Proceedings of the

10th European Lisp Symposium

Vrije Universiteit Brussel, Brussels, Belgium
April 3 – 4, 2017

In-cooperation with ACM

Co-located with <Programming 2017>

Alberto Riva (ed.)

ISBN-13: 978-2-9557474-1-4
ISSN: 2677-3465

Contents

Preface

Message from the Programme Chair

Welcome to the 10thth edition of the European Lisp Symposium!

The selection of papers for this year's edition of the European Lisp Symposium shows, once again, that the Lisp family of languages is vibrant and evolving, and that at the same time it maintains its historical role as a "programming languages laboratory". This is demonstrated by the wide variety of topics in the program, and is perfectly exemplified by the two invited talks, one of them dealing with the application of Common Lisp to bioinformatics, and the other one discussing persistence in the functional programming world.

I was personally impressed with the number and quality of submissions received, which allowed us to put together a strong and varied program. As in previous editions, the symposium will include a tutorial, demonstrations, and lightning talks in addition to regular papers. This year ELS is co-located with the new <Programming> conference, something that will surely increase its audience, and is further proof of Lisp's place in the programming languages world.

I would like to thank the ELS Steering Committee for offering me the opportunity to help bring this symposium to life, and all the members of the Program Committee for their timely and insightful reviews. Special thanks go to Didier Verna and Irène Durand: their assistance throughout the whole process was invaluable, and contributed to making this endeavor a pleasure. I would also like to acknowledge all our sponsors, and the organizers of the <Programming 2017> conference for handling all local organizational aspects.

Finally, thanks to all authors and symposium participants for their role in keeping the Lisp community alive.

I wish you all a great time in Brussels!

Alberto Riva, April 2017

Message from the Organizing Chair

Dear fellow Lispers,

2017 is a special year. Although technically, the name of Lisp first appeared academically in 58, Lisp itself is turning 60 this year, and as it happens, ELS is turning 10. I'm having a hard time realizing that we have be running the symposium for 10 years already, and I remember the first occurrence, in Bordeaux, France (my native town, by the way) like it was yesterday...
ELS has come a long way since then. We now have an average of 70 to 90 people attending every year (from all over the world), we have had the ACM In-Cooperation-With status for two years in a row, and we literally blew the number of submissions this time. That is why a lot of thanks are in order.
First of all, a big thank you to Alberto Riva for chairing the 2017 edition. Thanks to Irène Durand for helping with the ACM process. We also owe a debt of gratitude to our sponsors, both old-time regulars and new; without them, ELS would simply not be happening. Finally, a big thank you to you, the ELS crowd. It seems that no matter how hard we try, there is no way to make you stop from attending! And this is a good thing because without you, there would not be a symposium either. Seeing you coming every year is as gratifying as it gets.

So let's just continue with Lisp, and on for a new decade!

Didier Verna, Brussels, April 3 2017

Organization

Programme Chair

- Alberto Riva, University of Florida, USA

Programme Committee

- Marco Antoniotti – Università Milano Bicocca, Milano, Italy
- Marc Battyani – FractalConcept
- Theo D'Hondt – Vrije Universiteit Brussel Belgium
- Marc Feeley – Université de Montreal, Canada
- Erick Gallesio, Université de Nice Sophia-Antipolis, France
- Rainer Joswig – Independent Consultant, Germany
- António Menezes Leitão – Technical University of Lisbon, Portugal
- Nick Levine – RavenPack, Spain
- Henry Lieberman – MIT, USA
- Mark Tarver – Shen Programming Group
- Jay McCarthy – University of Massachusetts, Lowell, USA
- Christian Queinnec – Université Pierre et Marie Curie, France
- François-René Rideau – Bridgewater Associates, USA
- Nikodemus Siivola – ZenRobotics, Ltd
- Chris Stacy – CS Consulting, USA
- Alessio Stalla – ManyDesigns s.r.l, Italy

Sponsors

We gratefully acknowledge the support given to the 10thth European Lisp Symposium by the following sponsors:

Brunner Systemhaus
Schulstraße 8
35216 Biedenkopf
Germany
`www.systemhaus-brunner.de`

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

LispWorks Ltd.
St John's Innovation Centre
Cowley Road
Cambridge, CB4 0WS
England
`www.lispworks.com`

Interdisciplinary Center for Biotechnology Research
2033 Mowry Road
University of Florida
Gainesville, FL, 32610
USA
`biotech.ufl.edu`

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

Common Lisp Foundation
Van Zuylen van Nijeveltstraat 161
2242LD Wassenaar
The Netherlands
`cl-foundation.org`

Invited Contributions

Identity in a world of values

Hans Hübner, LambdaWerk GmbH, Germany

Data persistence can add a great deal of complexity to application software, and making the gap between application and storage has been a constant field of research, experiments and products. In the object oriented programming paradigm, persistence seems to be a natural extension to object behavior, and even though one could argue that many persistent object systems have flaws and leak their abstractions, there is a large body of prior art and research in that area.
In the functional programming world, persistence does not find as natural a partnering abstraction, and it is often either conceptionally pushed to the boundaries of the application, or treated in an ad-hoc fashion interleaved with the beauty and conceptional rigor of pure functions.
The presentation discusses these forces and explores how Clojure's Software Transactional Memory system can be used to implement application data persistence.

Hans Hübner has three decades of experience as a programmer. Starting his career with Basic, Forth and Pascal, he followed the rise of object oriented software with C++. After ten years as a professional Common Lisp programmer, he is now heading a software company in Berlin, specializing in Clojure based communication systems for the healthcare industry.

How the strengths of Lisp-family languages facilitate building complex and flexible bioinformatics applications

Bohdan B. Khomtchouk, University of Miami Miller School of Medicine

We present a rationale for expanding the presence of the Lisp family of programming languages in bioinformatics and computational biology research. Put simply, Lisp-family languages enable programmers to more quickly write programs that run faster than in other languages. Languages such as Common Lisp, Scheme and Clojure facilitate the creation of powerful and flexible software that is required for complex and rapidly evolving domains like biology. We will point out several important key features that distinguish languages of the Lisp family from other programming languages, and we will explain how these features can aid researchers in becoming more productive and creating better code. We will also show how these features make these languages ideal tools for artificial intelligence and machine learning applications.
We will specifically stress the advantages of domain-specific languages (DSLs): languages that are specialized to a particular area, and thus not only facilitate easier research problem formulation, but also aid in the establishment of standards and best programming practices as applied to the specific research field at hand. DSLs are particularly easy to build in Common Lisp, the most comprehensive Lisp dialect, which is commonly referred to as the "programmable programming language".
We are convinced that Lisp grants programmers unprecedented power to build increasingly sophisticated artificial intelligence systems that may ultimately transform machine learning and artificial intelligence research in bioinformatics and computational biology.

Bohdan B. Khomtchouk is an NDSEG Fellow in the Human Genetics and Genomics Graduate Program at the University of Miami Miller School of Medicine. His research interests include bioinformatics and computational biology applications in HPC, integrative multi-omics, artificial intelligence, machine learning, mathematical genetics, biostatistics, epigenetics, visualization, search engines and databases. He will be starting his postdoctoral research work at Stanford University this year.

Session I: Tools

Common Lisp UltraSpec - A Project For Modern Common Lisp Documentation

Michał Herda
Faculty of Mathematics and Computer Science
Jagiellonian University
ul. Łojasiewicza 6
Kraków, Poland
phoe@openmailbox.org

ABSTRACT

The Common Lisp programming language has many bodies of documentation—the language specification [2] itself, language extensions, and multiple libraries. I outline issues with the current state of Common Lisp documentation and propose an improvement in form of the Common Lisp UltraSpec. It is a project of modernizing the LaTeXsources of the draft standard [1] of Common Lisp and ultimately unifying it with other bodies of Common Lisp documentation.

This is achieved through semi-automatic parsing and formatting of the sources using a text editor, regular expressions, and other text processing tools. The processed text is then displayed in a web browser by means of a wiki engine.

The project is currently in its late phase with the greater part of the specification parsed and edited. A demo of the project is available at http://phoe.tymoon.eu/clus/.

CCS Concepts

•Software and its engineering → Documentation;

Keywords

Common Lisp, Documentation, Specification

1. INTRODUCTION

The current state of Common Lisp documentation in general is not satisfactory to me and many other people I have talked with throughout my relatively short experience as a Lisp programmer.

The bodies of documentation are often incomplete and out-of-sync—they do not reflect the current state of a particular library. They tend to be outdated and in need of modernization. They contain errors and mistakes and are either unmaintained or non-editable. They are non-hyperlinked and do not refer to each other in a way that makes it easy to navigate between them. They are scattered across the web and depend on multiple separated web hostings, increasing the risk of losing access to them if a particular server containing their data fails.

I see multiple issues this state of matters causes to Common Lisp programmers, newbies and seasoned Lispers alike. The experienced programmers are most likely used to these pieces of documentation and that fact might minimize the losses they suffer, but, from my personal experience, newcomers are often lost and disoriented because of general fragmentation of the documentation, lack of maintenance, poor visual appeal, and lack of an obvious way to fix this state of things for those willing to do so.

For the sake of justice, I must remark that the things I have mentioned above do not disqualify said documentation from being useful and invaluable for many Common Lisp programmers. The manuals specifying the language, its extensions and many libraries are good enough as a building material and the multitude of Common Lisp applications proves this point. But I consider this state of matters to be improvable.

This paper contains an idea and requirements for an improvement of the state of Common Lisp documentation. It also contains a description of an implementation of this idea, the sources I have used, the full technological stack I have utilized so far, the problems and issues I have encountered, the benefits and disadvantages of my approach, a mention of a notable mistake I have made and the plans I have for the future extension and expansion of the aforementioned work.

2. PREVIOUS WORK

2.1 ANSI CL Standard

2.1.1 Published standard

The ANSI Common Lisp standard[2] is the specification of the Common Lisp language, published in 1994.

The specification itself is a written document of over one thousand pages of formatted text. Such a large amount of technical data was a natural candidate to be turned into a digital database browsable by humans.

2.1.2 Derived work

2.1.2.1 Common Lisp HyperSpec.

The most famous work derived from the ANSI CL standard is the Common Lisp HyperSpec[1] (henceforth abbre-

[1]http://www.lispworks.com/documentation/common-lisp.html

viated as CLHS). It is a hyperlinked Web version of the original standard, which allows for easy navigation of the standard. Its HTML form also allows searching and quick access using external search engines, such as Google or IRC[2] bots.

The CLHS was itself created by an automated tool which converted TeX into HTML.

The CLHS is released under an essentially non-free license which allows verbatim copying of CLHS as a whole, but prohibits any changes to it or creating any derivative works based on it. It is therefore not possible to build a unified piece of Common Lisp documentation based on the CLHS.

Further information about the history of creation of the standard and the CLHS is available in the work by Kent M. Pitman, *Common Lisp: The Untold Story*[4].

2.1.2.2 Franz Online ANS.

Another notable work is the Franz Online ANS[3], created by Franz Inc. and also being "a semi-mechanical translation of the original TeX into HTML". The presentation of the language standard is copyrighted by Franz. I have not attempted to contact the owners of that presentation—I only learned of its existence very recently, at which point I had already done most of the parsing work.

2.2 Lisp content aggregators

The Common Lisp documentation spans far and wide beyond the Common Lisp standard. Even during the time of Common Lisp standardization, many extensions to the language existed, with their respective pieces of documentation.

From this arose the obvious need of aggregating Lisp content, both with respect to code and documentation. Below, I will outline three contemporary services which provide Lisp users with content.

2.2.1 l1sp.org

The l1sp.org[4] service is a content aggregation tool created by Zach Beane. Its main purpose is to enable lookup of symbols inside various pieces of documentation scattered around the web. It currently contains links to over 20 pieces of documentation, including the CLHS, documentation for various CL implementations and many commonly used language extensions and libraries, such as the Metaobject Protocol[5], ASDF[6] and Alexandria[7].

This redirection service is very useful and allows for easy lookup, but such an approach depends on the presence of all the pieces of documentation in their respective places all around the Web. Also, the pieces of documentation are not linked to each other; for example, it is impossible to reach the Common Lisp reference from within the Metaobject Protocol reference and vice versa. The individual pieces of documentation also greatly vary in style: both the typesetting and graphical layout and the textual form in which the information is presented to the reader.

2.2.2 Quicklisp

Quicklisp[8], created by Zach Beane, provides a centralized repository of Lisp libraries through a piece of Lisp code, which in turn allows the programmer to automatically resolve dependencies, download and compile a particular library on their Lisp system.

While Quicklisp is invaluable as a library repository, it does not provide any sort of documentation service—it is outside the scope of the Quicklisp project. Therefore it cannot be a direct aid in creating a Lisp documentation project.

2.2.3 Quickdocs

The Quickdocs service[9] is a content aggregation tool created by Eitaro Fukamachi expressly for automated collection and generation of documentation for Common Lisp libraries; therefore, it aids the issue outlined in the paragraph above by expanding Quicklisp with documentation capabilities. The documentation itself is generated automatically from the source code of libraries found in the Quicklisp repositories. It consists of the description of a Quicklisp system and a list of exported symbols along with the type of objects they refer to and any documentation strings they may contain.

Such automation provides a very good and aesthetically pleasing means of reading about the protocol of a given system. The issue with such automatic generation is that it forces the authors of libraries to follow a convention of documenting their libraries in a particular way, which must be recognizable by the tool parsing the Quicklisp systems. Otherwise, the documentation will not be visible in Quickdocs. Additionally, Quicklisp descriptions often contain little more than links to external websites documenting the code, which deprives Quickdocs of the ability to automatically generate documentation for it.

3. MY WORK

3.1 Idea

The idea of creating a unified, hyperlinked Common Lisp documentation that would additionally span over multiple language extensions and libraries has been growing in me since I began my journey with Common Lisp back in 2015. I have been irritated by the separation of particular bodies of Lisp knowledge and lack of connection between them. In the beginning of 2016, I started looking for means to improve this situation.

During my research, it became obvious to me that—no matter which particular way would be chosen in this case—the project of creating and maintaining a modern, unified repository of Common Lisp documentation would require substantial work. It would be necessary to choose the appropriate pieces of work the repository would consist of, find most recent versions of their documentation, solve any legal issues of creating derivative works of them, parse the existing documents, and keep the repository maintained in the face of the changing versions of Common Lisp libraries.

3.2 Requirements

The idea for building such a piece of documentation was presented at the European Lisp Symposium 2016 during a

[2]Internet Relay Chat.
[3]http://franz.com/search/search-ansi-about.lhtml
[4]http://l1sp.org
[5]http://metamodular.com/CLOS-MOP/
[6]Another System Definition Facility, https://common-lisp.net/project/asdf/
[7]https://common-lisp.net/project/alexandria/
[8]https://www.quicklisp.org/
[9]http://quickdocs.org/

lightning talk[3] that I gave. I would like to expand on a particular slide of that presentation, which outlines the qualities I expect of a Common Lisp documentation project.

1. **Editable:** It needs to be modifiable and extensible by anyone willing to expand it.
2. **Complete:** It should aim for completeness and maximum coverage of the Common Lisp universe.
3. **Downloadable:** It should be usable locally, without an Internet connection.
4. **Mirrorable/Clonable:** It should be easy to create mirrors and copies of it on the Internet and on hard drives.
5. **Versioned:** It should use version control.
6. **Modular:** It should be splittable into separate modules with cross-module hyperlinks breaking as the only side effect.
7. **Updatable:** It should be easy to update it to its newest version.
8. **Portable:** It should be exportable as a static HTML website.
9. **Unified:** It should be consistent in style.
10. **Community-based:** It should belong to the Lisp community and be further developed and extended there.

The implementation of this idea is a project that I have named the Common Lisp UltraSpec, henceforth abbreviated CLUS.

The dpANS[10] source[1] makes it **editable**.

Git[11] as version control makes it **downloadable**, **mirrorable/clonable**, **versioned** and **updatable**.

Hosting it on GitHub[12] allows it to be **community-based**.

DokuWiki[13] allows it to be **modular** and **portable**.

The goals are to make it **complete** and **unified**.

3.3 Source

The whole process was made possible by the availability of the TEX source code for "draft preview Americal National Standard", abbreviated as dpANS, for Common Lisp. These sources were put into public domain by Kent M. Pitman and other members of the X3J13 committee.

While not being the actual standard itself, the dpANS is close enough to it to be usable as a proper reference of Common Lisp while also being in the public domain, which allows me to create derivative works of it. It turned out to be a feasible source upon which I could begin implementing the first part of the UltraSpec.

[10]Draft preview American National Standard.
[11]https://git-scm.com/
[12]https://github.com
[13]https://www.dokuwiki.org/

3.4 Work done so far

At the moment of writing these words, I have translated all dictionary entries and the glossary from the dpANS sources into pages in DokuWiki markup syntax, corrected the pages, and hyperlinked the code examples found there.

Additionally, I have created a customized version of DokuWiki meant for displaying the CLUS content. While I have not yet published the source code of this modified DokuWiki instance, it was successfully deployed[14] with the specification data translated so far.

3.5 Used methods and tools

The presence of a feasible source for creating a unified and modernized piece of Common Lisp documentation allowed me to download the draft and start looking for means of parsing and processing it. The following subchapters describe the tools I have been using and explain the reasons for them being chosen.

3.5.1 Main method

My method of editing the original sources involves cutting the original TEXfiles into pieces, one per page, which is especially important for the dictionaries.

Then, I utilize regular expressions to remove comments, replace parts of TEXmarkup with their corresponding DokuWiki markup and remove the TEXmacros that do not make sense in the new context. I also automatically create hyperlinks to other pages.

After this semi-automatic process is done, I manually clean and realign the text wherever necessary, manually edit more complicated markup such as mathematical symbols, reindent the tables, and fix broken hyperlinks. I also do all required fixes in the specification text. I take care to ask other people for advice whenever I edit the original text, especially whenever I change the parts that construct the formal specification (and not e.g. notes, or examples.)

Once the files are parsed and cleaned, I create the proper file structure by renaming and moving the files into the DokuWiki directory tree. Simply moving a file into its proper location immediately allows DokuWiki to be able to render it, so creating or updating files is enough to update the CLUS version visible in the browser.

As noted elsewhere, I have not written any automated tool to do the required work for me; the ability to execute a series of regular expressions on multiple files works well enough for me.

This method was developed on the source files of the dpANS3 and might work to some extent on other TEXfiles without modification, as the choice of used TEXmacros varies between documents. Other pieces of documentation, which use other formats, such as HTML[15], will require further modifications to this method.

3.5.2 Notepad++—the text editor

When it came to the main editor for doing most of the parsing work, I could choose between Emacs[16] and Notepad++[17], a pair of GPL-licensed[18] programmer's editors.

[14]http://phoe.tymoon.eu/clus/
[15]HyperText Markup Language
[16]https://www.gnu.org/software/emacs/
[17]https://notepad-plus-plus.org/
[18]https://www.gnu.org/licenses/gpl-3.0.en.html

Emacs is a keyboard-oriented editor, available for all major operating systems; Notepad++ is a WYSIWYG, keyboard-and-mouse-oriented editor written for Windows that I was able to run on my Linux setup using the Wine[19] toolkit.

I ended up doing most of the actual processing of the sources in Notepad++. I found its bulk-editing RegEx features very handy and easy to learn and use. This paper was written in Emacs using its TeX modes.

3.5.3 DokuWiki—the engine for displaying HTML

DokuWiki is a GPL-licensed wiki software written in PHP[20]. In my experience, it was able to fulfill all the requirements I had for a displaying engine. It simply works and allows me to deliver the contents in a readable and aesthetically pleasing way. It does not need database access and instead relies on flat files, which allows for easy versioning of the data with Git. It has a simple markup syntax that I consider sane. It is extensible and hackable, which so far proves very useful. Also, I have had some previous experience in using and configuring it.

The displaying engine is also a part that can be replaced later for another solution - in this case, I am not bound to DokuWiki as the only displaying engine.

3.5.4 Regular expressions, GNU coreutils—tools for parsing the sources

The most important choice that I had to make in the beginning was how to parse the source files of dpANS. The source code is a large body of LaTeX code, created by multiple people over a large span of time. It contains highly customized TeX macros, used irregularly among the source code.

The initial research led me towards TeX parsers written in various languages, such as Parsec[21] written in Haskell[22]. My initial attempts of feeding the dpANS sources to the parsers I found were failures though; the individual bodies of code were too complex and my knowledge about these parsers was too little for me to succeed. I realized that, in order to properly parse the TeX source code of the draft, I would need to create a substantially large set of parsing rules; even afterwards, I would need to spend a lot of time doing manual polishing and fixing of the corner cases, such as TeX macros used only in a few places within the source files or actual mistakes within formatting, such as utilizing function markup for macros and vice versa.

Because of this, I decided to abandon the approach of parsing the standard with a parser capable of processing TeX directly and instead go for a simpler choice: utilizing a set of regular expressions to parse a subset of the used TeX. It would mean later polishing the preprocessed data by hand, though I would like to note that this last step—manual processing and fixing—would be necessary anyway regardless of the technique used.

My editor of choice, Notepad++, contained a powerful enough RegEx engine that was capable of guiding me through the process. Various bulk edits were also made through assorted Unix utilities: grep, sed, awk, rename.

[19]https://www.winehq.org/
[20]https://php.net/
[21]https://wiki.haskell.org/Parsec
[22]https://wiki.haskell.org/

3.5.5 Git—versioning system, GitHub—project hosting

The data for the whole project is kept in a Git repository, stored at GitHub[23] and publicly available. Because DokuWiki keeps all data as flat text files, I can easily modify and deploy new versions of data to upstream websites.

3.6 Problems encountered

Most of the problems I have encountered are connected with the dpANS sources being a big and complicated piece of documentation, and using regular expressions to parse the TeX sources.

As I have mentioned before, the source code had been created over a lengthy period of time with multiple people contributing to it. Because of that, many parts of the specification are formatted differently: they utilize different TeX macros, specific to the people creating the source and the part of the language that was worked upon. Despite the irregularities, I was able to employ the regular expressions and capabilities of my editor to fix most of the cases globally and fix the corner cases manually.

A significant part of the required work was hyperlinking. Although I was able to parse the code for TeX glossary entries, I also needed to take the English grammar into account, such as plural and past forms of glossary entries.

I have had some minor problems with DokuWiki's rendering and markup capabilities, though none of them have been significant enough to be mentioned in detail here.

3.7 Differences to source materials

I do not claim that my markup is better than the original TeX markup of dpANS—if anything, it is different, allowing me to directly display the contents inside the DokuWiki engine. Most of the work I have done does not regard the markup itself, but rather making tens of minor corrections within the specification regarding to spelling, linking and obvious mistakes[24].

If I may make such a bold claim here - as I worked through and with the text of the specification, I got a growing impression that the third draft of the specification, subsequently accepted by the X3J13 committee, lacked a final, global review from an external source that would catch all the minor mistakes and errors in the specification. If I, a person without much experience in Lisp and untrained in creating specifications, am able to catch and fix tens of them inside the TeX sources, then it was also possible twenty three years ago, when the test of the specification was turned into a standard.

A major part of my work consists of doing such a "final review", which puts me in the position of being one more editor to the Common Lisp specification.

4. CONCLUSIONS AND FUTURE WORK

4.1 Benefits and Disadvantages

The benefits of my approach come as logical continuations of the slogans used in section 3.2.

The most obvious one, which is also the goal of the project, is the construction of a contemporary source of Common

[23]https://github.com/phoe/clus-data
[24]Such as, quoting the specification page for `PROG2`: ***prog2*** *[returns] the primary value yielded by* ***first-form****.*

Lisp documentation and a single resource capable of containing most of the knowledge a Common Lisp programmer might need, under a license permitting its free modification, expansion and reuse.

Another upside is modernization of the specification by fixing its issues and bugs, expanding its examples sections, clarifying any inconsistencies and questions that have emerged since the creation of the standard and giving it a more aesthetically pleasing look. I consider it important for Lisp to have a modern, browsable documentation, especially in light of various comments that I have gathered throughout the Internet, which emphasize such a need.

A beneficial side effect of my approach is the generation of a version of the Common Lisp specification in a markup format. Such a format can then be easily parsed by automated tools to produce a document of any required typesetting qualities.

The disadvantages of my current approach occur on different layers.

First of all, it is easy to keep a single static website on the Web for years without any changes, but CLUS is far from static because of its design. The body of code that CLUS will turn into, as the time progresses, will require maintenance in order to remain clear and readable; it will require reviewers to check the input from anyone wanting to contribute to the CLUS repositories.

Second, although it does apply specifically to the dpANS sources, parsing and hyperlinking the chapters of the specification takes significant time. Additionally, because of the variety of forms other bodies of Lisp documentation have, it will be non-trivial to import them into CLUS. It will require a separate effort to have them parsed and prepared for inclusion.

Third, the legal status and licensing issues of the various pieces of documentation will require separate thought. Creating a compilation work of all these elements will be essentially creating a derivative of them all and legal caution will need to be taken in case of documents with unknown or confusing legal status. It might be required to negotiate the terms of inclusion of particular pieces of work into CLUS with the respective holders of rights to them.

4.2 Thoughts

4.2.1 dpANS as humanistic material

Among all the literature available for studying Common Lisp, I would like to mention the dpANS source files as a valuable reading matreial from a non-technical point of view.

The standard was created before the era of ubiquitous versioning systems. Because of this, the draft source contains many comments, some of them timestamped. They show the technical problems and decisions the langauge specifiers faced and solved in the process of creating a formal standard for a programming language. They also outline the features which were deprecated and removed—or, on the contrary, created and added along the way, some of which I personally find quite enlightening. What I want to emphasize here, though, is that they show X3J13 as a group of human beings working on a common goal. The comments there show various aspects of their work: from communicating messages between particular people, through decision-making and commented-out pieces of specification itself, to the in-jokes and humor of the people[25].

In my opinion, studying the original sources for all three draft previews (all of which are available online) might be valuable for any person who wants to research specification development or software development in general from a more human point of view as well as Lisp programmers who are interested in extending their background and the process through which Common Lisp came to life.

4.2.2 Translator or editor?

Another thought that I would like to mention here is the fact that, in the beginning, I had imagined my work as simple translation of the sources from their $\TeX$ format into wiki markup in order to let the DokuWiki engine format them into HTML. Reality has superseded these ideas—I quickly realized that the standard itself has its share of inconsistencies, bugs and other issues. It is of course expected for such a huge body of documentation to have issues and these issues do not undermine the value of the specification as a whole, but I have unexpectedly found myself to be able to fix them as I progress through the sources.

Suddenly, from a simple translator, I had become an editor of the Common Lisp standard itself. What I am creating right now is not the draft sources being translated into DokuWiki markup—it is an edited version which contains many improvements and fixes to many issues that were impossible to fix in the previous CL specifications based on the work of X3J13.

It is a very responsible role that has emerged—but also one that I consider very satisfying.

4.3 Mistakes

I have made a mistake that is important enough for me to want to point it out here. Here is where my lack of experience as an editor of formal documents shows—I have not done any formal editing work before. When I began parsing the specification, I decided to fix all obvious errors and mistakes on the fly, as I progress through the text. Because of this, I have not separated the stages of converting the specification to the new markup and editing its text; changes in the text appear because one person—me, as an editor—decides that they should be changed.

This is contrary to the experience of Kent M. Pitman—*"the primary qualification for Editorship was trustworthiness—particularly, the ability to resist the urge to 'meddle' in technical matters while acting in the role of 'neutral editor'"*.[4] Additionally, KMP noted that *"it was necessary that the community have complete trust that the only reason a change might be made to the meaning of the language was if there was a corresponding technical change voted by the committee"*.[4]

My situation is different from Kent's. The X3J13 had finished its work; the specification is done; there is no formal committee that an editor might need to obey or serve. I do not (yet) work on editing the specification with other people directly. Nonetheless, the Lisp community is whom that I attempt to serve here, trying to create a basis for an extensible Lisp documentation that might in turn accomodate various needs of the community itself. It is a responsible role that I do not want to do loosely.

[25]Such as: `%This list conjured by KMP, Barrett, and Loosemore. -kmp 14-Feb-92`

As I progressed through the text of the specification, I have made changes to its formal and informal parts as I saw fit and proper, asking the community every time I had any doubt about the original meaning or wanted to ask for support for a particular change. While my intention was to produce text that is clearer and better, it is obvious that such a frivolous and undocumented process applied to a strictly formal document produces text that is perhaps a better documentation - but I, from my own point of view, cannot call it a formal specification anymore.

This mistake that I have made has generated more work for me to do later. Once I finish converting the whole specification, I will need to make a review of the whole CLUS with the standard in one hand and the UltraSpec text in the other, pointing out the differences and collecting them out into a separate document listing the changes and differences between the two texts, so that such a list may later be reviewed by the Lisp community and the purpose of each change may be discussed. Only after such a document exists, I will consider the specification part of the UltraSpec to be formally complete.

4.4 Plans

It is impossible to speak of future plans without mentioning the Lisp community here.

The Common Lisp UltraSpec was meant from the start to be a community-based project, meaning that it belongs to the Lisp community and is meant to be utilized and expanded within it. I hope that other people will aid me in my process by suggesting changes, submitting patches, possibly integrating the documentation for respective Common Lisp libraries into the code and maintaining them later on.

There is an interesting project to convert the markup I have used into S-expressions[26] that can then be parsed and understood by Lisp—as data—and text editors—as editable, formatted text. I hope that it will make it possible one day to freely edit the CLUS source code in "lispy" editors, such as Emacs or Climacs[27].

Once the specification is completely integrated, I intend to extend its scope to include common facilities and extensions included and/or used in most contemporary Common Lisp implementations, such as the Metaobject Protocol, ASDF, Quicklisp and the compatibility libraries which provide cross-platform functionalities not included in the standard such as concurrency or networking.

There are collections of issues regarding the specifications, found throughout the years by the community. Some are organized, such as the collection on CLiki[28]; some, I hope, are in on the hard drives of the Lisp community. Previously, it was impossible to integrate them into proprietary texts of former editions of the specification, but with the creation of CLUS, this will no longer be the case.

I want to create quality standards for the respective types of pages and enforce them, in order to keep the quality of the documentation high and its style consistent across pages and modules.

In the far future, it might be possible that the CLUS might become a basis for a revised Common Lisp specification that might include changes like the CDRs[29] or the CL21 project[30]. I do not directly plan this far ahead, but I keep such a possibility in mind; I consider the kind of organic work that I am doing a required basis for any further attempts to improve the specification of Common Lisp.

4.5 Acknowledgements

I would like to thank Professor Robert Strandh, doctors Włodzimierz and Małgorzata Moczurad, and Philipp Marek for general support during the creation of this work, including, but not limited to, proofreading, advice, and moral support; one more thank you to Robert Strandh for allowing me to use his TEX layout for papers.

I would additionally like to thank the people from the `#lisp` and `#lisp-pl` IRC channel for a lot of support on this paper and the Common Lisp UltraSpec project in general.

5. REFERENCES

[1] *Draft proposed INCITS 226-1994 Information Technology, Programming Language, Common Lisp X3.226:199x. Draft 15.17, X3J13/94-101.* American National Standards Institute, 1994.

[2] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp.* American National Standards Institute, 1994.

[3] M. Herda. Common Lisp UltraSpec. Presentation slides for Lightning Talks of the European Lisp Symposium 2016, day 2.

[4] K. M. Pitman. Common Lisp: The Untold Story. Web version.

[26] https://www.reddit.com/r/lisp/comments/5xcafp/c/deje13k/

[27] https://github.com/robert-strandh/Second-Climacs

[28] http://www.cliki.net/proposed\%20ansi\%20revisions\%20and\%20clarifications

[29] https://common-lisp.net/project/cdr/

[30] http://cl21.org/

Loading Multiple Versions of an ASDF System in the Same Lisp Image

Vsevolod Domkin

vseloved@gmail.com

Abstract
In this paper, we present a proof-of-concept solution that implements consecutive loading of several versions of the same ASDF system in a single Lisp image. It uses package renaming to dynamically adjust the names of the packages loaded by a particular version of a system to avoid name conflicts on the package level. The paper describes the implementation, possible usage, and limitations of this approach. We also discuss the deficiencies of ASDF that impede its use as a basis for developing such alternative system manipulation strategies and potential ways to address them.

