing 8: Scala: Implementation of Eq (4)

```
M.forall{(x) =>
  M.forall{(y) =>
    M.forall{(z) =>
      op(op(x,y),z) == op(x,op(y,z))}}}
```

Being an expert programmer, Helen always tests her code thoroughly. In this case she tests on a particular example and finds that it surprisingly returns `false`. Helen does not know if the operation really fails to be associative or whether her implementation has a bug. She knows that for some triple (x, y, z), the operation (as coded) is not associative. How can she determine which (x, y, z) triple serves as a counterexample so she can look more closely to understand the discrepancy?

4.2 Technique 2: Using Existential Quantifiers

In Section 4.1, Helen used the built-in universal quantifiers in each programming language. Thus *true* is returned when the operation is found to be associative. However, Helen's code detected a failure to be associative; *false* was returned. Helen attempts to apply DeMorgan's law (Eq (3)) to her code, inverting the logic so that it returns *true* when the operation is not associative. The advantage of a *true* return value is that Helen obtains information in addition to the Boolean *true*. For example a triple (x, y, z) would be *true*.

Helen exploits the fact that the triple (x, y, z) is *true* in the lisp dialects, Clojure and Common Lisp as shown in Listings 5 and 6. However, she finds that the technique (Listings 7 and 8) fails for Python and Scala.

Listing 9: Clojure Listing 5 with Inverted Logic

```
(some (fn [x]
   (some (fn [y]
      (some (fn [z] (and (not (= (op x (op y z))
                                  (op (op x y) z)))
                      [x y z]))
              M))
      M))
   M)
```

Listing 10: Common Lisp Listing 6 with Inverted Logic

```
(some #'(lambda (x)
  (some #'(lambda (y)
    (some #'(lambda (z)
              (and (not (equal (op x (op y z))
                                (op (op x y) z)))
                (list x y z)))
        M))
  M)
```

Listing 11: Broken Python Listing 7 with Inverted Logic

```
# returns True or False, not the counterexample
any(op(op(x,y), z) != op(x, op(y,z)) and [x, y, z]
    for x in M
    for y in M
    for z in M)

# equivalent but more idiomatic
any([x, y, z]
    for x in M
    for y in M
    for z in M
    if op(op(x,y), z) != op(x, op(y,z)))
```

Listing 12: Broken Scala Listing 8 with Inverted Logic

```
1 // This code does not compile
2 M.exists{(x) =>
3   M.exists{(y) =>
4     M.exists{(z) =>
5       op(op(x,y),z) != op(x,op(y,z)) && (x,y,z)}}}`
```

The Python Listing 11 fails because, as documented, `any` returns `True` or `False` explicitly. In contrast to the lisp dialects, `any` unfortunately does not return the first *true* result it encounters.

The Scala Listing 12 fails to compile, because the function Helen has given to `M.exists...` (line 5) is expected to return a `Boolean`; whereas the tuple `(x,y,z)` is not a `Boolean`.

4.3 Technique 3: Sequence of Counterexamples

Helen, being dedicated and determined to find a solution, tries to refactor the code so that it returns an empty sequence if the operation is associative, otherwise a sequence of counterexamples if the operation fails to be associative.

This technique works for all the languages shown but with a notable disadvantage. Helen is generating exceedingly many counterexamples, when one would suffice to disprove associativity. The search for counterexamples is an n^3 search. Helen would like to terminate the search early once a counterexample is found.

Listing 13: Clojure: Sequence of Counterexamples

```
(for [x M
      y M
      z M
      :when (not (= (op (op x y) z)
                    (op x (op y z))))]
  [x y z])
```

Listing 14: Common Lisp: Sequence of Counterexamples

```
(loop :for x :in M
  :nconc (loop :for y :in M
           :nconc (loop :for z :in M
                    :unless (equal (op (op x y) z)
                                   (op x (op y z)))
                      :collect (list x y z))))
```

Listing 15: Python: Sequence of Counterexamples

```
[(x,y,z) for x in M
         for y in M
         for z in M
         if op(op(x,y), z) != op(x, op(y,z))]
```

Listing 16: Scala: Sequence of Counterexamples

```
for{x <- M
    y <- M
    z <- M
    if op(op(x,y),z) != op(x,op(y,z))
   } yield (x,y,z)
```

4.4 Technique 4: At Most One Counterexample

Not yet admitting defeat, Helen alters Listings 13 through 16 to terminate the loop on finding the first counterexample.

She again considers Listing 13. This expression, thanks to the `for` construct, returns a so-called *lazy-sequence* of all possible counterexamples. Helen tests whether the returned sequence is empty, and if not, calls `first` to obtain an actual counterexample. Unfortunately, Helen finds that so-called lazy-sequences in Clojure are greedily-lazy and sadly Clojure has continued to compute several additional counterexamples even though only the first one is consumed at the call site. Clojure experts do not consider this greedy-laziness to be a problem in practice—claiming that is it *usually* not a problem, which we interpret (via Demorgan's Law) to mean: sometimes *IS* a problem.

Helen, determined more than ever, addresses the problem of greedy-lazy-sequences by employing so-called transducers in Listing 17. This refactoring further exacerbates the extent to which the original code and the debugging code are asymmetric.

Listing 17: Clojure: Return one Counterexample

```
(->> (for [x M
           y M
           z M]
       [x y z])
     (transduce (comp (filter (fn [[x y z]]
                                (not (= (op x (op y z))
                                        (op (op x y) z)))))
                      (take 1))
                conj []))
```

On to the next programming language, Helen refactors Listing 14 using `block` / `return-from` to avoid collecting. This Common Lisp code now returns `nil` if the operation is associative; otherwise it returns a triple, indicating the first counterexample encountered.

Listing 18: Common Lisp: Return one Counterexample

```
(block nil
  (loop
    :for x :in M
    :do (loop
           :for y :in M
           :do (loop :for z :in M
                 :unless (equal (op (op x y) z)
                                (op x (op y z)))
                    :do (return-from nil
                           (list x y z))))))
```

For the next programming language, Python, Helen replaces the list comprehension from Listing 15 with a call to `next` Listing 19. The code now returns `None` if the operation is associative; otherwise it returns a triple which is the first counterexample encountered.

Listing 19: Python: Return one Counterexample

```
next(((x,y,z) for x in M
              for y in M
              for z in M
              if op(op(x,y), z) != op(x, op(y,z))),
     None)
```

Finally, Helen attacks the Scala Listing 16. She converts the input sequence `M`, to a lazy list with a call to `.to(LazyList)`. Now, the code in Listing 20 returns a possibly empty, lazy list. The lazy list is empty if the operation is associative, otherwise the call-site should invoke the method `.first` to obtain a counterexample.

Listing 20: Scala: Return one Counterexample

```
val lazyM = M.to(LazyList)
for{x <- lazyM
    y <- lazyM
    z <- lazyM
    if op(op(x,y),z) != op(x,op(y,z))
   } yield (x,y,y)
```

4.5 Summary of Standard Techniques

Each of the techniques which Helen has tried solves part of the problem, but there are significant disadvantages.

- In Technique 3 (Section 4.3) Helen is generating exceedingly many counterexamples, when one would suffice to disprove associativity.
- The code in Technique 4 has drastically diverged from Helen's first attempt in Listings 5 through 8.
- The code no longer resembles the elegant Equation 4 which Helen is attempting to verify.

5 HEAVY BOOLEAN SEMANTICS

We briefly and tersely describe the semantics of so-called *Heavy Booleans* and in the following sections give examples of their implement in various programming languages.

Let $\mathcal{H}$ be the set called *Heavy Booleans*. Partition $\mathcal{H}$ into $\mathcal{H}=\mathcal{T}\cup\mathcal{F}$. Call $\mathcal{T}$ the *truthy* values and $\mathcal{F}$ the *falsey* values. Let $t\in\mathcal{T}$, $f\in\mathcal{F}$, and $b\in\mathcal{H}$. Let S be an arbitrary set, and $h{::}s$ be a finite sequence of elements of S whose first element is h and remaining elements are the sequence s. Let $[]$ denote the empty sequence. Let $p:S\rightarrow\mathcal{H}$ be a so-called *heavy predicate* function.

Let $\mathcal{W}\supset\{\emptyset\}$ be a set of so-called *reasons*, and $why{:}\mathcal{H}{\rightarrow}\mathcal{W}$ be a function for which $why{:}\mathcal{T}{\rightarrow}\mathcal{W}$ and $why{:}\mathcal{F}{\rightarrow}\mathcal{W}$ are both bijections. Call $why(t)$ a *witness* and $why(f)$ a *counterexample*. Let $\bot\in\mathcal{F}$ and $\top\in\mathcal{T}$, such that such that $why(\bot)=why(\top)=\emptyset$. The operator $\vdash$ *decorates* a Heavy Boolean with an *additional* reason.

The following axioms hold for unary operator $\neg{:}\mathcal{T}{\rightarrow}\mathcal{F}$ and $\neg{:}\mathcal{F}{\rightarrow}\mathcal{T}$, non-commutative operators $\vee{:}\mathcal{H}{\times}\mathcal{H}{\rightarrow}\mathcal{H}$ and $\wedge{:}\mathcal{H}{\times}\mathcal{H}{\rightarrow}\mathcal{H}$, and quantification operators *exists* and *forall*.

$$why(\neg t) = why(t) \tag{5}$$
$$why(\neg f) = why(f) \tag{6}$$
$$f \vee b = b \tag{7}$$
$$t \vee b = t \tag{8}$$
$$f \wedge b = f \tag{9}$$
$$t \wedge b = b \tag{10}$$
$$exists(p, []) = \bot \tag{11}$$
$$forall(p, []) = \top \tag{12}$$
$$exists(p, h{::}s) = (p(h) \vdash h) \vee \; exists(p, s) \tag{13}$$
$$forall(p, h{::}s) = (p(h) \vdash h) \wedge \; forall(p, s) \tag{14}$$

We omit proofs of associativity, well-definedness, DeMorgan's Law, equalities such as: $\neg(\neg h)=h$, $\neg\bot=\top$, $\neg\top=\bot$, and more.

6 HEAVY BOOLEAN VALUES

Helen, being a resourceful programmer, stumbles upon the Heavy Boolean library at https://github.com/jimka2001/heavybool. She discovers that it provides a technique allowing her to verify certain quantified propositions or find counterexamples. She will be able verify Equation (4), all the while assuring that the code express the intent and remain symmetric with the mathematical formalism.

In this section we look at the implementation of Heavy Booleans in each of our target programming languages. A *Heavy Boolean* is an object which *represents true* or *false* in a Boolean context, but has meta data. The primary purpose of a Heavy Boolean is to be generated by universal and existential quantifiers. Because of the way Boolean values are treated in the various programming languages the design of these Heavy Boolean objects is subtly different in each case.

6.1 Clojure

Listing 21: Clojure Quantifier Syntax

```
(+forall [a M]
  (+forall [b M]
    (+forall [c M]
      (= (op (op a b) c)
         (op a (op b c))))))

;; equivalently by macro expansion
(+forall [a M
          b M
          c M]
  (= (op (op a b) c)
     (op a (op b c))))
```

In Clojure, Helen represents Eq (4) as in Listing 21.

If `op` is the `+` (addition) function, then this universal quantifier is satisfied it returns a Heavy Boolean indicating *true*. However, if `op` is the `-` (subtraction) function, then the universal quantifier returns a value indicating *false*, but which also indicates the counterexample: $a = 0$, $b = 0$, and $c = 1$. Notice that since `+forall` and `+exists` are macros, the macro has access to the variable names, `a`, `b`, and `c`, and hence is able to incorporate them into the return value. As will be seen in Sections 6.3 and 6.4, this subtlety is missing from the Python and Scala implementations of Heavy Boolean as those implementations do not include any meta-programming.

Listing 22: Clojure Quantifier Use in Boolean Context

```
(+if (+or (+forall [a M]
            (> a 10))
          (+exists [a M
                    b M]
            (< (* a b) 100)))
  "yes"
  "no")
```

In both Clojure and in Common Lisp (Section 6.2) the language dictates which values are *false* and that all other values are *true*. If Helen attempts to use a `heavy-bool` in a Boolean context, it will be considered *true*. For this reason, we provide wrappers around Boolean operators `+if`, `+or`, `+and`, `+not` and a few others. An example is shown in Listing 22.

Note that lines 17 and 9 of Listing 23 are exactly equivalent as the latter is a macro expansion of the former. Arguably line 9 better emphasizes how Heavy Booleans compose.