CCS Concepts: • **Software and its engineering~Software configuration management and version control systems** • *Software and its engineering~Software libraries and repositories* • *Software and its engineering~Software evolution*

1. Introduction

The problem of supporting simultaneous access to multiple versions of the same library in the same software artifact is relevant to the software projects that rely on many third-party components and/or have a long development time span. Due to the separate evolution of third-party libraries, the situations may arise when they may depend on different incompatible versions of software that share the same name. Besides, even the software project under direct control of the user itself may necessitate dependency of several versions of the same library that support different behaviors and functionality. This problem is often called dependency hell[1] (and, in different programming language environments, is known as "DLL hell," "jar hell" etc.) It manifests either in the inability to build the target software as a result of name conflicts or in the unsolicited redefinition of parts or whole functionality by the conflicting packages, which may happen silently or vocally, depending on the particular environment.

In Common Lisp, packages[2] provide namespacing capabilities to reduce the risk of name conflicts between symbols. The packages are first-class globally-accessible dynamic objects. Due to the existence of a centralized “registry” of known packages in the running Lisp image, name conflicts may arise when two independent software artifacts that include the definitions of the packages with the same names or nicknames are loaded into the same image. The conflict will manifest in the redefinition of the previously loaded package by the one loaded later, which will result in an extension of the package's external API and, possibly, an unexpected redefinition of parts of its functionality that have the same names (be it functions, classes or variables). This risk grows with the development of the library ecosystem, and such cases have been already reported[3] for the Quicklisp[4] distribution, which is the largest repository of Common Lisp open source libraries. An even higher risk of conflict exists not between independent pieces of software, but between different versions of the same software. In this case, a redefinition of the previous version of the package with a newer one may be intended (in case of upgrade), but if non-backward compatible changes are introduced, this will, potentially, mandate the upgrade of all of the package's dependents. Such situation may be undesired, especially in the case of third-party dependents that are not under the control of the user. Moreover, it may be beneficial to utilize both old and new versions of the upgraded package's functionality. The described risks are most critical for production software that is usually dependent on many external libraries and is produced via a process of automatic build (often using Continuous Integration[5] systems), not allowing for manual intervention in case of unexpected conflict.
The ways to approach dependency conflicts include administrative measures (adherence to a particular versioning or naming policy - see Semantic Versioning[6] or package renaming on incompatible changes proposal[7]) and

programmatic solutions. Not questioning the value of proper software development practices, it should still be noted that administrative measures have a crucial limitation of impossibility to fully regulate the activity of third-party developers, especially for the software that already exists and may not even be maintained at the moment. That is why a programmatic solution is essential, but, currently, there is no library or feature of an existing tool that allows dealing with them.

Most programming language environments do not provide a comprehensive user-friendly way of automating version conflict resolution due to the limitations of their namespacing capabilities (see, for example, the situation in Python[8]). One notable exception is the JVM, which allows to extend the standard classlloader[9] to dynamically load several classes that have the same name — the capability used by OSGi[10] to systematically handle version conflicts. Furthermore, the upcoming Java 9 will include the project Jigsaw[11] that introduces a new module system also capable of handling version conflicts by default. JavaScript is another interesting case as it initially lacked the concept of a package or module, and when it was later introduced via the Module pattern[12] and its derivatives, the standard objects were used to host modules with incapsulation of dependencies within the object's private scope that allows to not register the loaded dependency's name in the global public scope, when it is not necessary, thus preventing the version conflict altogether.

In Common Lisp, the low-level solution to conflicts of package name clashes is the standard `rename-package`[13] function. Using it allows possible to avoid name conflicts by changing the reference to the first of the conflicting artifacts before the second one is initialized. If the primarily loaded version of the package is renamed, a new one may be loaded without name conflicts. Such renaming, however, requires careful orchestration as the process of loading different packages is usually complex and not fully transparent, and the renaming should take place after the other packages, which are the users of the one being renamed, are loaded. This may not be possible in the general use case because of the potential redefinition and additions to the packages at program runtime. However, in the common case of loading the source code and then working with the image without any subsequent modifications to the dependencies, the renaming can be performed reliably.

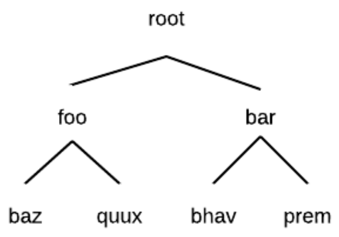
Packages are a source code level concept, while for the purpose of automation of the compilation and loading of the source code itself, a de facto standard abstraction provided in Common Lisp is a “system”[14]. It provides a way to group relevant source code files and other file-based resources and to specify the order of their compilation/loading. The currently adopted implementation of the system concept and related APIs is ASDF[15]. ASDF performs the similar role to make[16] and Ant[17] in other programming environments, and it allows for reproducible programmatic bundling, distribution, and initialization of both software libraries and applications. The system in ASDF supports the notion of version, which allows to logically distinguish different versions of the same software packages. It also allows specifying dependencies between systems (including versioned ones). Putting different packages (even with the same names) in different ASDF systems or putting different versions of the same-named package in different versions of an ASDF system allows to approach the problem of name conflicts, provided there is a way to control the loading of those systems and perform package renaming at the necessary points of the process. Currently, ASDF doesn't implement such functionality. Moreover, it has a number of key limitations preventing the implementation. First of all, at any moment in the running Lisp image, only a single version of a system may be accessible to ASDF. In case of an attempt to load another version (that may be discovered by ASDF even accidentally), several conflict resolution strategies may be utilized, the default being to load the system with the latest sysdef file access timestamp. This constraint is conditioned on the ASDF reliance on a central in-memory registry of known systems (similar to the package registry) that is a key-value store keyed by system names only, without the version information. Secondly, the ASDF approach to version conflict resolution is restricted to a single pre-defined strategy for determining the acceptable versions given a certain constraint[18].

To sum up, there is no end-to-end solution to potential system-level name and version conflicts in the Common Lisp environment, but it is desirable in order to support future growth of the Lisp library ecosystem and large-scale projects. The approach should support ASDF systems. Consequently, the proposed solution is based on the standard `rename-package` and low-level ASDF APIs.

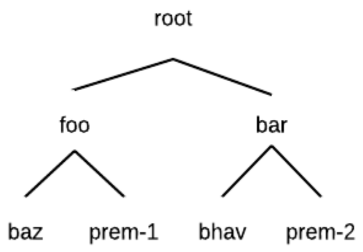
2. Possible conflict scenarios

In order to validate the correctness of a version conflict resolution approach, the following conflict scenarios should be analyzed. More complex possible configurations will be a combination of these primitive cases.

1. "Zero" scenario. No name conflicts. A fallback to `asdf:load-system` is expected.

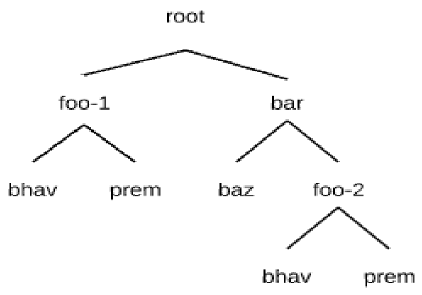


2. "Basic" scenario. There is a single name conflict `between` prem `v.1` `(required by` `foo`) and v.2 (required by

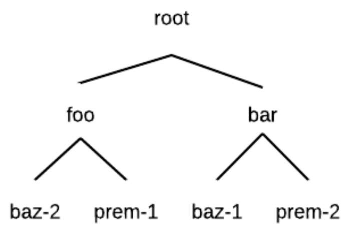


`bar`).

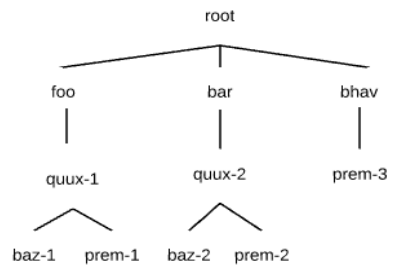
3. "Subroot" scenario. There is a single conflict (in system `foo`), and one of the conflicting packages is a direct dependency of the root system.



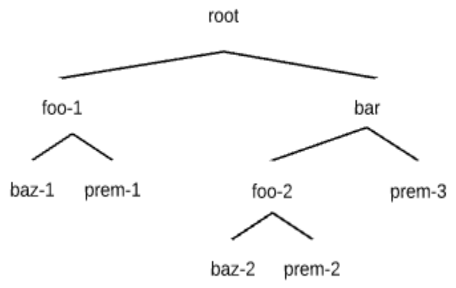
4. "Cross" scenario. There are 2 conflicting systems at the same level in the dependency tree: `prem` and `baz`.



5. "Inter" scenario. There are 3 conflicting systems with one of them (`quux`) being the dependent on the two others: `baz` and `prem`.



6. "Subinter" scenario. There are 3 conflicting systems with one of them (`foo`) being the dependent on the two others (`baz` and `prem`), and one of the conflicting systems (`foo`) a direct dependency of the root system.



3. Implementation

We propose an ASDF-compatible algorithm for conflict-free loading of a particular system's dependencies with on-demand renaming of their packages in case of discovered name/version conflicts happening at the right moment in the program loading sequence. The algorithm comprises of the following steps:

1. Assemble a dependency tree for the system to be loaded based on ASDF systems' dependency information and, using it, discover the dependencies,

which produce name conflicts.
2. In case of no conflicts, fallback to regular ASDF load sequence.
3. In case of conflicts, for each conflicting system determine the topmost possible user of the system in the dependency hierarchy that doesn't have two conflicting dependencies (the one, below the lowest common ancestor of the conflicting systems).
4. Determine the load order of systems using topological sort with an additional constraint that, among the children of the current node of the dependency tree, the ones that require conflict resolution will be loaded last.
5. Load the systems' components (plain load without loading the dependencies) in the selected order caching the fact of visiting a particular system to avoid multiple reloading of the same dependencies that are referenced from various systems in the dependency tree.
6. During the load process, record all package additions and associate them with the system being loaded.
7. After a particular system has been loaded, check whether it was determined as a point of renaming for one or more of its dependencies, and perform the renaming.

In step 4, load-last strategy is necessary for the renaming of the alternative system to happen before the load of the current one: in case of the opposite order, the current system will be loaded but not renamed, as the renaming will happen only after load of the parent node, which will result in a name conflict. This is relevant to the Subroot (4) and Subinter (6) text scenarios.

The algorithm is implemented in the function `load-system-with-renamings`[19] that is summarized in Figure 1. It operates on the instances of a `sys` structure that is used as a simple named tuple: `(defstruct sys name version parent)`.

```
(defun load-system-with-renamings (sys)
  (multiple-value-bind (deps reverse-load-order renamings)
      (traverse-dep-tree sys)
    (when (zerop (hash-table-count renamings))
      (return-from load-system-with-renamings (asdf:load-system sys)))
    (let ((already-loaded (make-hash-table :test 'equal))
          (dep-packages (make-hash-table)))
      ;; load dependencies one by one in topological sort order
      ;; renaming packages when necessary and caching the results
      (dolist (dep (reverse reverse-load-order))
        (let ((conflict (detect-conflict)))
          (when (or conflict
                    (not (gethash (sys-name dep) already-loaded)))
            (renaming-packages
             (if conflict
                 (load-system dep)
                 (load-components (asdf:find-system (sys-name dep)))))
          (unless conflict (setf (gethash name already-loaded) t))))))))
```

Figure 1. Source code for the `load-system-with-renamings` procedure

In the actual function, the `renaming-packages` and `detect-conflict` macros are implemented in-place, but, here, for the sake of clarity, they are extracted. `detect-conflict` is omitted as it is trivial to implement, and `renaming-packages` is listed separately (see Figure 2). That's why it references the seemingly free (but, in fact, the parent's) `dep` and `dep-packages` variables.

The `traverse-dep-tree`[19] function implements the first stage of the algorithm: building a dependency tree, discovering conflicts and arranging the dependencies in proper order for loading. It recurses on the current system's dependencies and keeps a set of the encountered systems and their versions to spot version conflicts via set intersection with a specialized `:key` function that takes into account different system versions.

We also provide an alternative to the ASDF's implementation of system loading facility in `load-system` and `load-components` functions.

```
(defmacro renaming-packages (&body body)
  `(let ((known-packages (list-all-packages)))
     ,@body
     ;; record newly added packages
     (setf (gethash dep dep-packages)
           (set-difference (list-all-packages) known-packages))
     ;; it's safe to rename pending packages now
     (dolist (d (gethash dep renamings)))
       (let ((suff (format nil "~:@(~A-~A-~A~)"
                           (sys-version d) (sys-name dep) (gensym))))
         (dolist (pkg (gethash d dep-packages))
           (rename-package pkg (format nil "~A-~A"
                                       (package-name package) suff)
                           (mapcar (lambda (nickname)
                                     (format nil "~A-~A" nickname suff))
                                   (package-nicknames pkg))))))
```

Figure 2. Source code for the `renaming-packages` macro

3. Working around ASDF

The initial assumption for the development of this algorithm was to build it on top of the public ASDF API as an alternative system loading strategy. However, during its implementation, several obstacles were encountered in ASDF, which forced us to develop alternative procedures to the existing ASDF public counterparts, using the low-level internal ASDF utilities.

The main blocker was an ASDF's core choice to have a central registry of known systems that uses unversioned system names as keys.

In a lot of ways, ASDF is very tightly-coupled and not transparent:

- The source code, in general, is rather extensive and abstraction-heavy, but not well-documented.
- Most of the ASDF `actions`, even the ones that could be implemented in a purely functional manner (for instance, `find-system`), trigger internal state changes.
- The ASDF operations class hierarchy is based on a number of abstract classes, such as `downward-`, `upward-` or `sideway-operations`, which form implicit interdependencies between concrete operations, but this is not documented in a clear manner.
- The ASDF operations are performed not directly, but according to an order specified in a `plan`[20] object. The plan API is also not documented.
- ASDF caching behavior is undocumented.

This makes ASDF a monolithic tool tuned towards implementing a particular strategy of handling systems, which is substantially hard to repurpose in order to support alternative strategies, using the existing machinery for system discovery, orchestration of compilation, and loading of single files. Consequently, there are many unexpected omissions from the ASDF public API. Here are a few that were encountered in the process of this work:

- There is no direct way to load a system from a specific filesystem location: only a system that is previously found using the ASDF algorithm can be loaded.

- There is no direct way to enumerate all potential candidate locations for loading a system: each ASDF system search function should terminate the discovery process as soon as it finds a candidate.
- There is no direct way to find a system with a specified version: the version argument to ASDF operations may only be used as a constraint that the current candidate system should satisfy, not as a guide for selecting the candidate.
- There is no direct way to load just the source files for the system's components without checking and, possibly, reloading its dependencies: calling `load-op` on a source file invokes implicit operation planning machinery that is specified partly in the operations hierarchy and partly in the associated generic functions, and it will cause the call to `prepare-op` on the same file, which triggers `prepare-op` on the whole system, which, in turn, checks the system's dependencies and may invoke their full reload. Overall, a simple call to `(operate 'load-op <component>)` may produce a call stack of 10 or more levels of just ASDF operations.
- It is impossible to read the contents of an ASDF system definition without changing the global state, although this is often needed to determine some property of the candidate ASDF system, like its version or set of dependencies.

As a result, we had to define a number of utility functions to patch the missing parts of ASDF, which, definitely, are not well-integrated with the current vision of how ASDF parts should play together, and which use a number of private ASDF utilities that might be changed or removed in the next versions. Such approach is, obviously, not scalable and, ideally, the implementation should be performed based solely on the ASDF public API. But, to allow that, the API has to be expanded significantly, which will require some major changes to ASDF core: adding deeper support for versions, decoupling some of the functions, and making others less dependent on side effects.

Below is an example of some of the alternatives to ASDF operations that we have developed. The `sysdef-exhaustive-central-registry-search` (see Figure 3) is a version of `asdf::sysdef-central-registry-search` that doesn't stop as soon as the first candidate ASD-file is found. It is used instead of `asdf:search-for-system-definition`, and has a drawback of limiting the search to only the legacy `*central-registry*` locations.

```
(defun sysdef-exhaustive-central-registry-search (system)
  (let ((name (asdf:primary-system-name system))
        rez)
    (dolist (dir asdf:*central-registry*)
      (let ((defaults (eval dir)))
        (when (and defaults (uiop:directory-pathname-p defaults))
          (let ((file (asdf::probe-asd name defaults
                                       :truename asdf:*resolve-symlinks*)))
            (when file (push file rez))))))
    (reverse rez)))
```

Figure 3. Source code for the `sysdef-exhaustive-central-registry-search` function

We, also, had to resort to an interesting way of getting a record for a specific system by its ASD-file (see Figure 4) as a part of an alternative implementation of `find-`

system. Unfortunately, there is no ASDF function that will load an ASD file and return a list of ASDF system objects for the systems defined in it.

```
(asdf:load-asd asd)
(cdr (asdf:system-registered-p system))
```

Figure 4. Source code for the `sysdef-exhaustive-central-registry-search` function

Finally, in Figure 5 you may find a workaround to shortcircuit the ASDF operations' interdependency mechanism and prevent it from performing any other actions except directly loading the components of a current system. In general, it is a sign of excessive code coupling when a simpler operation requires more code than a more complex one, which includes it.

```
(defparameter *loading-with-renamings* nil)
(defmethod asdf:component-depends-on :around ((o asdf:prepare-op)
                                              (s asdf:system))
  (unless *loading-with-renamings*
    (call-next-method)))
(defun load-components (sys)
  (let ((*loading-with-renamings* t))
    (dolist (c (asdf:module-components sys))
      (asdf:operate 'asdf:load-op c)))
  t)
```

Figure 5. Source code for the simplified mechanism of ASDF components loading

To sum up, the current version of ASDF is tightly-coupled and lacks referential transparency at core, while at the middle level it's not well-documented and lacks a comprehensive API that could be used for the development of alternative top-level system management utilities based on a solid foundation of ASDF's system discovery and individual component manipulation machinery.

4. Limitations of the Solution

The proposed solution is, primarily, intended for the use case of loading the whole target system at once without future modifications of its dependencies in-memory, which is necessitated by production build environments. The alternative system load scenarios, that are, mostly, interactive and allow for the programmer to remain in control of the environment, resorting, in case of conflicts, to manual intervention ranging from explicit renaming to changing the source code of the conflicting dependencies and "vendoring" them as part of the project, are not in such desperate need of an automatic solution.

Our approach has a number of limitations that should be listed to avoid unexpected and unexplainable edge cases. The risk of their manifestation in the intended environment is low, but, nevertheless, the users should be aware of the possible shortcomings.

The first limitation is the passive mechanism of capturing package changes after-the-fact, which is not transactional. Parallel invocation of `load-system-with-renamings` has a race condition. The critical section is the process of recording the changes to the global package table in `renaming-packages`. To remove the limitation, this part may be protected by a mutex. This is not done in the presented code to avoid additional complexity. Ideally, the sequential and parallel versions of this procedure should be provided with the sequential one being the default. An alternative solution would be to perform full source code analysis of the system to be loaded in order to determine, which packages will be defined in it. Such complexity is definitely an overkill.

Our approach also relies on the assumption that all the packages from the currently loaded systems where not defined previously. It is a reasonable constraint for the vanilla production environment, which may, however, be violated during an interactive session. Unfortunately, the only measure that may be taken here is a disciplined approach to package loading. At least, the Lisp compiler will issue a warning on package redefinition, which will alert the programmer that the name conflict has occurred. It is possible to expand the loading function to intercept this warning and terminate its operation, if necessary.

Elaborating on this point, it should be also obvious that this procedure will not be able to catch changes to existing packages (that may be regarded as monkey-patching, in this context). It is debatable, whether such changes should be prevented by our system, as their purpose is usually contradictory to the idea of immutable dependencies that our solution upholds.

Next potential issue is associated with implicit transitive dependencies: if a system *foo* depends on *bar* and *quux*, and *bar* also depends on *quux*, in ASDF system definition, it is sufficient to list only *bar* as *foo*'s direct dependency. This implicit dependency may break if *quux*'s packages are renamed during the loading process: according to the algorithm, the renaming will happen directly after loading of *bar*. In such situation, all the references to *quux*'s packages in *foo*'s code will be invalidated, as they will be read when the packages will not be accessible by the old canonical names. However, such situation is relevant only to the newly defined systems that are under full control of the developer, as for the deeper dependencies there should be no version conflicts, as they would not have allowed the system to be built by the normal ASDF procedure, and conflicts introduced when combining multiple dependencies are resolved at the topmost level by the algorithm, thus not effecting the dependency subtrees of the combined systems. Adding an explicit dependency on *quux* in *foo* allows solving the problem in a straightforward way.

Also, our approach doesn't address the possibility of two independent packages having the same name and version, but it, probably, should be handled not at the code-base level, but rather the social one. Additionally, our conflict-finding mechanism may be extended to catch such case.

Finally, the additional minor inconvenience is that the conflicting packages will be available under altered names, which can be discovered from the environment but are not apparent. This may impede interactive redefinition, monkey-patching, hot-patching and other interactive programming practices that might occasionally be of interest to the user. It will surely break the code relying on runtime manipulations using `intern` or `eval`: the references to the renamed packages in the code not yet evaled will be invalidated after the renaming, unlike the references in the code that was read and loaded, which will be associated with new names automatically.

5. Conclusions and Future Work

Our name conflict resolution algorithm and its proof-of-concept implementation provide a feasible solution to potential dependency hell problems for Common Lisp software, specifically targeted to production environments. The paper also explores its corner cases, which require special handling. Although this solution may not be final, it is already usable in the environments that are faced with dependency conflicts.

The proposed approach has several directions of improvement:

- It should be expanded to cover other non-load-based scenarios of system manipulation. In particular, for the compilation use case, the implementation should be almost identical.
- It should be made more compatible

with ASDF (provided ASDF is also changed to be less hostile to such solutions).

Exploration of the possible implementation strategies for this program also helped uncover the deficiencies in the current implementation of ASDF and showed one of the directions for its future development. ASDF is underutilizing its position as a de facto standard toolkit for dependency management in Common Lisp by not providing a comprehensive API for manipulation of both systems and system definition files. In order to allow for this algorithm and other possible non-default build strategies to be implemented on top of ASDF public API, a number of changes are necessary. In general, those should include decoupling of the ASDF code base (specifically from the assumptions of a 1-to-1 mapping of a system record in the ASDF registry to a system currently loaded), comprehensive documentation of the `plan` and `action` APIs, development of utility wrapper functions for common middle-level actions, and a general review of the public API according to the scenarios we'd like it to support.

References

[1] "Dependency Hell" definition - https://en.wikipedia.org/wiki/Dependency_hell
[2] Pitman, K. M., Common Lisp HyperSpec, 1996. Chapter 11. Packages - http://www.lispworks.com/documentation/lw50/CLHS/Body/11_.htm
[3] Nickname collision: bordeaux-threads and binary-types - https://github.com/quicklisp/quicklisp-projects/issues/296
[4] Beane, Z., Quicklisp - http://quicklisp.org
[5] "Continuous Integration" definition - https://en.wikipedia.org/wiki/Continuous_integration
[6] Semantic Versioning - http://semver.org/
[7] Vodonosov, A., Backward compatibility of libraries (case study in Common Lisp) - https://www.european-lisp-symposium.org/editions/2016/lightning-talks-1.pdf
[8] Kernfeld, P., The Nine Circles of Python Dependency Hell - https://tech.knewton.com/blog/2015/09/the-nine-circles-of-python-dependency-hell/
[9] Liang, S., Bracha, G., Dynamic class loading in the Java virtual machine - Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, OOPSLA '98, pages 36–44, 1998 - http://www.humbertocervantes.net/coursdea/DynamicClassLoadingInTheJavaVirtualMachine.pdf
[10] OSGi service platform core specification - http://www.osgi.org/
[11] Project Jigsaw. Project Jigsaw website - http://openjdk.java.net/projects/jigsaw/
[12] Herman, D., Tobin-Hochstadt, S., Modules for JavaScript Simple, Compilable, and Dynamic Libraries on the Web - homes.soic.indiana.edu/samth/js-modules.pdf
[13] Pitman, K. M., Common Lisp HyperSpec, 1996. Function RENAME-PACKAGE - http://www.lispworks.com/documentation/HyperSpec/Body/f_rn_pkg.htm
[14] ASDF System - http://www.cliki.net/ASDF+System
[15] ASDF - https://common-lisp.net/project/asdf/
[16] make. Gnu make website - https://www.gnu.org/software/make/
[17] ant. Apache ant website - http://ant.apache.org/
[18] Rideau, F.-R., Goldman, R.P., ASDF Manual. Chapter 7.4 Functions, Function: version-satisfies - https://common-lisp.net/project/asdf/asdf/Functions.html
[19] Domkin, V., ASDFx - https://github.com/vseloved/asdfx/blob/master/asdfx.lisp
[20] Rideau, F.-R., Goldman, R.P., ASDF Manual. Chapter 7. The Object model of ASDF - https://common-lisp.net/project/asdf/asdf/The-object-model-of-ASDF.html

Session II: Types

A Lisp Way to Type Theory and Formal Proofs

Frederic Peschanski
UPMC Sorbonne Universités – LIP6
4 place Jussieu
Paris, France
frederic.peschanski@lip6.fr

ABSTRACT

In this paper we describe the LaTTe proof assistant, a software that promotes the Lisp notation for the formalization of and reasoning about mathematical contents. LaTTe is based on type theory and implemented as a Clojure library with top-level forms for specifying axioms, definitions, theorems and proofs. As a pure library, LaTTe can exploit the advanced interactive coding experience provided by modern development environments. Moreover, LaTTe enables a form of proving in the large by leveraging the Clojar/Maven ecosystem. It also introduces a very simple and concise domain-specific proof language that is deeply rooted in natural deduction proof theory. And when pure logic is not enough, the system allows to take advantage of the host language: a Lisp way to proof automation.

CCS Concepts

•**Theory of computation** → **Logic; Type theory;**

Keywords

Logic; Type Theory; Proof Assistant; Clojure

1. INTRODUCTION

Proof assistants realize an ancient logician's dream of (re)constructing mathematics based on purely mechanical principles. Most proof assistants (perhaps with the exception of [6]) are complex pieces of software.

One important factor of this complexity is that proof assistants generally try to mimic the common mathematical notation, which is a complex parsing problem that very often get in the way of the user. LaTTe, in comparison, totally surrenders to the simplicity (and sheer beauty) of a Lisp notation. This is also the case of the ACL2 theorem prover [7]. One immediate gain is that the complex issue of parsing vanishes. It also makes the definitions more explicit and less ambiguous than if written with more traditional (and highly informal) computerized variants of mathematical notations.

ELS '16 April AA–BB, 2017, Brussels, Beldium
ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

Another important characteristic is that LaTTe is implemented as a library. The activity of doing mathematics is here considered as a form of (rather than an alternative to) programming. We see this as a clear advantage if compared to proof assistants designed as standalone tools such as Isabelle [11] or Coq [14]. First, this gives a better separation of concerns, only a small library is to be maintained, rather than a complex tool-chain. Moreover, modern Lisp implementations often come with very advanced tools for interactive programming that enables *live-coding mathematics*[1].

Another important factor of complexity in proof assistants, both for the developers and the users, is the language in which proofs are written. The LaTTe proof language is from this respect extremely small and simple. It only introduces two constructs: **assume** and **have** with obvious semantics. The language is also arguably less ad-hoc than most other proof languages in that it is deeply rooted in natural deduction proof theory [12]. As a summary, LaTTe aims at *minimalism.* It has less features than most other proof assistants and each implemented feature strives for simplicity and concision. We argue, however, that this set of features is already plenty enough for the purpose of formalizing mathematics on a computer.

The outline of the paper is as follows. First, in section 2 LaTTe is introduced from a user perspective. The general principles underlying the LaTTe kernel are then discussed in section 3. In particular, we provide a bidirectional type systems that is used in LaTTe to perform both proof/type-checking and type inference. Perhaps the most significant feature of LaTTe is the introduction of a *domain-specific language* (DSL) for proof scripts. This is presented and illustrated with examples in section 4. In section 5 we discuss the way parts of proofs can be automated, the Lisp-way. In section 6 we discuss some (non-)features of LaTTe in comparison with other proof assistants.

2. A LOGICIAN'S DREAM

Quoting [5], the main functionality of a proof assistant is to:

- formalize mathematical contents on a computer
- assist in proving theorems about such contents

Mathematical contents, as can be found in textbooks and proof assistants, is mostly based on definitions, and statement of axioms and theorems. Consider for example the

[1]cf. https://www.youtube.com/watch?v=5YTCY7wm0Nw

```
(proof compose-injective
  :script
  ;; Our hypothesis is that f and g are injective.
  (assume [Hf (injective U V f)
           Hg (injective T U g)]
    ;; We now have to prove that the composition is injective.
    ;; For this we consider two arbitrary elements x and y
    ;; such that f ∘ g(x) = f ∘ g(y)
    (assume [x T
             y T
             Hinj (equal V (f (g x)) (f (g y)))]
      ;; Since f is injective we have: g(x) = g(y).
      (have ⟨a⟩ (equal U (g x) (g y))
            :by (Hf (g x) (g y) Hinj))
      ;; And since g is also injective we obtain: x = y.
      (have ⟨b⟩ (equal T x y) :by (Hg x y ⟨a⟩))
      ;; Since x and y are arbitrary, f ∘ g is thus injective.
      (have ⟨c⟩ (injective T V (λ [x T] (f (g x))))
            :discharge [x y Hinj ⟨b⟩]))
    ;; Which is enough to conclude the proof.    □
    (qed ⟨c⟩)))
```

Table 1: A declarative proof script in LaTTe.

notion of an *injective function* for which every element of the codomain is the image of at most one element of the domain. This can be formalized as follows in LaTTe:

```
(definition injective
  "An injective function."
  [[T ⋆] [U ⋆] [F (⟹ T U)]]
  (forall [x y T]
    (⟹ (equal U (F x) (F y))
         (equal T x y))))
```

Similarly to the ACL2 theorem prover [7], LaTTe uses a Lisp notation (with Clojure extensions and Unicode characters) for the definition of mathematical contents. With a bit of practice and a good editor, this notation should become a Mathematician's best friend.

After defining some mathematical concepts as definitions, the next step is the statement of theorems. An important property of injective functions is stated below.

```
(defthm compose-injective
  "The composition of two injective functions
  is injective."
  [[T ⋆] [U ⋆] [V ⋆] [f (⟹ U V)] [g [⟹ T U]]]
  (⟹ (injective U V f)
      (injective T U g)
      (injective T V (λ [x T] (f (g x))))))
```

The **defthm** form only declares a theorem. In the next step, we must provide a formal proof so that the theorem can be used in further developments.

There are two main families of proof languages. First there is the LCF-style tactic languages[2] as found in e.g. Coq [14] or HOL light [6]. This is an imperative formalism that works on a goal-state to conduct the proofs. The main drawback of such an approach is that the proofs are then very remote from the mathematical practice. Moreover, they cannot be understood without interacting with the tool. Proofs do not have a life on their own in such environments. The declarative proof languages, such as Isar[3] in Isabelle [11], on the contrary, are designed to be closer to standard "pencil and paper" proofs. In LaTTe a proof can be written in two ways: either by supplying a lambda-term (as explained in the next section), or more interestingly using a declarative proof script. The language for proof scripts is probably the most distinctive feature of LaTTe. It is described more thoroughly in section 4, but we illustrate its use by the proof of the theorem compose-injective, given in Table 1. One important characteristic of this proof is that the formal arguments (in Lisp forms) are in close correspondence with the informal proof steps (in comments). An important difference with a proof language such as Isar is of simplicity: only two constructs are introduced: **assume** and **have**. And the underlying proof theory is just standard natural deduction [12].

[2]As a historical remark, the first tactic languages for LCF were developed in Lisp. However this lead to the creation of the ML programming language. Modern variants of ML are often used nowadays to implement theorem provers.

```
(definition D "⟨doc⟩"
  [[x₁ t₁] ... [xₙ tₙ]]
  ⟨term⟩)
```

⟨term⟩	t, u	::=	$\Box$	(kind)
			$\mid \star$	(type)
			$\mid x$	(variable)
			$\mid [t\ u]$	(application)
			$\mid (\lambda\ [x\ t]\ u)$	(abstraction)
			$\mid (\Pi\ [x\ t]\ u)$	(product)
			$\mid (D\ t_1\ t_2\ \ldots\ t_n)$	(instantiation)

Table 2: The syntax of the LaTTe calculus

3. LAMBDA THE ULTIMATE

The kernel of the LaTTe library is a computerized version of a simple, although very expressive lambda-calculus. It is a variant of λD as described in the book [9], which corresponds to the *calculus of constructions* [4] (without the prop/type distinction) enriched with definitional features.