Listing 23: Clojure Quantifier Sample Output

```
1  (def M [0 1 2 3 4])
2
3  (+forall [a M
4            b M
5            c M]
6    (= (+ (+ a b) c)
7       (+ a (+ b c))))   => [true ()]
8
9  (+forall [a M
10           b M
11           c M]
12   (= (- (- a b) c)
13      (- a (- b c))))  => [false ({:witness 0, :var a}
14                                  {:witness 0, :var b}
15                                  {:witness 1, :var c})]
16
17 (+forall [a M]
18   (+forall [b M]
19     (+forall [c M]
20       (= (- (- a b) c)
21          (- a (- b c)))))) => [false
22                                ({:witness 0, :var a}
23                                 {:witness 0, :var b}
24                                 {:witness 1, :var c})]
```

6.2 Common Lisp

The Heavy Boolean implementation in Common Lisp, Listing 24, resembles that in Clojure, except that Heavy Boolean objects are represented by instances of the CLOS [7] class `heavy-bool` and in particular its two direct subclasses, `heavy-true` and `heavy-false`.

Listing 24: Common Lisp Quantifier Sample Output

```
(setf  M '(0 1 2 3 4))

(+forall (a M
          b M
          c M)
  (= (+ (+ a b) c)
     (+ a (+ b c)))) => #<HEAVY-TRUE T>

(+forall (a M
          b M
          c M)
  (= (- (- a b) c)
     (- a (- b c)))) => #<HEAVY-FALSE F
                          ((:WITNESS 0 :VAR A)
                           (:WITNESS 0 :VAR B)
                           (:WITNESS 1 :VAR C))>
```

6.3 Python

In Python we have implemented Heavy Booleans with the class named `HeavyBool` with two subclasses `HeavyTrue` and `HeavyFalse`. The Python language has an interesting feature not available to the lisp dialects discussed above. We define the `__bool__` method on `HeavyBool` so that instances of `HeavyTrue` and `HeavyFalse` behave like the Boolean *true* and *false* respectively in Boolean contexts, such as with `if`, `and`, `or`, *etc.*.

An unfortunate (but documented) feature of the `any` function is that it explicitly returns `True` if any of the generated element is *true* and `False` otherwise. Likewise, `all` returns `False` on encountering a *false* element, otherwise returns `True`. *I.e.*, when Helen calls `any` or `all`, she cannot detect which element was the culprit of early termination. To solve this problem, we have implemented `anyM` and `allM`. The `anyM` function iterates over its input, expecting to find instances of `HeavyBool`, and returns the first instance of `HeavyTrue` it encounters, otherwise returns an undecorated instance of `HeavyFalse`. Analogously, `allM` returns the first `HeavyFalse` instance encountered, otherwise returns an undecorated instance of `HeavyTrue`.

Using `allM`, Helen implements something similar to Listings 23 and 24. The `anyM` functions works analogously.

Listing 25: Python: Existential Quantifier

```
1  allM(HeavyBool(((a - b) - c) == (a - (b - c)),
2                 {"a":a, "b":b, "c":c})
3       for a in M
4       for b in M
5       for c in M)  => False[[{'a': 0,
6                               'b': 0,
7                               'c': 1}]]
8
9  # Alternative to annotate only counterexamples
10 forallM(M, lambda a:
11   forallM(M, lambda b:
12     # Cannot put arbitrary code here.
13     # May only use single expression.
14     # Cannot declare local variables.
15     forallM(M, lambda c:
16       HeavyBool(((a - b) - c) == (a - (b - c))
17                 ).annotateFalse({"a":a, "b":b, "c":c}))))
18
19 => False[[{'a': 0, 'b': 0, 'c': 1},
20           {'witness': 1},
21           {'witness': 0},
22           {'witness': 0}]]
23
24 # python quantifiers use in Boolean context
25 if allM(HeavyBool(a > 10)
26         for a in M) or \
27    anyM(HeavyBool(a*b < 100)
28         for a in M
29         for b in M):
30    print("yes")
31 else:
32    print("no")
```

When contrasting Listing 25 line 2 with Listings 23 and 24, we see that Helen must construct the *reason* argument herself `{"a":a, "b":b, "c":c}` whereas the macro-based lisp implementation of `+forall` and `+exists` auto-construct it.

While the code in Listing 25 lines 1 through 5 resembles corresponding code using the built-in `all` (*e.g.*, Listing 11), each iteration wastefully constructs a `HeavyBool` with a populated `reason` argument. The code in Listing 25 line 17, as maladroit as it is, shows how to annotate the `HeavyBool` only in the counterexample case.

Even if some aspects of the Python implementation of Heavy Booleans are awkward, some parts do seem elegant: *e.g.*, the fact that `HeavyBool` behaves like built-in `bool` with respect to many built-in operators—illustrated in Listing 25 lines 24 through 32. We have not implemented heavy-Boolean-specific versions of `if`, `and`, *etc.*. as was the case in Listing 22.

The `forallM` (and `existsM`) implementation restricts Helen to the egregiously limited `lambda` syntax, prohibiting most coding patterns including prohibition of variable declarations (See Listing 25). This syntax is not imposed on the lisp programmer, as the macros `+exists` and `+forall` allow arbitrary code in the given code body.

Listing 26: Scala Quantifier Syntax

```
1  val M = 0 to 4
2
3  forallM("a",M){(a) =>
4    forallM("b",M){(b) =>
5      forallM("c",M){(c) =>
6        (a - b) - c == a - (b - c)}}}
7
8  => false[(witness->0, tag->a);
9           (witness->0, tag->b);
10          (witness->1, tag->c)]
11
12 // used in Boolean context
13 if (forallM("a",M){(a) =>
14       forallM("b",M){(b) =>
15         forallM("c",M){(c) =>
16           (a + b) + c == a + (b + c)}}})
17   "yes"
18 else
19   "no"
20
21 => "yes"
22
23 (forallM("a", M){(a) => a < 10}
24   && existsM("a", M){(a)=>
25        existsM("b", M){(b) =>
26          a*b < 100}})
27
28 => true[(witness->0, tag->a); (witness->0, tag->b)]
```

6.4 Scala

In Scala we implemented Heavy Booleans as an Algebraic Data Type (ADT) [11] named `HeavyBool`. An ADT in Scala is a class for which the compiler can assume all subclasses are fixed and known at compile time. The two subclasses `HeavyTrue` and `HeavyFalse` implement Heavy Boolean *true* and *false*. The `HeavyBool` class provides methods `forallM` and `existsM` which can be used as shown in Listing 26.

As is seen in Listing 26, the `forallM` method (and also `existsM`) takes a string argument where Helen must specify the variable name. This redundancy is because we are not using any Scala meta-programming [3] to auto-extract the variable name from the code. Meta-programming in Scala is exceedingly complicated, far beyond our expertise, and makes application code incompatible between major Scala compiler releases.

Instances of `HeavyBool` may be used in Boolean contexts such as with `if`, `and`, `or`, *etc.*. as seen in Listing 26 lines 13 through 19.

7 EXTENDED USE CASE

The example discussed in Section 3 may not be convincing to programmers who are more concerned with concrete program development. We provide here a second example and solution which illustrates how the Clojure implementation of `heavy-bool` solves an annoying and perhaps serious limitation in unit testing.

Helen Wheels is now attempting to write a unit test using the standard, well-loved `clojure.test` testing framework. She begins by writing the test shown in Listing 27. The test follows the principle of PBT (property based testing) [4, 5] which rather than testing specific examples, instead tests expected invariants of functions based on randomly generated or exhaustively generated input data.

Listing 27: Test Case Using Standard API

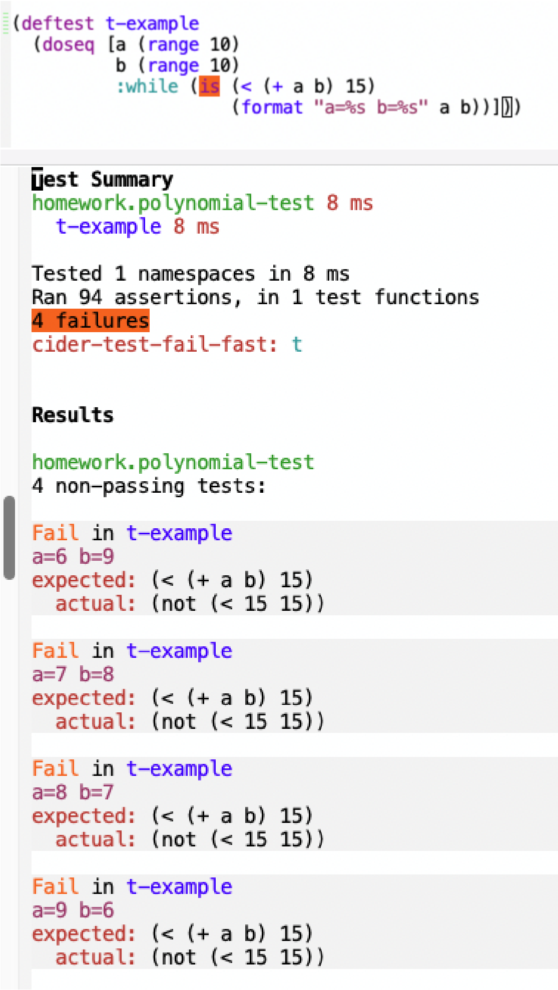
```
1  (deftest t-plus-associative
2    (doseq [p1 polynomials
3            p2 polynomials
4            p3 polynomials]
5      (is (sut/==
6             (sut/+ (sut/+ p1 p2) p3)
7             (sut/+ p1 (sut/+ p2 p3)))
8          (format "non-associative: p1=%s\np2=%sp3=%s"
9                  p1 p2 p3))))
```

In Clojure, the `sut/` prefix is used to indicate symbols from the *System Under Test*. The unit test tries to verify that the function `sut/+` is associative. If a counterexample is found, the side-effecting call to `clojure.test/is` (line 5) registers the counter example, but the `doseq` loop continues. Thus, if n is the length of `polynomials`, then worst case the loop will report n^3 many violations. If running interactively, the IDE freezes while trying to construct a formatted string annotating thousands of almost identical counterexamples.

7.1 1st Vain Attempted Fix, `:while`

In the first attempt to fix this problem, Helen evokes the `:while` modifier as shown on line 5 in Listing 28.

This attempt works somewhat, but `:while` does not exactly stop the iteration. Instead, the `:while` modifier causes the inner most loop `p3` to exit, evoking the successive iteration of `p2`. The computation of success values of the expression continues for some



```
(deftest t-example
  (doseq [a (range 10)
          b (range 10)
          :while (is (< (+ a b) 15)
                     (format "a=%s b=%s" a b))])
```

```
Test Summary
homework.polynomial-test 8 ms
  t-example 8 ms

Tested 1 namespaces in 8 ms
Ran 94 assertions, in 1 test functions
4 failures
cider-test-fail-fast: t


Results

homework.polynomial-test
4 non-passing tests:

Fail in t-example
a=6 b=9
expected: (< (+ a b) 15)
  actual: (not (< 15 15))

Fail in t-example
a=7 b=8
expected: (< (+ a b) 15)
  actual: (not (< 15 15))

Fail in t-example
a=8 b=7
expected: (< (+ a b) 15)
  actual: (not (< 15 15))

Fail in t-example
a=9 b=6
expected: (< (+ a b) 15)
  actual: (not (< 15 15))
```

Figure 1: Example Test Failure

time after the first `false` value of `(is ...)`. The end effect is that we'll have n^2 rather than n^3 many failures.

Listing 28: 1st Vain Attempt to Fix Problem in Listing 27

```
(deftest t-plus-associative
  (doseq [p1 polynomials
          p2 polynomials
          p3 polynomials
          :while (is (sut/==
                       (sut/+ (sut/+ p1 p2) p3)
                       (sut/+ p1 (sut/+ p2 p3)))
                     (format
                       "non-associative: p1=%s\np2=%sp3=%s"
                       p1 p2 p3))]))
```

Figure 1 is a simpler example which shows that the `(doseq ... :while ....)` loop does not abort on the first failure.

7.2 2nd Vain Attempted Fix, `every?`

Helen wants the test to simply fail on the first violation, so she rewrites the code as shown in Listing 29, using the built-in universal quantifier, `every?`.

Listing 29: 2nd Vain Attempt to Fix Problem in Listing 27

```
1  (deftest t-plus-associative-b
2    (is (every? (fn [p1]
3          (every? (fn [p2]
4            (every? (fn [p3]
5              (sut/== (sut/+ (sut/+ p1 p2)  p3)
6                      (sut/+ p1  (sut/+ p2 p3))))
7              polynomials))
8            polynomials))
9          polynomials)
10       ;; regrettably, p1, p2, and p3 are out of scope
11       "WHAT TEXT TO PUT HERE?")))
```

By pulling the `clojure.test/is` outside the loop, the test fails early, when it discovers one failure. However, Helen cannot construct the second argument of 'is' (line 11) which should indicate the values of `p1`, `p2`, and `p3` constituting the counterexample.

7.3 3rd Vain Attempted Fix, `some`

Helen notices that part of the problem with Listing 29 was that although the execution stops as soon as a counterexample was found, the iterations variables were no longer in scope. She decides to apply DeMorgan's law as was done in Listing 9, and refactor the test, writing Listing 30[2].

Listing 30: 2nd Vain Attempt to Fix Problem in Listing 27

```
(deftest t-minus-associative
  (is (empty?
        (some (fn [[x y z]]
                (when (not (= (op x (op y z))
                              (op (op x y) z)))
                  [x y z]))
              (for [p1 polynomials
                    p2 polynomials
                    p3 polynomials]
                [p1 p2 p3])))))
```

This refactoring leads to a successful detection of violation of associativity by displaying a counterexample.

```
expected: (empty?
            (some
             (fn [[x y z]]
           (when (not (= (op x (op y z)) (op (op x y) z)))
                [x y z]))
             (for [p1 M p2 M p3 M] [p1 p2 p3])))
  actual: (not (empty? [{0 1} {} {0 1}]))
```

Even though a counterexample is successfully discovered, the cost is that Helen refactored the code so that diverged from the mathematical expression (Eq 4) she was attempting to verify.

[2] Thanks to Ed Bowler aka https://github.com/l0st3d for providing this clever solution

7.4 Proposed Fix, `heavy-bool`

Using the `heavy-bool` library, Helen writes the test shown in Listing 31.

Listing 31: Proposed Fix for Problem in Listing 27

```
(deftest t-plus-associative-b
  (is (hb/+bool
        (hb/+forall [p1 polynomials
                     p2 polynomials
                     p3 polynomials]
           (hb/+heavy-bool (sut/==
                             (sut/+ (sut/+ p1 p1)
                                    p3)
                             (sut/+ p1
                                    (sut/+ p2 p3))))))))
```

If this test fails, we'll see a message such as the following. Cryptic, but all the information is there.

The `:var` and `:witness` tags of the error message indicates that when `p1 = {0 1}` and `p2 = {}` and `p3 = {0 1}` the associativity fails. The `:associative false` key/value pair indicates that it is the associativity check which is failing.

```
Fail in t-plus-associative-b
hb: plus associativity

expected: (hb/+bool
           (hb/+forall
            [p1 polynomials
             p2 polynomials
             p3 polynomials]
           (hb/+heavy-bool
            (sut/==
             (sut/+ (sut/+ p1 p1) p3)
             (sut/+ p1 (sut/+ p2 p3))))))
  actual: (not
           (hb/+bool
            [false
             ({:var p1, :witness {0 1}}
              {:var p2, :witness {}}
              {:var p3, :witness {0 1}})]))
```

8 CONCLUSION

The existential and universal quantifiers offered by the Heavy Boolean implementations provide useful alternatives to the built-in quantifiers of the languages we have investigated: Clojure, Common Lisp, Python, and Scala. In the lisp dialects the quantifiers `+exists` and `+forall` provide expressions which closely match the mathematical notation, express the intent, and abort an otherwise polynomial complexity search once a witness or counterexample is found. In addition, the quantifiers provided in lisp, provide additional elegance in that they provide useful information such as variable names allowing reflection for debugging or constructing error messages. A limitation of the lisp dialects is that they do not support overloading *false* values, thus the heavy-boolean objects cannot be used in the ordinary Boolean contexts such as `if`, `cond`, `and`, `or`, `not`, *etc.*. Because of this limitation, the packages provide *wrapped* versions such as `+if`, `+and`, *etc.*.

The Python and Scala languages do allow extending Boolean values, including *false*, which allows Heavy Boolean objects to be used in place of built in Boolean primitives such as `if`, `and`, *etc.*. However, the lack of meta programming means the user must provide redundant information such as variable names.

As explained in Section 6.3 the Python implementation imposes strict limitation on user expressiveness which is not an inconvenience experienced by the lisp programmer.

REFERENCES

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

[2] Merrie Bergmann, James Moore, and Jack Nelson. *The Logic Book, Sixth Edition.* McGraw-Hill, 2015. ISBN 978-0-07-803841-9.

[3] Eugene Burmako. Scala macros: let our powers combine! on how rich syntax and static types work with metaprogramming. In *Proceedings of the 4th Workshop on Scala*, SCALA '13, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450320641. doi: 10.1145/2489837.2489840. URL https://doi.org/10.1145/2489837.2489840.

[4] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581132026. doi: 10.1145/351240.351266. URL https://doi.org/10.1145/351240.351266.

[5] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. *SIGPLAN Not.*, 35(9):268–279, September 2000. ISSN 0362-1340. doi: 10.1145/357766.351266. URL https://doi.org/10.1145/357766.351266.

[6] J. Malenfant F.-N. Demers. Reflection in logic, functional and object-oriented programming: a short comparative study. In *IJCAI'95, Workshop on Reflection and Metalevel Architectures and their Applications in AI*, pages 29–38, 1995. URL http://fparreiras/papers/reflectionlogicfuncoocomparative.pdf.

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

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

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

[10] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide.* Artima Incorporation, USA, 1st edition, 2008. ISBN 0981531601, 9780981531601.

[11] Filip Sieczkowski, Sergei Stepanenko, Jonathan Sterling, and Lars Birkedal. The essence of generalized algebraic data types. *Proc. ACM Program. Lang.*, 8(POPL), January 2024. doi: 10.1145/3632866. URL https://doi.org/10.1145/3632866.

[12] Guy L Steele Jr, Scott E Fahlman, Richard P Gabriel, David A Moon, Daniel L Weinreb, Daniel G Bobrow, Linda G DeMichiel, Sonya E Keene, Gregor Kiczales, Crispin Perdue, et al. *Common LISP: The Language (2nd Ed.).* Digital Press, Bedford, Massachusetts, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

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

Tuesday, 20 May 2025

A Brief Perspective on Deep Learning Using Common Lisp

Martin Atzmueller
Semantic Information Systems Group, Osnabrück University &
German Research Center for Artificial Intelligence (DFKI)
Osnabrück, Germany
martin.atzmueller@uos.de

Abstract

Artificial neural networks (ANN) and deep learning (DL) are prominent areas in artificial intelligence and specifically machine learning. This paper provides a brief perspective on a respective set of frameworks and libraries as potential options to be applied using Common Lisp. We discuss a targeted selection and provide a brief categorization as well as illustrating implementation examples. We conclude with a discussion of open issues and further perspectives.

CCS Concepts

• Software and its engineering → Software libraries and repositories; **• Computing methodologies → Neural networks**.

Keywords

Common Lisp, Deep Learning, Framework, Library, Integration

1 Introduction

Common Lisp [25, 46, 47] features a flexible and powerful approach for solving complex problems, in particular due to its extensibility and exploratory as well as dynamic development capabilities. It has been successfully applied in many areas of artificial intelligence (AI), in particular symbolic AI, e. g., in knowledge representation and reasoning [17, 18, 24], expert systems [36, 41], planning [4, 16], theorem proving [29, 34], robotics [27, 51], and computer vision [44, 45]. While Common Lisp is also being used in data analysis and machine learning, in areas such as deep learning most frameworks and/or libraries focus on implementations via programming languages such as Python [49], R [37] or C++ [26]. However, for approaching deep learning using Common Lisp there are different options being enabled with the availability of specific frameworks and different interfaces: With its strengths, e. g., in rapid-prototyping, developing complex applications [30], its symbolic processing capabilities, and the availability of industrial strength compilers (e. g., Steel Bank Common Lisp, SBCL) [38] for creating efficient (native) code, approaches integrating with Common Lisp can potentially provide options beyond those above. Furthermore, this also relates to implementing and using targeted domain specific languages which enable advanced approaches in dynamic and exploratory programming, interactive experimentation as well as rapid prototyping, which is often crucial, e. g., in teaching and educational contexts.

In this paper, we focus on ANN and DL frameworks enabling the application of Deep Learning using Common Lisp. These, in particular, relate to frameworks being directly implemented *in*, or allowing access *from* Common Lisp such that Deep Learning approaches can be implemented using existing libraries. Regarding the level of integration there are different options: On the one hand, for example, basic computational components facilitating fast matrix multiplication can be integrated via foreign function interfaces while higher level machine learning is then implemented in Common Lisp. On the other hand, existing (foreign) libraries could be integrated at higher levels, such that only a thin Common Lisp API is implemented on top. Both approaches favor specific advantages as well as disadvantages.

Overall, we present a brief perspective towards targeted approaches, sketch and discuss implementation examples, and outline open issues and perspectives. In particular, we aim to target frameworks which are readily accessible in the language for enabling direct access and customization of the respective framework components, i. e., architecture and functional interface. Furthermore, we also address options for hybrid approaches, e. g., using loose coupling to frameworks implemented in other languages, or a tighter coupling using foreign function interfaces. While this typically allows the extension of the respective application programming interface (API), providing domain specific languages using Common Lisp, and building on existing efficient libraries, e. g., implemented in C/C++, such approaches typically only favor extensibility on the mid-level to higher-level API.

Then, such facilities allow efficient and effective development and deployment and can provide extensible approaches for implementation. Using these, for example, we can potentially address research on new deep learning/symbolic architectures, support teaching on those and further advanced concepts in deep learning frameworks and optimization, as well as enable further use in complex Common Lisp AI and ML applications. In general, for example, advanced approaches such as neuro-symbolic learning methods or physics-informed machine learning approaches could benefit from such a (deep) integration, regarding the adaptation and integration into learning processes, general extensibility of the respective basic machine learning components, and overall system integration.

The rest of the paper is structured as follows: Section 2 starts with a brief summary of some background on deep learning, before we set out to describe exemplary fundamental as well as related frameworks. Afterwards, Section 3 presents a brief perspective on deep learning using Common Lisp, discussing frameworks and approaches, which are complemented by brief exemplary implementation sketches. Next, Section 4 provides a discussion on those. Finally, Section 5 concludes with a summary and outlines several interesting open issues and further perspectives

ELS'25, Zurich, Switzerland

https://doi.org/10.5281/zenodo.15455278

2 Deep Learning

Below, we first briefly sketch the background of artificial neural networks as well as Deep Learning. After that, we summarize some general frameworks for Deep Learning, before we focus on the Common Lisp based ones in the next section. Finally, we discuss related frameworks for Common Lisp both for fundamental components for Deep Learning as well as for general machine learning.

2.1 Background

In the area of AI and machine learning, DL focuses on ANNs that are made up of relatively simple computational nodes, i. e., neurons arranged in layered network structures with many connections depending on the applied architecture [32, 42]. Artificial neural networks were initially inspired by biological neural networks in the human brain [31, 58]. A neural network can be regarded as a composition of linear and specifically non-linear mathematical functions, such that (composed) function(s) are assigned to the layers of the network, enabling learning complex functions. Essentially, without non-linearity, a neural network would only be able to model linear functional relationships. The resulting function, modeled by the network structure, transforms input data into respective output data. During learning with known input-output data relations the internal parameters (so-called weights) of the neurons of the network are adapted. This is done using the so-called backpropagation algorithm [39] which utilizes a specific loss (or cost) function to estimate the error between expected and given output which is then backpropagated through the network's structure. The obvious goal is to minimize the loss (or cost) function. Backpropagation utilizes the gradient of the loss function for adjusting the network parameters (weights) via gradient descent; in practice often a further optimization algorithm such as stochastic gradient descent is applied. The output of a neural (deep) network is obtained by forward propagation, i. e., through feeding the input data through the network, layer by layer, e. g., in order to obtain a prediction. For a more detailed discussion, we refer to [15, 32, 42, 48].

From this brief description, it becomes obvious that there are certain internal parameters which are important for implementation but also for tweaking and adapting algorithms, e. g., learning methods. This includes, e. g., the types of nodes contained within an ANN, layer and connection structures, the weights of the network, loss and regularization functions, the optimization strategies, etc.

Fundamental components for implementation include methods for handling vectors, arrays - and in general tensors, for fast operations on those, in particular, multiplication; essentially this relates to supporting fast linear algebra operations on vectors, matrices and tensors. For these, the respective frameworks either implement this by themselves, or make use of high-performance libraries for linear algebra. Also, auto-differentiation is rather important, e. g., regarding gradient descent and (dynamic) computational graphs. Regarding computational efficiency, in addition to typical CPU processing, graphics processing units (GPU) can be applied for enabling highly performant calculations, provided by respective toolkits for such processing units. Compared to using only CPU processing, this can speed up calculations considerably [33]. Another practical point for deployment often also includes mobile devices with specific – often resource-constrained – requirements, e. g., [7, 56].