3.1 Syntax

The basic syntax of the LaTTe calculus is given in Table 2. There are various syntactic sugars that will for some of them be introduced later on, and there is also a **defnotation** mechanism to introduce new notations when needed (e.g. for the existential quantifier, which is a derived principle in type theory). Perhaps the most notable feature of the calculus is that it is compatible with the *extended data notation*[4], i.e. it is a subset of the Clojure language[5]. As a dependently-typed lambda-calculus, there is no syntactic distinction between terms and types. A type is simply a term whose type is $\star$, called the sort of types. The type of $\star$ itself is a sort called a *kind* and denoted by $\Box$[6]. The kernel of any lambda-calculus is formed by the *triptych*: variable occurrences $x, y, \ldots$, function applications $[t\ u]$ and abstractions $(\lambda\ [x\ t]\ u)$. The latter expresses a function with an argument x of type t (hence a term of type $\star$) and

[3]Isar stands for "Intelligible semi-automated reasoning".
[4]cf. https://github.com/edn-format/edn
[5]This means that the lambda-terms in LaTTe can be quoted in Clojure, and thus used to feed macros.
[6]For the sake of logical consistency, the kind $\Box$ has no type, which makes LaTTe an *impredicative* type theory.

with term u as body. The type of a lambda-abstraction is called a *product* and is denoted by $(\Pi\ [x\ t]\ u)$. The intended meaning is that of universal quantification: "for all x of type t, it is the case that u". As an example, the term $(\lambda\ [A\ \star]\ (\lambda\ [x\ A]\ x))$ corresponds to a type-generic identity function. Its type is $(\Pi\ [A\ \star]\ (\Pi\ [x\ A]\ A))$. An important syntactic sugar that we will largely exploit is that if in $(\Pi\ [x\ t]\ u)$ the variable x has no free occurrence in the body u, then we can rewrite the product as $(\Longrightarrow t\ u)$. This can be interpreted ass an arrow type of functions that from inputs of type t yield values of type u. Alternatively, and in fact *equivalently* this is the logical proposition that "t implies u". When such a logical point of view is adopted, the universal quantified symbol $\forall$ can be used instead of the more esoteric Π. For the type-generic identity function, this finally gives $(\forall\ [A\ \star]\ (\Longrightarrow A\ A))$, i.e. for any type (proposition) A, it is the case that A implies A. This gives a first glimpse of the tight connection between computation and logic in such a typed lambda-calculus, namely the *Curry-Howard correspondence* [13].

Because LaTTe is aimed at *practice* and not only theory, the basic lambda-calculus must be enriched by *definitional principles.* First, parameterized definitions can be introduced using the **definition** form. Then, such definitions can be *instantiated* to produce *unfolded* terms. In LaTTe, parenthesized (and desugared) expressions that do not begin with λ or Π are considered as instantiations of definitions.

For example, we can introduce a definition of the type-generic identity function as follows:

```
(definition identity
  "the identity function"
  [[A ★][x A]]
  x)
```

Then, an expression such as (identity nat 42) would be instantiated to 42 (through δ-reduction, as discussed below). In theory, explicit definitions and instantiations are not required since they can be simulated by lambda-abstractions and applications, but in practice it is very important to give names to mathematical concepts (as it is important to give names to computations using function definitions).

3.2 Semantics

The semantics of lambda-calculus generally rely on a rewriting rule named β-reduction and its application under a context:

- (conversion) $[(\lambda[x\ t]u)\ v] \xrightarrow{\beta} u\{v/x\}$
- (context) if $t \xrightarrow{\beta} t'$ then $C[t] \xrightarrow{\beta} C[t']$, for any *single-hole context C.*

The notation $u\{v/x\}$ means that in the term u all the free occurrences of the variable x must be substituted by the term v. For example, we have $[a\ [(\lambda\ [x]\ [bx])\ [c\ d]]] \xrightarrow{\beta} [a\ [b\ [c\ d]]]$. This is because if we let $t = [(\lambda\ [x]\ [bx])\ [c\ d]]$ and $t' = [b\ [c\ d]]$ then $t \xrightarrow{\beta} t'$ by the conversion rule. And if we define the context $C[X] = [a\ X]$ with hole X, then $C[t] \xrightarrow{\beta} C[t']$ by the context rule. While β-reduction seems trivial, it is in fact *not* the case, at least at the implementation level. One difficulty is that lambda-terms must be considered up to α-equivalence. For example, $(\lambda\ [x\ t]\ u) \equiv_\alpha (\lambda\ [y\ t]\ u)$ because we do not want to distinguish the lambda-terms based on their bound variables. Reasoning about such issues is in fact not trivial, cf. e.g. [3]. A lambda-calculus aimed at logical reasoning has to fulfill two important requirements:

- *strong normalization*: no lambda-term t yields an infinite sequence of β-reductions
- *confluence*: if $t \xrightarrow{\beta^*} t_1$ and $t \xrightarrow{\beta^*} t_2$ then there exist a term u such that $t_1 \xrightarrow{\beta^*} u$ and $t_2 \xrightarrow{\beta^*} u$ (up-to α-equivalence)[7]

As a consequence, each lambda-term t possesses a unique *normal form* $\widetilde{t}$ (up-to α-equivalence). Thus, two terms t_1 and t_2 are β-equivalent, denoted by $t_1 =_\beta t_2$, iff $\widetilde{t_1} \equiv_\alpha \widetilde{t_2}$. In a proof assistant based on type theory, the α-equivalence and β-reduction relations are not enough, for example to implement the definitional features. The LaTTe kernel uses a δ-reduction relation, similar to that of [9], to allow the instantiation of definitions.

If we consider a definition D of the form given in Table 2 then the rules are as follows:

- (instantiation) $(D\ t_1\ \ldots\ t_n) \xrightarrow{\delta} u\{t_1/x_1, \ldots, t_n/x_n\}$
- (context) if $t \xrightarrow{\delta} t'$ then $C[t] \xrightarrow{\delta} C[t']$, for any *single-hole context C.*

At the theoretical level, the overlap between β and δ-reductions is relatively unsettling but in practice, β-reduction works by copying terms while δ-reduction uses names and references, and is thus much more economical. Moreover, in mathematics giving names to definitions and theorems is of primary importance so the issue must be addressed with rigour. LaTTe here still roughly follows [9].

LaTTe introduces a further σ-reduction relation for **defspecial**'s. This is discussed in section 5.

3.3 Type inference

There are three interesting algorithmic problems related to the typing of lambda-terms. First, there is the *type checking* problem. Given a term t and a term u, check that u is the type of t. In LaTTe this problem occurs given:

- a *definitional environment* Γ containing an unorded map of definition names to definitions. For example, if D is a definition then $\Gamma[D(t_1, \ldots, t_n)]$ gives the lambda-term corresponding to the definition contents.
- a *context* Δ containing an ordered list of associations from (bound) variable names to types. If x is a variable in the context, then $\Delta[x]$ is its type.

A term t that has type u in environment Γ and context Δ is denoted by: $\Gamma; \Delta \vdash t :: u$. It is not very difficult to show that type checking in LaTTe is decidable. This would be a relatively straightforward elaboration for λD in [9]. Suppose that we know only the type part. Thus, we have to find a term to replace the question mark in $\Gamma; \Delta \vdash ? :: u$. This *term synthesis* problem is not decidable in LaTTe and the intuition is that we would then have an algorithmic way to automatically find a proof for an arbitrary theorem. Term synthesis can still be partially solved in some occasions, and

[7]The notation $t \xrightarrow{\beta^*} t'$ means zero or more β-reductions from t to t', it is the reflexive and transitive closure of the relation of β-reduction under context.

$$\frac{}{\Gamma;\Delta \vdash \star :> \square}\ (type) \qquad \frac{\Gamma;\Delta \vdash \widetilde{A} :: s \quad s \in \{\star,\square\}}{\Gamma;\Delta, x :: A \vdash x :> A}\ (var) \qquad \frac{\Gamma;\Delta \vdash A :> s_1 \quad \Gamma;\Delta, x :: A \vdash B :> s_2 \quad \widetilde{s_1}, \widetilde{s_2} \in \{\star,\square\}}{\Gamma;\Delta \vdash (\Pi\ [x\ A]\ B) :> s_2}\ (prod)$$

$$\frac{\Gamma;\Delta, x :: A \vdash t :> B \quad \Gamma;\Delta \vdash (\Pi\ [x\ A]\ B) :> s \quad \widetilde{s} \in \{\star,\square\}}{\Gamma;\Delta \vdash (\lambda\ [x\ A]\ t) :> (\Pi\ [x\ A]\ B)}\ (abs) \qquad \frac{\Gamma;\Delta \vdash t :> (\Pi\ [x\ A]\ B) \quad \Gamma;\Delta \vdash u :: A}{\Gamma;\Delta \vdash [t\ u] :> B\{u/x\}}\ (app)$$

$$\frac{\begin{array}{c}\Gamma[D] :: [x_1\ t_1]\ [x_2\ t_2]\ \cdots\ [x_n\ t_n] \to t \\ \Gamma;\Delta \vdash e_1 :: t_1 \quad \Gamma;\Delta, x_1 :: t_1 \vdash e_2 :: t_2 \quad \cdots \quad \Gamma;\Delta, x_1 :: t_1, x2 :: t_2, \ldots, x_{m-1} :: t_{m-1} \vdash e_m :: t_m\end{array}}{\Gamma;\Delta \vdash (D\ u_1\ u_2\ \ldots\ u_m) :> (\Pi\ [x_{m+1}\ t_{m+1}]\ \ldots\ (\Pi\ [x_n\ t_n]\ t\{u_1/x_1, u_2/x_2, \ldots, u_m/x_m\})\cdots)}\ (ref)$$

Table 3: Type inference rules

```
(defn type-of-var [def-env ctx x]
  (if-let [ty (ctx-fetch ctx x)]
    (let [[status sort] (let [ty' (norm/normalize def-env ctx ty)]
                          (if (stx/kind? ty')
                            [:ok ty']
                            (type-of-term def-env ctx ty)))]
      (if (= status :ko)
        [:ko {:msg "Cannot calculate type of variable." :term x :from sort}]
        (if (stx/sort? sort)
          [:ok ty]
          [:ko {:msg "Not a correct type (super-type is not a sort)" :term x :type ty :sort sort}])))
    [:ko {:msg "No such variable in type context" :term x}]))


(example
 (type-of-var {} '[[bool ⋆] [x bool]] 'x) ⇒ '[:ok bool])

(example
 (type-of-var {} '[[x bool]] 'x)
 ⇒ '[:ko {:msg "Cannot calculate type of variable.",
          :term x :from {:msg "No such variable in type context", :term bool}}])
```

Table 4: The Clojure source for the (var) inference rule.

it is an interesting approach for proof automation. On the other hand, one may want to replace the question mark in the following problem: $\Gamma;\Delta \vdash t ::?$. Now we are looking for the type of a given term, which is called the *type inference* problem. LaTTe has been designed so that this problem is decidable and can be solved efficiently. If the inferred type of term t is A, then we write: $\Gamma;\Delta \vdash t :> A$.

Table 3 summarizes the type inference rules used in LaTTe. Each rule corresponds to a Clojure function, we will take the (var) rule as an example. Its implementation is a function named type-of-var, whose complete definition is given in Table 4. For a variable x present in the context Δ (parameter ctx in the source code) with type A, the (var) rule first normalizes A (using the norm/normalize function) and compares its type with a sort $\star$ or $\square$. This checks that A is effectively a type. In the conclusion of the rule, the notation $x :> A$ is to be interpreted as "the inferred type for x is A". In the source code, this corresponds to the value of the variable ty. Note that only the denormalized version of the type is inferred, which is an important memory saving measure. The other rules are connected similarly to a rather straightforward Clojure function. One subtlety in the (app) rule for application is that the operand term u must be checked against an inferred type A. It is possible to implement a separate type-checker. For one thing, type-checking can be done more efficiently than type inference. Moreover, it is a simpler algorithm and is useful for separate proof checking. However, there is a large overlap between the two algorithms and it is not really worth the duplication. Indeed, a type-checking algorithm can be obtained "for free" using the following fact:

$$\Gamma;\Delta \vdash t :: u \text{ iff } \Gamma;\Delta \vdash t :> v \text{ and } v =_\beta u$$

The complete implementation of the type inference algorithm is less than 400 lines of commented code, and is available on the github repository[8].

4. A DSL FOR PROOF SCRIPTS

The language of mathematical proofs is very literary and remote from the formal requirements of a computer system. As discussed in [5], a proof should be not just a *certificate of truthiness* but also, and perhaps most importantly, an *explanation* about the theorem under study. Proof assistants that use a tactic language (such as Coq or HOL) do not produce readable proofs. To understand the proof, one generally has to replay it step-by-step on a computer. A language such Isabelle/Isar allows for declarative proof scripts, that with some practice can be read and understood like classical proofs on papers. However Isar is arguably a complex

[8]cf.https://github.com/latte-central/LaTTe/blob/master/src/latte/kernel/typing.clj

$\langle\text{proof}\rangle\ P$	::=	**(proof** *thm* :term t)	(direct proof)
		| **(proof** *thm* :script ρ)	(proof script)
$\langle\text{script}\rangle\ \rho$	::=	$\sigma\ \rho$	(proof step σ)
		| **(assume** [$H\ t$] ρ)	(global assumption)
		| **(qed** t)	(proof term t)
$\langle\text{step}\rangle\ \sigma$	::=		
		(assume [$H\ t$] ρ)	(local assumption)
		| **(have** $\langle a\rangle\ A$:by t)	(proof of A with term t)
		| **(have** $\langle a\rangle\ A$:discharge [$x_1 \cdots x_n\ t$])	(discharge assumptions)

Table 5: The proof language of LaTTe

an rather *ad hoc* language, with only informal semantics. The domain specific language (DSL) for declarative proof scripts in LaTTe is in comparison very simple. It is an implementation, in the context of type theory and LaTTe, of *fitch-style* natural deduction proofs [12], and is thus deeply rooted in logic and proof theory. The syntax of the proof language is very concise (cf. Table 5) and with simple and formal semantics (cf. Table 6).

As an illustration, we consider the following proposition:

$$\phi \equiv ((P \implies Q) \land (\neg R \implies \neg Q)) \implies (P \implies R)$$

A natural deduction proof of ϕ, said in Fitch-style and adapted from [12], is given in Table 7. We will now see how to translate such a proof to the LaTTe proof language. Initially, the environment Γ contains at least the theorem to prove, but without a type, i.e. something of the form: $thm(P :: \star, Q :: \star, R :: \star) \lhd ? : \phi$. The context Δ contains the three bindings: $P :: \star, Q :: \star, R :: \star$

The beginning of our LaTTe proof is as follows:

```
(proof thm :script
  (assume(H (and (⟹ P Q)
                 (⟹ (not P) (not Q))))
    ...
```

According to rule (*glob*) of Table 6, the hypothesis H and its type (the stated proposition) is introduced in the context Δ. The term u generated by the body of the **assume** block will be propagated. The first step is as follows:

```
... continuing
(have ⟨a⟩ (⟹ P Q) :by (p/and-elim-left% H))
...
```

The justification (and-elim-left% H) is a **defspecial** that will be discussed more precisely in the next section. But it is simply a function that takes the proof of a conjunction and generates the proof of the left operand of the conjunction. The result of a **have** step, handled by the rule named (*by*), is to add a new definition to the environment Γ of the form:

$$\langle a\rangle \lhd t :: (\implies\ P\ Q)$$

with t the left-elimination of assumption H. Of course, this only works if the type-checker agrees: each **have** step is checked for correctness.

Ultimately, each accepted step is recorded as a local theorem recorded in Γ.

The hypothesis x of type P is assumed and in the next steps we have:

```
... continuing
(assume [x P]
  (have ⟨b⟩ Q :by (⟨a⟩ x))
  (have ⟨c⟩ (⟹ (not R) (not Q))
        :by (p/and-elim-right% H))
    ...
```

Now, still through rule (*by*) the environment Γ is extended with definitions $\langle b\rangle$, obtained by applying x on $\langle a\rangle$, and $\langle c\rangle$, obtained by right-elimination of H. For the moment we remain very close to the original Fitch-style proof. In the next step, the objective is to perform a *reductio ab absurdum*. We first state $\neg R$ and derive a contradiction from it. This gives:

```
... continuing
(assume [Hr (not R)]
  (have ⟨d⟩ (not Q) :by (⟨c⟩ Hr))
  (have ⟨e⟩ absurd :by (⟨d⟩ ⟨b⟩))
  (have ⟨f1⟩ (not (not R))
        :discharge [Hr ⟨e⟩]))
...
```

In the type theory of LaTTe, the proposition (not P) corresponds to ($\Longrightarrow$ P absurd) with absurd an type without inhabitant, classically: ($\forall$ [A $\star$] A). Hence after having obtained (not Q) through step $\langle c\rangle$ we obtain step $\langle e\rangle$. The :discharge step $\langle f1\rangle$ corresponds to the generation of a lambda term of the form: (λ [Hr (not R)] $\langle$e$\rangle$) hence a term of type (==> (not R) absurd), thus (not (not R)). Since we discharged the hypothesis Hr we can close the corresponding **assume** scope[9].

In the Fitch-style proof at step $\langle f\rangle$ we deduce R by contradiction. This reasoning step can only be performed in classical logic. In fact the proposition ($\Longrightarrow$ (not (not R)) R) is equivalent to the axiom of the excluded middle, and is thus classical in essence. In LaTTe, we must rely on the classic namespace to perform the corresponding step, as follows.

```
... continuing
(have ⟨f2⟩ R
      :by ((classic/not-not-impl R) ⟨f1⟩))
...
```

At this point we are able to assert the conclusion of the rule, and finish the proof.

```
  ... continuing
  (have <g> (==> P R) :discharge [x ⟨f2⟩]))
(qed ⟨g⟩))))
```

The term synthetized at step $\langle g\rangle$ is propagated to the (*script*) rule using the rule (*qed*). Finally, the type-checking problem $\langle g\rangle :: \phi$ is decided, which leads to the acceptation or refutal of the proof. Hence, the natural deduction proof script is only to elaborate, step-by-step, a proof candidate. Ultimately, the type-checker will decide if the proof is correct or not.

5. PROOF AUTOMATION, THE LISP WAY

Proof automation is an important part of the proof assistant experience. In a semi-automated theorem prover such as ACL2 [7], the objective is to minimize the need for interaction with the tool. This requires strong restrictions on the

[9]In the current version of LaTTe the :discharge steps are performed automatically when closing the assume blocks. Thus, they do not have to (and in fact cannot) be written by the user.

$$\frac{\Gamma; x_1 :: t_1, \ldots, x_n :: t_n \vdash t :: P}{\begin{array}{c}\Gamma, thm(x_1 :: t_1, \ldots, x_n :: t_n)\lhd ? :: P \\ \vdash (\textsf{proof}\ thm\ \textsf{:term}\ t) \dashv \Gamma, thm \lhd t :: P\end{array}}\ (term) \qquad \frac{\begin{array}{c}\Gamma; x_1 :: t_1, \ldots, x_n :: t_n \vdash \rho \Rrightarrow t \dashv \Gamma' \\ \Gamma; x_1 :: t_1, \ldots, x_n :: t_n \vdash t :: P\end{array}}{\begin{array}{c}\Gamma, thm(x_1 :: t_1, \ldots, x_n :: t_n)\lhd ? :: P \\ \vdash (\textsf{proof}\ thm\ \textsf{:script}\ \rho) \dashv \Gamma, thm \lhd t :: P\end{array}}\ (script)$$

$$\frac{\Gamma; \Delta \vdash \sigma \Rrightarrow t \dashv \Gamma' \quad \Gamma'; \Delta \vdash \rho \Rrightarrow u \dashv \Gamma''}{\Gamma; \Delta \vdash \sigma\ \rho \Rrightarrow u \dashv \Gamma''}\ (step) \quad \frac{\Gamma; \Delta, H :: t \vdash \rho \Rrightarrow u}{\Gamma; \Delta \vdash (\textsf{assume}\ [H\ t]\ \rho) \Rrightarrow u}\ (glob) \quad \frac{}{\Gamma; \Delta \vdash (\textsf{qed}\ t) \Rrightarrow t}\ (qed)$$

$$\frac{\Gamma; \Delta, H :: t \vdash \rho \Rrightarrow u \dashv \Gamma'}{\Gamma; \Delta \vdash (\textsf{assume}\ [H\ t]\ \rho) \dashv \Gamma'}\ (loc) \qquad \frac{\Gamma; \Delta \vdash t :: A}{\Gamma; \Delta \vdash (\textsf{have}\ \langle a \rangle\ A\ \textsf{:by}\ t) \dashv \Gamma, \langle a \rangle \lhd t :: A}\ (by)$$

$$\frac{\Gamma; \Delta \vdash t' :: A \quad t' \equiv (\lambda\ [x_1 :: t_1]\ \cdots\ (\lambda\ [x_n :: t_n]\ t) \cdots)}{\Gamma; \Delta, x_1 :: t_1, \ldots, x_n :: t_n \vdash (\textsf{have}\ \langle a \rangle\ A\ \textsf{:discharge}\ [x_1\ \cdots\ x_n\ t]) \dashv \Gamma, \langle a \rangle \lhd t' :: A}\ (hyp)$$

Table 6: The semantics of LaTTe proof scripts

```
│ H. (P ⟹ Q) ∧ (¬R ⟹ ¬Q)
├────────────────────────
│ ⟨a⟩ P ⟹ Q                    ∧ Elim:H
│ │ x. P
│ ├──────
│ │ ⟨b⟩ Q                       ⟹ Elim:⟨a⟩, x
│ │ ⟨c⟩ ¬R ⟹ ¬Q                ∧ Elim:H
│ │ │ Hr. ¬R
│ │ ├──────
│ │ │ ⟨d⟩ ¬Q                    ⟹ Elim:⟨c⟩, Hr
│ │ │ ⟨e⟩ Q                     Repeat:⟨b⟩
│ │ ⟨f⟩ R                       Absurd:Hr
│ ⟨g⟩ P ⟹ R                    ⟹ Intro:x, ⟨f⟩
```

Table 7: A Fitch-style proof (from [12])

logic manipulated by the tool. Tactic-based proof assistants often provide complex decision procedures implemented as dedicated tactics. In LaTTe, proof automation relies on a less imperative notion of *special* that we discuss in this section.

In LaTTe proof scripts, each **have** step of the form (**have** ⟨a⟩ A :by t) involves the following chain of events:

1. stating a proposition as a type A
2. finding a candidate term t
3. checking that the term t effectively has type A

In the normal usage, the user needs to perform steps 1 and 2, and LaTTe automatically performs step 3. In some situations, the user can benefit from the LaTTe implementation to either state the proposition, or even receive help in finding a candidate term.

Given the term t, the type inference algorithm of Table 3 may be used to obtain proposition A automatically. The syntax of such a proof state is: (**have** ⟨a⟩ _ :by t). In many situations, it is not recommended because it may make a proof unintelligible, however sometimes this is useful to avoid redundancies in the proofs.

The most interesting situation is the converse: when the proposition A is known but it remains to find the candidate term t. The term synthesis problem is not decidable in general, but it is of course possible to help in the finding process.

The LaTTe proof assistant follows the Lisp tradition of allowing users to write extensions of the system in the host language itself (namely Clojure). This is the purpose of the **defspecial** form that we introduce on a simple example.

The left-elimination rule for conjunction is declared as follows in LaTTe:

```
(defthm and-elim-left "..."
  [[A :type] [B :type]]
  (==> (and A B)
       A))
```

When using this theorem in a **have** step, one needs to provide the types A and B as well as a proof of (and A B), i.e. something of the form:

```
(have ⟨a⟩ A :by ((and-elim-left A B) p))
```

with p a term of type (and A B). But if p has a conjunction type, then it seems redundant having to state propositions A and B explicitely. This is where we introduce a special rule and-elim-left% as follows.

```
(defspecial and-elim-left% "..."
  [def-env ctx and-term]
  (let [[status ty]
            (type-of-term def-env ctx and-term)]
    (if (= status :ko)
      (throw (ex-info "Cannot type term." ...))
      (let [[status A B]
             (decompose-and-type def-env ctx ty)]
        (if (= status :ko)
          (throw (ex-info "Not an 'and'-type." ...))
          [(list #'and-elim-left A B) and-term])))))
```

A **defspecial** is similar to a regular Clojure function, except that it may only be called during a **have** proof step. It receives as arguments the current environment and context as well as the argument calls. In the case of the left elimination rule, only one supplementary argument is passed to the special: the term whose type must be a conjunction (parameter and-term in the code above). In the first step, the type of the term is calculated using the inference algorithm. If a type has been successfully derived, an auxiliary function named decompose-and-type analyzes the type to check if it is a conjunction type. If it is the case then the two conjuncts A and B are returned. Ultimately, a **defspecial** form must either throw an exception or return a synthesized term. In our case, the non-special term ((and-elim-left A B) and-term) is returned.

In a proof, the **have** step for left-elimination is now simpler:

```
(have ⟨a⟩ A :by (and-elim-left% p))
```

This is only a small example, there are many other use of specials in the LaTTe library.

The **defspecial** form is quite expressive since the computational content of a special can exploit the full power of the host language. We might wonder if allowing such computations within proofs is safe. Thanks to the ultimate type-checking step, there is no risk of introducing any inconsistency using specials. In fact the only "real" danger is to introduce an infinite loop (or a too costly computation) in the proof process. But then the proof cannot be finished so we are still on the safe side of things. For the moment, there is no complex decision procedure implemented using *specials* so it is difficult to compare the approach with the common tactic-based one.

6. DISCUSSIONS

In this section we discuss a few common features of proof assistants, and the way they are supported (or not) in LaTTe.

Implicit arguments

Proof assistants such as Coq [14] and Agda [2] allow to make some arguments of definitions *implicit*. The idea is that such arguments may be filled automatically using a unification algorithm. The advantage is that the notations can thus be simplified, which removes some burden on the user. A first drawback is that because higher-order unification is not decidable, it is sometimes required to fill the arguments manually. Moreover the implicit arguments may hide some important information: it is not because an argument can be synthesized automatically that it is not useful in its explicit form. In LaTTe all arguments must be explicit. However, it is possible to refine definitions by partial applications. For example, the general equality predicate in LaTTe is of the form (equal T x y), which states that x and y of the same type T are equal. In the arithmetic library[10], the equality on integer is defined as follows:

```
(definition =
  "The equality on integers."
  [[n int] [m int]]
  (equal int n m))
```

Hence, we can write (= n m) instead of (equal int n m) when comparing integers. In the next (upcoming) version of LaTTe a more general form of implicit arguments will be supported. Instead of relying on a somewhat unpredictable unification algorithm, we will simply allow the user to specify the synthesis of arguments as a Clojure function. The following is a definition of an *implicit*:

```
(defimplicit equal [[x tx] [y _]]
  (list 'equal tx x y))
```

[10]cf. https://github.com/latte-central/latte-integers

When an expression (equal e1 e2) is encountered (at typing time), the *implicit* is called with x (resp. y) bound to e1 (resp. e2) and tx (resp. ty) to its type. The body of the *implicit* form simply produces the correct term of arity 3. At the technical level, this feature is very close to the implementation of *specials*.

Holes in proofs

The proof assistant Agda [2] allows to put holes (i.e. unification variables) in proof terms, which gives an alternative way to perform proofs in a step-wise way. Such a partial proof can be type-checked (assuming the holes have the correct type), and suggestions (or even completions) for holes can be provided by a synthesis algorithm. For the moment, LaTTe does not integrate such a feature but it is planned for the next version of the proof engine. The idea is to reject a proof but return possible mappings for the holes.

Inductives and Σ-terms

In Coq [14] and Agda [2] the term languages is much more complex than that of LaTTe. In particular *inductives* and Σ-terms are proposed. It is then much easier to introduce inductive types and recursive functions in the assistants. Moreover, this gives a way of performing *proofs by (recursive) computation*. The main disadvantage is that the uniqueness of typing is lost, and of course the underlying implementation becomes much more complex. Moreover, *universe levels* must be introduced because inductives do not seem to deal well with *impredicativity*. In LaTTe we adopt the approach of Isabelle [10] and HOL-light [6] (among others) of introducing inductive sets and recursion as mathematical libraries. Proof automation is then needed to recover a form of proof by computation. In LaTTe we just started to implement inductive sets and recursion theorems [11]. The next step will be to automate recursive computations using *specials*. The Σ-types are much easier to implement than inductives. They offer a way to encode *subsets*, i.e. a term $\Sigma x : T.P(x)$ is the subset of the elements of type T that satisfies the predicate P. This is not needed in type theory and LaTTe since such a subset can simply be coded by a predicate $(\lambda\ [x\ T]\ (Px))$. We haven't found any strong argument in favor of Σ-types in the literature.

User interfaces

Most proof assistants are provided with dedicated user interfaces, in general based on an extensible editor such as Emacs. An example of such an environment is *Proof general* [1] that is working with Coq and was also working with Isabelle until version 2014. Proof general was also working with other proof assistants, but support has been dropped. The major weak points are maintainability and evolvability. There is in general much more motivation to work on the kernel of a proof assistant rather than its user interface. The user interfaces for proof assistants can be seen as *live-coding environments*. In most Lisps, and of course Clojure, development environments are designed for a thorough live-coding experience. This observation is one of the two reasons why LaTTe was designed as a library and not as a standalone tool. Our experience is that the Clojure coding environments (Emacs/cider, Cursive, Gorilla Repl, etc.) are

[11]cf. https://github.com/latte-central/fixed-points

perfectly suited for proof assistance. In a way LaTTe has a very powerful interactive environment, maintained by rather large communities, and all this for free!

Proving in the large

The second reason of the design of LaTTe as library is to leverage the *Clojure ecosystem* for proving in the large. Mathematical content can be developed as Clojure libraries, using *namespaces* for modularization. The mathematical libraries can be very easily deployed using Clojars (and Maven) and then used as dependencies in further development. Since all proof forms are macros, the proof checking is performed at compile-time and thus the deployed libraries are already checked for correctness. In this way, although LaTTe is not (yet) a very popular proof assistant, its features for proving in the large are already much more advanced if compared to all the proof assistants we know of. This is of course obtained *for free* thanks to the way Clojure and its ecosystem is (very thoroughly) designed.

7. CONCLUSION AND FUTURE WORK

In this paper we described the LaTTe proof assistant in much details. The ways a dependent type theory might be implemented in practice is not very often described in the literature, a notable exception being [8]. In this paper, we provide all the key concepts to understand the actual implementation of the LaTTe kernel. LaTTe is a minimalist implementation of a proof assistant based on type theory. It is not, however, just a toy implementation for demonstration purpose. It has been used, and is used, to formalize various theories such as a part of typed set-theory, the foundations of integer arithmetic and some developments about fixed points and inductive sets. These are available on the project page[12].

Beyond the formalization of important parts of mathematics (especially the real numbers), we have a few plans concerning the implementation itself. The terms manipulated in type theory can become quite large in the case of long proofs. This is a rather sparsely studied aspect of type theory, as most of the implementation aspects. We already experimented a more efficient term representation, but the performance gains were limited and the price to pay – giving up the internal Lisp representation – much too high. We also introduced a memoization scheme for the type inference algorithm (which is a known bottleneck) but the ratio memory increase vs. CPU gains is not very good. The best way to circumvent this performance issues is to split the proof in separately-compiled subproofs. An automatic proof splitting algorithm was recently experimented with much higher performance gains. Note, however, that these performance issues only occur at compile-time because this is when the proofs are checked for correctness. This has no impact when using the mathematical libraries because they are deployed in compiled form. Most of the other planned features revolve around higher-order pattern matching and (inherently partial) unification. One functionality that would then be possible is the notion of proof refinement using holes. This would also enable the development of search algorithms for theorems.

[12]https://github.com/latte-central

8. REFERENCES

[1] D. Aspinall. Proof general: A generic tool for proof development. In *TACAS 2000*, volume 1785 of *LNCS*, pages 38–42. Springer, 2000.

[2] A. Bove, P. Dybjer, and U. Norell. A brief overview of agda - A functional language with dependent types. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

[3] A. Charguéraud. The locally nameless representation. *J. Autom. Reasoning*, 49(3):363–408, 2012.