2.2 Overview: Common Frameworks

There is a variety of frameworks for DL, c. f., [13, 50]. Common frameworks (mostly in Python/C++/Java) include, e. g., TensorFlow [1, 2], PyTorch [35], Keras [8], JAX [6], and DeepLearning4J.

- TensorFlow[1] is one of the most prominent open-source Deep Learning frameworks. It facilitates the creation, training, and deployment of DL models via a flexible computational graph structure, and also supports both CPU and GPU computation. It also features a variant (TensorFlow Lite) for mobile and embedded devices, and also includes a C interface.[2]
- PyTorch[3] is also one of the most prominent open source Deep Learning framework, which also supports CPU and GPU computation, and also provides a C++ API/interface[4]. In particular, PyTorch provides dynamic computational graph construction for DL modeling and implementation. There is also some support for enabling models on mobile devices.[5]
- JAX[6] is another open-source framework for fast numerical computing which can also be applied for DL. It offers automatic differentiation together with GPU acceleration for optimizing the performance of complex numerical computations, thus addressing the basic DL components directly.
- Keras[7] is another major framework abstracting on a higher-level API for ease of use, modular high-level implementation and fast prototyping. Keras supports different backends such as those mentioned above (TensorFlow, PyTorch, and JAX). It also allows for deploying models an mobile devices.
- Eclipse DeepLearning4J[8] is a framework for DL in Java. It includes a variety of tools for preprocessing as well, in addition to DL modeling. It features GPU support, and can also be applied on mobile devices.

2.3 Related Frameworks

For DL in Common Lisp, there is a large number of related frameworks, e. g., focusing on DL fundamentals such as linear algebra and fast matrix multiplication functionality. Here, often LAPACK [11] / BLAS [14] as fundamental math libraries for scientific computing, e. g., implemented in OpenBLAS[9] [53] are included. For Common Lisp, e. g., LLA[10], magicl[11] or MGL-MAT[12] are available. Furthermore, there are general machine learning frameworks, e. g., clml[13] or cl-mlep[14]. However, those do not focus on DL. Finally, related frameworks include general parallel computing frameworks, e. g., Petalisp[15] [19–21] for which implementing DL is possible. Compared to those frameworks discussed above, this typically requires a higher-level approach (e. g., [23] built on an early Petalisp version).

[1] https://github.com/tensorflow/tensorflow
[2] https://www.tensorflow.org/install/lang_c
[3] https://github.com/pytorch/pytorch
[4] https://pytorch.org/cppdocs/
[5] https://pytorch.org/mobile/home/
[6] https://github.com/google/jax
[7] https://github.com/keras-team/keras
[8] https://github.com/deeplearning4j/deeplearning4j
[9] https://www.openblas.net/
[10] https://github.com/Lisp-Stat/lla
[11] https://github.com/quil-lang/magicl
[12] https://github.com/melisgl/mgl-mat
[13] https://github.com/mmaul/clml
[14] https://github.com/fzalkow/cl-mlep
[15] https://github.com/marcoheisig/Petalisp

3 Deep Learning using Common Lisp

Below, we briefly summarize options regarding DL using Common Lisp. We discuss three frameworks, distinguishing between one which is specifically implemented in Common Lisp regarding Deep Learning and thus allows direct integration. In addition, we discuss two prominent options regarding a Common Lisp – Python bridge for enabling access to some of the general (Python) frameworks discussed above. Hence, those frameworks are not targeted ones for DL, but for enabling access to and/or integration of the respective Python libraries. In that sense, they do not directly provide deep learning methods but facilitate access to such. Therefore, they enable a relatively flexible selection of deep learning libraries. In that sense, they thus, in principle, also allow users to harness the large set of available options specifically for Python as outlined above.

In summary, as discussed in [3], there have been previous approaches for connecting Common Lisp to Python – enabling to leverage respective frameworks and libraries. In particular, the Py4CL[16] framework facilitates calling Python from Common Lisp using a subprocess, in a stream-based interprocess communication (IPC) approach. Building on that, the library py4cl2[17] incorporates Python's introspection capabilities to further ease the import of function signatures, also allowing a more flexible type mapping. As discussed below, Py4cl2-CFFI[18] is a successor which establishes an FFI bridge to Python libraries using the C-API of CPython. As shown in [3] this enables a more efficient approach concerning run-time, compared to its predecessor libraries/frameworks. For a more detailed discussion, we refer to [3]. In the following, we will briefly present an example using Py4CL and Py4cl2-CFFI, thus distinguishing between a stream-based vs. an approach based on foreign function interface (FFI) access.

A brief overview of the considered approaches is given in Table 1. We distinguish between a low-level vs. high-level API (integration) in Lisp. This is also reflected regarding the extensibility of the respective frameworks, where those integrating Python also need to be extended in Python regarding the low-level functionality, while overall (higher-level) extensibility can be enabled via Common Lisp.

Table 1: Selected framework options for Common Lisp enabling direct or mediated integration of DL libraries

	MGL	Py4CL	Py4CL-CFFI
Low/High Level	Low/High	High	High
Lisp / Bridge	Common Lisp	Bridge (stream-based)	Bridge (FFI-based)
Basic Models	MLP, RNN, RBM, DBM, DBN, GP	*depends on the used library*	*depends on the used library*
Backend/GPU	via CL-CUDA		
Extensibility	Common Lisp	Common Lisp (+ Python)	Common Lisp (+ Python)

3.1 MGL

MGL is a machine learning library, providing several basic models for DL, including

- Backpropagation neural networks [32, 42]: Feed-forward multi-layer perceptron (MLP) networks, as well as recurrent neural networks (RNN).
- Boltzmann Machines [40, 55]: Restricted Boltzmann Machines (RBM, Deep Boltzmann Machines (DBM), as well as Deep Belief Networks (DBN).
- Gaussian processes [10, 43]: These are implemented using the MGL components, in particular using a feed-forward neural network Gaussian Processes.

Listing 1: MGL DL (MNIST) example, adapted from: https://github.com/melisgl/mgl/blob/master/example/mnist.lisp

```
(defun build-relu-mnist-mlp (&key (n-layer-1 256)
                                  (n-layer-2 256)
                                  (n-layer-3 256))
  (build-fnn (:class 'mnist-mlp :max-n-stripes 100)
    (inputs (->input :size 784 :dropout 0.2))
    (layer1-activations
     (->activation inputs :name 'layer1 :size n-layer-1))
    (layer1* (->relu layer1-activations))
    (layer1 (->dropout layer1*))
    (f2-activations
     (->activation layer1 :name 'layer2 :size n-layer-2))
    (layer2* (->relu layer2-activations))
    (layer2 (->dropout layer2*))
    (f3-activations
     (->activation layer2 :name 'layer3 :size n-layer-3))
    (layer3* (->relu layer3-activations))
    (layer3 (->dropout layer3*))
    (prediction (build-softmax layer3))))

(defun build-softmax (inputs)
    (->softmax-xe-loss
     (->activation inputs :name 'classification :size 10)
      :name 'classification))
```

MGL features GPU support via CL-CUDA[19] and relies on MGL-MAT supporting efficient multi-dimensional array computation. MGL supports several basic models as outlined above, in particular feed-forward as well as recurrent neural networks, and Boltzmann machines. In short, feed-forward networks, for example, are composed of a set of *Lumps* (corresponding to layers) which can also be composite lumps (*CLUMPS*) composed of several components thus forming a network by itself. In this way, more complex structures can be compiled. Also, different activation functions, (e. g., rectified linear activation – *relu* or *softmax*), loss, and regularization (e. g., dropout) are supported, in addition to different optimization strategies. Furthermore, MGL also supports several standard machine learning components such as methods for sampling, model training, accuracy estimation, and validation. Listing 1 shows an example for the classic MNIST dataset [12]: Here, a three-layer *MLP* network is constructed for an 28 × 28 input size in order to classify the data and obtain an output (in terms of 10 different classes) – relating to the task of classifying input data of handwritten digit images into 10 classes, which relates to the individual 10 digits. For that, three fully connected layers of (default) 256 nodes are constructed, also including specific *dropout* layers (for excluding specific nodes in computation regarding regularization) and *relu* activation functions.

[16] https://github.com/bendudson/py4cl
[17] https://github.com/digikar99/py4cl2
[18] https://github.com/digikar99/py4cl2-cffi
[19] https://github.com/takagi/cl-cuda

Listing 2: Py4CL Neural Network (CNN) example – i. e., a Common Lisp version of the Keras MNIST example; adapted from https://github.com/keras-team/keras/blob/master/examples/demo_mnist_convnet.py

```
(defpackage :py4cl-example (:use :cl :py4cl))
(in-package :py4cl-example)

(import-module "numpy" :as "np")
(import-module "keras")
(import-module "keras.layers" :as "layers")
(import-module "keras.datasets.mnist")
(import-module "keras.utils")

(defparameter *num-classes* 10)
(defparameter *input-shape* '(28 28 1))

(let ((score nil))
  (remote-objects
    (let* ((data (keras.datasets.mnist:load_data)) ;;  ((x-train y-train) (x-test y-test))
     (x-train (np:expand_dims (chain data ([] 0) ([] 0) (__truediv__ 255)) -1))
     (y-train (keras.utils:to_categorical (chain data ([] 0) ([] 1)) *num-classes*))
     (x-test (np:expand_dims (chain data ([] 1) ([] 0) (__truediv__ 255)) -1))
     (y-test (keras.utils:to_categorical (chain data ([] 1) ([] 1)) *num-classes*))
     (batch-size 128) (epochs 3) (model (keras:sequential)))

      (python-method model 'add (layers:input :shape *input-shape*))
      (python-method model 'add (layers:conv2d :filters 32 :kernel_size '(3 3) :activation "relu"))
      (python-method model 'add (layers:maxpooling2d :pool_size '(2 2)))
      (python-method model 'add (layers:conv2d :filters 64 :kernel_size '(3 3) :activation "relu"))
      (python-method model 'add (layers:maxpooling2d :pool_size '(2 2)))
      (python-method model 'add (layers:flatten))
      (python-method model 'add (layers:dropout 0.5))
      (python-method model 'add (layers:dense *num-classes* :activation "softmax"))

      (print (python-method model 'summary))

      (chain model (compile :optimizer "adam" :loss "categorical_crossentropy" :metrics #("accuracy")))
      (chain model (fit x-train y-train :batch_size batch-size :epochs epochs :validation_split 0.1))

      (setf score (chain model (evaluate x-test y-test :verbose 0)))))
  (format t "Test loss: ~A~%"  (chain score ([] 0)))
  (format t "Test accuracy: ~A~%" (chain score ([] 1))))
```

3.2 Python DL Integration Using Py4CL

As outlined above, there are also options beyond directly implementing/building on libraries in Common Lisp (i. e., as a deep coupling of DL approaches in our context) by a more loose coupling via using frameworks/libraries implemented in other languages/system which are made accessible to be used via Common Lisp. Here, in particular, this relates to C++ and/or Python-based options, where, e. g., either an approach based on foreign function interface (FFI) integration or stream-based inter-process communication approaches are possible. This is the approach taken by Py4CL which was one of the earlier options for connecting between Common Lisp and Python, thus allowing access to libraries in the Python ecosystem.

Essentially, Py4CL uses a stream-based approach for connecting Common Lisp to interact with Python code, such that the Common Lisp process communicates with a separately launched (external) Python process. This is the approach which we sketch in the following. Here, essentially when starting Py4CL a connection from the Common Lisp system to an (external) Python process is initiated, which allows for importing Python function and modules, function/method calls and respective instantiations of data and objects. One challenge is then the exchange/communication in terms of data/objects which can either be done via value or reference – enabling or preventing respective tighter integration, as already briefly discussed above.

Listing 2 shows a simple example of a DL model – again for the MNIST dataset using a convolutional neural network architecture: The Py4CL framework is applied using Keras (in Python), illustrating the respective Py4CL primitives. The model is constructed and trained (in the Python process) while the results can be accessed via the Py4CL connection in Common Lisp. In the example, the respective Python modules are first imported to be accessible in the Common Lisp system. Then, data is first loaded and preprocessed. Next, the model is constructed via the Keras *sequential* API, adding the different convolutional network layers. Again, *relu* activation, pooling (for downsampling, data reduction), final dropout and flatten layer (turning its respective input from the convolutional/pooling layers into a single vector), followed by the *softmax* activation as before. Finally, the model is trained and evaluated.

The example illustrates one important point, which is valid in general for any such approaches where such a loosely coupled framework is applied: In the example, the *py4cl:remote-objects* macro is used for keeping the data/objects in the macro's scope in Python, such that only references (handles) are passed to Common Lisp. This enables an efficient approach – otherwise the referenced data structures would have to be converted, i. e., serialized and passed via stream-communication, which is rather infeasible for large objects/data. The example also shows how the result of an evaluation (the score) can then be evaluated further in Common Lisp.

Listing 3: Py4CL2-CFFI Neural Network (CNN) example – i. e., a Common Lisp version of the Keras MNIST example; adapted from https://github.com/keras-team/keras/blob/master/examples/demo_mnist_convnet.py

```
(defpackage :py4cl2-cffi-example (:use :cl :py4cl2-cffi))
(in-package :py4cl2-cffi-example)

(defun py-sym (name)
  (if (stringp name)
      name
      (string-downcase (symbol-name name))))

(defmacro py. (obj method-name &rest args)
  `(pymethod ,obj ,(py-sym method-name) ,@args))

(defpyfun "load_data" "keras.datasets.mnist")
(defpyfun "expand_dims" "numpy")
(defpyfun "to_categorical" "keras.utils")
(defpyfun "Sequential" "keras")

(defpyfun "Input" "keras.layers")
(defpyfun "Conv2D" "keras.layers")
(defpyfun "MaxPooling2D" "keras.layers")
(defpyfun "Flatten" "keras.layers")
(defpyfun "Dropout" "keras.layers")
(defpyfun "Dense" "keras.layers")

(defparameter *num-classes* 10)
(defparameter *input-shape* '(28 28 1))

(defun get-data (data index1 index2)
  (py. (py. data "__getitem__" index1) "__getitem__" index2))

(let ((score nil))
  (with-remote-objects
      (let* ((data (load-data)) ;; ((x-train y-train) (x-test y-test))
             (x-train (expand-dims :a (py. (get-data data 0 0) "__truediv__" 255) :axis -1))
             (y-train (to-categorical (get-data data 0 1) *num-classes*))
             (x-test (expand-dims :a (py. (get-data data 1 0) "__truediv__" 255) :axis -1))
             (y-test (to-categorical (get-data data 1 1) *num-classes*))
             (batch-size 128) (epochs 3) (model (sequential)))
        (py. model add (input :shape *input-shape*))
        (py. model add (conv-2d :filters 32 :kernel_size '(3 3) :activation "relu" :name "conv2d1"))
        (py. model add (max-pooling-2d :pool_size '(2 2) :name "maxpool1"))
        (py. model add (conv-2d :filters 64 :kernel_size '(3 3) :activation "relu" :name "conv2d2"))
        (py. model add (max-pooling-2d :pool_size '(2 2) :name "maxpool2"))
        (py. model add (flatten))
        (py. model add (dropout 0.5))
        (py. model add (dense *num-classes* :activation "softmax"))

        (print (py. model summary))

        (py. model compile :optimizer "adam" :loss "categorical_crossentropy" :metrics #("accuracy"))
        (py. model fit x-train y-train :batch_size batch-size :epochs epochs :validation_split 0.1)

        (setf score (py. model evaluate x-test y-test :verbose 0))))
  (format t "Test loss: ~A~%"  (item score 0))
  (format t "Test accuracy: ~A~%" (item score 1)))
```

3.3 Python DL Integration Using Py4CL-CFFI

As discussed above, Py4CL-CFFI takes a slightly different approach than Py4CL, namely access to Python via a FFI-based approach. Hence, as discussed in [3] communication/calling Python is faster, which is particularly important for processes/calls of short to medium duration, whereas longer running processes potentially do not benefit that much from the reduced communication overhead. For a broader description and benchmarks we refer to [3] for a more detailed evaluation and discussion. In our case of DL, specifically when constructing larger DL models, most of the time is spent in training and testing the respective DL model.

In line with the examples presented above, Listing 3 illustrates a simple example of a DL model implemented using Py4CL-CFFI. Here, we again refer to the standard example for the MNIST dataset using a convolutional neural network architecture. Like in the previous example using Py4CL, the Py4CL-CFFI framework is applied using Keras (in Python). The syntax for importing modules and accessing/calling Python callables slightly differs from Py4CL, nevertheless the overall structure is rather similar. In this example, several helper functions are included, which enable both easier readability and a more streamlined applicability of the respective Python functionality.

4 Discussion

Regarding the three presented frameworks/options for DL in Common Lisp, there are different levels of integration with respective advantages and drawbacks. With frameworks directly implemented in Common Lisp, for example, it is possible to more directly interface regarding (also low-level) computations which can be beneficial for advanced applications such as physics-informed machine learning [9, 28] or neuro-symbolic (hybrid) [5, 22, 54] approaches. Here, then also the advanced capabilities of efficient Common Lisp compilers, for example, SBCL can be utilized, since those functions can be compiled to efficient native code – in contrast to, for example, interpreted Python code. Regarding extensibility, there is, e. g., the option of adapting functions directly, or to tweak the behavior of specific classes and/or methods. However, when specific functionality is not available in Common Lisp libraries (yet), then other framework interfacing options are often necessary.

Regarding such frameworks which are made accessible to Common Lisp via foreign function or stream-based inter-process integration, there are also both advantages and drawbacks. There is a variety of such (foreign) DL frameworks available – a potentially clear advantage. However, the respective method of integration needs to take into account specific requirements of the application, which need to be matched to the possible options. For instance, regarding stream-based approaches, we already pointed out challenges regarding data exchange using stream-based interaction mechanisms. Then, specific wrappers in the foreign language (such as Python) can abstract away specific functionality which cannot be enabled on the Common Lisp side due to efficiency issues in data conversion. In our example, this was kind of indicated by only considering the remote objects on the Python side, such that the relatively large datasets were only handled there and did not need to be converted to a serialized representation to be sent to Common Lisp via the stream-based interface. Also, in cases where the interaction happens via the stream inter-process communication, there is some limit to the possible number of function calls in a given timeframe. Some analysis on this is given in [3]. Also, interaction is typically focused on function and method calls, such that extensibility (as discussed above) is limited.

In addition to the stream-based interface, often also other foreign function integration options are possible, in particular, when there is a C-interface available. Then, FFI bindings can be generated and applied, e. g., using *c2ffi*[20] or *cl-autowrap*.[21] A portable option is provided, for example, using the (common) foreign function interface (CFFI).[22] Compared to using Py4CL, an important option for a tighter foreign framework integration is provided by the Py4CL2-CFFI project[23] as discussed above. It enables to connect the Python shared library via CFFI, as illustrated above, which results in a Common Lisp image using one python instance with tighter integration regarding data passing and specifically more efficient calling options regarding functions/methods, since this is handled on the FFI level. Another option is given by ABCL[24], in particular, when connecting to Java frameworks/libraries is important.

5 Conclusions

As we have presented in this paper, there are several frameworks and options available for DL in Common Lisp. In particular, this relates to direct (deep) integration into the Common Lisp system itself as well as connecting to foreign libraries and/or frameworks. For those, there are several promising approaches enabling a rich set of implementation options.

We have provided a brief perspective on a selection of respective fameworks/options regarding DL using Common Lisp, specifically on a framework allowing the mentioned direct access to the modeling elements for engineering ANN/DL architectures including network structure, loss functions etc" and provided a targeted implementation example. In addition, we have sketched two further options using frameworks enabling access to (foreign) Python-based libraries, and also discussed a standard example (MNIST) in more detail. Interesting options for future work include, for example, detailed evaluations of the respective frameworks and libraries as well as extensions concerning more DL architectures.

There are also several further open issues and perspectives:

(1) The mentioned MGL framework implemented in Common Lisp currently does not extend to some recent architectures for DL, e. g., graph neural networks [52, 57]. This can be addressed, e. g., by specialized extensions, or by using foreign libraries and respective bridging frameworks. Then, high-level API access via Common Lisp is enabled, potentially using additional interfacing layers on the library side.
(2) Frameworks such as Petalisp can enable promising options as efficient backends for Common Lisp based frameworks using the provided abstractions for efficient parallel scientific computational support – potentially in a uniform way.
(3) The latter point is also relevant for specialized computational GPU backend support. In general, for DL applications GPU support is rather important, which is also enabled by various frameworks. As discussed above, most (foreign-library-enabled) frameworks provide sufficient GPU support, such that including those can also help to address this issue, whenever the application requirements allow for the Common Lisp to the foreign library setting, e. g., in terms the implemented foreign function and data structure integration etc.
(4) Mobile devices pose specific challenges regarding the deployment of DL models. Therefore, options for running Common Lisp implementations on mobile devices are also important here. Interesting perspectives involve optimizations for such mobile devices using specific models, and/or providing specific (cross-)compilation steps to specialized model variants.
(5) Advanced DL-based approaches such as physics-informed machine learning (e. g., [9, 28]) or the integration of symbolic and sub-symbolic (neural) approaches (e. g., [5, 22, 54]) are potentially promising applications fields, where Common Lisp can potentially demonstrate its strengths in the future. Here, deep integration of the respective frameworks can potentially enable considerable advantages compared to the currently available options due to the adaptation and integration capabilities into low-level processes, the general extensibility of the respective basic (low-level) components, as well as the overall system integration.

[20] https://github.com/rpav/c2ffi
[21] https://github.com/rpav/cl-autowrap
[22] https://github.com/cffi/cffi
[23] https://github.com/digikar99/py4cl2-cffi
[24] https://abcl.org/

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems, 2015. URL http://tensorflow.org/. Software available from tensorflow.org.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-Scale Machine Learning. In *Proc. USENIX symposium on operating systems design and implementation*, pages 265–283, 2016.

[3] Shubhamkar Ayare. py4cl2-cffi: Using CPython's C API to call Python callables from Common Lisp. In *Proc. European Lisp Symposium (ELS)*, pages 52–59, 2024.

[4] Tamara Babaian and James G Schmolze. Efficient open world reasoning for planning. *Logical Methods in Computer Science*, 2, 2006.

[5] Tim Bohne, Anne-Kathrin Patricia Windler, and Martin Atzmueller. A Neuro-Symbolic Approach for Anomaly Detection and Complex Fault Diagnosis Exemplified in the Automotive Domain. In *Proc. K-Cap*, pages 35–43, 2023.

[6] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/google/jax.

[7] Jiasi Chen and Xukan Ran. Deep Learning With Edge Computing: A Review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[8] François Chollet. keras. https://github.com/fchollet/keras, 2015.

[9] Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific machine learning through physics–informed neural networks: Where we are and what's next. *Journal of Scientific Computing*, 92(3):88, 2022.

[10] Andreas Damianou and Neil D Lawrence. Deep gaussian processes. *Artificial intelligence and statistics*, pages 207–215, 2013.

[11] James Demmel. Lapack: A portable linear algebra library for high-performance computers. *Concurr. Comput. Pract. Exp.*, 3(6):655–666, 1991.

[12] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[13] Kai Dinghofer and Frank Hartung. Analysis of criteria for the selection of machine learning frameworks. In *Proc. ICNC*, pages 373–377. IEEE, 2020.

[14] Iain S Duff, Michael A Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):239–267, 2002.

[15] Jianqing Fan, Cong Ma, and Yiqiao Zhong. A selective overview of deep learning. *Stat. Sci.*, 36(2):264, 2021.

[16] Robert P Goldman and Ugur Kuter. Hierarchical Task Network Planning in Common Lisp: The Case of SHOP3. In *Proc. European Lisp Symposium (ELS)*, pages 73–80, 2019.

[17] Volker Haarslev and Ralf Möller. Description of the RACER System and its Applications. *Description Logics*, 49, 2001.

[18] Volker Haarslev, Kay Hidde, Ralf Möller, and Michael Wessel. The RacerPro knowledge representation and reasoning system. *Sem. Web*, 3(3):267–277, 2012.

[19] Marco Heisig. Petalisp: A common lisp library for data parallel programming. In *Proc. European Lisp Symposium*, pages 4–11, 2018.

[20] Marco Heisig. An Introduction to Array Programming in Petalisp. In *Proc. European Lisp Symposium, ELS*, pages 18–21, 2024.

[21] Marco Heisig and Harald Köstler. Petalisp: run time code generation for operations on strided arrays. In *Proc. ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, pages 11–17, 2018.

[22] Pascal Hitzler, Aaron Eberhart, Monireh Ebrahimi, Md Kamruzzaman Sarker, and Lu Zhou. Neuro-symbolic approaches in artificial intelligence. *National Science Review*, 9(6):nwac035, 2022.

[23] Michael Holzmann. *LispNet: A machine learning framework for Petalisp and its application to Multigrid PDE.* Master's thesis, Chair CS10, FAU, Germany, 2022.

[24] Ian Horrocks and Peter F Patel-Schneider. FaCT and DLP. In *Proc. International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pages 27–30. Springer, 1998.