[4] T. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988.

[5] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[6] J. Harrison. HOL Light: An overview. In *TPHOLs 2009*, volume 5674 of *LNCS*, pages 60–66, Munich, Germany, 2009. Springer-Verlag.

[7] M. Kaufmann, J. S. Moore, and P. Manolios. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[8] A. Löh, C. McBride, and W. Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inform.*, 102(2):177–207, 2010.

[9] R. Nederpeldt and H. Geuvers. *Type Theory and Formal Proof: An Introduction.* Cambridge University Press, 2014.

[10] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[11] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer-Verlag, 2002.

[12] F. J. Pelletier and A. P. Hazen. A history of natural deduction. In *Logic: A History of its Central Concepts*, volume 11 of *Handbook of the History of Logic*, pages 341–414. Elsevier, 2012.

[13] M. H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism,*. Elsevier, 2006.

[14] The Coq development team. The coq proof assistant. https://coq.inria.fr/.

Programmatic Manipulation of Common Lisp Type Specifiers

Jim E. Newton
jnewton@lrde.epita.fr

Didier Verna
didier@lrde.epita.fr

Maximilien Colange
maximilien.colange@lrde.epita.fr

EPITA/LRDE
14-16 rue Voltaire
F-94270 Le Kremlin-Bicêtre
France

ABSTRACT

In this article we contrast the use of the s-expression with the BDD (Binary Decision Diagram) as a data structure for programmatically manipulating Common Lisp type specifiers. The s-expression is the *de facto* standard surface syntax and also programmatic representation of the type specifier, but the BDD data structure offers advantages: most notably, type equivalence checks using s-expressions can be computationally intensive, whereas the type equivalence check using BDDs is a check for object identity. As an implementation and performance experiment, we define the notion of maximal disjoint type decomposition, and discuss implementations of algorithms to compute it: a brute force iteration, and as a tree reduction. The experimental implementations represent type specifiers by both aforementioned data structures, and we compare the performance observed in each approach.

CCS Concepts

•**Theory of computation → Data structures design and analysis;** *Type theory;* •**Computing methodologies → Representation of Boolean functions;** •**Mathematics of computing →** *Graph algorithms;*

1. INTRODUCTION

Common Lisp programs which manipulate type specifiers have traditionally used s-expressions as the programmatic representations of types, as described in the Common Lisp specification [4, Section 4.2.3]. Such choice of internal data structure offers advantages such as homoiconicity, making the internal representation human readable in simple cases, and making programmatic manipulation intuitive, as well as enabling the direct use of built-in Common Lisp functions such as `typep` and `subtypep`. However, this approach does present some challenges. Such programs often make use of ad-hoc logic reducers—attempting to convert types to canonical form. These reducers can be complicated and difficult to debug. In addition run-time decisions about type equivalence and subtyping can suffer performance problems.

In this article we present an alternative internal representation for Common Lisp types: the Binary Decision Diagram (BDD) [6, 2]. BDDs have interesting characteristics such as representational equality; *i.e.* it can be arranged that equivalent expressions or equivalent sub-expressions are represented by the same object (`eq`). While techniques to implement BDDs with these properties are well documented, an attempt apply the techniques directly to the Common Lisp type system encounters obstacles which we analyze and document in this article.

In order to compare performance characteristics of the two data structure approaches, we have constructed a problem called Maximal Disjoint Type Decomposition (MDTD): decomposing a given set of potentially overlapping types into a set of disjoint types. Although MDTD is interesting in its own right, we do not attempt, in this paper, to motivate in detail the applications or implications of the problem. We consider such development and motivation a matter of future research. Our use of the MDTD problem in this article is primarily a performance comparison vehicle.

We present two algorithms to compute the MDTD, and separately implement the algorithms with both data structures s-expressions and BDDs (4 implementations in total). Finally, we report performance characteristics of the four algorithms implemented in Common Lisp.

Key contributions of this article are:

- A description of how to extended known BDD related implementation techniques to represent Common Lisp types and facility type based calculations.
- Performance comparison of algorithms using traditional s-expression based type specifiers *vs.* using the BDD data structure.
- A graph based algorithm for reducing the computational complexity of MDTD.

2. DISJOINT TYPE DECOMPOSITION

In presenting the problem of decomposing a set of overlapping types into non-overlapping subtypes, we start with an example intended to convey an intuition of the problem. We continue by defining precisely what we intend to calculate. Then in sections 2.1 and 2.2 we present two different algorithms for performing that calculation.

ELS '10 April 3–4, 2016, Brussels, Belgium

ACM ISBN 978-1-4503-2138-9.

DOI: 10.1145/1235

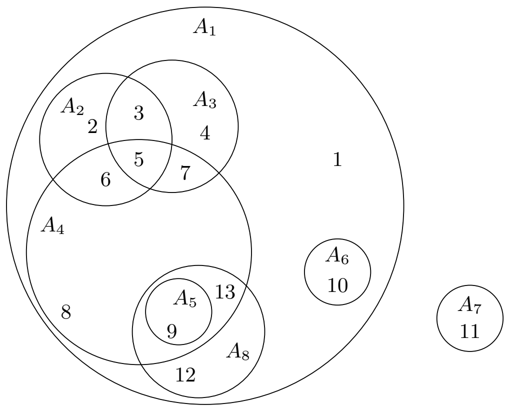


Figure 1: Example Venn Diagram

Disjoint Set	Derived Expression
X_1	$A_1 \cap \overline{A_2} \cap \overline{A_3} \cap \overline{A_4} \cap \overline{A_6} \cap \overline{A_8}$
X_2	$A_2 \cap \overline{A_3} \cap \overline{A_4}$
X_3	$A_2 \cap A_3 \cap \overline{A_4}$
X_4	$A_3 \cap \overline{A_2} \cap \overline{A_4}$
X_5	$A_2 \cap A_3 \cap A_4$
X_6	$A_2 \cap A_4 \cap \overline{A_3}$
X_7	$A_3 \cap A_4 \cap \overline{A_2}$
X_8	$A_4 \cap \overline{A_2} \cap \overline{A_3} \cap \overline{A_8}$
X_9	A_5
X_{10}	A_6
X_{11}	A_7
X_{12}	$A_8 \cap \overline{A_4}$
X_{13}	$A_4 \cap A_8 \cap \overline{A_5}$

Figure 2: Disjoint Decomposition of Sets from Figure 1

In the Venn diagram in Figure 1, $V = \{A_1, A_2, ..., A_8\}$. We wish to construct logical combinations of those sets to form as many mutually disjoint subsets as possible. The resulting *decomposition* should have the same union as the original set. The maximal disjoint decomposition $D = \{X_1, X_2, ..., X_{13}\}$ of V is shown in Figure 2.

NOTATION 1. *We use the symbol, $\perp$, to indicate the disjoint relation between sets. I.e., we take $A \perp B$ to mean $A \cap B = \varnothing$. We also say $A \not\perp B$ to mean $A \cap B \neq \varnothing$.*

NOTATION 2. *We use the notation, $A \subset B$, ($A \supset B$) to indicate that A is either a strict subset (superset) of B or is equal to B.*

DEFINITION 1. *Let U be a set and V be a set of subsets of U. The Boolean closure of V, denoted $\widehat{V}$, is the (smallest) super-set of V such that $\alpha, \beta \in \widehat{V} \implies \{\alpha \cap \beta, \alpha \cap \overline{\beta}\} \subset \widehat{V}$.*

DEFINITION 2. *Let U be a set, and let V and D be finite sets of non-empty subsets of U. D is said to be a disjoint decomposition of V, if the elements of D are mutually disjoint, $D \subset \widehat{V}$, and $\bigcup_{X \in D} X = \bigcup_{A \in V} A$. If no larger set fulfills those properties, D is said to be the maximal disjoint decomposition of V.*

We claim without proof that there exists a unique maximal disjoint decomposition of a given V. A more complete discussion and formal proof are available [14].

The MDTD problem: Given a set U and a set of subsets thereof, $V = \{A_1, A_2, ..., A_M\}$, suppose that for each pair (A_i, A_j), we know which of the relations hold: $A_i \subset A_j$, $A_i \supset A_j$, $A_i \perp A_j$. We would like to compute the maximal disjoint decomposition of V.

In Common Lisp, a type is a set of (potential) values [4, Section Type], so it makes sense to consider the maximal disjoint decomposition of a set of types.

2.1 The RTE Algorithm

We first encountered the MDTD problem in our previous work on regular type expressions (RTE) [15]. The following algorithm was the one presented in that paper, where we pointed that the algorithm suffers from significant performance issues. Performance issues aside, a notable feature of the RTE version of the MDTD algorithm is that it easily fits in 40 lines of Common Lisp code, so it is easy to implement and easy to understand.

1. Let U be the set of sets. Let V denote the set of disjoint sets, initially $D = \varnothing$.
2. Identify all the sets which are disjoint from each other and from all the other sets. ($\mathcal{O}(n^2)$ search) Remove these sets from U and collect them in D.
3. If possible, choose X and Y, for which $X \not\perp Y$.
4. Remove X and Y from U, and add any of $X \cap Y$, $X \backslash Y$, and $Y \backslash X$ which are non-empty. *I.e.*, $U \leftarrow (U \backslash \{X, Y\}) \cup (\{X \cap Y, X \backslash Y, Y \backslash X\} \backslash \{\varnothing\})$
5. Repeat steps 2 through 4 until $U = \varnothing$, at which point we have collected all the disjoint sets in D.

2.2 The graph based algorithm

One of the sources of inefficiency of the algorithm explained in Section 2.1 is at each iteration of the loop, an $\mathcal{O}(n^2)$ search is made to find sets which are disjoint from all remaining sets. This search can be partially obviated if we employ a little extra book-keeping. The fact to realize is that if $X \perp A$ and $X \perp B$, then we know *a priori* that $X \perp A \cap B$, $X \perp A \backslash B$, $X \perp B \backslash A$. This knowledge eliminates some of useless operations.

This algorithm is semantically similar to the algorithm shown in Section 2.1, but rather than relying on Common Lisp primitives to make decisions about connectivity of types, it initializes a graph representing the initial relationships, and thereafter manipulates the graph maintaining connectivity information. This algorithm is more complicated in terms of lines of code, 250 lines of Common Lisp code as opposed to 40 lines for the algorithm in Section 2.1.

Figure 3 shows a graph representing the topology (connectedness) of the diagram shown in Figure 1. Nodes ①, ②, ... ⑧ in Figure 3 correspond respective to A_1, A_2, ... A_8 in Figure 1. Blue arrows correspond to subset relations, pointing from subset to superset, and green lines correspond to other non-disjoint relations.

To construct this graph first eliminate duplicate sets. *I.e.*, if $X \subset Y$ and $X \supset Y$, then discard either X or Y. It is necessary to consider each pair (X, Y) of sets, $\mathcal{O}(n^2)$ loop.

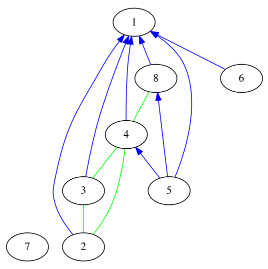


Figure 3: Topology graph

- If $X \subset Y$, draw a blue arrow $X \rightarrow Y$
- Else if $X \supset Y$, draw a blue arrow $X \leftarrow Y$
- Else if $X \not\perp Y$, draw green line between X and Y.
- If it cannot be determined whether $X \subset Y$, assume the worst case, that they are non-disjoint, and draw green line between X and Y.

The algorithm proceeds by breaking the green and blue connections, in explicit ways until all the nodes become isolated. There are two cases to consider. Repeat alternatively applying both tests until all the nodes become isolated.

2.2.1 *Subset relation*

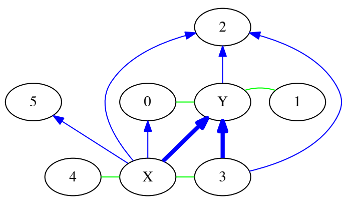
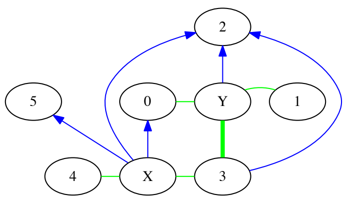
Before	
After	
Node	**Re-labeled Boolean expression**
X	X
Y	$Y \cap \overline{X}$

Figure 4: Subset before and after mutation

A blue arrow from X to Y may be eliminated if X has no blue arrow pointing to it, in which case Y must be relabeled as $Y \cap \overline{X}$ as indicated in Figure 4.

Figure 4 illustrates this mutation. Node Ⓨ may have other connections, including blue arrows pointing to it or from it, and green lines connected to it. However node Ⓧ has no blue arrows pointing to it; although it may have other blue arrows pointing away from it.

If X touches (via a green line) any sibling nodes, *i.e.* any other node that shares Y as super-class, then the blue arrow is converted to a green line. In the *before* image of Figure 4 there is a blue arrow from ③ to Ⓨ and in the *after* image this arrow has been converted to a green line.

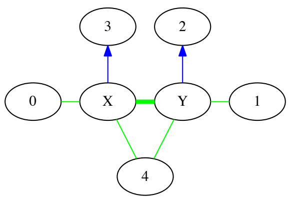
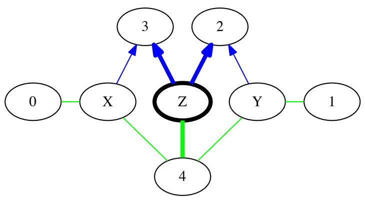
Before	
After	
Node	**Re-labeled Boolean expression**
X	$X \cap \overline{Y}$
Y	$\overline{X} \cap Y$
Z	$X \cap Y$

Figure 5: Touching connections before and after mutation

2.2.2 *Touching connections*

A green line connecting X and Y may be eliminated if neither X nor Y has a blue arrow pointing to it. Consequently, X and Y must be relabeled and a new node must be added to the graph as indicated in Figure 5. The figure illustrates the step of breaking such a connection between nodes Ⓧ and Ⓨ by introducing the node Ⓩ.

Construct blue arrows from this node, Z, to all the nodes which either X or Y points to (union). Construct green lines from Z to all nodes which both X and Y connect to (intersection). If this process results in two nodes connected both by green and blue, omit the green line.

3. TYPE SPECIFIER MANIPULATION

To correctly implement the MDTD by either strategy described above, we need operators to test for type-equality, type disjoint-ness, subtype-ness, and type-emptiness. Given a *subtype* predicate, the other predicates can be constructed.

The emptiness check: $A = \emptyset \iff A \subset \emptyset$. The disjoint check: $A \perp B \iff A \cap B \subset \emptyset$. Type equivalence $A = B \iff A \subset B$ and $B \subset A$.

Common Lisp has a flexible type calculus making type specifiers human readable and also related computation possible. Even with certain limitations, s-expressions are an intuitive data structure for programmatic manipulation of type specifiers in analyzing and reasoning about types.

If `T1` and `T2` are Common Lisp type specifiers, the type specifier `(and T1 T2)` designates the intersection of the types. Likewise `(and T1 (not T2))` is the type difference. The empty type and the universal type are designated by `nil` and `t` respectively. The `subtypep` function serves as the subtype predicate. Consequently `(subtypep '(and T1 T2) nil)` computes whether `T1` and `T2` are disjoint.

There is an important caveat however. The `subtypep` function is not always able to determine whether the named types have a subtype relationship [5]. In such a case, `subtypep` returns `nil` as its second value. This situation occurs most notably in the cases involving the `satisfies` type specifier. For example, to determine whether the `(satisfies F)` type is empty, it would be necessary to solve the halting problem, finding values for which the function `F` returns true.

As a simple example of how the Common Lisp programmer might manipulate s-expression based type specifiers, consider the following problem. In SBCL 1.3.0, the expression `(subtypep '(member :x :y) 'keyword)` returns `nil,nil`, rather than `t,t`. Although this is compliant behavior, the result is unsatisfying, because clearly both `:x` and `:y` are elements of the `keyword` type. By manipulating the type specifier s-expressions, the user can implement a smarter version of `subtypep` to better handle this particular case. Regrettably, the user cannot force the system to use this smarter version internally.

```
(defun smarter-subtypep (t1 t2)
  (multiple-value-bind (T1<=T2 OK) (subtypep t1 t2)
    (cond
      (OK
       (values T1<=T2 t))
      ;; (eql obj) or (member obj1 ...)
      ((typep t1 '(cons (member eql member)))
       (values (every #'(lambda (obj)
                          (typep obj t2))
                      (cdr t1))
               t))
      (t
       (values nil nil)))))
```

As mentioned above, programs manipulating s-expression based type specifiers can easily compose type intersections, unions, and relative complements as part of reasoning algorithms. Consequently, the resulting programmatically computed type specifiers may become deeply nested, resulting in type specifiers which may be confusing in terms of human readability and debuggability. The following programmatically generated type specifier is perfectly reasonable for programmatic use, but confusing if it appears in an error message, or if the developer encounters it while debugging.

```
(or
 (or (and (and number (not bignum))
          (not (or fixnum (or bit (eql -1)))))
     (and (and (and number (not bignum))
               (not (or fixnum (or bit (eql -1)))))
          (not (or fixnum (or bit (eql -1))))))
 (and (and (and number (not bignum))
           (not (or fixnum (or bit (eql -1)))))
      (not (or fixnum (or bit (eql -1))))))
```

This somewhat obfuscated type specifier is semantically equivalent to the more humanly readable form `(and number (not bignum) (not fixnum))`. Moreover, it is possible to write a Common Lisp function to *simplify* many complex type specifiers to simpler form.

There is a second reason apart from human readability which motivates reduction of type specifiers to canonical form. The problem arises when we wish to programmatically determine whether two s-expressions specify the same type, or in particular when a given type specifier specifies the `nil` type. Sometimes this question can be answered by calls to `subtypep` as in `(and (subtypep T1 T2) (subtypep T2 T1))`. However, as mentioned earlier, `subtypep` is allowed to return `nil,nil` in some situations, rendering this approach futile in many cases. If, on the other hand, two type specifiers can be reduced to the same canonical form, we can conclude that the specified types are equal.

We have implemented such a function, `reduce-lisp-type`. It does a good job of reducing the given type specifier toward a canonical form, by repeatedly recursively descending the expression, re-writing sub-expressions, incrementally moving the expression toward a fixed point. We choose to convert the expression to a disjunctive normal form, *e.g.*, `(or (and (not a) b) (and a b (not c)))`. The reduction procedure follows the models presented by Sussman and Abelson [1, p. 108] and Norvig [16, ch. 8].

4. BINARY DECISION DIAGRAMS

A challenge using s-expressions for programmatic representation of type specifiers is the need to after-the-fact reduce complex type specifiers to a canonical form. This reduction can be computationally intense, and difficult to implement correctly. We present here a data structure called the Binary Decision Diagram (BDD) [6, 2], which obviates much of the need to reduce to canonical form because it maintains a canonical form by design. Before looking at how the BDD can be used to represent Common Lisp type specifiers, we first look at how BDDs are used traditionally to represent Boolean equations. Thereafter, we explain how this traditional treatment can be enhanced to represent Common Lisp types.

4.1 Representing Boolean expressions

Andersen [3] summarized many of the algorithms for efficiently manipulating BDDs. Not least important in Andersen's discussion is how to use a hash table and dedicated constructor function to eliminate redundancy within a single BDD and within an interrelated set of BDDs. The result of Andersen's approach is that if you attempt to construct two BDDs to represent two semantically equivalent but syntactically different Boolean expressions, then the two resulting BDDs are pointers to the same object.

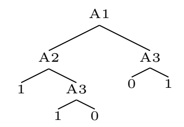


Figure 6: BDD for $(A_1 \wedge A_2) \vee (A_1 \wedge \neg A_2 \wedge A_3) \vee (\neg A_1 \wedge \neg A_3)$

Figure 6 shows an example BDD illustrating a function of three Boolean variables: A_1, A_2, and A_3. To reconstruct the DNF (disjunctive normal form), collect the paths from the root node, A_1, to a leaf node of 1, ignoring paths terminated by 0. When the right child is traversed, the Boolean complement ($\neg$) of the label on the node is collected (*e.g.* $\neg A_3$), and when the left child is traversed the non-inverted parent is collected. Interpret each path as a conjunctive clause, and form a disjunction of the conjunctive clauses. In the figure the three paths from A_1 to 1 identify the three conjunctive clauses $(A_1 \wedge A_2)$, $(A_1 \wedge \neg A_2 \wedge A_3)$, and $(\neg A_1 \wedge \neg A_3)$.

4.2 Representing types

Castagna [8] explains the connection of BDDs to type theoretical calculations, and provides straightforward algorithms for implementing set operations (intersection, union, relative complement) of types using BDDs. The general recursive algorithms for computing the BDDs which represent the common Boolean algebra operators are straightforward.

Let B, B_1, and B_2 denote BDDs, $B_1 = (if\ a_1\ C_1\ D_1)$ and $B_2 = (if\ a_2\ C_2\ D_2)$.

C_1, C_2, D_1, and D_2 represent BDDs. The a_1 and a_2 are intended to represent type names, but for the definition to work it is only necessary that they represent labels which are order-able. We would eventually like the labels to accommodate Common Lisp type type names, but this is not immediately possible.

The formulas for $(B_1 \vee B_2)$, $(B_1 \wedge B_2)$, and $(B_1 \setminus B_2)$ are similar to each other. If $\circ \in \{\vee, \wedge, \setminus\}$, then

$$B_1 \circ B_2 = \begin{cases} (if\ a_1 & (C_1 \circ C_2) & (D_1 \circ D_2)) & \text{for } a_1 = a_2 \\ (if\ a_1 & (C_1 \circ B_2) & (D_1 \circ B_2)) & \text{for } a_1 < a_2 \\ (if\ a_2 & (B_1 \circ C_2) & (B_1 \circ D_2)) & \text{for } a_1 > a_2 \end{cases}$$

There are several special cases, the first three of which serve as termination conditions for the recursive algorithms.

- $(t \vee B)$ and $(B \vee t)$ reduce to t.
- $(nil \wedge B)$, $(B \wedge nil)$, and $(B \backslash t)$ reduce to nil.
- $(t \wedge B)$, $(B \wedge t)$, $(nil \vee B)$, and $(B \vee nil)$ reduce to B.
- $(t \setminus (if\ a\ B_1\ B_2))$ reduces to $(if\ a\ (t \backslash B_1)\ (t \backslash B_2))$.

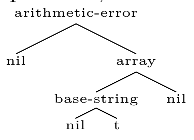
4.3 Representing Common Lisp types

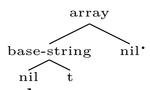
We have implemented the BDD data structure as a set of CLOS classes. In particular, there is one leaf-level CLOS class for an internal tree node, and one singleton class/instance for each of the two possible leaf nodes, *true* and *false*.

The label of the BDD contains a Common Lisp type name, and the logical combinators (`and`, `or`, and `not`) are represented implicitly in the structure of the BDD.

A disadvantage BDDs present when compared to s-expressions as presented in Section 3 is the loss of homoiconicity. Whereas, s-expression based type-specifiers may appear in-line in the Common Lisp code, BDDs may not.

A remarkable fact about this representation is that any two logically equivalent Boolean expressions have exactly the same BDD structural representation, provided the node labels are consistently, totally ordered. Andersen[3] provides a proof for this claim. For example, the expression from Figure 6, $(A_1 \wedge A_2) \vee (A_1 \wedge \neg A_2 \wedge A_3) \vee (\neg A_1 \wedge \neg A_3)$ is equivalent to $\neg((\neg A_1 \vee \neg A_2) \wedge (\neg A_1 \vee A_2 \vee \neg A_3) \wedge (A_1 \vee A_3))$. So they both have the same shape as shown in the Figure 6. However, if we naïvely substitute Common Lisp type names for Boolean variables in the BDD representation as suggested by Castagna, we find that this equivalence relation does not hold in many cases related to subtype relations in the Common Lisp type system.

An example is that the Common Lisp two types `(and (not arithmetic-error) array (not base-string))` *vs.* `(and array (not base-string))` are equivalent, but the naïvely constructed BDDs are different: 

vs. 
.

In order to assure the minimum number of BDD allocations possible, and thus ensure that BDDs which represent equivalent types are actually represented by the same BDD, the suggestion by Andersen [3] is to intercept the BDD constructor function. This constructor should assure that it never returns two BDD which are semantically equivalent but not `eq`.

4.4 Canonicalization

Several checks are in place to reduce the total number of BDDs allocated, and to help assure that two equivalent Common Lisp types result in the same BDD. The following sections, 4.4.1 through 4.4.5 detail the operations which we found necessary to handle in the BDD construction function in order to assure that equivalent Common Lisp type specifiers result in identical BDDs. The first two come directly from Andersen's work. The remaining are our contribution, and are the cases we found necessary to implement in order to enhance BDDs to be compatible with the Common Lisp type system.

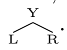
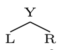
4.4.1 *Equal right and left children*

An optimization noted by Andersen is that if the left and right children are identical then simply return one of them, without allocating a new BDD [3].

4.4.2 *Caching BDDs*

Another optimization noted by Andersen is that whenever a new BDD is allocated, an entry is made into a hash table so that the next time a request is made with the exactly same label, left child, and right child, the already allocated BDD is returned. We associate each new BDD with a unique integer, and create a hash key which is a list (a triple) of the type specifier (the label) followed by two integers corresponding to the left and right children. We use a Common Lisp `equal` hash table for this storage, although we'd like to investigate whether creating a more specific hash function specific to our key might be more efficient.

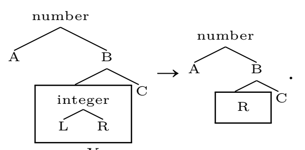
4.4.3 *Reduction in the presence of subtypes*

Since the nodes of the BDD represent Common Lisp types, other specific optimizations are made. The cases include situations where types are related to each other in certain ways: subtype, supertype, and disjoint types. In particular there are 12 optimization cases, detailed in Table 1. Each of these optimizations follows a similar pattern: when constructing a BDD with label X, search in either the left or right child to find a BDD, 
. If X and Y have a particular relation, different for each of the 12 cases, then the 
BDD reduces either to L or R. Two cases, 5 and 7, are further illustrated below.

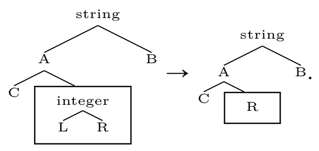
Case	Child to search	Relation	Reduction
1	$X.left$	$X \perp Y$	$Y \to Y.right$
2	$X.left$	$X \perp \overline{Y}$	$Y \to Y.left$
3	$X.right$	$\overline{X} \perp Y$	$Y \to Y.right$
4	$X.right$	$\overline{X} \perp \overline{Y}$	$Y \to Y.left$
5	$X.right$	$X \supset Y$	$Y \to Y.right$
6	$X.right$	$X \supset \overline{Y}$	$Y \to Y.left$
7	$X.left$	$\overline{X} \supset Y$	$Y \to Y.right$
8	$X.left$	$\overline{X} \supset \overline{Y}$	$Y \to Y.left$
9	$X.left$	$X \subset Y$	$Y \to Y.left$
10	$X.left$	$X \subset \overline{Y}$	$Y \to Y.right$
11	$X.right$	$\overline{X} \subset Y$	$Y \to Y.left$
12	$X.right$	$\overline{X} \subset \overline{Y}$	$Y \to Y.right$

Table 1: BDD optimizations

Case 5: If $X \supset Y$ and $\overset{Y}{L\ \ R}$ appears in $X.right$, then $\overset{Y}{L\ \ R}$ reduces to R. *E.g.*, $integer \subset number$; if $X = number$ and $Y = integer$; thus

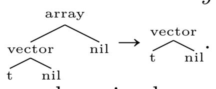


Case 7: If $\overline{X} \supset Y$ and $\overset{Y}{L\ \ R}$ appears in $X.left$, then $\overset{Y}{L\ \ R}$ reduces to R. *E.g.*, $integer \subset \overline{string}$; if $X = string$ and $Y = integer$; thus



4.4.4 Reduction to child

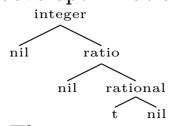
The list of reductions described in Section 4.4.3 fails to apply in cases where the root node itself needs to be eliminated. For example, since $vector \subset array$ we would like the following reductions:



The solution which we have implemented is that before constructing a new BDD, we first ask whether the resulting BDD is type-equivalent to either the left or right children using the `subtypep` function. If so, we simply return the appropriate child without allocating the parent BDD. The expense of this type-equivalence is mitigated by the memoization. Thereafter, the result is in the hash table, and it will be discovered as discussed in Section 4.4.2.

4.4.5 More complex type relations

There are a few more cases which are not covered by the above optimizations. Consider the following BDD:



This represents the type `(and (not integer) (not ratio) rational)`, but in Common Lisp `rational` is identical to `(or integer ratio)`, which means `(and (not integer) (not ratio) rational)` is the empty type. For this reason, as a last resort before allocating a new BDD, we check, using the Common Lisp function `subtypep`, whether the type specifier specifies the `nil` or `t` type. Again this check is expensive, but the expense is mitigated in that the result is cached.

5. MDTD IN COMMON LISP

When attempting to implement the algorithms discussed in Sections 2.1 and 2.2 the developer finds it necessary to choose a data structure to represent type specifiers. Which ever data structure is chosen, the program must calculate type intersections, unions, and relative complements and type equivalence checks and checks for the empty type. As discussed in Section 3, s-expressions (*i.e.* lists and symbols) is a valid choice of data structure and the aforementioned operations may be implemented as list constructions and calls to the `subtypep` predicate.

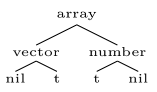


Figure 7: BDD representing `(or number (and array (not vector))`

As introduced in Section 4, another choice of data structure is the BDD. Using the BDD data structure along with the algorithms described in Section 4 we can efficiently represent and manipulate Common Lisp type specifiers. We may programmatically represent Common Lisp types largely independent of the actual type specifier representation. For example the following two type specifiers denote the same set of values: `(or number (and array (not vector)))` and `(not (and (not number) (or (not array) vector)))`, and are both represented by the BDD shown in Figure 5. Moreover, unions, intersections, and relative complements of Common Lisp type specifiers can be calculated using the reduction BDD manipulation rules also explained in Section 4.

We have made comparisons of the two algorithms described in Sections 2.1, 2.2. One implementation of each uses s-expressions, one implementation of each uses BDDs. Some results of the analysis can be seen in Section 6.

Using BDDs in these algorithms allows certain checks to be made more easily than with the s-expression approach. For example, two types are equal if they are the same object (pointer comparison, `eq`). A type is empty if it is identically the empty type (pointer comparison). Finally, given two types (represented by BDDs), the subtype check can be made using the following function:

```
(defun bdd-subtypep (bdd-sub bdd-super)
  (eq *bdd-false*
      (bdd-and-not bdd-sub bdd-super)))
```

This implementation of `bdd-subtype` should not be interpreted to mean that we have obviated the need for the Common Lisp `subtypep` function. In fact, `subtypep`, is still useful in constructing the BDD itself. However, once the BDDs have been constructed, and cached, subtype checks may at that point avoid calls to `subtypep`, which in some cases might otherwise be more compute intensive.

6. PERFORMANCE OF MDTD

Sections 2.1 and 2.2 explained two different algorithms for calculating type decomposition. We look here at some performance characteristics of the two algorithms. The algorithms from Section 2.1 and Section 2.2 were tested using both the Common Lisp type specifier s-expression as data structure and also using the BDD data structure as described in Section 5. Figures 9 and 8 contrast the four effective algorithms in terms of execution time vs sample size.

We attempted to plot the results many different ways: time as a function of input size, number of disjoint sets in the input, number of new types generated in the output. Some of these plots are available in the technical report [14]. The plot which we found heuristically to show the strongest visual correlation was calculation time vs the integer product of the number of given input types multiplied by the number of calculated output types. *E.g.*, if the algorithm takes a list of 5 type specifiers and computes 3 disjoint types in 0.1 seconds, the graph contains a point at (15,0.1). Although we don't claim to completely understand why this particular plotting strategy shows better correlation than the others we tried, it does seem that all the algorithms begin a $\mathcal{O}(n^2)$ loop by iterating over the given set of types which is incrementally converted to the output types, so the algorithms in some sense finish by iterating over the output types. More research is needed to better understand the correlation.