[25] American National Standards Institute. Incits 226-1994[s2008] information technology, programming language, common lisp. Standard, 1994.

[26] ISO/IEC. Programming Languages — C++. Draft International Standard N4660, March 2017. URL https://web.archive.org/web/20170325025026/http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4660.pdf.

[27] Rob Janssen, Erik van Meijl, Daniel Di Marco, René van de Molengraft, and Maarten Steinbuch. Integrating planning and execution for ROS enabled service robots using hierarchical action representations. In *Proc. International Conference on Advanced Robotics (ICAR)*, pages 1–7. IEEE, 2013.

[28] George Em Karniadakis, Ioannis G Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. Physics-informed machine learning. *Nature Reviews Physics*, 3(6):422–440, 2021.

[29] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23 (4):203–213, 1997.

[30] Bohdan B Khomtchouk, Edmund Weitz, Peter D Karp, and Claes Wahlestedt. How the strengths of lisp-family languages facilitate building complex and flexible bioinformatics applications. *Brief. Bioinform.*, 19(3):537–543, 2018.

[31] Anders Krogh. What are artificial neural networks? *Nature biotechnology*, 26(2): 195–197, 2008.

[32] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, 2015.

[33] Sparsh Mittal and Shraiysh Vaishay. A survey of techniques for optimizing deep learning on GPUs. *Journal of Systems Architecture*, 99:101635, 2019.

[34] J Strother Moore. Milestones from the Pure Lisp theorem prover to ACL2. *Formal Aspects of Computing*, 31(6):699–732, 2019.

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proc. NIPS*, pages 8024–8035. Curran Associates, Inc., 2019.

[36] Frank Puppe. Knowledge reuse among diagnostic problem-solving methods in the shell-kit D3. *Int. J. Hum. Comput. Stud.*, 49(4):627–649, 1998.

[37] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2021.

[38] Christophe Rhodes. SBCL: A sanely-bootstrappable common lisp. In *Proc. Workshop on Self-sustaining Systems*, pages 74–86. Springer, 2008.

[39] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[40] Ruslan Salakhutdinov and Geoffrey Hinton. Deep boltzmann machines. In *Artificial intelligence and statistics*, pages 448–455. PMLR, 2009.

[41] Eric Sandewall. Knowledge-based systems, lisp, and very high level implementation languages. *The Knowledge Engineering Review*, 7(2):147–155, 1992.

[42] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

[43] Matthias Seeger. Gaussian processes for machine learning. *International journal of neural systems*, 14(02):69–106, 2004.

[44] Kai Selgrad, Alexander Lier, Jan Dörntlein, Oliver Reiche, and Marc Stamminger. A High-Performance Image Processing DSL for Heterogeneous Architectures. In *Proc. European Lisp Symposium*, pages 39–46, 2016.

[45] Benjamin Seppke and Leonie Dreschler-Fischer. Tutorial: Computer vision with allegro common lisp and the vigra library using vigracl. *Fundação Calouste Gulbenkian, Lisbon, May 6-7, 2010*, page 53, 2010.

[46] Guy L Steele and Richard P Gabriel. The evolution of lisp. In *History of programming languages—II*, pages 233–330. 1996.

[47] Guy L Steele Jr. An overview of common lisp. In *Proceedings of the 1982 ACM Symposium on LISP and Functional Programming*, pages 98–107, 1982.

[48] Ruo-Yu Sun. Optimization for deep learning: An overview. *Journal of the Operations Research Society of China*, 8(2):249–294, 2020.

[49] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual.* CreateSpace, Scotts Valley, CA, 2009. ISBN 1441412697.

[50] Zhaobin Wang, Ke Liu, Jian Li, Ying Zhu, and Yaonan Zhang. Various frameworks and libraries of machine learning and deep learning: a survey. *Archives of computational methods in engineering*, pages 1–24, 2019.

[51] Johannes Wienke and Sebastian Wrede. A middleware for collaborative research in experimental robotics. In *2011 IEEE/SICE International Symposium on System Integration (SII)*, pages 1183–1190. IEEE, 2011.

[52] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

[53] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 BLAS performance optimization on Loongson 3A processor. In *2012 IEEE 18th international conference on parallel and distributed systems*, pages 684–691. IEEE, 2012.

[54] Dongran Yu, Bo Yang, Dayou Liu, Hui Wang, and Shirui Pan. A survey on neural-symbolic learning systems. *Neural Networks*, 2023.

[55] Nan Zhang, Shifei Ding, Jian Zhang, and Yu Xue. An overview on restricted boltzmann machines. *Neurocomputing*, 275:1186–1199, 2018.

[56] Tianming Zhao, Yucheng Xie, Yan Wang, Jerry Cheng, Xiaonan Guo, Bin Hu, and Yingying Chen. A survey of deep learning on mobile devices: Applications, optimizations, challenges, and research opportunities. *Proceedings of the IEEE*, 110(3):334–354, 2022.

[57] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

[58] Jinming Zou, Yi Han, and Sung-Sau So. Overview of artificial neural networks. *Artificial neural networks: methods and applications*, pages 14–22, 2009.

Porting the Steel Bank Common Lisp Compiler and Runtime to the Nintendo Switch NX Platform

Charles Zhang

Yukari “Shinmera” Hafner

charleszhang99@yahoo.com

shinmera@tymoon.eu

Shirakumo.org

Zürich, Switzerland

Abstract

We present our efforts to adapt the SBCL runtime and compiler to deploy applications onto the NX platform. The Nintendo Switch (NX) is a 64-bit ARM-based platform for video games with a proprietary micro-kernel operating system. Notably this system does not give programs the ability to mark pages as executable at run time or expose access to thread signal handlers, both of which present a significant hurdle to SBCL’s intended bootstrap process and runtime operation. To work around these hurdles we modify SBCL’s build system to bootstrap on a Linux ARM system, which is similar enough to the NX to be able to compile user code. We then use a technique called “shrinkwrapping” to combine the Lisp code and data with the C runtime compiled for the NX to produce a final, fully static ELF executable that can be run on the NX, without the need for runtime code generation. We further use a restricted version of “safepoints” to synchronise threads during garbage collection.

CCS Concepts

• **Software and its engineering** → **Runtime environments**; **Dynamic compilers**; *Garbage collection*; Software creation and management.

Keywords

Common Lisp, SBCL, porting, ARM, aarch64, Nintendo Switch, NX, Experience Report

ACM Reference Format:

Charles Zhang and Yukari “Shinmera” Hafner. 2025. Porting the Steel Bank Common Lisp Compiler and Runtime to the Nintendo Switch NX Platform. In *Proceedings of the 18th European Lisp Symposium (ELS’25).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.5281/zenodo.15384117

Figure 1: The Nintendo Switch (NX) handheld game console

1 Introduction

The Nintendo Switch (codename NX) is a handheld video game console based on the ARM 4 Cortex-A57 64-bit architecture[5]. It features a proprietary micro-kernel operating system called the “Nintendo Switch system software”[8] (NX OS), and normally runs only software that has been licensed, approved, and digitally signed and encrypted by Nintendo.

Developing licensed software for the NX has to be done via Nintendo’s own proprietary Software Development Kit (SDK), which they distribute only under a non-disclosure agreement (NDA). An open-source third-party alternative to the SDK is available[1], but this cannot be used for licensed software.

The OS-provided runtime environment is directly suitable only for C and C++ software. Other game engines such as Unity, Unreal, and Godot do provide exporting functionality to the NX, meaning that ports for the runtime environments they rely on such as .NET/Mono (C#), Lua, and GDScript have been developed, but remain closed-source.

ELS’25, Zürich, Switzerland

© 2025 Copyright held by the owner/author(s).

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM

https://doi.org/10.5281/zenodo.15384117

Many high-performance native-code Common Lisp implementations operate by compiling and loading user code at runtime into the same process (the state of which is traditionally termed an ‘image’ in the Lisp world). These implementations typically provide a way to dump an image in a way where the runtime of a new process can load it. This implementation technique contrasts with ahead-of-time compilation languages such as C where code is compiled and linked ‘offline’ into an executable whose entire code and runtime gets loaded into its own process by the operating system. On the NX, the SBCL runtime cannot compile and load new code at runtime after an executable is loaded by the operating system on the NX due to platform security restrictions. Because large portions of the system such as the compiler of such image-based Common Lisp implementations are also written in Common Lisp, this restriction means that the implementation must be bootstrapped and all user code compiled ahead of time off of the NX, before the compiled Lisp code in the image can then somehow be linked into a runtime built for the NX by the operating system, rather than relying on the implementation runtime to load the image. We discuss the intricacies of this problem for an image-based implementation such as SBCL in section 3 and section 4.

A port of a Common Lisp implementation such as ECL[3], where Lisp code is compiled to C and easily linked into a C runtime, is conceivable and presents far fewer challenges for building applications for the NX than with an image-based implementation. However, due to the high performance requirements of video games and the relatively low-power platform of the NX, we decided to try and port SBCL, since SBCL produces much faster code than ECL especially with regards to CLOS. In addition, some of the obstacles presented by the NX apply in general to how most Common Lisp implementations function.

For example, the NX OS does not provide user signal handlers. This lack of signal handlers is a problem for the Garbage Collector (GC), as SBCL relies on inter-thread signalling to park threads during garbage collection. While SBCL does provide a safepoints mechanism for GC that is used on Windows which similarly lacks signal handlers, this mechanism is not well tested on other platforms, and as-is still did not meet the requirements of the locked-down NX platform. We discuss the GC in detail in section 5.

While we can now compile and deploy complex Common Lisp applications to the NX, some parts of the runtime remain unsupported, and we discuss our future efforts in this regard in section 7.

2 Related Work

Rhodes[7] outlines the methodology behind the general SBCL bootstrapping process, which we extend for our locked-down target platform.

A Common Lisp bootstrapping process as Durand et al.[4] outline where the whole target image is created on the host would work to avoid requiring compiling and executing code at runtime on the NX; however, there is not a complete implementation of this technique yet. User libraries which require querying foreign functions at compile time where the details of target architecture matter also cannot be handled with this technique.

Citing information on the operating system of the NX is difficult as it is a closed-source platform with all usual information placed under NDA. All publicly available information is from security research such as by Roussel-Tarbouriech et al.[8] and reverse-engineering[1].

Particularly, we are unaware of any publication about the porting of other runtime environments to the NX, such as C#, JavaScript, Lua, etc.

Patton[6] and Schwartz[9] describe some details about SBCL's garbage collector which we modify for our port.

3 Build System

The usual process to build the SBCL compiler and runtime proceeds in several distinct phases, some of which need to be run on a “host system” and others on the “target system”. This process does allow for limited cross-compilation, wherein the “host system” steps can be run on an operating system and platform that is not that of the target we are trying to compile for. However, the “target system” steps are supposed to run on the target architecture, and involve compiling, loading, and executing new system code in an existing image. As mentioned in the introduction, we cannot load and execute code in the same process that compiled the code on the NX, as the NX OS does not allow us to map new executable pages.

The basic idea of our solution to this issue is to replace the NX as *initial* target with an ARM64-based Linux. Once everything, including user code, has been compiled on this Linux target, we extract all Lisp code and data out using a process called *shrinkwrapping* and link it together with the SBCL C runtime as compiled for the NX. This process ultimately results in a fully static ELF executable that does not perform any dynamic memory mapping or compilation.

Our build still functions largely the same as the one outlined by Rhodes[7], though being run roughly twice with special configurations to accomplish the hybrid build.

(1) `build-config` (NX)
This step gathers whatever build configuration options for the target and spits them out into a readable format for the rest of the build process. We run this on some host system (which may not be our ARM64 Linux intermediary), using a special flag to indicate that we're building for the NX. We also enable the *fasteval* contribution, which we need to step in for any place where we would usually invoke the compiler at runtime.

(2) `make-host-1` (NX)
Next we build the cross-compiler with the host Lisp compiler, and at the same time emit C header files describing Lisp object layouts in memory as C structs for the next step.

(3) `make-target-1` (NX)
Now we use the C compiler the Nintendo SDK provides for us, which can cross-compile the SBCL C runtime for the NX. We had to make adjustments to the C runtime bits, as the NX OS is not POSIX compliant and lacks a few features the SBCL C runtime usually takes advantage of. The SBCL C runtime contains the GC and OS glue bits that the Lisp code needs. The

artefacts from this stage will later be used in step (10) to attach the C runtime to the shrinkwrapped core.

(4) `build-config` (Linux~NX)
We now create an almost-normal ARM64 Linux system with the same feature set as for the NX. This process involves the usual steps as for a "normal" Linux ARM64 build, though with a special flag to inform some parts of the Lisp process that we're going to ultimately target the NX.

(5) `make-host-1` (Linux~NX)
This step proceeds as usual.