6.1 Performance Test Setup

The type specifiers used in Figure 9 are those designating all the subtypes of `fixnum` such as. `(member 2 6 7 9)` and `(member 1 2 8 10)`. The type specifiers used in Figure 8 are those designating a randomly selected set of subtypes of `cl:number` and `cl:condition` together with programmatically generated logical combinations thereof such as `(and number (not bit))` and `(or real type-error)`.

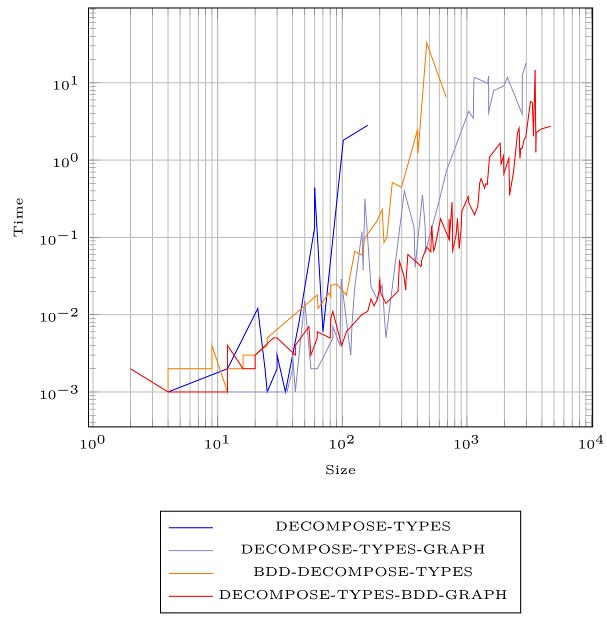


Figure 8: Combinations of number and condition

The performance tests comprise starting with a list of randomly selected type specifiers from a pool, calling each of the four functions to calculate the disjoint decomposition, and recording the time of each calculation. We have plotted in Figures 9 and 8 the results of the runs which took less than 30 seconds to complete. This omission does not in any way effect the presentation of which algorithms were the fastest on each test.

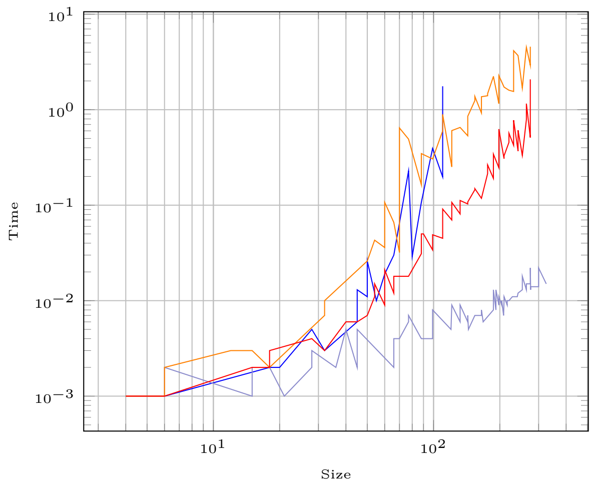


Figure 9: Subtypes of fixnum

The tests were performed on a MacBook 2 GHz Intel Core i7 processor with 16GB 1600 MHz DDR3 memory, and using SBCL 1.3.0 ANSI Common Lisp.

6.2 Analysis of Performance Tests

There is no clear winner for small sample sizes. But it seems the tree based algorithms do very well on large sample sizes. This is not surprising, as the graph based algorithm was designed with the intent to reduce the number of passes, and take advantage of subtype and disjointness information.

Often the better performing of the graph based algorithms is the BDD based one as shown in Figure 8. However there is a notable exception shown in Figures 9 where graph algorithm using s-expressions performs best.

7. RELATED WORK

Computing a disjoint decomposition when permitted to look into the sets has been referred to as *union find* [17, 11]. MDTD differs in that we decompose the set without knowledge of the specific elements; *i.e.* we are not permitted to iterate over or visit the individual elements. The correspondence of types to sets and subtypes to subsets thereof is also treated extensively in the theory of semantic subtyping [9].

BDDs have been used in electronic circuit generation[10], verification, symbolic model checking[7], and type system models such as in XDuce [12]. None of these sources discusses how to extend the BDD representation to support subtypes.

Decision tree techniques are useful in the efficient compilation of pattern matching constructs in functional languages[13]. An important concern in pattern matching compilation is finding the best ordering of the variables which is known to be NP-hard. However, when using BDDs to represent Common Lisp type specifiers, we obtain representation (pointer) equality, simply by using a consistent ordering; finding the *best* ordering is not necessary for our application.

8. CONCLUSION AND FUTURE WORK

The results of the performance testing in Section 6 lead us to believe that the BDD as data structure for representing Common Lisp type specifiers is promising, but there is still work to do, especially in identifying heuristics to predict its performance relative to more traditional approaches.

It is known that algorithms using BDD data structure tend to trade space for speed. Castagna [8] suggests a lazy version of the BDD data structure which may reduce the memory footprint, which would have a positive effect on the BDD based algorithms. We have spent only a few weeks optimizing our BDD implementation based on the Andersen's description [3], whereas the CUDD [18] developers have spent many years of research optimizing their algorithms. Certainly our BDD algorithm can be made more efficient using techniques of CUDD or others.

Although, we do not attempt, in this paper, to motivate in detail the applications or implications of MDTD, we suspect there may be a connection between the problem, and efficient compilation of `type-case` and its use in improving pattern matching capabilities of Common Lisp. We consider such development and motivation a matter of future research.

An immediate priority in our research is to formally prove the correctness of our algorithms, most notably the graph decomposition algorithm from Section 2.2. Experimentation leads us to believe that the graph algorithm always terminates with the correct answer, nevertheless we admit there may be exotic cases which cause deadlock or other errors.

It has also been observed that in the algorithm explained in section 2.2 that the convergence rate varies depending on the order the reduction operations are performed. We do not yet have enough data to characterize this dependence. Furthermore, the order to break connections in the algorithm in Section 2.2. It is clear that many different strategies are possible, (1) break busiest connections first, (2) break connections with the fewest dependencies, (3) random order, (4) closest to top of tree, etc. These are all areas of ongoing research.

We plan to investigate whether there are other applications MDTD outside the Common Lisp type system. We hope the user of Castagna's techniques [8] on type systems with semantic subtyping may benefit from the optimizations we have discussed.

A potential application with Common Lisp is improving the `subtypep` implementation itself, which is known to be slow in some cases. Section 5 gave a BDD specific implementation of `bdd-subtypep`. We intend to investigate whether existing Common Lisp implementations could use our technique to represent type specifiers in their inferencing engines, and thereby make some subtype checks more efficient.

9. REFERENCES

[1] H. Abelson and G. J. Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, MA, USA, 2nd edition, 1996.

[2] S. B. Akers. Binary decision diagrams. IEEE Trans. Comput., 27(6):509–516, June 1978.

[3] H. R. Andersen. An introduction to binary decision diagrams. Technical report, Course Notes on the WWW, 1999.

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

[5] H. G. Baker. A decision procedure for Common Lisp's SUBTYPEP predicate. Lisp and Symbolic Computation, 5(3):157–190, 1992.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. IEEE Transactions on Computers, 35:677–691, August 1986.

[7] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. Inf. Comput., 98(2):142–170, June 1992.

[8] G. Castagna. Covariance and contravariance: a fresh look at an old issue. Technical report, CNRS, 2016.

[9] G. Castagna and A. Frisch. A gentle introduction to semantic subtyping. In Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, PPDP '05, pages 198–199, New York, NY, USA, 2005. ACM.

[10] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems, pages 365–373, London, UK, UK, 1990. Springer-Verlag.

[11] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. Commununication of the ACM, 7(5):301–303, may 1964.

[12] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. ACM Trans. Program. Lang. Syst., 27(1):46–90, Jan. 2005.

[13] L. Maranget. Compiling pattern matching to good decision trees. In Proceedings of the 2008 ACM SIGPLAN Workshop on ML, ML '08, pages 35–46, New York, NY, USA, 2008. ACM.

[14] J. Newton. Analysis of algorithms calculating the maximal disjoint decomposition of a set. Technical report, EPITA/LRDE, 2017.

[15] J. Newton, A. Demaille, and D. Verna. Type-Checking of Heterogeneous Sequences in Common Lisp. In European Lisp Symposium, Kraków, Poland, May 2016.

[16] P. Norvig. Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp. Morgan Kaufmann, 1992.

[17] M. M. A. Patwary, J. R. S. Blair, and F. Manne. Experiments on union-find algorithms for the disjoint-set data structure. In P. Festa, editor, Proceedings of 9th International Symposium on Experimental Algorithms (SEA'10), volume 6049 of Lecture Notes in Computer Science, pages 411–423. Springer, 2010.

[18] F. Somenzi. CUDD: BDD package, University of Colorado, Boulder.

Type Inference in Cleavir

Alex Wood

ABSTRACT

Type inference is an essential technique for obtaining good runtime performance if code is written in a dynamically typed language such as Common Lisp.

We describe a type-inference technique that works on an intermediate representation of the code in the form of an instruction graph. Unlike more traditional, but similar, intermediate representations, ours manipulates only Common Lisp objects. In other words, low-level computations such as address calculations are not exposed, making it possible for the compiler to determine type information for all lexical variables.

Our technique is expressed as a traditional forward dataflow computation, making it possible to use existing algorithms for such analysis.

The technique described in this paper is part of the Cleavir compiler framework, and it is used in the compiler of the Clasp Common Lisp system.

CCS Concepts

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

Keywords

Common Lisp, compiler optimization, type inference

1. INTRODUCTION

Type inference is an essential technique for implementing high-level languages. In modern statically-typed programming languages such as ML [7] or Haskell [5], type inference is a requirement for a program to be possible to compile. The most common technique for type inference in such languages is known as "Hindley-Milner" [4]. The Hindley-Milner system is efficient and correct, but imposes serious constraints on the form of the language and its type system that make it unsuitable for Common Lisp.

In a dynamically typed language such as Common Lisp [1], type inference is optional, and is used to avoid unnecessary runtime type checks when the compiler can prove the outcome of such type checks at compile time, rather than for semantic effect.

2. PREVIOUS WORK

2.1 The Nimble type inferencer

Henry Baker describes the Nimble type inferencer [1]. His technique works for the pre-standard Common Lisp language, and works by annotating source code with type information in the form of *type declarations*.

Nimble tracks control flow in both directions and maintains both upper and lower bounds on inferred type information, and could take exponential time. This was compensated for by Baker's team by reducing type computations to specialized linear bit-vector arithmetic, but the algorithm was still noticeably slow (on the computers of the time). A comparison of efficiency with our technique has not been made.

The advantage of Baker's technique is that it can be used with any conforming Common Lisp implementation, except that it would need some minor work in order to be applicable to code in the standardized language.

2.2 SBCL

The SBCL implementation of Common Lisp is known to have excellent type-inference capabilities. Its "constraint propagation" algorithm is similar to that described here, but instead operates at control-flow level of basic blocks, and can propagate constraints other than types. Rather than inlining, SBCL relies on type derivation functions associated with higher-level built-in operators. These are necessarily more complex, but removing the necessity of inlining may decrease code size.

3. HIR

Type inference in Cleavir is done after the program has been reduced to a "high-level intermediate representation", or HIR. The HIR consists of nodes called *instructions* connected to each other by arcs to form a graph of the program control flow. Each instruction has zero or more variable or constant inputs, zero or more variable outputs, and represents some small operation that reads only from the inputs and writes only to the outputs.

[1]Unpublished technical report. See for instance http://www.pipeline.com/ hbaker1/TInference.html.

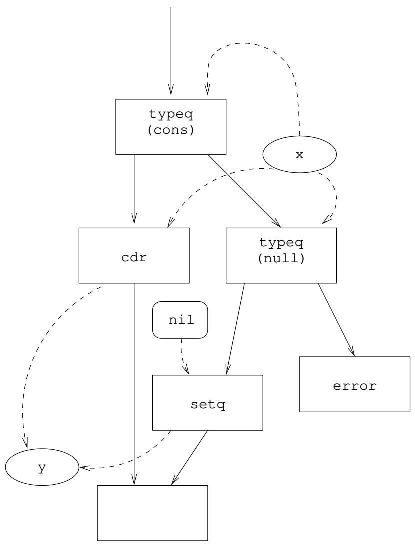


Figure 1: Implementation of `cdr` in HIR. Rectangles represent instructions, ellipses represent variables, and rounded rectangles represent constants. Solid line arrows represent control flow, while dashed lines are data input and output.

This representation is called "high-level" because almost all values for variables are Common Lisp objects, and low-level computations such as address calculations are not involved. Further stages of compilation, including optimizations from type inference, can refine the intermediate representation to a lower-level form.

Most HIR instructions have undefined behavior on values not of certain types. For example, the `cdr` instruction has one input, which must be a cons. To represent the Common Lisp `cdr` function, which can also be validly called on `nil`, type descrimination is necessary.

HIR includes type declaration information with the `the` instruction, which corresponds to the Common Lisp special operator of the same name. A `the` instruction has one input and no outputs, and an associated type. It has no operational effect, but informs the type inferencer that the input is of that type at that control point. After type inference, all `the` instructions can be removed.

Explicit type checks are represented in HIR by the `typeq` instruction. Each typeq instruction has a type associated with it when the HIR graph is produced. When run, typeq branches to one instruction if its one input is of the given type, and to the other if it is not.

For example, `cdr` might be represented as shown in Figure 1. The first typeq's left branch continues to the behavior for cons operands, and the second typeq's left branch continues to the behavior on null operands. The remaining branch is reached only if the operand is neither a cons nor null, and therefore signals a type error.

4. TYPE INFERENCE METHOD

Because Common Lisp types are arbitrary sets, perfect inference of Common Lisp types would be equivalent to the halting problem[2]; additionally, only some type information is useful for an optimizing compiler. Therefore, a small finite lattice is used for type inference.

Elements of the lattice are called *type descriptors*, or just "descriptors". Every Common Lisp type can be conservatively approximated by a descriptor. That is, the type will not contain any elements not also contained in the descriptor. There is a descriptor for the top type, bottom type, and any type that is useful for optimization.

Descriptors are collected into *bags*, which are associative maps from variables to descriptors. Bags are equipped with equality and union operations, which are both only defined for bags with the same set of variables, and consist of the equivalent operation on descriptors mapped over the variables.

Type inference begins by associating each arc with a bag. Initially, every descriptor in every bag is the top type descriptor. As an optimization, only those variables which are *live* (used as inputs in some instruction later in control flow) during the arc are included in each bag.

This map of arcs to bags represents all useful type information, and an altered map is the ultimate result of the type inference procedure. Type inference consists of successively shrinking the descriptors of variables in the bags from the top descriptor.

To proceed with type inference, all instructions with predecessors are iterated over in arbitrary order. For each instruction, a specialized *transfer function* is executed. The transfer function receives as arguments the instruction and a bag. The bag is the union of bags in arcs directly preceding the instruction. The transfer function then computes a bag for each succeeding arc of the instruction based on these arguments.

If the bags so computed are distinct from the bags already associated with those arcs, new information is available. The succeeding instructions are then marked to be iterated over again. This continues until no instructions are left to iterate over, and at this point the current association of arcs with bags is the result of type inference.

For example, a Common Lisp programmer may annotate their program with type information by using the `the` special operator: `(the fixnum foo)` indicates that whenever this form is evaluated, the variable `foo` is of type `fixnum`. `the` is represented in HIR by a `the-instruction`, which has one predecessor, one successor, and one input, the variable `foo`. During type inference, the transfer function for the `the-instruction` returns the information that in the successor arc, `foo` must fit the type descriptor most closely approximating the Common Lisp type `fixnum`, and transfer functions for instructions farther along in the control flow can propagate this information.

5. OPTIMIZATIONS

Inferred type information can be used to remove runtime type discrimination (typeq instructions, as described in Section 3) from the code. If the type of an input to typeq is inferred, it could be that only one branch of the typeq is ever taken, and thus the typeq itself can be removed.

For example, consider the code:

[2]The type of the return value of a pure function could be inferred to be a singleton set (an EQL type) or the empty set (NIL), depending on whether the function halts.

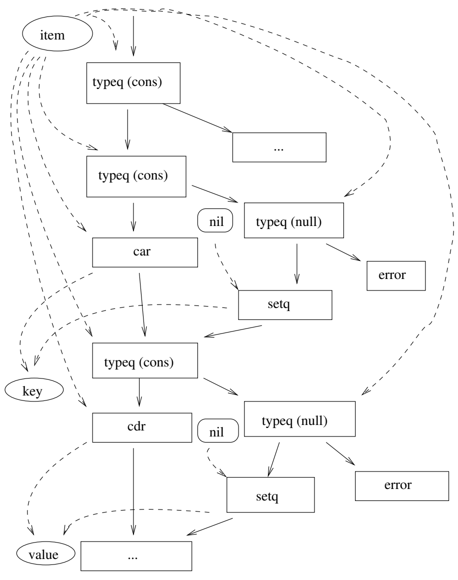


Figure 2: Unoptimized HIR fragment.

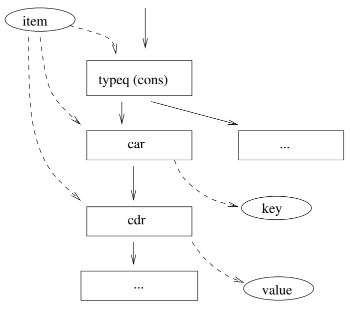


Figure 3: HIR fragment after removing redundant typeqs.

```
(if (consp item)
    (let ((key (car item))
          (value (cdr item)))
      ...)
    ...)
```

This would initially appear in HIR as Figure 2, with the `consp` being inlined as a `typeq`. The `car` and `cdr` calls involve two additional typeqs each, though they are obviously redundant.

The inferencer can determine from the earliest typeq that "item" must be a cons in the first if branch. The other four typeqs can then be eliminated, resulting in the HIR in Figure 3. The semantics of the original program are preserved, but it is implemented without unnecessary branching.

In the Clasp implementation of Common Lisp, this optimization has been tested to reduce the execution time (over ten million iterations) of a Fibonacci function by 56%.

6. CONCLUSIONS AND FUTURE WORK

We have described a technique for type inference of intermediate code resulting from the translation of Common Lisp code to an intermediate representation that manipulates only objects of the source language. Our technique uses a straightforward dataflow algorithm, which is also the most natural way of thinking about the evolution of type information in a Common Lisp program.

Type inference can be much more effective if the intermediate code is first converted to SSA form [2, 3] or some other notation that preserves values of lexical variables across assignments. Assigning to a lexical variable may lose existing type information associated with that variable, so that the type of the new value must be tested again at some later point in the program. Our technique works whether the intermediate code is transformed this way or not, but we have not yet tested the effect of such transformations on the effectiveness of our technique.

At present, the type inference only operates within function, and does not use any information about other functions that isn't explicitly declared. The algorithm can be extended to incorporate type information from local functions without much change, but using inferred types of global functions will require more work due to the semantics of redefinition.

Determining precise dataflow in a Common Lisp program is complicated when a lexical variable is shared between nested closures, and arbitrary dataflow may invalidate essential type information, thereby limiting the effectiveness of type inference. Currently, Cleavir only contains a very rudimentary *escape analysis* phase, forcing our type-inference technique to be very conservative by treating all shared variables as being of unknown type. We intend to improve the quality of the escape analysis so as to improve the effectiveness of our type-inference technique.

Currently, the dataflow computation is implemented with an ad hoc approach. However, Kildall [6] designed a general approach for solving a large spectrum of problems, and our type-inference technique is a good candidate for using that general approach. We already have a general implementation of this approach, and it can be customized by providing a *domain* in the form of a standard instance[3] that provides the details of the required lattice operations. In order to improve maintainability, we intend to adapt our type-inference technique so that it uses this implementation of Kildall's approach.

Finally, the type lattice used by our technique is currently very coarse. A more populated lattice can greatly improve the effectiveness of our technique, though care must be taken to avoid excessive processor time in the type inferencer itself.

After the submission of this paper, considerable progress has been made in implementing the algorithm in terms of Kildall's algorithm, and using an expanded lattice with more joins, but it has not yet been tested extensively.

7. ACKNOWLEDGEMENTS

We thank Robert Strandh for the original design of this method and contributions to this paper, Christian Schafmeister for a timing example and Lisp implementation to test in, and Stas Boukarev for information about SBCL's constraint propagation.

[3]Recall that a *standard instance* is an instance of (a subclass of) the class named `standard-class`. Some authors use the term "CLOS class", but that term does not exist in the Common Lisp standard.

8. REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp.* American National Standards Institute, 1994.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[4] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. ACM.

[5] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[6] G. A. Kildall. A unified approach to global program optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '73, pages 194–206, New York, NY, USA, 1973. ACM.

[7] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML.* MIT Press, Cambridge, MA, USA, 1997.

Session III: Demonstrations

Delivering Common Lisp Applications with ASDF 3.3

Robert P. Goldman
SIFT
rpgoldman@sift.net

Elias Pipping
FU Berlin
elias.pipping@fu-berlin.de

François-René Rideau
TUNES
fare@tunes.org

Abstract

ASDF is the *de facto* standard build system for Common Lisp (CL). In this paper, we discuss the most important improvements in ASDF versions 3.2 and 3.3. ASDF's ability to deliver applications as a single executable file now allows the static linking of arbitrary code written in C. We substantially improved ASDF's portability library UIOP, so its interface to spawn and control external processes now supports asynchronous processes. ASDF permits programmers to extend ASDF's build processes in an object-oriented way; until ASDF 3.2, however, ASDF did not correctly handle updates to these extensions during incremental builds. Fixing this involved managing the multiple phases in an ASDF build session. We also improved ASDF's source finding: it provides more usable default behaviors without any configuration; power users willing to manage its location caching can speed it up; and it offers better compliance with standard configuration locations.

CCS Concepts •**Software and its engineering** →*Software maintenance tools;*

Keywords ASDF, Build System, Common Lisp, Portability, Application Delivery, Demo

1 Introduction

Common Lisp (CL) is a general-purpose programming language with over ten active implementations on Linux, Windows, macOS, etc. ASDF, the *de facto* standard build system for CL, has matured from a wildly successful experiment to a universally used, robust, portable tool. While doing so, ASDF has maintained backward compatibility through many major changes, from Daniel Barlow's original ASDF in 2002 to François-René Rideau's largely rewritten versions, ASDF 2 in 2010, ASDF 3 in 2013, and now ASDF 3.3 in 2017. ASDF is provided as a loadable extension by all actively maintained CL implementations; it also serves as the system loading infrastructure for Quicklisp, a growing collection of now over 1,400 CL libraries. In this paper, we present some of the most notable improvements made to ASDF since we last reported on it [4], focusing on improvements to application delivery and subprocess management, better handling of ASDF extensions, and source location configuration refinements.

2 Application Delivery

ASDF 3 introduced *bundle operations*, a portable way to deliver a software system as a single, bundled file. This single file can be either: (1) a source file, concatenating all the source code; (2) a FASL file, combining all compiled code; (3) a saved image; or (4) a standalone application. In the first two cases, the programmer controls whether or not the bundle includes with a system all the other systems it transitively depends on.

We made bundle operations stable and robust across all active CL implementations and operating systems. We also extended these operations so that ASDF 3.2 supports single-file delivery of applications that incorporate arbitrary C code and libraries. This feature works in conjunction with CFFI-toolchain, an extension which we added to the *de facto* standard foreign function interface CFFI. CFFI-toolchain statically links arbitrary C code into the Lisp runtime. As of 2017, this feature works on three implementations: CLISP, ECL, and SBCL.

Loading a large Lisp application, either from source or from compiled files, can take multiple seconds. This delay may be unacceptable in use cases such as small utility programs, or filters in a Unix pipe chain. ASDF 3 can reduce this latency by delivering a standalone executable that can start in twenty milliseconds. However, such executables each occupy tens or hundreds of megabytes on disk and in memory; this size can be prohibitive when deploying a large number of small utilities. One solution is to deliver a "multicall binary" à la Busybox: a single binary includes several programs; the binary can be symlinked or hardlinked with multiple names, and will select which entry point to run based on the name used to invoke it. Zach Beane's `buildapp` has supported such binaries since 2010, but `buildapp` only works on SBCL, and more recently CCL. `cl-launch`, a portable interface between the Unix shell and all CL implementations, also has supported multicall binaries since 2015.

3 Subprocess Management

ASDF has always supported the ability to synchronously execute commands in a subprocess. Originally, ASDF 1 copied over a function `run-shell-command` from its predecessor `mk-defsystem` [2]; but it could not reliably capture command output, it had a baroque calling convention, and was not portable (especially to Windows). ASDF 3 introduced the function `run-program` that fixed all these issues, as part of its portability library UIOP. By ASDF 3.1 `run-program` provided a full-fledged portable interface to synchronously execute commands in subprocesses: users can redirect and transform input, output, and error-output; by default, `run-program` will throw CL conditions when a command fails, but users can tell it to `:ignore-exit-status`, access and handle exit code themselves.

ASDF 3.2 introduces support for asynchronously running programs, using new functions `launch-program`, `wait-process`, and `terminate-process`. These functions, available on capable implementations and platforms only, were written by Elias Pipping, who refactored, extended and exposed logic previously used in the implementation of `run-program`.

With `run-program` and now `launch-program`, CL can be used to portably write all kind of programs for which one might previously have used a shell script. Except CL's rich data structures, higher-order functions, sophisticated object system, restartable conditions and macros beat the offering of its scripting alternatives [4] [5].

ELS 2017, Brussel, Belgium
2017. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of The 10th European Lisp Symposium, April 3–4, 2017*.

4 Build Model Correctness

The original ASDF 1 introduced a simple "plan-then-perform" model for building software. It also introduced an extensible class hierarchy so ASDF could be extended in Lisp itself to support more than just compiling Lisp files. For example, some extensions support interfacing with C code.

Unfortunately, these two features were at odds with one another: to load a program that uses an ASDF extension, one would in a first phase use ASDF to plan then perform loading the extension; and one would in a second phase plan then perform loading the target program. Of course, there could be more than just two phases: some extensions could themselves require other extensions in order to load, etc. Moreover, the same libraries could be used in several phases.

In practice, this simple approach was effective in building software from scratch, though not necessarily as efficient as possible since libraries could sometimes unnecessarily be compiled or loaded more than once. However, in the case of an incremental build, ASDF would overlook that a change in one phase could affect the build in a later phase, and fail to invalidate and re-perform actions accordingly. Indeed it failed to even consider loading a system definition as an action that may be invalidated and re-performed when it depended on code that had changed. The user was then responsible for diagnosing the failure and forcing a rebuild from scratch.

ASDF 3.3 fixes this issue by supporting the notion of a session in which code is built and loaded in multiple phases. It tracks the status of traversed actions across phases of a session, whereby an action can independently be (1) considered up-to-date or not at the start of the session, (2) considered done or not for the session, and (3) considered needed or not during the session. When ASDF 3.3 merely checks whether an action is still valid from previous sessions, it uses a special traversal that carefully avoids either loading system definitions or performing any other actions that are potentially either out-of-date or unneeded for the session.

Build extensions are a common user need, though most build systems fail to offer proper dependency tracking when they change. Those build systems that do implement proper phase separation to track these dependencies are usually language-specific build systems (like ASDF), but most of them (unlike ASDF) only deal with staging macros or extensions inside the language, not with building arbitrary code outside the language. An interesting case is Bazel, which does maintain a strict plan-then-perform model yet allows user-provided extensions (e.g. to support Lisp [6]). However, its extensions, written in a safe restricted DSL, are not themselves subject to extension using the build system.

To fix the build model in ASDF 3.3, some internals were changed in backward-incompatible ways. Libraries available on Quicklisp were inspected, and their authors contacted if they were (ab)using those internals; those libraries that are still maintained were fixed.

5 Source Location Configuration

In 2010, ASDF 2 introduced a basic principle for all configuration: *allow each one to contribute what he knows when he knows it, and do not require anyone to contribute what he does not know* [1]. In particular, everything should "just work" by default for end-users, without any need for configuration, but configuration should be possible for "power users" and unusual applications.

ASDF 3.1, now offered by all active implementations, includes `~/common-lisp/` as well as `~/.local/share/common-lisp/` in its source registry by default; there is thus always an obvious place in which to drop source code such that ASDF will find it: under the former for code meant to be visible to end-users, under the latter for code meant to be hidden from them.

ASDF 2 and later consult the XDG Base Directory environment variables [3] when locating its configuration. Since 2015, ASDF exposes a configuration interface so all Lisp programs may similarly respect this Unix standard for locating configuration files. The mechanism is also made available on macOS and Windows, though with ASDF-specific interpretations of the standard: XDG makes assumption about filesystem layout that do not always have a direct equivalent on macOS, and even less so on Windows.

Finally, a concern for users with a large number of systems available as source code was that ASDF could spend several seconds the first time you used it just to recursively scan filesystem trees in the source-registry for `.asd` files — a consequence of how the decentralized ASDF system namespace is overly decoupled from any filesystem hierarchy. Since 2014, ASDF provides a script `tools/cl-source-registry-cache.lisp` that will scan a tree in advance and create a file `.cl-source-registry.cache` with the results, that ASDF will consult. Power users who use this script can get scanning results at startup in milliseconds; the price they pay is having to re-run this script (or otherwise edit the file) whenever they install new software or remove old software. This is reminiscent of the bad old days before ASDF 2, when power users each had to write their own script to do something equivalent to manage "link farms", directories full of symlinks to `.asd` files. But at least, there is now a standardized script for power users to do that, whereas things just work without any such trouble for normal users.

6 Conclusions and Future Work

We have demonstrated improvements in how ASDF can be used to portably and robustly deliver software written in CL. While the implementation is specific to CL, many of the same techniques could be applied to other languages.

In the future, there are many features we might want to add, in dimensions where ASDF lags behind other build systems such as Bazel: support for cross-compilation to other platforms, reproducible distributed builds, building software written in languages other than CL, integration with non-Lisp build systems, etc.

Bibliography

[1] François-René Rideau and Robert Goldman. Evolving ASDF: More Cooperation, Less Coordination. 2010.

[2] Mark Kantrowitz. Defsystem: A Portable Make Facility for Common Lisp. 1990.

[3] Waldo Bastian, Ryan Lortie and Lennart Poettering. XDG Base Directory Specification. 2010.

[4] François-René Rideau. Why Lisp is Now an Acceptable Scripting Language. 2014.

[5] François-René Rideau. Common Lisp as a Scripting Language, 2015 edition. 2015.

[6] James Y. Knight, François-René Rideau and Andrzej Walczak. Building Common Lisp programs using Bazel or Correct, Fast, Deterministic Builds for Lisp. 2016.

Radiance - A Web Application Environment

Nicolas Hafner
Shirakumo.org
Zürich, Switzerland
shinmera@tymoon.eu

ABSTRACT

Radiance[1] is a set of libraries that provide an environment for web applications. By putting its focus on running multiple web services or applications within the same environment, it has developed features that set it apart from traditional web frameworks. In particular, it shows a new approach to the way different framework features are provided, and how the routing to content is performed. These differences allow applications written against Radiance to transparently share common resources such as the HTTP server, user accounts, authentication, and so forth with each other.

CCS Concepts

•Software and its engineering → Development frameworks and environments; •Information systems → *Web applications;*

Keywords

Common Lisp, web framework, web development, encapsulation, interfacing

ELS'17 April 3–4, 2017, Brussel, Belgium

© 2017 Copyright held by the owner/author(s).

ACM ISBN .

DOI:

1. INTRODUCTION

As the internet evolved, websites began to evolve and became more and more dynamic. Content is no longer a set of static webpages, but rather automatically updated, or even individually composed for the specific user viewing the website. Creating such dynamic websites requires a lot of extra work, much of which is usually done in much the same way for every website. The requested content needs to be retrieved somehow, transformed as needed, and finally assembled into deliverable HTML.

In order to handle these common aspects, web frameworks have been created. These frameworks can come in all kinds of sizes, from the very minimal set of an HTML templating engine and a web server as seen in "micro-frameworks"[2], to extensive sets of utilities to handle user interaction and data of all kinds.

Typically these frameworks are constructed with the intent of helping you develop a single web application, which is then deployed and run standalone. However, this paradigm can lead to issues when multiple applications should be deployed side-by-side. For example, if the framework does not provide explicit support for a particular feature such as user accounts, the two applications will likely have implemented ad-hoc solutions of their own, which are incompatible with each other and thus can't be trivially merged together. Large, "macro-frameworks" may avoid the problems introduced by ad-hoc feature implementation, but instead run the risk of introducing too many features that remain unused.

Radiance initially grew out of the desire to write web applications that could be run together in such a way, that users would not have to create multiple accounts, track multiple logins, and so forth. Another primary goal of it was to write it in such a way, that it would be possible to exchange various parts of the framework without needing to change application code. This separation would allow flexibility on the side of the administrator, to pick and choose the parts that best fit their environment.

2. EXAMPLE REQUIREMENTS OF AN APPLICATION

In order to illustrate the reasoning behind the individual components present in the Radiance system, we are going to make use of a simple, imaginary example web application throughout the paper. This application should provide a "blog platform," wherein users can create text posts to which they can link their friends. They should also be able to edit the posts again at a later point and customise the look and feel to their preference.

When talking about such an application in the context of Radiance, we think of it as a form of library that is then loaded together with Radiance to form a full webserver. This approach is in contrast to the deployment models of many other frameworks, where the "library" is deployed bundled tightly together with the framework as a single application.

In order to write our blog application, we are going to want a database that stores the posts, a system to handle user accounts and authentication, a cache system for performance, an HTTP server, and a template engine.

Imagine now, if you will, that an administrator also wanted to run a forum on the same host as well, perhaps in order to allow people to discuss and engage with each other about the various posts on the main blog. Requiring users to maintain a login for each service would be annoying, to say the least. Furthermore, as an administrator, we would like to make sure that as much data is kept in the same place as possible, and arranged in a similar way, to ease maintenance.

For our blog to share as many resources as it can with potential third-party applications residing in the same installation, it needs to rely on the framework to provide all of our required features, except perhaps for the template engine. Additionally, it needs to have some system that divides up the URL namespace between applications. As the application writer, we cannot have any presumptions about what the final setup will look like — what kinds of applications will be deployed together, and how the public URLs should resolve.

3. THE RADIANCE SYSTEM

The core of the Radiance system is rather small. Despite this, it can provide a plethora of features in the spirit of a macro-framework, if needed. Most of its features are pluggable, which is done through interfaces.

The division of the URL namespace is provided through the routing system, which allows both a convenient view for developers to work with, and a powerful way for administrators to configure the system to their liking without having to touch application code.

Finally, the management of a Radiance installation is handled through so-called "environments," which specify the configuration of the individual components.

3.1 Interfaces

Interfaces represent a form of contract. They allow you to define the signatures of constructs exposed from a package, such as functions, macros, classes, and so forth. These function signatures are accompanied by documentation that specifies their public behaviour. Through this system, Radiance can provide several "standard interfaces" that specify the behaviour of many features that are commonly needed in web applications, without actually having to implement them.

In order to make this system more concrete, let's look at an interface that might provide some sort of caching mechanism.

```
(define-interface cache
  (defun invalidate (name)
    "Causes the cached value of NAME to be re-computed.")
  (defmacro with-caching (name invalidate &body body)
    "Caches the return value if INVALIDATE is non-NIL."))
```

This construct defines a new package called `cache`, exports the symbols `invalidate` and `with-caching` from it, and installs stub definitions for the respective function and macro. With this interface specification in place, an application can start writing code against it. In our imaginary blog service, we could now cache the page for a specific post like so:

```
(defun post-page (id)
  (cache:with-caching id NIL
    (render-post (load-post id))))
```

However, as it currently is, this function will obviously not work. The `with-caching` macro is not implemented with any useful functionality. As the writer of the blog application, we don't need to know the specifics of how the caching is implemented, though. All we need to do is tell Radiance that we need this interface. This can be done by adding the interface as a dependency to our application's ASDF[3] system definition.

```
(asdf:defsystem blog-service
  ...
  :depends-on (...
               (:interface :cache)))
```

Radiance extends ASDF's dependency resolution mechanism to allow for this kind of declaration. When the `blog-service` system is now loaded, Radiance will notice the interface dependency and resolve it to an actual system that implements the desired caching functionality. This resolution is defined in the currently active environment, and can thus be configured by the administrator of the Radiance installation.

For completeness, let's look at an implementation of this cache interface. The implementation must be loadable through an ASDF system, so that the above dependency resolution can be done. The actual implementation mechanism is handed to us for free, as Lisp allows redefinition.

```
(defvar cache::*caches* (make-hash-table))

(defun cache:invalidate (name)
  (remhash name *caches*))

(defmacro cache:with-caching (name invalidate &body body)
  (once-only (name)
    `(or (and (not ,invalidate) (gethash ,name *caches*))
         (setf (gethash ,name *caches*)
               (progn ,@body)))))
```

This implementation is a particularly primitive and straightforward one, in order to keep things short. The function and macro are provided by just overriding the stub definitions that were in place previously. The variable definition is not part of the official interface, and is instead exposed through an unexported symbol, denoting an implementation-dependant extension.

Using direct overwriting of definitions means that all applications must use the same implementation. However, usually this effect is intended, as we want to maximise the sharing between them. If an implementation should have special needs, it can always bypass the interfaces and make direct use of whatever it might depend on. This approach with interfaces does bring some benefits, however. For one, it allows the implementations to be as efficient as possible, as there is no intermediate layer of redirection.

Radiance provides standard interface definitions for all of the components we require for the blog application, and more. A full list of the interfaces and their descriptions is available in the Radiance documentation[1]. Thus, Radiance can be used like a macro-framework, but does not load any features unless specifically required. Additionally, any of the implementations can be exchanged by a system administrator for ones that more closely match their requirements, without having to change anything about the application code.

Finally, Radiance provides a system that allows the programmer to optionally depend on an interface. This mechanism is useful to model something like the user preferences mentioned in our example application. An administrator might not always want to provide an administration or settings panel. To make this dependency optional, we can defer the compilation and evaluation of our relevant logic to a later point, after an implementation has been loaded. For an imaginary `user-settings` interface, an example might look like the following:

```
(define-implement-trigger user-settings
  (user-settings:define-option ...))
```

Since all the symbols are already provided by the interface definition, there are no problems when the reader parses the source. The forms can thus simply be saved away to be evaluated once Radiance notices that an implementation of the interface in question has been loaded.

Ultimately, interfaces are a form of compromise between providing all possible features at once, and almost no features at all. The usefulness of an interface heavily depends on its specification, and for implementations to be really exchangeable without modifying the application code, each implementation and application must strictly adhere to the specification.

3.2 Routes

In order to allow the administrator to change where pages are accessible from, an application cannot hard-code its resource locations and the links in its templates. It is doubly important to do this when it comes to housing multiple applications in one, as the system needs to be set up in such a way that the applications do not clash with each other or potentially confuse pages of one another.

[1]https://github.com/Shirakumo/radiance/#interface

In many frameworks, like for example Symfony[4], this problem is solved by naming every resource in the system by a tag, and then allowing the configuration of what each tag resolves to individually. Radiance takes a different approach. It introduces the idea of two separate namespaces: an external one, which is what a visitor of a website sees and interacts with, and an internal one, which is what the programmer of an application deals with.

This separation allows application programmers to model their pages from a perspective that looks much more like the external view. Instead of dealing with named tags, they deal in concrete internal URLs that, in a development setup, often have a relatively direct one-to-one mapping to the external URLs. This makes it easier to visualise and think about the application structure. From the administrator's point of view this setup is more convenient too, as they simply need to think in terms of translations, rather than specific tag instances.

The translation between the two namespaces is the responsibility of the routing system. The way it is connected into the life-cycle of a request is illustrated in Figure 1.

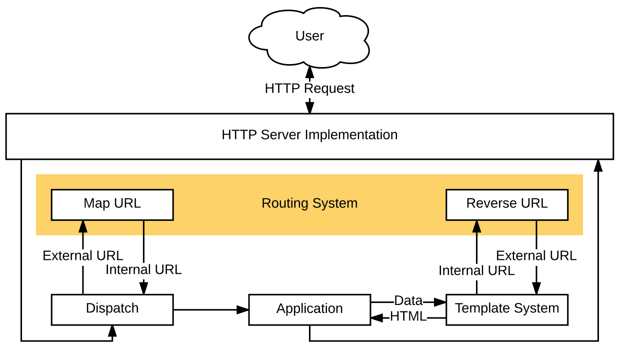


Figure 1: Standard request life-cycle

When a request is dispatched by Radiance, it first parses the request URL into an object presentation that makes it easier to modify. It then sends it through the mapping function of the routing system, which turn this external URL into an internal one. The request is then dispatched to the application that "owns" the URL in the internal representation.

Since Radiance has full control over the organisation of the internal representation, it can make strict demands as to how applications need to structure their URLs. Specifically, it requires each application to put all of its pages on a domain that is named after their application.

In our sample application, we might write the view page something like this, where `blog/view` is the internal URL that the view page is reachable on.

```
(define-page view "blog/view" ()
  (render-template (template "view.html")
                   (get-post (get-var "post-id"))))
```

If, as an administrator of an installation of this blog application, we now wanted to reach this page through the

URL `www.example.com/blog/view`, the mapping route functions would have to strip away the `www.example.com` domain, recognise the `blog` folder, and put that as the URL's domain instead. We can achieve this behaviour relatively easily with the following definition.

```
(define-route blog :mapping (uri)
  (when (begins-with "blog/" (path uri))
    (setf (domains uri) '("blog")
          (path uri) (subseq (path uri) 5))))
```

Alternatively, we could also use the even simpler string-based definition.

```
(define-string-route blog :mapping
  "/blog/(.*)" "blog/\\1")
```

Naturally, if you wanted different behaviour depending on which domain the request came from, you'd have to write a more specific translation.

With the mapping alone the case is not yet solved, though. When a page is emitted that contains links, the links must be translated as well, but in the opposite way. Unfortunately, because routes can be arbitrarily complex, it is not possible for the system to figure out a reversal route automatically. The logic of a reversal route will usually be much the same as it was for the mapping route and should thus not be a problem to figure out, though.

The reversal of URLs should receive special support from the templating system, as it is very frequently needed and should thus be short. Radiance does not dictate any specific template system, but offers extensions to some existing systems like Clip[5] to simplify this process. As a brief example, in Clip, URLs can be denoted through special attributes like this:

```
<a @href="blog/list">Latest Blog Posts</a>
<form method="post">
  <textarea name="text"></textarea>
  <input type="submit" @formaction="blog/submit" />
</form>
```

The templating engine takes care of reading out the `@` attribute values, running them through the reversal functions of the routing system, and putting the resulting URL into the corresponding attribute to produce a valid link.

3.3 Environments

Radiance's "environment" encapsulates the configuration of a Radiance installation. Through it, applications can provide settings that the administrator can set to their liking. It also provides the information necessary to figure out which implementation to use for an interface.

The environment itself simply dictates a directory that contains all of these configuration files. Each application in the system automatically receives its own directory in which it can store both configuration and temporary data files. By simply switching out this environment directory, the system can then be loaded into completely different configurations, for example allowing you to easily keep "development" and "production" setups.

An administrator of a system will not have to touch any source code, and instead can configure the system through a set of human-readable configuration files. It is, of course, still possible to configure the system through usual Lisp code as well, should one prefer this approach.

In our example application we might want to give the administrator the ability to define a global blog title. We could provide a default value for it like this:

```
(define-trigger startup ()
  (defaulted-config "Irradiant Blogs" :title))
```

We need to stick the configuration update into a trigger, in order to defer the evaluation to when Radiance is being started up and the environment has been decided. During the loading of the system, the environment might not have been set yet, and we would not be able to access the configuration storage.

While Radiance does provide default implementations for all of its interfaces, it is likely that some of them are not usable for a production setting. In order to change, say, the implementation of the database interface, we would then have to modify the Radiance core's configuration file. We can either modify the file directly like so:

```
((:interfaces (:database . "i-postmodern")
              ...)
 ...)
```

. . . or instead use the programmatical way, like so:

```
(setf (mconfig :radiance-core :interfaces :database)
      "i-postmodern")
```

`i-postmodern` here is the name of a standard implementation of the database interface for PostgreSQL databases.

Routes can also be configured through the core configuration file. Our previous example mapping route could be set up in the configuration file like this.

```
((:routes (blog :mapping "/blog/(.*)" "blog/\\1"))
 ...)
```

Radiance will take care of converting the configuration data into an actual route definition like we saw above.

The individual configuration of every application can be changed in much the same way.

4. CONCLUSION

Radiance provides a web framework that adjusts itself depending on how many features the application requires. By separating the applications from the implementations of these features with an interface, it allows the application programmer to write their software against a well-specified API, and retains the ability for the administrator to decide which implementation is most suitable for their setup.

By maintaining a strict separation of the URL namespace and providing an automated URL rewriting mechanism, Radiance allows for easy sharing of the namespace between an arbitrary number of applications, while at the same time giving the administrator a convenient way to modify the behaviour of the translation to fit their specific server configuration.

Through the environment system, Radiance standardises the way each application is configured by the administrator and how the system is pieced together when it is loaded. As a consequence of that standardisation it becomes trivial to switch between different setups of a Radiance installation. The simple, human-readable configuration format used allows even users without intimate knowledge of Lisp to set up an instance.

5. FURTHER WORK

Currently, while it is possible to dynamically load applications, it is not possible to dynamically unload them. Doing so is troublesome, as the system of an application might potentially modify any other part of the Lisp process. However, if a constraint on what kind of changes can be rolled back is introduced, it should be possible to provide a usable form of this kind of feature.

Radiance also does not allow you to change the implementation of an interface on the fly. This feature has much the same problems as the previous issue. In addition, though, the system needs to ensure that all dependant applications, including optionally dependant ones, are reloaded to make macro redefinitions take effect. Furthermore, since some interfaces expose classes and instances thereof, the system would either have to be able to migrate objects between interface implementations, or somehow invalidate the old instances if they are retained in another part of the application.

Ultimately it should be made possible to switch out the environment of a Radiance installation on the fly, for example to switch between development and production setups without having to restart completely. Doing so could massively improve the time needed to discover differences between different setups.

6. ACKNOWLEDGEMENTS

Thanks to Joram Schrijver, Till Ehrengruber, and Robert Strandh for feedback, corrections, and suggestions on the paper.

7. REFERENCES

[1] N. Hafner. Radiance, a web application envrionment. [Online; accessed 2016.12.16] http://shirakumo.org/projects/radiance.

[2] Various. Microframework. [Online; accessed 2016.12.16] https://en.wikipedia.org/wiki/Microframework.

[3] Various. Another system definition facility. [Online; accessed 2017.3.14] https://en.wikipedia.org/wiki/Another_System_Definition_Facility.

[4] Various. Symfony. [Online; accessed 2017.3.13] https://en.wikipedia.org/wiki/Symfony.

[5] N. Hafner. Clip, a common lisp html templating engine. [Online; accessed 2016.12.16] https://github.com/Shinmera/clip.

Teaching Students of Engineering some Insights of the Internet of Things using Racket and the RaspberryPi

Daniel Brunner
Systemhaus Brunner & Brunner Software
Schulstr. 8
Biedenkopf 35216, Germany
daniel@dbrunner.de

Stephan Brunner
Systemhaus Brunner & Brunner Software
Schulstr. 8
Biedenkopf 35216, Germany
stephan.brunner@systemhaus-brunner.de

ABSTRACT

We gave a course to teach students of engineering some insights into the Internet of Things (IoT). We started with an introduction into programming using Racket and the *Beginner Student Language (BSL)* teachpack. After that we introduced the RaspberryPi[1] and showed how to read data from a thermal sensor and switch a LED. With this knowledge we taught students to implement a simple publish-subscribe pattern where the RaspberryPi collected some thermal data and a program on their PC monitored these data and could switch the LED depending on the measured temperature. With this setup we explained several aspects of distributed computing and the Internet of Things in particular.

CCS CONCEPTS

•**Social and professional topics** →**Computing education;** •**Applied computing** →*Engineering;* •**Computer systems organization** →*Sensors and actuators;*

KEYWORDS

Racket, BSL, RaspberryPi, IoT, distributed computing

ACM Reference format:
Daniel Brunner and Stephan Brunner. 2016. Teaching Students of Engineering some Insights of the Internet of Things using Racket and the RaspberryPi. In *Proceedings of 10th European Lisp Symposium, Brussels, Belgium, April 3–4, 2017 (10th ELS),* 2 pages.
DOI:

1 INTRODUCTION AND GOAL

Nowadays a lot of news is written about the "Internet of Things" (IoT). In Germany these news are often related to the so-called "Industrie 4.0", a term which describes a new way to organize processes of manufacturing. Although the term is often used in newspapers and even at our universities, a precise and widely accepted definition does not yet exist. But most authors would agree that digitalization influences manufacturing and leads to new forms of manufacturing or even to additional product-related services. Therefore we took this as a starting point and our goal was to teach students of engineering some basic principles on how to design a distributed application.

The basic idea was to have a small device (RaspberryPi with a thermal sensor) which monitors some state of an imaginary machine and sends this data to a central message broker. A separate monitoring system (on the student's PC) should subscribe to these messages and check if the monitored device is in some "healthy" state. If the measured data was out of a given range, some action should be initiated (switch on a LED which was connected to the RaspberryPi).

Although there are a lot of IoT suites available, we wanted to accomplish this setup with very basic tools. On the other hand we wanted to avoid diving into assembler programming or system programming with C. Therefore we chose Racket along with the Beginner Student Language and the *universe* teachpack from the "How to Design Programs" (HtDP) teaching materials.

With these in hand we gave a two-day course at the dual study program *StudiumPlus*[2] of the University of Applied Sciences of Central Hesse (Technische Hochschule Mittelhessen[3]). Most of our students only had little knowledge in programming. Some of them took a C++ course at the beginning of their studies. Therefore we could not rely on any programming skills.

2 BASIC PROGRAMMING

To teach some basic ideas of programming we chose the approach of "How to Design Programs (2nd edition)" (see [2]). We taught some of the material of the first chapters and emphasized on the structural design recipe (see [1]). Students could follow and perform their exercises with Racket's IDE, DrRacket, which works on the student's PC as well as on the RaspberryPi. This took about one third of the whole course.

We tried to limit the material to what really is essential to build a small distributed application. Therefore we taught students about some basic data type of *BSL*: images, numbers, strings, booleans as well as some functions on how to work with these data types. After introducing booleans we went on to teach some logical operators and ended up with conditionals (`cond` and `if` and some predicates). These concepts were introduced using the *universe* teachpack which

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

[2]http://www.studiumplus.de
[3]http://www.thm.de

implements interactive, graphical programs (so-called *world* programs). Along with the data types we introduced the definition of functions and constants using `define`, comments, defining and using structs and how to use `require`.

3 SETUP

After teaching some basics in programming we set up the following components to establish a simple publish-subscribe pattern using the *universe* teachpack, e.g. connecting the *world* programs to the *universe server*, a central control program. To keep things very simple we omitted any authentication, encryption etc. We built teams consisting of two students. Each team received a RaspberryPi.

3.1 Student's PC and RaspberryPi

We provided the students with two modules: one with some basic struct definitions and constants to establish the publish-subscribe pattern and a second one where we hid some system calls to obtain the temperature using the I^2C bus or switching the LED via the GPIO. This was done by simple calls to the command line tools `i2cget` and `gpio`.

We prepared the RaspberryPi with a thermal sensor (LM75, see [3]), a red LED and the newest Raspbian[4] image which contains a graphical user interface. On top of that we installed DrRacket. Therefore students could use a simple VNC viewer to connect to the RaspberryPi and develop their programs directly on the RaspberryPi using the same IDE. Consequently students did not have to bother with Linux' command line programs.

Alternatively they could use the VNC viewer's function to upload files. Furthermore having a GUI is a requirement for the *universe* teachpack because the *world* programs need a `to-draw` clause.

The task for the student's PC was to develop a program which subscribed to the messages of their RaspberryPi, check if it is in a given range of temperatures and publish a suitable "change state" message.

After setting up the PC, the RaspberryPi and the above-described programs the students should monitor their sensor's temperature and watch for the LED to light up which it would if the temperature was out of a given range (e.g. using some cooler spray on the sensor).

3.2 Instructor's PC

The instructor's PC was running the message broker, a simple *server* program that implemented the described protocol. The source code of this component was not shown to the students. They were only taught how to publish and subscribe to data. To give a good overview over all RaspberryPis and their states, we built a small program that subscribed to all messages from the RaspberryPis and showed their states to the whole class via projector.

[4] http://www.raspbian.org

4 DISCUSSION

After the setup was working, we could explain several aspects of distributed systems and IoT in particular like the availability and type of network, low latency, bandwith restrictions, network congestion, authentication, message signing as well as topics like data format of the sensor, issues with time zones, different protocols and file formats. We ended up giving some advice on how to debug such distributed systems if something goes wrong.

Using Racket and the *universe* teachpack (and omitting lots of security measures) resulted in very short programs: Our sample implementation for student's components consisted of 67 lines for the program on the RaspberryPi and 95 lines for monitoring the messages. The modules that hid some implementation details on the structs and system programming (LM75, LED) used another 176 lines. Therefore students were not distracted by an overhead caused by libraries or the programming language itself. On the instructor's side more code was necessary: the message broker took 321 lines and our "overview" another 107 lines. To sum up: although only a fraction of Racket's BSL plus the *universe* teachpack was used, we were able to come up with a very short working solution.

After the two-day course students reported that they learned very fast and were suprised, they could achieve some results with only little knowledge in programming. We hoped that omitting authentication and encryption would encourage some of the students to hack or at least play with some of the other RaspberryPis. But that did not happen. Maybe they should be given more time to experiment and to think about possible shortcomings.

In a future course we are going to spend more time on the aspects of distributed computing, e.g. in terms of the *universe* teachpack: connecting *world* programs with a *universe.* This should broaden the understanding of the IoT and help the students to implement some of these techniques in their professional life.

ACKNOWLEDGMENTS

We would like to thank two anonymous reviewers for their helpful advice and Gerd Manthei of *StudiumPlus* at Technische Hochschule Mittelhessen who made this course possible.

REFERENCES

[1] Matthias Felleisen. 2015. Growing a Programmer. (8 September 2015). http://www.ccs.neu.edu/home/matthias/Thoughts/Growing_a_Programmer.html
[2] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. 2014. *How to Design Programs* (2nd ed.). MIT Press, Cambridge MA. http://www.ccs.neu.edu/home/matthias/HtDP2e/
[3] National Semiconductor 2001. *LM75: Digital Temperatur Sensor and Thermal watchdog with Two-Wire Interface.* National Semiconductor. http://esd.cs.ucr.edu/labs/temperature/LM75.pdf

Interactive Functional Medical Image Analysis

A demonstration using Racket and the *vigracket* library to detect sickle-cell anaemia

Benjamin Seppke
University of Hamburg
Dept. Informatics
Vogt-Kölln-Str. 30
Hamburg, Germany 22527
seppke@informatik.uni-hamburg.de

Leonie Dreschler-Fischer
University of Hamburg
Dept. Informatics
Vogt-Kölln-Str. 30
Hamburg, Germany 22527
dreschler@informatik.uni-hamburg.de

ABSTRACT

This article demonstrates the functional application of Computer Vision methods by means of a prototypical process chain. For this demo, we have selected the application area of medical image analysis, in detail the classification of blood cells using microscopic images. We further focus on the task of detecting abnormal sickle-shaped red blood cells, which are an indicator for a relatively common disease in countries around the equator: the so-called sickle-cell anaemia. From the functional languages, we have chosen Racket and the *vigracket* Computer Vision library [3]. Although this demo just scratches the surface of medical image processing, it provides a good motivation and starting point.

CCS CONCEPTS

•Computing methodologies →Image segmentation; •Software and its engineering →Functional languages; •Applied computing →*Bioinformatics;*

KEYWORDS

Functional Programming, Racket, Medical Image Processing, Computer Vision, Language Interoperability

ACM Reference format:
Benjamin Seppke and Leonie Dreschler-Fischer. 2017. Interactive Functional Medical Image Analysis. In *Proceedings of European Lisp Symposium 2017, Brussels, Belgium, April 2017 (ELS'2017),* 2 pages.

1 INTRODUCTION

Functional programming and functional languages have a long tradition in research and teaching at the Department of informatics at the University of Hamburg. Since 8 years, we successfully combine the Computer Vision library *vigra* [1] with functional programming languages. As an example in [4] we have presented how well Computer Vision approaches may be introduced into the Racket programming language. To reach a broader range of users, it shall not be unmentioned that the used C-wrapper library in platform-independent and has been tested and proven to work under Windows, Mac OS X and Linux.

After the latest optimisations, the Computer Vision wrapper libraries are even more powerful with respect to performance and image segmentation tasks [2]. Functional languages natively offer interactive development cycles, generic modelling and probably the most powerful garbage collectors.

ELS'2017, Brussels, Belgium 2017.

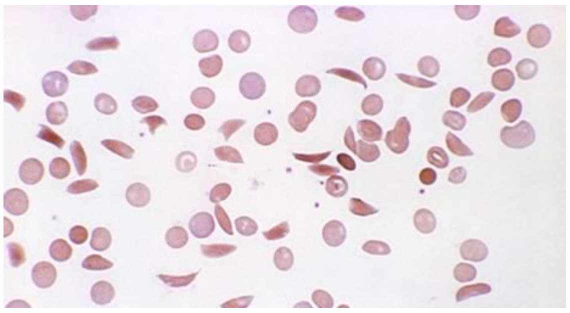


Figure 1: Blood sample of a patient with sickle-cell anaemia. The sickle-shaped cells occur among the regular, circle-shaped cells. (©Getty Images/Photoresearchers)

2 DEMONSTRATION

Sickle-cell anaemia or sickle-cell disease is genetic disease occurring in many parts of Africa and other countries. Besides its negative effects, there also seems to be a protective effect against Malaria. A comprehensive report on the sickle-cell disease and on the research progress can be found in [5]. We have chosen the detection of the sickle-cell disease for this demonstration as it requires shape detection and description of red blood cells on microscopy images. As an example, Figure 1 shows a typical image of a patient's red blood cells. For this demonstration we will distinguish between the main working phases of the process chain in logical order:

2.1 Loading images

Before loading images, it might be useful to define a working folder. By changing this folder, we can easily adapt our demo to different users or other folders. The image is then be loaded by *vigracket* in a three element list containing arrays of foreign memory for the red, green and blue channel.

```
(require vigracket)

(define dir (current-directory))
(define img (loadimage (build-path dir "cells.jpg")))
```

2.2 Preprocessing

The first step for the analysis of the different imaged blood cells is the division of the image contents into foreground (the cells) and background. This can e.g. be achieved by applying a threshold to one or more channels of the image.

Table 1: Semantics of the columns for RGB region-wise feature extraction. Each row corresponds to one segment.

Columns	Features
0	segment size
1, 2	upper left x and y-coordinates of segment
3, 4	lower right x and y-coordinates of segment
5, 6	mean x and y-coordinates of segment
...	other statistics of segment

The threshold is then used to pre-classify the pixels of the image into foreground (grey value = 255) and background pixels (grey value = 0). Here, we have chosen the red-channel and a fixed threshold of 222 to generate the mask, described above. After the thresholding we apply a morphologic opening filter to suppress the influence of smaller artefacts:

```
(define mask (image-map (λ(x) (if (< x 222.0) 255.0 0.0))
                        (image->red img)))
(define omask (openingimage mask 1.0))
```

2.3 Divison in segments

To analyse the single cells' properties, we have to divide the foreground into corresponding segments. If we define a segment as a connected component of (masked) pixels, we can assign unique labels for each segment. The `#t` tells the function to use eight-pixel connectivity for the component detection and `0.0` denotes the background value:

```
(define labels  (labelimage omask #t 0.0))
```

Instead of 0 and 255, this function assigns increasing values from 1 to the number of connected components found for each foreground classified pixel.

2.4 Segment analysis and classification

To decide whether a segment represents a sickle-cell or an circle-shaped cell, we need to get statistics, also called features, for each segment. This process can be automised by vigracket's latest functional extensions. We use the `extractfeatures` method to derive RGB-based statistics using the image and the label image.

```
(define stats (extractfeatures img labels))
```

The result, `stats`, is also an image, but with slightly changed semantics. Each row represents on region with its extracted features. The semantic with respect to the columns is shown in Table 1. For this demonstration, we find it sufficient to check whether the dimensions of each segment are roughly circle-like. We further use the aspect ratio of each region's bounding box plus one threshold, to accept a circle-like structure.

```
(define (circle-like? segment threshold)
    (let* [(width  (- (image-ref stats 3 segment 0)
                      (image-ref stats 1 segment 0)))
           (height (- (image-ref stats 4 segment 0)
                      (image-ref stats 2 segment 0)))
           (a (max width height))
           (b (min width height))]
       (< (/ a b) threshold)))
```

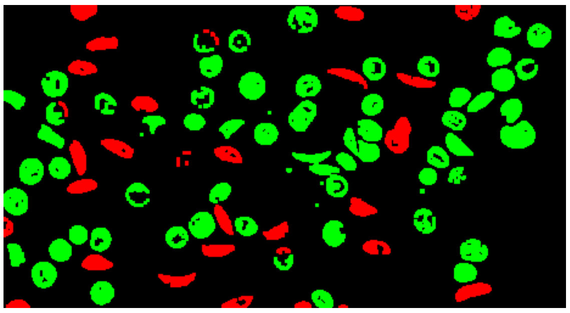


Figure 2: Classification result of Figure 1 using the demonstrated approach. Red: classified sickle-cells, green: classified circle-like cells, black: background.

2.5 Extraction and usage of results

Using the `circle-like?` function, we are now able to classify each region. Since the classification is a binary decision, we may simply filter the list of all segments:

```
(define maxlabel (image-reduce max 0.0 labels))
(define label-ids (build-list maxlabel values))
(define filtered-ids (filter (curryr circle-like? 4/3)
                             label-ids))
```

Now `filtered-ids` contains the ids of all segments which are circle-like (aspect ratio below 4/3) and thus not correspond to sickle-cells. This list can be used to quantify the ratio of different cell-types, to mark them in the image (see Figure 2) or to derive special statistics like color for these cells. Although the resulting classification leaves still place for improvements.

3 CONCLUSIONS

We demonstrated the use of interactive functional Computer Vision in a medical context. Although this demonstration has been performed using Racket and the *vigracket* library, it might also be performed by means of Common Lisp and the *vigracl* library.

Due to the limitations of this demo, the quality of the results is limited, too. However, this should not be seen as a comprehensive and best-possible segmentation and classification approach for the selected application area. Instead, it should be a motivation for all readers to utilise the power and simplicity of functional programming languages and powerful libraries for interdisciplinary areas of research. Although imperative languages are very popular at these fields at the moment, many tasks can be solved using functional languages while benefitting from their advantages, too.

REFERENCES

[1] Ullrich Köthe. 2017. The VIGRA homepage. Retrieved January 30, 2017. (2017). http://ukoethe.github.io/vigra/
[2] Benjamin Seppke. 2016. *Near-Realtime Computer Vision with Racket and the Kinect sensor.* Technical Report. University of Hamburg, Dept. Informatics.
[3] Benjamin Seppke. 2017. The vigracket homepage. Retrieved January 30, 2017. (2017). https://github.com/bseppke/vigracket
[4] Benjamin Seppke and Leonie Dreschler-Fischer. 2015. Efficient Applicative Programming Environments for Computer Vision Applications: Integration and Use of the VIGRA Library in Racket. In *Proceedings of the 8th European Lisp Symposium.*
[5] Graham R. Serjeant. 2010. One hundred years of sickle cell disease. *British Journal of Haematology* 151, 5 (2010), 425–429. DOI: http://dx.doi.org/10.1111/j.1365-2141.2010.08419.x

Session IV: Applications

Parallelizing Femlisp

Marco Heisig
Chair for Applied Mathematics 3
FAU Erlangen-Nürnberg
Cauerstraße 11
91058 Erlangen
marco.heisig@fau.de

Dr. Nicolas Neuss
Chair for Applied Mathematics 3
FAU Erlangen-Nürnberg
Cauerstraße 11
91058 Erlangen
neuss@math.fau.de

ABSTRACT

We report on the parallelization of the library FEMLISP [4], which is a Common Lisp framework for solving partial differential equations using the finite element method.

CCS Concepts

•**Software and its engineering** → **Software libraries and repositories;** •**Applied computing** → *Mathematics and statistics;*

Keywords

Parallelization, MPI, distributed computing, partial differential equations

1. INTRODUCTION

Finite elements (FE) are a particularly successful method for solving partial differential equations, which, in turn, model many important everyday problems. Since the discretization of the three-dimensional continuum leads to very large discrete problems, the parallel solution of these problems is a necessity.