(6) `make-target-1` (Linux~NX)
This step proceeds as usual.

(7) `make-host-2` (Linux~NX)
With the target runtime built, we build the target Lisp system (compiler and the standard library) using the Lisp cross-compiler built by the Lisp host compiler in `make-host-1` (Linux~NX). This step produces a "cold core" that the runtime can jump into, and can be done purely on the host machine. This cold core is not complete, and needs to be executed on the target machine with the target runtime to finish bootstrapping, notably to initialise the object system, which requires runtime compilation.

(8) `make-target-2` (Linux~NX)
The cold core produced in the last step is loaded into the target runtime, and finishes the bootstrapping procedure to compile and load the rest of the Lisp system. After the Lisp system is loaded into memory, the memory is dumped out into a "warm core", which can be loaded back into memory in a new process with the target runtime. From this point on, new code can be loaded and images can be dumped at will.

(9) `user`
For user code we now perform some tricks to make it think it's running on the NX, rather than on Linux. In particular we modify `*features*` to include `:nx` and not `:linux`, `:unix`, or `:posix`. Once that is set up and ASDF has been sufficiently tricked into believing we are on the NX, we can compile our program "as usual" and at the end dump out a new core.

(10) `shrinkwrap`
Once all our code has been loaded and a final core has been produced, we *shrinkwrap* the image to produce assembly files. The details of this technique are outlined in section 4. These assembly files can then be linked together with the SBCL C runtime compiled for the NX in step (3) with the Nintendo SDK toolchain, producing a final ELF executable.

(11) `package`
The final step is to run all the other SDK toolchain bits to produce a signed application package, which can be deployed to a Nintendo Switch development kit to be run.

A notable extra wrinkle in this procedure is that the SDK is available only for Windows. To accommodate this platform restriction, the custom build system we developed to automate these steps can be run either directly from Windows using another ARM machine remotely to run the Linux bits, or it can be run from a Linux system with the ARM bits being run either locally or remotely and the Windows bits being run either remotely or through Wine[2].

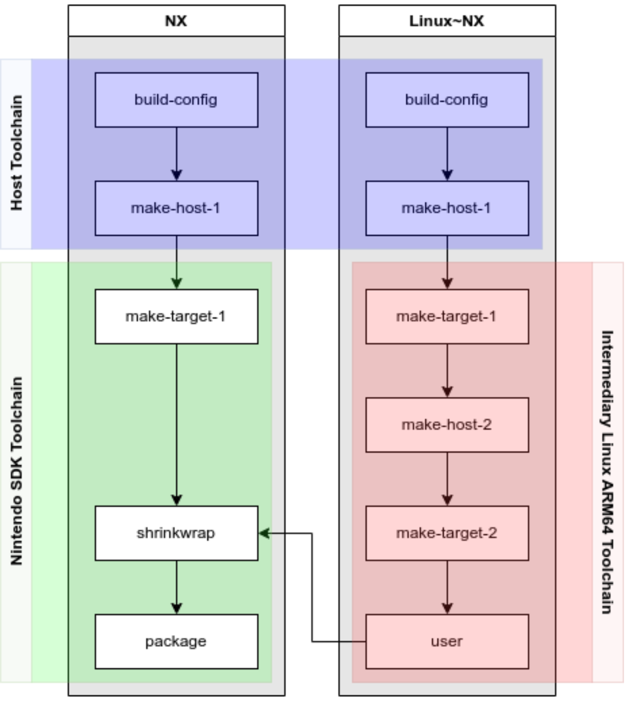


Figure 2: A diagram of the build steps and their dependencies

4 Relocation

Lisp objects in memory contain absolute pointers, with the result that dumping the in-process state of all Lisp memory spaces to disk results in a *core image* filled with absolute pointers. Traditionally, image-based Lisp implementations such as SBCL map their address spaces to fixed addresses for two reasons: First, reloading an image into a process is easier this way, since objects can be mapped back in the new image directly to where they were in the process that dumped the image. Second, some architecture backends optimize code generation by hardwiring the address of certain Lisp objects directly into the machine code.

4.1 Address Space Layout Randomization

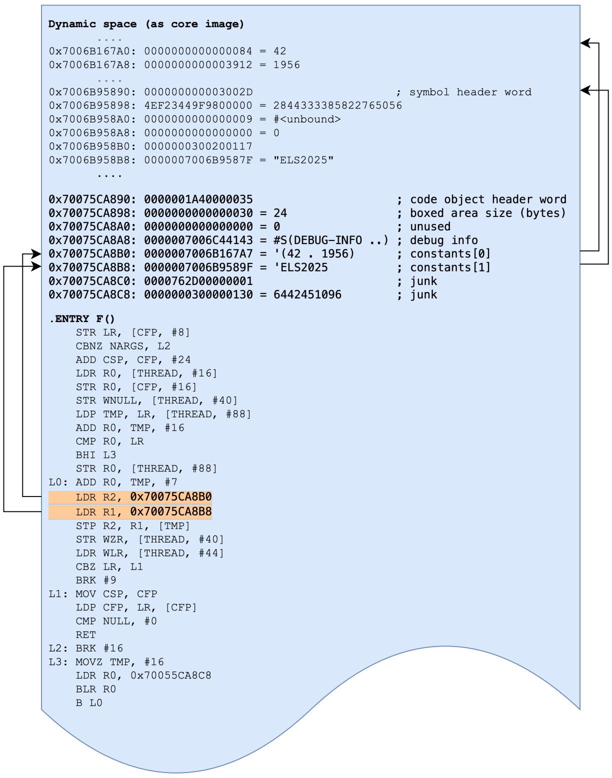
Because the NX operating system enforces address space layout randomization, the Lisp runtime cannot rely on being able to map its address spaces to fixed addresses. Hence, the runtime must relocate all objects present in the dumped image to the address spaces placed randomly by the operating system. As part of our work, we have extended the existing support for heap relocation to allow all other Lisp spaces used by SBCL to be relocated as well, with the result being that the runtime can fix up all absolute pointers mapped in from the image to point to the random addresses given to the runtime by the operating system. In addition, we modified the Lisp code generation to produce strictly position-independent code.

However, code objects still present a special problem on the NX for this relocation process. A code object in SBCL is represented like other Lisp objects in-memory, in that it has a header word describing the type and size of code object. The code object also contains slots pointing to the debug information associated with the code object as well as other boxed objects which represent constants referenced by the code instructions. This is so that the garbage collector can scan the code object and fix up these boxed pointers as it does with other Lisp objects. Finally, these slots are followed by the raw machine instructions and function entry points representing the compiled Lisp code. The code object associated with the function defined in Listing 1 is illustrated in Figure 3.

```
(defun f ()
  (cons '(42 . 1958) 'els2025))
```

Listing 1: A Lisp function referencing the constants `'(42 . 1958)` and `'ELS2025`.

Notably, the code object contains both executable machine code and absolute pointers to Lisp objects. Because the machine instructions are to be executed by the CPU, the pages the code object is allocated on must be marked executable by the runtime.



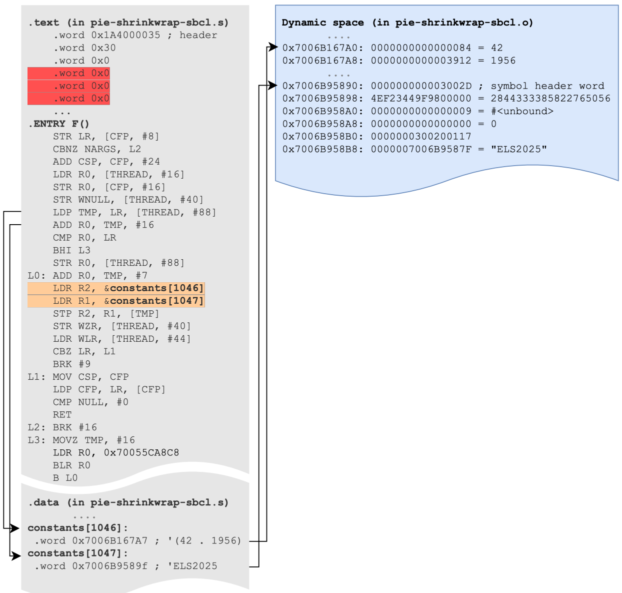
```
Dynamic space (as core image)
       ....
0x7006B167A0: 0000000000000084 = 42
0x7006B167A8: 0000000000003912 = 1956
       ....
0x7006B95890: 000000000003002D                      ; symbol header word
0x7006B95898: 4EF23449F9800000 = 2844333385822765056
0x7006B958A0: 0000000000000009 = #<unbound>
0x7006B958A8: 0000000000000000 = 0
0x7006B958B0: 0000000300200117
0x7006B958B8: 0000007006B9587F = "ELS2025"
       ....

0x70075CA890: 0000001A40000035                      ; code object header word
0x70075CA898: 0000000000000030 = 24                 ; boxed area size (bytes)
0x70075CA8A0: 0000000000000000 = 0                  ; unused
0x70075CA8A8: 0000007006C44143 = #S(DEBUG-INFO ..) ; debug info
0x70075CA8B0: 0000007006B167A7 = '(42 . 1956)       ; constants[0]
0x70075CA8B8: 0000007006B9589F = 'ELS2025           ; constants[1]
0x70075CA8C0: 0000762D00000001                      ; junk
0x70075CA8C8: 0000000300000130 = 6442451096         ; junk

.ENTRY F()
    STR LR, [CFP, #8]
    CBNZ NARGS, L2
    ADD CSP, CFP, #24
    LDR R0, [THREAD, #16]
    STR R0, [CFP, #16]
    STR WNULL, [THREAD, #40]
    LDP TMP, LR, [THREAD, #88]
    ADD R0, TMP, #16
    CMP R0, LR
    BHI L3
    STR R0, [THREAD, #88]
L0: ADD R0, TMP, #7
    LDR R2, 0x70075CA8B0
    LDR R1, 0x70075CA8B8
    STP R2, R1, [TMP]
    STR WZR, [THREAD, #40]
    LDR WLR, [THREAD, #44]
    CBZ LR, L1
    BRK #9
L1: MOV CSP, CFP
    LDP CFP, LR, [CFP]
    CMP NULL, #0
    RET
L2: BRK #16
L3: MOVZ TMP, #16
    LDR R0, 0x70055CA8C8
    BLR R0
    B L0
```

Figure 3: Representation of the compiled code object (in boldface) for the function `#'f` in memory, along with the cons and symbol objects it references.

However, executable pages are not writable on the NX, meaning the pointers to boxed Lisp data for the debug information and code constants cannot be fixed up as part of the relocation process on start-up. Furthermore, only the system loader is able to allocate executable pages by loading code from on-disk ELF .text sections, meaning it isn't possible to first fix up boxed object pointers in the code object *before* marking the code pages executable. In either case, the garbage collector would not be able to fix up these pointers either. Therefore, the code and data need to be separated, at first so that the runtime can relocate the absolute pointers to the Lisp objects, and later so that the moving garbage collector is able to fix up those references. This restriction requires the shrinkwrapping procedure to separate the data from the code and to rewrite the code in the image offline so that any references in code to code constants[1] are into a dislocated read/write space instead of into the code object itself, which is on an executable page and hence not writable on the NX. To demonstrate this process in more detail, our modified offline shrinkwrapping process produces two artefacts from a normal Lisp image:



```
.text (in pie-shrinkwrap-sbcl.s)
    .word 0x1A4000035 ; header
    .word 0x30
    .word 0x0
    .word 0x0
    .word 0x0
    .word 0x0
    ...
.ENTRY F()
    STR LR, [CFP, #8]
    CBNZ NARGS, L2
    ADD CSP, CFP, #24
    LDR R0, [THREAD, #16]
    STR R0, [CFP, #16]
    STR WNULL, [THREAD, #40]
    LDP TMP, LR, [THREAD, #88]
    ADD R0, TMP, #16
    CMP R0, LR
    BHI L3
    STR R0, [THREAD, #88]
L0: ADD R0, TMP, #7
    LDR R2, &constants[1046]
    LDR R1, &constants[1047]
    STP R2, R1, [TMP]
    STR WZR, [THREAD, #40]
    LDR WLR, [THREAD, #44]
    CBZ LR, L1
    BRK #9
L1: MOV CSP, CFP
    LDP CFP, LR, [CFP]
    CMP NULL, #0
    RET
L2: BRK #16
L3: MOVZ TMP, #16
    LDR R0, 0x70055CA8C8
    BLR R0
    B L0

.data (in pie-shrinkwrap-sbcl.s)
       ....
constants[1046]:
 .word 0x7006B167A7 ; '(42 . 1956)
constants[1047]:
 .word 0x7006B9589f ; 'ELS2025
```

```
Dynamic space (in pie-shrinkwrap-sbcl.o)
       ....
0x7006B167A0: 0000000000000084 = 42
0x7006B167A8: 0000000000003912 = 1956
       ....
0x7006B95890: 000000000003002D ; symbol header word
0x7006B95898: 4EF23449F9800000 = 2844333385822765056
0x7006B958A0: 0000000000000009 = #<unbound>
0x7006B958A8: 0000000000000000 = 0
0x7006B958B0: 0000000300200117
0x7006B958B8: 0000007006B9587F = "ELS2025"
```

Figure 4: Representation of the same code object and the cons and symbol objects it references after code/data segregation and the rest of the shrinkwrapping process.

- `pie-shrinkwrap-sbcl.s`: A textual assembly file is generated with all code objects in the original core image listed under `.text` sections. The pointers to debug information and code constants inside the code object are zeroed out, and the code constants referenced via the ARMv8 `LDR` instruction are displaced into a r/w `.data` section shared by all code objects. The `LDR` instructions are rewritten to access the constants from this section instead.
- `pie-shrinkwrap-sbcl.o`: A binary object file with the rest of the Lisp data from the core image section is generated. The

[1] As for the per-code-object slot containing debug information, we modified the Lisp system so that debug information is not accessed through the code object at all but rather through a weak hash table keyed by code object.

majority of the Lisp image is thus 'shrinkwrapped' into a large r/w `.data` section.

The reason for producing both a textual and a binary artefact is that it is easier to rewrite and produce textual assembly for code, and it is faster to process and link a binary artefact for the rest of the data. Figure 4 illustrates how the code object for the function `#'f` gets split up. The shrinkwrapping process then links both artefacts with the rest of the SBCL runtime to produce the final executable. The upshot of this process is that both the relocation process and the garbage collector can now scan and fix up all boxed pointers in the image at runtime without running into the read-only constraint for executable pages.

4.2 Linkage between Lisp and foreign code

Another part of the system impacted by the need for relocation is linkage between Lisp and foreign code. SBCL uses a linkage table that indirects calls to foreign code from Lisp so that foreign code can be loaded by the operating system at any address on startup and resolved in the linkage table. However, for callbacks into Lisp from foreign code, Lisp must create function pointers to snippets of assembly allocated in static (i.e. never moved by GC) Lisp space. These pointers can then be passed to and called from foreign code. To ensure that such function pointers in a saved image are valid even after the Lisp spaces containing the callback entry points are relocated, the function pointers must somehow be fixed up. We managed to solve this problem by having the offline shrinkwrapping process move the callback entry code into another `.text` ELF section while recording fixup information. The operating system can then load the callback entry code into executable pages, and the Lisp runtime can rewrite the static function pointers to point to where the code was loaded using the recorded fixup information.

5 Garbage Collection

SBCL's default garbage collector, *gencgc*, is a stop-the-world collector, meaning it must stop or *park* other threads in the process. Doing so is necessary so that the threads don't access any memory that the collector might move or change during collection. On RISC architectures like the ARM the Nintendo Switch runs on, the collector is also precise and must be able to identify which storage locations contain Lisp objects, so it is important that the threads are parked at code locations where the collector can safely do so.

In the discussion that follows, a *Lisp thread* is a native thread registered to the Lisp runtime that can either be executing Lisp or foreign code. Native threads created by Lisp are immediately registered to the runtime. A native thread created by foreign code that calls into Lisp via e.g. a callback gets a Lisp thread associated with it by the runtime when it starts executing Lisp code for the first time.

On Unix systems, the POSIX signal mechanism is used by default to park all Lisp threads besides the thread executing GC. By sending a signal to every other thread, each thread will enter a signal handler, which can then park the thread. This method is advantageous, since it leverages an operating system mechanism to interrupt thread execution and frees the thread from checking whether it should park. The garbage collector can also use the signal context mechanism to read and process the values of all registers and the Lisp stack at the point where the thread stopped. On the flip side, Lisp code must be generated in such a way that the GC almost always knows how to parse Lisp objects from registers or the stack at any code location. However, certain instruction sequences in the generated code then need to behave atomically in the sense that it is not safe to stop for GC at those locations, so extra bookkeeping is done to defer interrupts in those sequences. Lisp threads in foreign code are stopped and restarted just as Lisp threads are in Lisp code, though only the contents of its stack are scavenged.

On Windows, no equivalent signal mechanism is available, so a strategy using safepoints must be employed instead. A *safepoint* is a location in code which is known to be GC-safe, so the collector has enough information at that location to correctly function. The Lisp compiler then injects code (here termed *yieldpoint* code) at these safepoints, such as at function call boundaries and loop returns, which causes the thread to check whether it should yield for GC. Yieldpoint code should be inserted strategically so that it doesn't take too long for threads to park after a given thread decides to collect garbage, but also so that the overhead of checking whether to park the thread is as low as possible. Lisp threads in foreign code do not encounter yieldpoint code generated by the GC, but also do not need to be parked anyway, as foreign code is assumed to not access Lisp data. Its Lisp stack can still be safely scavenged while the thread is executing foreign code. However, since the thread is not stopped, it is possible that the thread may re-enter Lisp during a collection, in which case code for re-entry into Lisp must check if GC is running and park the thread and wait for GC to complete before the thread can execute Lisp code again. A similar consideration applies for Lisp threads in Lisp which enter foreign code while the collecting thread is waiting for all Lisp threads in Lisp code to stop; the Lisp thread must communicate to the GC that it is entering foreign code and hence the collecting thread need not wait for it to yield anymore, since the Lisp thread is about to start executing foreign code. For the purposes of this paper, we will call these points of communication at foreign function call boundaries *foreign yieldpoints*.

To summarize, the safepoint and signal mechanisms can be seen as inverses of each other: whereas code generation for the signal mechanism must ensure that the majority of code locations are safe for GC, the safepoint mechanism requires only that strategically chosen locations are safe for GC. The safepoint mechanism also allows foreign code to run without stopping at the cost of extra communication at foreign call boundaries, while the signal mechanism stops all Lisp threads regardless of whether they are executing foreign code or not.

In the existing safepoint ports, SBCL exploits hardware memory protection facilities provided by the operating system in order to make generated yieldpoint code as small as possible: GC toggles the read and write permission bits of pages in virtual memory in order to communicate to the thread various GC state transitions. Non-foreign yieldpoint code then consists of a single read instruction from a global page that a GC marks as non-readable when it wants all Lisp threads to park. The resulting page fault traps into trap handler code that yields the thread and progresses the state of the GC. Foreign yieldpoint code consists of a write instruction onto a thread-local page, informing the GC whether the Lisp thread is entering or leaving foreign code; the GC is then able to selectively cause the relevant threads to trap and wait for the state of the GC to progress or allow the relevant threads to continue toggling the write permissions of the thread-local page accordingly. One other advantage of using page permissions is that it allows race-free examine + wait + use in the latter scenario where the 'entry into/return from foreign code' flag word in memory needs to be read and appropriately acted on.

5.1 Challenges for the NX

As the NX also does not expose a POSIX signal mechanism, we chose to use the safepoint strategy as a starting point. However, as mentioned previously, even the safepoint strategy relies on hardware memory protection facilities to aid inter-thread communication between the collecting thread and other threads. The NX does not provide ways to install trap handlers for memory faults in release mode, so a different mechanism had to be devised.

For non-foreign yieldpoint code, it sufficed to replace the trapping read instruction with a polling instruction sequence which simply checks if a global 'stop now' word is true and if so, branches into a trampoline which calls into the trap handling code. Since there is no hardware trap and associated register context under this scheme, the trampoline also serves to spill all registers onto the Lisp stack so that the garbage collector correctly scavenges and fixes up the roots contained in the registers at the safepoint.

For foreign yieldpoint code, we replace the page permission scheme by designating an additional 'permission' word on the thread local page in addition to the 'entry into/return from foreign code' flag word to emulate the effect of the page permission scheme: the additional word flags whether write access to the other word is allowed. Since the thread containing the page and the collector thread may both race to access the two words in memory, the code sequence examining the permission word and dispatching to the correct GC state handler based on the value of the flag word requires explicit synchronization primitives. For now, we use locks to ensure race-free communication between the collector thread and any threads which are entering or exiting foreign code.

In addition to the existing safepoint mechanism requiring hardware page protection facilities, the garbage collector also used certain virtual-memory tricks to quickly zero out and/or return large chunks of memory back to the operating system. As the NX does not expose such sophisticated virtual memory management to the application developer, we had to rely on slower and more portable means of achieving the same effect.

6 Conclusion

We have demonstrated that it is possible to port SBCL even to a very restrictive platform that is hostile towards dynamic runtimes such as used for Lisp.

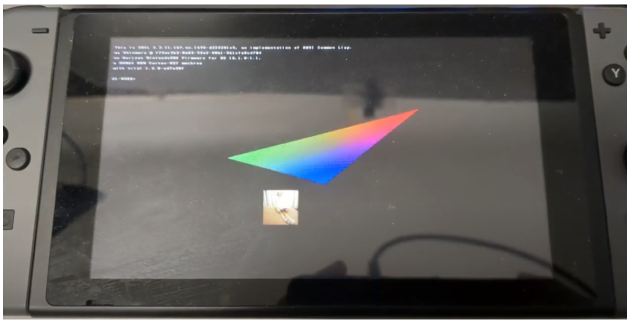


Figure 5: A demo running in the Trial Common Lisp game engine on the Nintendo Switch development hardware

Using a substitute build host that is similar enough to the target platform, we can compile all code ahead of time, then substitute the base runtime and rewrite the resulting code in order to create an executable suitable for the target platform.

We have also managed to update the garbage collector to work on operating systems with more restrictive feature sets than are found on a typical desktop operating system.

7 Further Work

Currently, we rely on the *fasteval* system to circumvent runtime compilation restrictions. This is especially vital for CLOS, since the discriminating function is compiled only on first call of a generic function. However, since we know that we don't dynamically add or remove methods, we should be able to pre-compile all of these functions as well. We'd like to add such a CLOS freeze step to the build pipeline, perhaps using a technique similar to *satiation* as outlined by Strandh et al.[10]

We also unfortunately have not yet had the time to successfully adapt Shirakumo Games' previous title, Kandria, to work on the NX. This port is currently in progress, and it is likely that further minor incompatibilities or bugs in our current SBCL port will surface as part of this effort.

The follow-up console to the Nintendo Switch, called "Switch 2" has also recently been announced. The device is backwards compatible with the NX and we expect that the operating environment will be very similar if not identical to that of the NX, just with more RAM and a better CPU and GPU. We hope to receive developer access to the Switch 2 and ensure that the port works for that, too.

Finally there are a number of improvements we've made that we would like to upstream to lessen the maintenance burden. We've already contributed a bunch of the changes back upstream, but a lot of it is also tied to the proprietary SDK and cannot be open-sourced due to the NDA, and some other changes are too contentious for the rest of the SBCL team to want to upstream.

8 Acknowledgements

We would like to thank Douglas Katzman for his help and advice for various parts of the porting effort, as well as the rest of the SBCL maintenance team for their continuous improvements to the

SBCL platform. We would also like to thank Robert Strandh and Jan Moringe for their feedback on the draft paper.

References

[1] Switchbrew. URL https://switchbrew.org/.

[2] Bob Amstadt and Michael K Johnson. Wine. *Linux Journal*, 1994(4es):3–es, 1994.

[3] Guiseppe Attardi. The embeddable common lisp. In *Papers of the fourth international conference on LISP users and vendors*, pages 30–41, 1994.

[4] Irène A Durand and Robert Strandh. Bootstrapping common lisp using common lisp. In *EUROPEAN LISP SYMPOSIUM*, 2019.

[5] Thomas Morgan. New switch mod delivers real-time cpu, gpu and thermal monitoring – and the results are remarkable. *Eurogamer*, 2020.

[6] Hayley Patton. Parallel garbage collection for sbcl. 2023.

[7] Christophe Rhodes. Sbcl: A sanely-bootstrappable common lisp. In *Workshop on Self-sustaining Systems*, pages 74–86. Springer, 2008.

[8] Gauvain Tanguy Henri Gabriel Roussel-Tarbouriech, Noel Menard, Tyler True, Tini Vi, et al. Methodically defeating nintendo switch security. *arXiv preprint arXiv:1905.07643*, 2019.

[9] Ethan H Schwartz. Dynamic optimizations for sbcl garbage collection. In *Proceedings of the 11th European Lisp Symposium on European Lisp Symposium*, pages 12–14, 2018.

[10] Robert Strandh. Resolving metastability issues during bootstrapping. In *Proceedings of ILC 2014 on 8th International Lisp Conference*, pages 103–106, 2014.