The library FEMLISP [4] is a framework written completely in Common Lisp (CL). This gives the user all the benefits of CL: for example, compact and flexible code as well as interactivity. However, until recently, FEMLISP was a serial program, which was a major impediment for many applications. This only changed in 2016.

In this contribution, we report on some aspects of how we parallelized FEMLISP. In Section 2, we describe the library DDO, on which FEMLISP parallelization is based. In Section 3, we sketch the most important steps of the parallelization process and, in Section 3.1, we give some numerical results, showing that our efforts have been successful.

2. DYNAMIC DISTRIBUTED OBJECTS

Parallel computing in Common Lisp is usually based on the portable libraries BORDEAUX-THREADS and LPARALLEL for shared-memory parallelization, and CL-MPI and LFARM for distributed-memory parallelization. Taking these tools as a basis, we enhance them with a library called *Dynamic Distributed Objects* (DDO).

This library works as follows:

- The objects to be shared between different Lisp images are modified by inheriting from the mixin class DISTRIBUTED-OBJECT. Note that only a relatively small number of interface objects have to be "identified" in this way.
- Synchronization of these so-called *distributed objects* occurs only at certain synchronization points so that the communication overhead remains reasonably small.
- The basic administrative data structure is a relation implemented with the help of red-black binary trees connecting three items: an index ("local index") for the distributed object on this processor, the neighboring processor, and the index of the corresponding distributed object on that neighboring processor.
- The *local index* refers to its associated distributed object via a weak hash table. This implies that an entry in this table does *not* prevent the garbage collector from removing the object if it is not needed locally any more. However, as soon as a distributed object has been garbage-collected, a finalizer is called which ensures that the fact that this object has vanished is communicated to the neighboring processors.

3. PARALLELIZATION OF FEMLISP

One goal when parallelizing FEMLISP was to keep the serial (or rather shared-memory parallel) code working as before and to introduce the treatment of the interfaces as an augmented library. This augmented library contains the following:

1. Mesh generation can generate distributed coarse meshes (i.e. meshes where only part of the mesh is located on each processor with correctly identified interface parts).
2. Mesh refinement ensures that the refined interfaces are again identified correctly.
3. Discretization is automatically parallelized, because we decided to work with so-called *inconsistent* matrices A_i and vectors f_i (see [1]) and consequently no information has to be exchanged to obtain the global stiffness

10th European Lisp Symposium 2017, Brussels, Belgium
© 2017 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

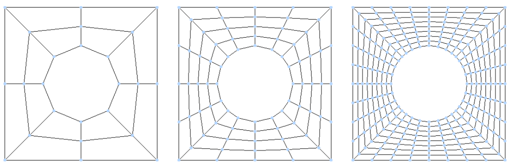


Figure 1: Three refinement levels in 2D.

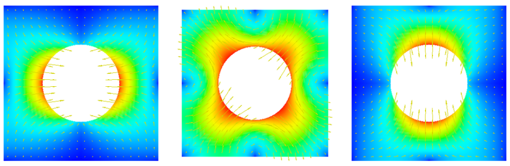


Figure 2: Three of four vector solutions in 2D.

matrix (to make an inconsistent vector or matrix consistent, the interface parts would have to be added up).

4. The result of this discretization step is a very large sparse distributed linear system with unknowns associated with certain geometric entities in the distributed mesh. This sparse linear system has the important property that its matrix graph only couples unknowns which are geometrically not too far apart. Nevertheless, solving it efficiently is a non-trivial task on which much research has been carried out in recent decades. In particular, it is now well known that an optimal solver must be hierarchical and therefore has to contain both local and non-local communication in an appropriate way. For our calculations here, we chose a BPX-preconditioned CG algorithm which is well known for both optimality and relatively easy parallelization (see e.g. [2]).

3.1 Numerical results

As a model problem, we compute the effective elastic behavior of a periodically perforated composite. Such materials are often encountered in modern technology, and exhibit large-scale behavior which is different from that of each of the components and which depends both on the properties of the components as well as on their geometric arrangement.

The calculation of the effective behavior can be done by solving d^d vector PDE problems on a d-dimensional representative cell. In the simple case of a ball-like perforation, this representative cell is a unit cube with a ball-like hole (or, in the $d = 2$ case, a unit square with a circle-like hole). This calculation is not a real challenge in the two-dimensional case, and yields the effective coefficient with high accuracy in only a few minutes. For example, using the meshes from Figure 1, choosing a suitable discretization, and solving the discrete problem with the above-mentioned BPX scheme, leads to the solutions shown in Fig. 2.

In the case $d = 3$, however, solving the same problem is much more challenging and therefore an interesting opportunity for testing the quality of our parallelization.

In the following, we report some results from [3]. Our architecture of choice is a compute server at our institute featuring 256 GB RAM and a NUMA architecture with an AMD Opteron 6180 SE CPU (4 sockets with 12 cores each

Cells	Unknowns	Matrix entries	$\hat{A}_{11}^{11}$
48	192.024K	18.5712M	2.6231458888
384	1.51991M	148.704M	2.6231424485
3072	12.0165M	1.18752G	2.6231424309

Table 1: Approximation when refining the mesh (K,M,G are abbreviations for $10^3, 10^6, 10^9$).

	Sockets/MPI workers			
Cells	1	2	3	4
48	42	25 (1.7)	20 (2.1)	17 (2.5)
384	300	170 (1.8)	130 (2.3)	105 (2.9)
3072	1985	960 (2.1)	685 (2.9)	565 (3.5)

Table 2: Execution times in seconds and speedups (in parentheses)

and no hyperthreading).

Solving this problem results in Table 1, from which we can see that we can already obtain a high accuracy of the effective coefficient on a mesh with 384 cells (which correspond to about 1.5 million unknowns and 150 million matrix entries). This value is finally checked using a calculation on a refined mesh (which corresponds to about 12 million unknowns and one billion matrix entries).

The execution times needed to obtain these results can be found in Table 2. We can clearly see that the speedup is almost optimal as soon as the problem becomes large enough (so that the communication matters less). Note also that these results only show the improvements due to our distributed-memory parallelization, and that every MPI worker process already employs 12 OS threads. Indeed, a completely serial calculation would take more than four times as long than the results shown in the first column of Table 2 (see [3], where we study the parallelization of FEMLISP in more detail starting from the serial code).

4. ACKNOWLEDGMENTS

We would like to thank the KONWIHR project (http://www.konwihr.uni-erlangen.de/about-konwihr.shtml) for their support as well as Eberhard Bänsch and Balthasar Reuter for helpful discussions and suggestions.

5. REFERENCES

[1] P. Bastian, K. Birken, K. Johannsen, S. Lang, N. Neuss, H. Rentz-Reichert, and C. Wieners. UG – a flexible software toolbox for solving partial differential equations. *Comput. Visual. Sci.*, 1:27–40, 1997.

[2] J. H. Bramble and X. Zhang. The analysis of multigrid methods. In P. G. Ciarlet and J.-L. Lions, editors, *Handbook of Numerical Analysis*, volume 7, pages 173–415. North–Holland, Amsterdam, 2000.

[3] M. Heisig and N. Neuss. Making a Common Lisp Finite Element library high-performing — a case study. *(submitted)*, 2017.

[4] N. Neuss. `http://www.femlisp.org`.

Tutorials

General Game Playing in Common Lisp

Steve Losh
Reykjavík University
Menntavegi 1
Reykjavík, Iceland
steve@stevelosh.com

ABSTRACT

Common Lisp has a rich history in the field of artificial intelligence. One subfield of AI that is currently an area of active research is general game playing, which focuses on writing players that can learn to play *any* game intelligently given only its rules. We provide a short introduction to the field of general game playing and present `cl-ggp`, a framework for writing general game players in Common Lisp. We then implement a simple general game player in about forty lines of code to show the framework in action.

CCS CONCEPTS

•Computing methodologies →Game tree search; *Logic programming and answer set programming;* **•Software and its engineering →Application specific development environments;**

KEYWORDS

General game playing, Common Lisp

1 OVERVIEW OF GENERAL GAME PLAYING

Traditional game AI research has focused on creating agents for individual games, such as Deep Blue[2] for Chess and AlphaGo[7] for Go. The field of general game playing has a higher-level goal: creating agents capable of playing *any* game intelligently given only its rules. Instead of programmers using domain knowledge to create a new AI for each game from scratch, a general game player must be able to receive a set of rules and be ready to play strategically almost immediately[1].

While playing any game is a laudable goal, in practice there are some restrictions imposed on the games for practicality. Most existing research in the field of general game playing deals with finite, simultaneous-move, complete-information games, and the International General Game Playing Competition focuses on these types of games[3]. Relaxing these restrictions (e.g. playing games with incomplete information like Texas hold'em) is an area of active research.

[1]Typically players are given anywhere from thirty seconds to several minutes of preparation time after receiving the rules for any preprocessing they might want to perform.

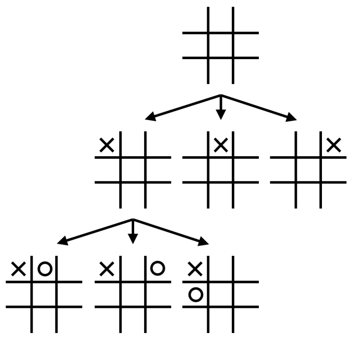


Figure 1: A partial game tree for tic-tac-toe.

Games of this type can be thought of as a tree of states[2], the leaves of which are terminal states where the game has ended. Figure 1 shows an example of a partial game tree for the game of tic-tac-toe.

2 OVERVIEW OF GAME DESCRIPTION LANGUAGE

Pitting general game players against each other in competitions requires a standardized way to describe game rules. A simple logic programming language called Game Description Language (GDL) was created for this purpose[3]. GDL is a variant of Datalog with several extensions and an s-expression-based syntax that Lisp users will appreciate. We present a basic overview here, but readers who want more details should consult the full specification[4].

A game state in GDL is described by a series of logical facts of the form `(true . . . )`. Listing 1 shows the state corresponding to the lower-left node in the example game tree.

The definition of a game in GDL consists of logical facts and rules. The roles and initial state of the game are given as facts as shown in Listing 2.

Other aspects of a game are described as logical rules of the form `(<= <head> <body. . . >)` as shown in Listing 3. Logic variables are named with a `?` prefix.

In lines `1-11` we define `row`, `line`, and `open` predicates which will be useful in defining the rest of the game rules. The definitions for `column` and `diagonal` have been omitted to save space.

[2]It may be possible to reach an identical game state from multiple paths, so the "game tree" would more properly be called the "game directed acyclic graph".
[3]GDL was intended to be a heavily restricted language to make it easier to reason about, but unfortunately turned out to be Turing complete[5]. In practice this is usually not a problem.

Listing 1: An example state for a game of tic-tac-toe.

```
(true (control x))
(true (cell 1 1 x))
(true (cell 1 2 o))
(true (cell 1 3 blank))
(true (cell 2 1 blank))
(true (cell 2 2 blank))
(true (cell 2 3 blank))
(true (cell 3 1 blank))
(true (cell 3 2 blank))
(true (cell 3 3 blank))
```

Listing 2: Role and initial state definition for tic-tac-toe.

```
(role x)
(role o)
(init (control x))
(init (cell 1 1 blank))
(init (cell 1 2 blank))
(init (cell 1 3 blank))
(init (cell 2 1 blank))
(init (cell 2 2 blank))
(init (cell 2 3 blank))
(init (cell 3 1 blank))
(init (cell 3 2 blank))
(init (cell 3 3 blank))
```

The `terminal` predicate (lines 13-17) determines whether a particular state is terminal. A game of tic-tac-toe terminates when one player has completed a line or there are no open spaces left on the board.

For all terminal states the `(goal <player> <value>)` predicate (lines 19-31) must describe a single goal value for every role. We give a player a score of 100 if they win by marking a line, 0 if their opponent wins, or 50 for a draw.

The `(legal <player> <move>)` predicate (lines 33-41) gives the legal moves for each player at any given state. All players must have at least one move in every non-terminal state. In our definition of tic-tac-toe there are two types of moves: when a player has control they must mark a blank cell, and when the other player has control they can only perform a “no-op” move.

When all players have made a move they are added to the state of the game as `(does <player> <move>)` facts. The successor state can then be computed from the `(next <fact>)` predicate (lines 43-60).

Tic-tac-toe has two kinds of information that must be tracked for each state: the contents of the board and which player has control. A new state is computed from the current state in four parts:

- Control changes hands every turn (lines 45-46).
- Cells that are already marked stay marked (lines 48-50).
- If a player uses their move to mark a cell, that cell is marked in the next state (lines 52-54).
- All other blank cells remain blank (lines 56-60).

There are several other restrictions on game definitions that we will not cover here. Interested readers should consult the full GDL specification[4] to learn all the details of playing games written in GDL.

Listing 3: Rules for tic-tac-toe.

```
 1 (<= (row ?n ?mark)
 2   (true (cell ?n 1 ?mark))
 3   (true (cell ?n 2 ?mark))
 4   (true (cell ?n 3 ?mark)))
 5
 6 (<= (line ?mark) (row ?n ?mark))
 7 (<= (line ?mark) (column ?n ?mark))
 8 (<= (line ?mark) (diagonal ?n ?mark))
 9
10 (<= open
11   (true (cell ?r ?c blank)))
12
13 ;;;; Terminal
14
15 (<= terminal (line x))
16 (<= terminal (line o))
17 (<= terminal (not open))
18
19 ;;;; Goal Values
20
21 (<= (goal ?player 100)
22   (line ?player))
23
24 (<= (goal ?player 0)
25   (line ?other)
26   (distinct ?player ?other))
27
28 (<= (goal ?player 50)
29   (not (line x))
30   (not (line o))
31   (not open))
32
33 ;;;; Legal Moves
34
35 (<= (legal ?player (mark ?row ?col))
36   (true (cell ?row ?col blank))
37   (true (control ?player)))
38
39 (<= (legal ?player noop)
40   (true (control ?other))
41   (distinct ?player ?other))
42
43 ;;;; State Transitions
44
45 (<= (next (control x)) (true (control o)))
46 (<= (next (control o)) (true (control x)))
47
48 (<= (next (cell ?row ?col ?player))
49   (true (cell ?row ?col ?player))
50   (distinct ?player blank))
51
52 (<= (next (cell ?row ?col ?player))
53   (true (cell ?row ?col blank))
54   (does ?player (mark ?row ?col)))
55
56 (<= (next (cell ?row ?col blank))
57   (true (cell ?row ?col blank))
58   (does ?player (mark ?x ?y))
59   (or (distinct ?row ?x)
60       (distinct ?col ?y)))
```

3 WRITING GENERAL GAME PLAYERS IN COMMON LISP

To write a general game player capable of competing with other players we can split the implementation into three distinct parts:

(1) The HTTP-based GGP network protocol, for connecting to and communicating with a central game server.
(2) Parsing GDL game descriptions and reasoning about states to determine legal moves, terminality, goal values, etc.
(3) An AI to search the game tree and find moves that will lead to a win for the player.

`cl-ggp` is a library written in Common Lisp to handle the tedious parts of this process. It is installable with Quicklisp[4]. The code is available as a Mercurial[5] or Git[6] repository[7], and is released under the MIT license. We present a short guide to its usage here, but for a much more thorough introduction readers should refer to the documentation[8].

The library contains two separate ASDF systems: `cl-ggp` and `cl-ggp.reasoner`.

3.1 The `cl-ggp` System

The main `cl-ggp` system handles the GGP network protocol and manages the basic flow of games. It takes a simple object-oriented approach similar to that of the `ggp-base` package[9] for the JVM.

To create a general game player users define a CLOS subclass of the `ggp-player` class and implement four methods to handle the main flow of the game:

`(player-start-game <player> <rules> <role> <deadline>)` is called by the framework when a new game begins. Each player will only ever be running a single game at a time. The method receives as arguments the GDL description of the game (a list of s-expressions), the role it has been assigned, and the time limit for any initial processing it may wish to do[10].

`(player-update-game <player> <moves>)` is called by the framework at the beginning of each turn. It receives as an argument the list of moves done by players in the previous turn (except for the first turn, in which `moves` will be `nil`). Players will typically use this method to compute the new state of the game.

`(player-select-move <player> <deadline>)` is called by the framework directly after `player-update-game`, and must return a move to perform before the given `deadline`.

`(player-stop-game <player>)` is called by the framework when a game has ended. Players can use it to trigger any cleanup they might require.

Once all the necessary methods have been defined, an instance can be created with `(make-instance <class> :name <name> :port <port number>)` and given to `start-player` to begin listening on the given port. `stop-player` can be used to stop a player and relinquish the port.

[4]As of the time of this writing it is not in a Quicklisp dist, so you'll need to use Quicklisp's local project support.
[5]https://bitbucket.org/sjl/cl-ggp/
[6]https://github.com/sjl/cl-ggp/
[7]The most recent commit hashes at the time of this writing are `abdfc9d` (Mercurial) and `749651e` (Git).
[8]https://sjl.bitbucket.io/cl-ggp/
[9]https://github.com/ggp-org/ggp-base/
[10]Symbols in rules are interned in the `ggp-rules` package to avoid polluting other namespaces.

3.2 The `cl-ggp.reasoner` System

The `cl-ggp.reasoner` system implements a basic Prolog-based reasoning system to use as a starting point if desired. Under the hood it uses the Temperance[11] logic programming library to compute and reason about states. Temperance is an implementation of the Warren Abstract Machine[8][12] in pure Common Lisp.

The included reasoner is intended to be a simple starting point for experimentation. Users who want better performance or want more access to the reasoning process are encouraged to write their own reasoning systems.

The reasoner API consists of six functions:

`(make-reasoner <rules>)` Creates and returns a reasoner object for reasoning about the given GDL rules.

`(initial-state <reasoner>)` Returns the initial state of the game as described by the `(init ...)` facts in the GDL.

`(next-state <reasoner> <state> <moves>)` Returns the successor state of the given state, assuming the given moves were taken.

`(terminalp <reasoner> <state>)` Returns `t` if the given state is terminal, `nil` otherwise.

`(legal-moves-for <reasoner> <state> <role>)` Returns a list of all legal moves for the given role in the given state.

`(goal-value-for <reasoner> <state> <role>)` Returns the goal value for the given role in the given state.

4 IMPLEMENTING A RANDOM PLAYER

The basic framework and included reasoner are enough to write a simple general game player that can play any GDL game legally (though not particularly intelligently). Such a player is shown in Listing 4.

We first define a subclass of `ggp-player` called `random-player` with slots for holding the information it will need to play a game.

In `player-start-game` we create a reasoner with the rules passed along by the framework. We store the reasoner and the assigned role in the player instance.

In `player-update-game` we compute the current state of the game. If `moves` is `nil` this is the first turn in the game, and so we simply request the initial state from the reasoner. Otherwise we compute the next state from the current one and the moves performed. We store the current state in the player for later use.

In `player-select-move` we compute the legal moves our role can perform in the current state and choose one at random. This ensures we always play legally, but is not usually a very effective strategy. A more intelligent player would use the given time to search the game tree and try to find which moves lead to high goal values for its role.

Finally in `player-stop-game` we clear out the slots of the player so the contents can be garbage collected.

[11]https://bitbucket.org/sjl/temperance/
[12]A virtual machine designed for compiling and running Prolog code.

Listing 4: A simple general game player capable of playing any GDL game legally.

```lisp
(defclass random-player (ggp-player)
  ((role          :accessor p-role)
   (current-state :accessor p-current-state)
   (reasoner      :accessor p-reasoner)))

(defmethod player-start-game
    ((player random-player) rules role deadline)
  (setf (p-role player) role
        (p-reasoner player) (make-reasoner rules)))

(defmethod player-update-game
    ((player random-player) moves)
  (setf (p-current-state player)
        (if (null moves)
          (initial-state (p-reasoner player))
          (next-state (p-reasoner player)
                      (p-current-state player)
                      moves))))

(defmethod player-select-move
    ((player random-player) deadline)
  (let ((moves (legal-moves-for
                 (p-reasoner player)
                 (p-current-state player)
                 (p-role player))))
    (nth (random (length moves)) moves)))

(defmethod player-stop-game
    ((player random-player))
  (setf (p-current-state player) nil
        (p-reasoner player) nil
        (p-role player) nil))

(defvar *random-player*
  (make-instance 'random-player
                 :name "RandomPlayer"
                 :port 4000))

(start-player *random-player*)
```

Once the four required methods are defined we create an instance of the player and start it listening on the given port.

5 IMPROVING THE PLAYER

Classical game tree search strategies like minimax can be used to search the game tree for promising moves, and for small games they produce acceptable results. However, because a general game player must be able to play *any* game it cannot have a heuristic function hard-coded into it[13], which makes many traditional search techniques much more difficult.

One strategy that has recently gained popularity is simulation-based search, such as Monte-Carlo Tree Search[1]. MCTS works by running many random playouts from a state to determine its approximate value without expanding the entire game tree, operating under the assumption that a state where random playouts tend to produce good results is a good state to be in.

MCTS relies on running many random playouts to provide data, so the faster playouts can be run the better its results will be. In practice the bottleneck in running random simulations is reasoning about and computing states, so another improvement would be to optimize the reasoner or write an entirely new one, possibly not using Prolog-style reasoning at all (e.g. a propositional network[6]).

[13]Any heuristic that works for one game could potentially backfire in a different game.

6 CONCLUSION

General game playing involves the creation of AI agents capable of playing any game intelligently given only its rules. After giving an overview of the field we presented `cl-ggp`, a framework for writing general game players in Common Lisp, and showed how to create a simple player with it. We hope this framework will reduce the friction involved in creating players and encourage more people to experiment with Common Lisp as a platform for research in general game playing.

ACKNOWLEDGMENTS

We would like to thank Stephan Schiffel for providing feedback on a draft of this paper.

REFERENCES

[1] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intellig. and AI in Games ()* 4, 1 (2012), 1–43.
[2] Murray Campbell, A Joseph Hoane Jr., and Feng-hsiung Hsu. 2002. Deep Blue. *Artificial Intelligence* 134, 1-2 (Jan. 2002), 57–83.
[3] Michael R Genesereth and Yngvi Björnsson. 2013. The International General Game Playing Competition. *AI Magazine* (2013).
[4] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. 2008. *General Game Playing: Game Description Language Specification.* Technical Report.
[5] Abdallah Saffidine. 2014. The Game Description Language Is Turing Complete. *IEEE Transactions on Computational Intelligence and AI in Games* 6, 4 (2014), 320–324.
[6] Eric Schkufza, Nathaniel Love, and Michael R Genesereth. 2008. Propositional Automata and Cell Automata - Representational Frameworks for Discrete Dynamic Systems. *Australasian Conference on Artificial Intelligence* (2008).
[7] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529, 7587 (2016), 484–489.
[8] David H D Warren. 1983. An Abstract Prolog Instruction Set. (Oct. 1983), 1–34.

Session V: Languages and meta-languages (1)

Fast, Maintainable, and Portable Sequence Functions [*]

Irène Durand
Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
irene.durand@u-bordeaux.fr
robert.strandh@u-bordeaux.fr

ABSTRACT

The Common Lisp sequence functions are challenging to implement because of the numerous cases that need to be taken into account according to the keyword arguments given and the type of the sequence argument, including the element type when the sequence is a vector.

For the resulting code to be fast, the different cases need to be handled separately, but doing so may make the code hard to understand and maintain. Writing tests that cover all cases may also be difficult.

In this paper, we present a technique that relies on a good compiler to optimize each separate case according to the information available to it with respect to types and values of keyword arguments. Our technique uses a few custom macros that duplicate a general implementation of the body of a sequence function. The compiler then specializes that body in different ways for each copy.

CCS Concepts

•Software and its engineering → Abstraction, modeling and modularity; Software performance; Compilers;

Keywords

Common Lisp, Compiler optimization, Portability, Maintainability

1. INTRODUCTION

The Common Lisp [1] sequence functions are challenging to implement for several reasons:

- They take several keyword parameters that modify the behavior in different ways. Several special cases must therefore be taken into account according to the value of these keyword parameters.
- In order for performance to be acceptable, different variations may have to be implemented according to the type of the sequence to be traversed.
- When the sequence is a vector, it may be necessary to specialize the implementation according to the element type of the vector, and according to whether the vector is a *simple array* or not.

For reasons of maintainability, it is advantageous to create a small number of versions, each one containing a single loop over the relevant elements. In each iteration of the loop, tests would determine the exact action based on current values of keyword arguments. In the case of a vector, the general array accessor `aref` would be used to access the elements.

On the other hand, for reasons of performance, it is preferable to create a large number of different versions of each function, each version being specialized according to the exact values of the keyword arguments given. In the case of a vector, it is also advantageous to have versions specialized to the available element types provided by the implementation. However, in this case, maintenance is problematic, because each version has to be maintained and tested separately.

A compromise used by some implementations is to use the Common Lisp macro system to abstract some of the specialization parameters as macro arguments. With this technique, a special version is created by a call to some general macro, providing different cases for keyword parameters, element types, test functions, etc. We find that this technique results in code that is extremely hard to understand, and therefore to be perceived as correct by maintainers.

In this paper, we present a different technique. We use the Common Lisp macro system, but not in order to create macros that, when called, create special versions of a sequence function. Instead, our technique makes it possible to write very few versions of each sequence function, thus keeping a high degree of maintainability. Most of our macros have no apparent role in our functions, so do not require the maintainer to understand them. Instead, they serve the sole purpose of allowing the compiler to generate efficient code.

Our technique was developed as part of the SICL project[1] which aims to supply high quality implementation independent code for a large part of the Common Lisp standard.

2. PREVIOUS WORK

[*]The code presented in this article is available at https://github.com/idurand/cl-portable-find.git

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

10th ELS April 3–4, 2017, Brussels, Belgium

Most implementations process list elements in reverse order when `:from-end` is true only when the specification requires it, i.e., only for the functions `count` and `reduce`.

We designed a technique [3] that allows us to always process list elements in reverse order very efficiently when `:from-end` is true. Since that paper contains an in-depth description of our technique, and in order to keep the presentation simple, in this paper, no example traverses the sequence from the end.

2.1 ECL and Clasp

The sequence functions of ECL have a similar superficial structure to ours, in that they take advantage of custom macros for managing common aspects of many functions such as the interaction between the `test` and `test-not` keyword arguments, the existence of keyword arguments `start` and `end`, etc. But these macros just provide convenient syntax for handling shared aspects of the sequence functions. They do not assist the compiler with the optimization of the body of the code.

For functions for which the Common Lisp specification allows the implementation to process elements from the beginning of the sequence even when `from-end` is *true*, ECL takes advantage of this possibility. For the `count` function applied to a list, ECL simply reverses the list before processing the elements.

The Common Lisp code base of Clasp is derived from that of ECL, and the code for the sequence functions of Clasp is the same as that of ECL.

2.2 CLISP

The essence of the code of the sequence functions of CLISP are written in C, which makes them highly dependent on that particular implementation. For that reason, in terms of previous work, CLISP is outside the scope of this paper. Our technique is still applicable to CLISP, of course, since it uses only standard Common Lisp features.

2.3 SBCL

The sequence functions of SBCL are implemented using a mixed approach.

Macros are used to create special versions for the purpose of better performance. Transformations during compilation can replace a general call to a sequence function by a call to a special version when additional information is available such as when the sequence is a specialized vector, or when some keyword argument has a particular explicit value in the call.

Macros are also used to abstract details of combinations of values of keyword arguments.

However, when little information is available at the call site, a call to the general purpose function is maintained, and no particular attempt has been made to optimize such calls. As a result, in order to obtain high performance with the SBCL sequence functions, the programmer has to supply additional explicit information about the element type (in case of a vector) and explicit keyword arguments to such calls.

2.4 Clozure Common Lisp

The sequence functions of Clozure Common Lisp are implemented according to the approach where each function has a number of special versions according to the type of the sequence and the combination of the values of the keyword arguments.

However, the code in Clozure Common Lisp contains very few attempts at optimizing performance. For example, while there is an explicit test for whether a vector to be used as a sequence is a simple array, there is no attempt to specialize according to the element type of the vector.

3. OUR TECHNIQUE

We illustrate our technique with a simplified version of the function `find`. Recall that this function searches a sequence for the first occurrence of some item passed as an argument, and that the behavior can be altered as usual with parameters for determining the comparison function, a key function to apply to each element, the direction of the search, and the interval to search.

In the begining of this section, our version is simplified in the following way:

- The only type of sequence handled is `vector`.
- The `test` function is fixed to be `eql`.
- The interval to search is the entire vector.
- The key function to apply to each element is fixed to be `identity`.
- The search is from the beginning of the vector.

In the general version of the `find` function, all these parameters must of course be taken into account, and then our technique becomes even more applicable and even more important. But the general version does not require any additional difficulty beyond what is needed for the special case, and the general case would only clutter the presentation, hence the special version which we will call `find-vector`.

Clearly, in terms of portability and maintainability, it would be desirable to implement `find-vector` like this:

```
(defun find-vector-1 (item vector)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (loop for index from 0 below (length vector)
        for element = (aref vector index)
        when (eql item element)
          return element))
```

Unfortunately, most implementations would have difficulties optimizing this version, simply because the exact action required by the function `aref` depends on the *element type* of the vector, and whether the vector is a *simple-array*. This information is clearly *loop invariant*, but most compilers do not contain adequate optimization passes in order to duplicate and specialize the loop.

To improve code layout, in what follows, we assume the following type definition:

```
(deftype simple-byte-vector ()
  `(simple-array (unsigned-byte 8)))
```

To help the compiler, one can imagine a version like this:

```
(defun find-vector-2 (item vector)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (if (typep vector 'simple-byte-vector)
      (loop for index from 0 below (length vector)
            for element = (aref vector index)
```

```
         when (eql item element)
           return element)
      (loop for index from 0 below (length vector)
            for element = (aref vector index)
            when (eql item element)
              return element)))
```

Here, we have illustrated the specialization with a non-standard element type, so that either an implementation-specific type predicate has to be used, or (as in our example) a call to typep is needed.

Whether a local declaration of the type of the vector in addition to the call to typep is required for the compiler to optimize the call to aref is of course implementation specific. Similarly, whether a special version (possibly implementation-specific) of aref is required also depends on the implementation.

Not only do we now have implementation-specific code, but we also have a maintenance problem. The loop will have to be duplicated for each sequence function, and for every specific type that the implementation can handle. This duplication requires separate tests for each case so as to guarantee as much coverage as possible. Given the number of combinations of types, plus the additional parameters we have omitted, this requirement quickly becomes unmanageable.

To solve this problem, we introduce a macro with-vector-type that abstracts the implementation-specific information and that takes care of duplicating the loop:

```
(defmacro with-vector-type (vector-var &body body)
  `(macrolet ((vref (array index)
                `(aref ,array ,index)))
     (if (typep ,vector-var 'simple-byte-vector)
         (locally (declare (type simple-byte-vector
                                 ,vector-var))
           ,@body)
         (progn
           ,@body))))
```

Here, we have introduced a new operator named vref in the form of a local macro, and that is globally defined to expand to aref. This global definition works for SBCL, but different implementations may need different expansions in different branches. For example, some implementations might need for the macro to expand a call to sbit in a branch where the vector is a simple bit vector.

We have also introduced a local declaration for exact type of the vector in the specialized branch. Each implementation must determine whether such a declaration is necessary.

Using this macro, we can now write our function find-vector like this:[2]

```
(defun find-vector-4 (item vector)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (with-vector-type vector
    (loop for index from 0 below (length vector)
          for element = (vref vector index)
          when (eql item element)
            return element)))
```

We notice a couple of essential properties of this code:

[2]In our examples, as in our real code, we frequently use a value of 0 for the safety optimize quality. However, this is the case only for auxilary functions called from protocol functions that have verified all parameters before calling such an auxiliary function. Overall, the combination is therefore still safe as seen by the application programmer.

- The exact set of available vector types in the implementation is hidden inside the macro with-vector-type, which would have a different version in different Common Lisp implementations, but there will be a single occurrence of this macro for all the sequence functions.
- The maintenance problem resulting from duplicating the loop has disappeared, because the macro with-vector-type is in charge of the duplication, making it certain that the copy is exact.

For a second example in the same spirit, consider how the keyword parameter end is handled when the sequence is a list. Again, we illustrate our technique with a simplified version of the find function.

As for the previous example, in terms of portability and maintainability, it would be desirable to implement find-list like this:

```
(defun find-list-1 (item list &key end)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (loop for index from 0
        for element in vector
        when (and (not (null end)) (>= index end))
          return nil
        when (eql item element)
          return element))
```

As with the previous example, most Common Lisp implementations would have difficulties optimizing the code, even though the test (null end) is loop invariant. We solve this problem by introducing the following macro:

```
(defmacro with-end (end-var &body body)
  `(if (null ,end-var)
       (progn ,@body)
       (progn ,@body)))
```

The code for find-list can now be written like this:

```
(defun find-list-2 (item list &key end)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (with-end end
    (loop for index from 0
          for element in vector
          when (and (not (null end)) (>= index end))
            return nil
          when (eql item element)
            return element))
```

We notice that the loop body looks the same as in the portable and maintainable version shown before, and the only difference is that the loop has been wrapped in a call to the macro with-end. A good compiler will now specialize each of the two copies of the loop introduced by the with-end macro according to the value (i.e., nil or not) of the variable end. In the first copy, the entire first when clause of the loop will be removed. In the second copy, the test in the first when clause of the loop is reduced to the comparison between index and end.

4. PROPERTIES OF OUR TECHNIQUE

4.1 Performance

We compared the performance of our technique for the find function shown in Appendix B to the performance of the find function shipped with SBCL.

We tested the performance of our technique only on SBCL because it is one of the few implementations that has a compiler that implements all the optimizations that our technique requires in order to perform well. In Section 5 we discuss what these optimizations are.

The results show a significant performance gain compared to the `find` function of SBCL. In fact, as it turns out, the SBCL sequence functions often require the programmer to declare the element type (when the sequence is a vector) in order for performance to be improved. We have not attempted to compare our technique to this case, for the simple reason that one of the advantages of our technique is precisely that no additional information is required in order for performance to be acceptable.

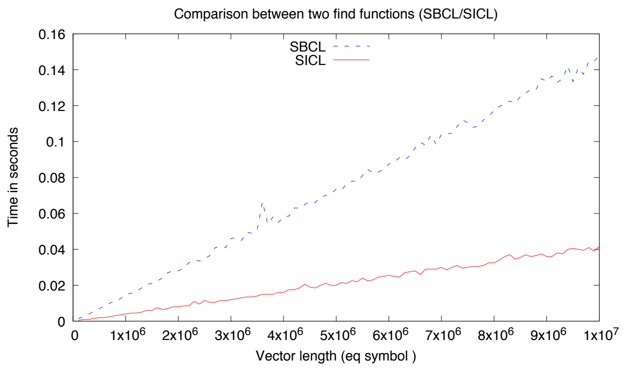
Most of our tests use relatively fast `test` functions such as `eq` or `eql`. This is a deliberate choice, as we want to compare the performance of the traversal of the sequence, and a more time-consuming test function would dominate the execution time. Because of the inherent imprecision in timing results, using more time-consuming test functions would require us to subtract two large timing results, thereby making the difference even less precise.

In this paper, we report only execution time, and no other performance parameter such as memory use or amount of allocated memory. In fact, the only situation where memory consumption is more than a small constant is when the stack is used as temporary storage when `:from-end` is true and the sequence is a list. This situation is fully covered in our paper dedicated to processing lists in reverse order [3]. Similarly, there is no allocation of memory involved in our technique. Client-supplied functions for tests and key computation may of course allocate memory.

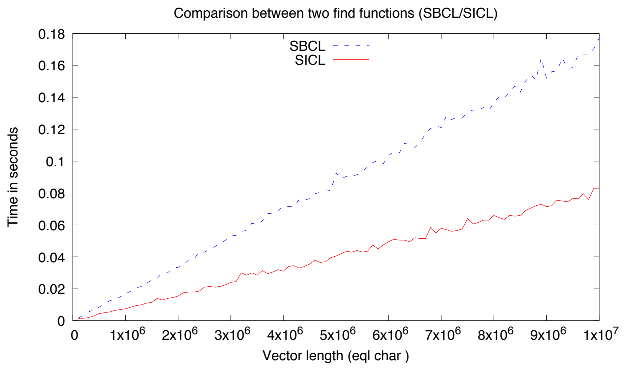
4.1.1 Results on vectors

When the sequence is a vector, the main performance consideration has to do with the element type of the vector. The parameters `start`, `end`, and `from-end` do not significantly alter the way the traversal is implemented. The `key` function may influence performance, but for the cases that we treat specially, only unspecialized vectors are concerned.

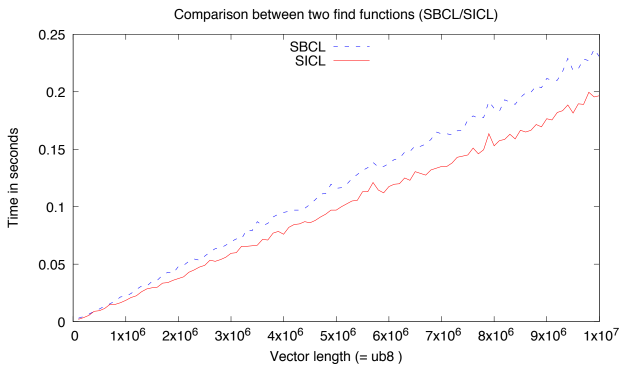
Our first test shows the performance comparison for an unspecialized vector with the `key` function being `identity` and the `test` function being `eq`. As shown in the diagram below, our function is around three times as fast as that of SBCL.



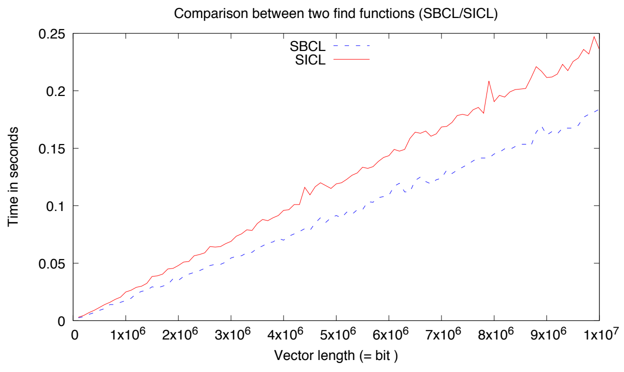
The next test uses a vector whose element type is `character`. The `key` function is `identity`, and the `test` function is `eql`. As shown in the diagram below, our function is around twice as fast as that of SBCL.



The next test shows a vector with the element type being `(unsigned-byte 8)` and the test function being `=`. In this case, our function is only around 20% faster than that of SBCL. This modest improvement can be explained by the fact that this test function is not one of the functions that we treat in a special way. An implementation that wishes better performance for this case can modify the macros to reflect this desire.



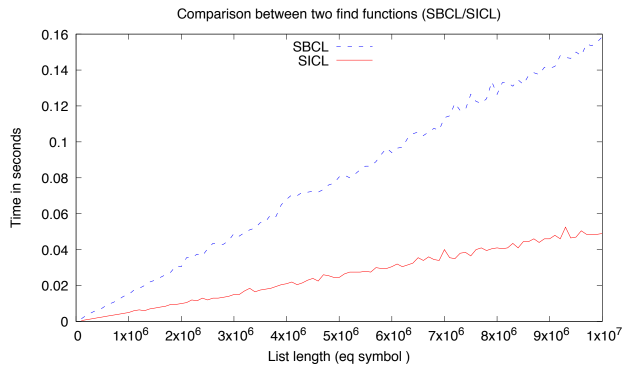
For completeness, we finally show a test for a vector with the element type being `bit`. In this case, our technique is slow, because it accesses the elements one at a time, whereas a good, native implementation of `find` would use available processor instructions to handle an entire word at a time [2].



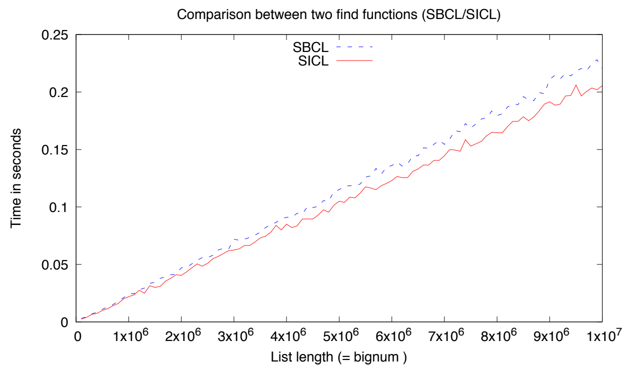
4.1.2 Results on lists

When the sequence is a list, the concept of element type is not applicable. The `key` function is important, because the sequence functions may be used for association lists. For that reason, we include a test with `car` as a key function. Also, for lists, the parameter `end` may influence the performance. In our implementation, we specialize the loops according to whether this parameter is `nil` or a number, allowing for two different specialized versions of the main traversal loop.

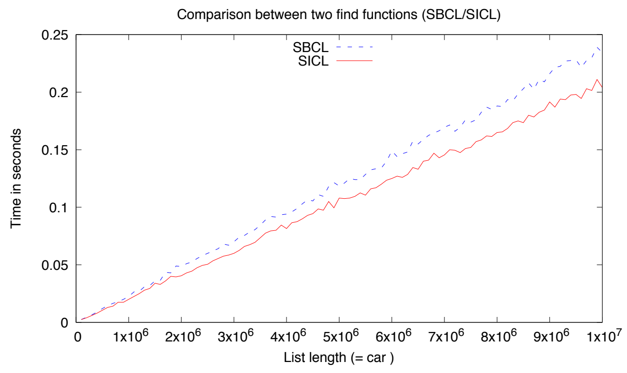
Our first test, like the first one on a vector, traverses a list of symbols. The test function is `eq`, and no `end` parameter has been given, which is equivalent to giving it the value `nil`. Again, our implementation is around three times as fast as that of SBCL.



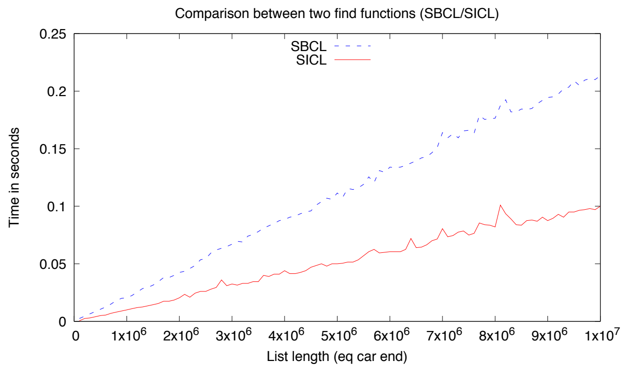
In the next test, the sequence is a list containing only bignums. The `test` function is `=`. As the diagram shows, our technique is only moderately faster than that of SBCL.



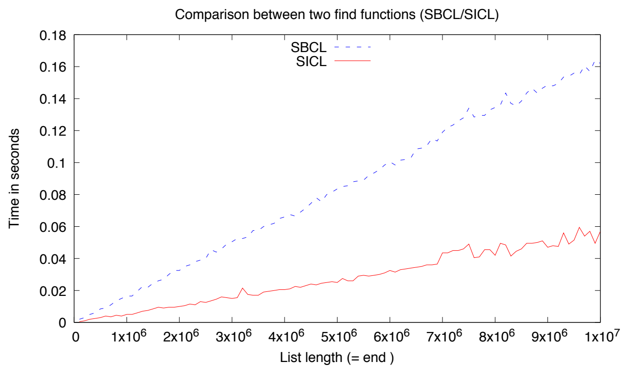
In the next test, the sequence is a list where each element is a pair where the first element is a bignum, so that we can use `car` as a `key` function. The `test` function is still `=`. There seems to be no significant difference for this case, compared to the previous one, suggesting that the native implementation of `car` is so fast that calling `=` will dominate the computation.



In the next test, the sequence is a list of pairs of symbols; it uses `car` as key function, `eq` as test function and a non `nil` value for the `end` parameter. Our implementation is at least twice as fast as that of SBCL.



Our final test uses a non `nil` value for the `end` parameter. Despite the fact that we use a slightly more expensive `test` function (namely `=`), the performance of our implementation is very good; it is around three times as fast as that of SBCL.



4.2 Maintainability

From the point of view of maintainability, there are clear advantages to our technique. With only a small amount of macro code, we are able to hide the implementation details of the functions, without sacrificing performance.

The small amount of macro code that is needed to make our technique work is clearly offset by the considerable decrease in the code size that would otherwise have been required in order to obtain similar performance.

4.3 Disadvantages

There are not only advantages to our technique.

For one thing, compilation times are fairly long, for the simple reason that the body of the function is duplicated a large number of times. Ultimately, the compiler eliminates most of the code, but initially the code is fairly large. And the compiler must do a significant amount of work to determine what code can be eliminated. To give an idea of the orders of magnitude, in order to obtain fully-expanded code on SBCL, we had to increase the inline limit from 100 to 10 000, resulting in a compilation time of tens of seconds for a single function. For the case of a vector, the inner loop will be replicated for each element type, then each of these replicas will again be replicated for each special case of the `test` and `test-not` functions, and again for each special case of a `key` function, and finally for the two cases of processing from the beginning or the end. The resulting number of replicas of the inner loop depends on the number of such special cases, but can easily exceed 1000. Each replica will ultimately be stripped down by the compiler because redundant tests will be eliminated for each one.

Another disadvantage of our technique is that it doesn't lend itself to applications using short sequences. For such applications, it would be advantageous to inline the sequence functions, but doing so would make each call site suffer the same long compilation times as we now observe for the ordinary callable functions.

Not all compilers are able to optimize the main body of a function according to some enclosing condition. For a Common Lisp implementation with a more basic compiler, no performance improvement would be observed. In addition, the duplication of the main body of the function would result in a very large increase of the code size, compared to a simpler code with the same performance.

For the special case of bit vectors, our technique will not be able to compete with a good native implementation of the sequence functions. The reason is that, despite the optimizations that the compiler can perform with our technique, the body of a typical sequence function still consists of a loop where each iteration treats a single element. A native implementation would not treat a single element in an iteration. Instead, it would take advantage of instructions that exist in most processors for handling an entire *word* at a time, which on a modern processor translates to 64 bits. An implementation that uses our technique would then typically handle bit vectors as a special case, excluded from the general technique.

5. CONCLUSIONS AND FUTURE WORK

We have presented a technique that allows implementations of most of the Common Lisp sequence functions that are simultaneously fast, maintainable, and portable, provided the compiler supplied by the implementation is sufficiently sophisticated to apply certain standard optimization techniques.

The main exception for which our technique is generally unable to compete with a native implementation is when the sequence is a bit vector. Any implementation that accesses the elements of the bit vector one at a time, rather than using native instructions that can handle an entire word at a time, is unable to match the native performance [2]. On the other hand, our technique allows the Common Lisp implementation to treat bit vectors as an exceptional case, and use our general technique for the other cases.

We have yet to perfect the exact declarations to include in our implementation, and the exact places where these declarations should be added. Different Common Lisp implementations have different requirements in this respect, so this work may have to be repeated for different implementations.

At the moment, we have been working exclusively with SBCL, for the simple reason that the SBCL compiler does provide the optimizations that are required in order for our technique to yield excellent performance. We intend to experiment with other major implementations as well in order to determine which ones are suited for our technique. For our technique to provide fast code, the compiler must be able to remove redundant tests. A test T_2 is redundant if it is dominated[3] by a test T_1 testing the exact same condition, and T_2 can be reached from only one of the two branches of T_1. In our technique, an outer macro provides T_1, whereas T_2 occurs in the inner loop of the function.

[3]Dominance is a graph-theory concept that is frequently used in optimizing compilers to transform intermediate code in the form of an instruction graph.

The Cleavir compiler framework of the SICL project will ultimately include a technique for *path replication* in intermediate code, that, while not specifically meant for the kind of optimization required for the technique presented in this paper, will have the same effect as more direct techniques currently used in advanced compilers.

Our technique is well adapted to processing sequences with a relatively large number of elements. When the sequence contains few elements, the overhead of the call and of processing the keyword arguments may be significant. Also, we do not take advantage of any declaration of element type, in the case when the sequence is a vector. We plan to investigate the possibility of modifying our macros so that definitions of specialized functions are automatically generated, leaving a fairly small general function that can then be inlined.

6. ACKNOWLEDGMENTS

We would like to thank Bart Botta, Pascal Bourguignon, and Philipp Marek for providing valuable feedback on early versions of this paper.

7. REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp.* American National Standards Institute, 1994.

[2] H. G. Baker. Efficient implementation of bit-vector operation in common lisp. *SIGPLAN Lisp Pointers*, III(2-4):8–22, Apr. 1990.

[3] I. Durand and R. Strandh. Processing list elements in reverse order. In *Proceedings of the 8th European Lisp Symposium*, ELS '15, 2015.

APPENDIX

A. PROTOCOL

In this appendix, we describe the macros and functions that are part of the protocol of our technique, used for implementing most of the sequence functions.

`apply-key-function` *element key-function* [*Function*]

This function takes an element of the sequence and a function to apply in order to obtain the object to use for comparison. For performance reasons, this function should be inlined.

A typical definition of this function might look like this:

```
(defun apply-key-function (element key-function)
  (declare (optimize (speed 3) (debug 0) (safety 3)))
  (declare (type function key-function))
  (cond ((eq key-function #'identity)
         element)
        ((eq key-function #'car)
         (car element))
        ((eq key-function #'cdr)
         (cdr element))
        (t
         (funcall key-function element))))
```

`canonicalize-key` *key-var* [*Macro*]

This macro takes a single argument which must be a variable that holds the value of the `&key` keyword argument. Its role is to make sure the contents of the variable is a function. A typical implementation might look like this:

```
(defmacro canonicalize-key (key-var)
  `(cond ((null ,key-var)
          (setf ,key-var #'identity))
         ((not (functionp ,key-var))
          (setf ,key-var (fdefinition ,key-var)))
         (t nil)))
```

`with-key-function` *key-function-var* `&body` *body* [*Macro*]

This macro takes a single argument which must be a variable that holds the value of the canonicalized `&key` keyword argument. It is used to duplicate *body* for different typical values for the *key* argument to many sequence functions. A typical implementation of this macro looks like this:

```
(defmacro with-key-function (key-function-var &body body)
  `(cond ((eq ,key-function-var #'identity)
          ,@body)
         ((eq ,key-function-var #'car)
          ,@body)
         ((eq ,key-function-var #'cdr)
          ,@body)
         (t
          ,@body)))
```

In each clause of the `cond` form in this macro, the inlined version of the function `apply-key-function` will be simplified in a different way by the compiler, resulting in a specialized loop.

`for-each-relevant-cons`
(cons-var index-var list start end from-end) `&body` *body* [*Macro*]

This macro executes *body* for each *relevant* **`cons`** *cell.* It takes into account the values of *start* and *end* to restrict the execution to a particular sub-sequence, and it takes into account the value of *from-end* to determine the order in which the relevant `cons` cells are supplied to the *body* code. The parameter *cons-var* is the name of a variable that contains a reference to the relevant `cons` cell for each execution of *body*. Similarly, the parameter *index-var* is the name of a variable that contains the index of the particular `cons` cell to be processed.

Because of the size of the definition of this macro, due mainly to the code for processing `cons` cells in reverse order [3], we do not show its definition here.

`with-test-and-test-not`
(test-var test-not-var) `&body` *body* [*Macro*]

The role of this macro is to supply certain special cases for the possible values of the keyword parameters `test` and `test-not` of a typical sequence function. It is assumed that it has already been verified that at most one of the two keyword arguments has a value other than `nil`. A typical implementation might look like this:

```
(defmacro with-test-and-test-not
    ((test-var test-not-var) &body body)
  `(cond ((null ,test-not-var)
          (locally (declare (type function ,test-var))
            (cond ((eq ,test-var #'eq)
                   ,@body)
                  ((eq ,test-var #'eql)
                   ,@body)
                  (t
                   ,@body))))
         ((null ,test-var)
          (locally (declare (type function ,test-not-var))
            (cond ((eq ,test-not-var #'eq)
                   ,@body)
                  ((eq ,test-not-var #'eql)
                   ,@body)
                  (t
                   ,@body))))
         (t nil)))
```

`with-from-end` *from-end-var* `&body` *body* [*Macro*]

This macro duplicates *body* for the two cases where the value of the argument variable *from-end-var* is either *true* or *false*. A typical implementation looks like this:

```
(defmacro with-from-end (from-end-var &body body)
  `(if ,from-end-var
       (progn ,@body)
       (progn ,@body)))
```

`satisfies-two-argument-test-p`
item element test test-not [*Function*]

This function is typically inlined. It provides special cases for common values of the `test` and `test-not` keyword arguments of a typical sequence function. All but one of these cases will be eliminated in each branch of the macro `with-test-and-test-not` in which this function is located. A typical implementation might look like this:

```
(defun satisfies-two-argument-test-p
    (item element test test-not)
  (declare (optimize (speed 3) (debug 0) (safety 3)))
  (cond ((null test-not)
         (locally (declare (type function test))
           (cond ((eq test #'eq)
                  (eq item element))
                 ((eq test #'eql)
                  (eql item element))
                 (t
                  (funcall test item element)))))
        ((null test)
         (locally (declare (type function test-not))
           (cond ((eq test-not #'eq)
                  (not (eq item element)))
                 ((eq test-not #'eql)
                  (not (eql item element)))
                 (t
                  (not (funcall test-not item element))))))
        (t nil)))
```

`for-each-relevant-element`
element-var index-var vector start end
from-end `&body` *body* [*Macro*]

This macro is used to traverse the elements of a vector. The argument *element-var* is a symbol that is bound to each element during the execution of *body*. Similarly, *element-var* is a symbol that is bound to the index of the relevant element. The *vector* argument is an expression that must evaluate to a vector. The arguments *start* and *end* are expressions that evaluate to the two indices of the interval to traverse. Finally, `from-end` is a generalized Boolean that indicates whether the traversal is to be done from the end of the relevant interval. A typical implementation of this macro might look like this:

```
(defmacro for-each-relevant-element
    ((elementv indexv vector start end from-end)
     &body body)
  (let ((vectorv (gensym))
        (startv (gensym))
        (endv (gensym)))
    `(let ((,vectorv ,vector)
           (,startv ,start)
           (,endv ,end))
       (declare (type fixnum ,startv ,endv))
       (if ,from-end
           (loop for ,indexv downfrom (1- ,endv)
                   to ,startv
                 do (let ((,elementv
                            (aref ,vectorv ,indexv)))
                      ,@body))
           (loop for ,indexv from ,startv below ,endv
                 do (let ((,elementv
                            (aref ,vectorv ,indexv)))
                      ,@body))))))
```

`with-simple` *vector* `&body` *body* [*Macro*]

This macro simply checks whether *vector* is a `simple-array`, and duplicates *body* in each branch of the test. A typical implementation might look like this:

```
(defmacro with-simple (vector &body body)
  `(if (typep ,vector 'simple-array)
       (progn ,@body)
       (progn ,@body)))
```

`with-vector-type` *vector-var* `&body` *body* [*Macro*]

This macro duplicates *body* for each possible value of `array-upgraded-element-type` that the implementation provides. It also provides a local definition for the macro `vref` which we use instead of `aref` to access the elements of the vector in *body*. If the compiler of the implementation is unable to specialize `aref` according to the element type, then the implementation may provide different definitions of the macro `vref` for different element types. Since the supported element types vary from one implementation to another, we do not provide an example of how this macro may be implemented.

B. EXAMPLE IMPLEMENTATION

As an example of how the sequence functions might be implemented using the functions and macros in Appendix A, we show our implementation of `find-list` which is called from `find` when the sequence is known to be a list:

```
(defun find-list
    (item list from-end test test-not start end key)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (declare (type list list))
  (with-bounding-indices-list (start end)
    (with-key-function key
      (with-test-and-test-not (test test-not)
        (with-from-end from-end
          (for-each-relevant-cons
              (cons index list start end from-end)
            (let ((element (apply-key-function
                            (car cons) key)))
              (when (satisfies-two-argument-test-p
                     item element test test-not)
                (return-from find-list element)))))))))
```

DIY Meta Languages with Common Lisp

Alexander Lier   Kai Selgrad   Marc Stamminger

Computer Graphics Group, Friedrich-Alexander University Erlangen-Nuremberg, Germany

{alexander.lier, kai.selgrad, marc.stamminger}@fau.de

ABSTRACT

In earlier work we described C-Mera, an S-Expression to C-style code transformator, and how it can be used to provide high-level abstractions to the C-family of programming languages. In this paper we provide an in-depth description of its internals that would have been out of the scope of the earlier presentations. These implementation details are presented as a toolkit of general techniques for implementing similar meta languages on top of Common Lisp and illustrated on the example of C-Mera, with the goal of making our experience in implementing them more broadly available.

CCS CONCEPTS

•Software and its engineering →Source code generation; Preprocessors; *Translator writing systems and compiler generators;*

KEYWORDS

Code Generation, Common Lisp, Macros, Meta Programming

ACM Reference format:
Alexander Lier, Kai Selgrad, and Marc Stamminger. 2017. DIY Meta Languages with Common Lisp. In *Proceedings of European Lisp Symposium, Brussel, Belgium, April 2017 (ELS'17),* 8 pages.
DOI: 10.475/123_4

1 INTRODUCTION

In this paper we describe techniques that we employed to implement C-Mera [19], a meta language for C (and C-like languages), embedded in Common Lisp. C-Mera provides a Lisp-like syntax for C, that is, it is not a compiler from Lisp to C, but from C written in Lisp-form to regular C, i.e. there is no inherent cross-language compilation. An exemplary C-Mera (C++) program that simply prints all of its command-line arguments looks as follows:

```
1  (include <iostream>)
2
3  (defmacro println (&rest args)
4    `(<< #:std:cout ,@args #:std:endl))
5
6  (function main ((int argc) (char *argv[])) -> int
7    (for ((int i = 1) (< i argc) ++i)
8      (println " - " argv[i]))
9    (return 0))
```

The mapping to C++ is straightforward for the most part, and readers familiar with Lisp will recognize that lines 3-4 show a very simple macro that is then used in the main function. For a more thorough description and many more examples see our earlier work [12, 17–19], note, however, that even the above code shows features not present in many projects similar to C-Mera, e.g. inline type annotations (i.e. pointers), idiomatic C shorthands such as the post-increment and seamless integration with standard Common Lisp macros.

On the side of the language's user the most important features of C-Mera are its flexibility and extensibility, especially via Common Lisp macros. Using those, custom abstractions can be built easily, and with zero cost at run-time, which is very important when working in high-performance application domains. As these abstractions are quickly and easily attained such meta languages can be a valuable tool for prototyping, research, and when working on tight deadlines. Examples from this point of view can be found in previous work on C-Mera and its application [12, 17–19].

In this paper we provide a more in-depth description of C-Mera from the language implementor's side. One of the key features of C-Mera from this vantage point is the simplicity of its architecture. Due to its embedding in Common Lisp (and adoption and thus exploitation of its syntax) the problem of defining a system suitable for highly involved meta programming in C and C-like languages is reduced to constructing C-programs from S-Expressions, pretty-printing the internal representation in form of C-code and configuring the Common Lisp environment such that any inconsistencies with our target languages are resolved appropriately. The implementation of these details is described on a much more technical level than the scope of previous work allowed.

We believe that summarizing these details and documenting the design decisions behind them can be valuable to projects with similar goals, even when applied to different target languages or application domains. Especially since most of the implementation details described are not tied to C at all, they can be applied to help construct other meta languages on top of Common Lisp. Section 2 also lists a few Lisp-based projects that follow a similar path as C-Mera and those could naturally find inspiration from this detailed description.

In the remainder of this paper we first provide context for our work, starting with C-Mera and similar Lisp-based approaches over works that employ similar concepts in other languages to more general compiler technology and how it is used towards the same ends (Section 2). Following that, we detail the design goals we set up for C-Mera and the evaluation process of a C-Mera program, from how the source is read over its internal representation to tree traversal during C-code generation (Section 3). We then describe our package setup in more detail, noting the intricacies of overloading C-Mera and Common Lisp symbols (Section 4). Finally, we provide some very technical details on how to find a balance between the idiosyncrasies of the Common Lisp and C-family syntaxes (Section 5) and conclude with a short summary (Section 6).

2 RELATED WORK

Our description of techniques for implementing meta languages in Common Lisp is naturally founded on our experience of working on C-Mera. Following the initial description [19] that demonstrated meta programming for stencil computations, we showed how it can be used to provide higher-level programming paradigms to the realm of C-like languages [18]. We also presented two real-world use cases. Firstly, a domain-specific language for high-performance image-processing applications [17] and, secondly, how C-Mera can be used to explore a vast space of implementation variants of a given algorithm [12]. With this paper we go back towards describing our base system presented earlier [19], however, the focus of this work is not on a description of the concept and providing examples to illustrate its versatility, but on the actual, low-level implementation details and design choices involved in the process.

Convenient, fully fledged macros and therefore extensive and easy meta programming is most prominently featured in the Lisp family of languages such as Racket, Scheme and Common Lisp. For this reason, these languages are host to many similar projects: Parenscript [16] generates JavaScript, whereas C-Amplify [6], Cl-Cuda [22], Lisp/c [2], and C-Mera target C, C++ and similar C-syle languages (with varying degrees of language support and maturity). While reaping the benefits of straightforward embedding in a powerful host language, following this approach the language designer is not as free as when starting out from scratch.

Some rather new languages, such as Rust [13], are designed to also support Lisp-style macros. However, as long as such languages show a less uniform syntax larger-scale meta programming (in the example of Rust using procedural macros) comes at a higher cost of engineering.

Naturally, the more ambitious and free option to write a language from scratch is, in principle, always available. Specific tools, for example Yacc [8], Lex [11] or ANTLR [14], and libraries such as Spirit [7] can ease the process of constructing an appropriate parser. However, building a consistent language and implementing powerful meta-programming capabilities are still the responsibility of the language designer.

Extending an existing language for meta programming provides a more efficient solution. For example, MetaOcaml [4] provides facilities for multi-stage programming with OCaml. Another example is Terra [5], which provides meta-programming capabilities by utilizing Lua as the host language. Lua functions can be applied to adjust, extend, and write Terra code and the embedded code can reference variables and call functions defined in Lua. Terra's sytanx is based on Lua and processed with a just-in-time compiler and can optionally be modified further with Lua prior to eventual compilation.

Such approaches require considerable effort to be realized, especially when targeting syntactically hard languages (e.g. C++). Other approaches utilize available language resources that were originally not intended for meta programming on that scale. C++ Template Meta Programming [25] (TMP), for example, exploits the template mechanism for extensive abstractions. The demand for such abstractions is visible from the field started by this work [1, 24], especially in the face of it being generally considered very hard [6, 10, 20].

In contrast, utilizing Lisp as an code generator is generally a straightforward task, but unleashing the full potential of Lisp's built-in functions and macro system while allowing convenient and naturally written input code (from the perspective of the C, as well as the Lisp programmer) can become rather tricky. In the most naïve approach, every syntactical element of a meta language implemented in, e.g., Common Lisp, will be mapped to S-Expressions, leading to highly verbose code. This can go on well unto the level of specifying how symbols have to be rewritten [2]. Sections 4 and 5 will describe the compromise found during C-Mera's implementation to have more free-form code while not suffering a loss in generality.

3 EVALUATION SCHEME

In this section we first define the most important characteristics that we wanted our meta language to exhibit (Section 3.1). We then provide a short overview of our intended syntax and mode of evaluation, exemplified with C-Mera (Section 3.2). Following that we describe the evaluation scheme in more detail, starting with how the internal representation is constructed (Section 3.3), kept and finally written out again (Section 3.4).

3.1 Design Goals

The most fundamental requirement for our language was being able to seamlessly interact with Common Lisp's macro system. This way we ensured that it is meta programmable to the same degree and not limited, e.g., to some specific form of templating. Interaction with Common Lisp's macro system also entails that writing our own macro-expansion routines was never intended, that is, our problem statement is much simpler as it seems at first glance. We also wanted to provide a system as accessible to C-programmers as possible, given our primary objective. For us, this entails to have the language properly keep the case of symbols (while not making the Common Lisp code in macros more awkward than necessary), to provide as many idiomatic C shorthands as possible (e.g., increments, declaration decorators), to interact with Common Lisp code (honoring lexical scope), to avoid quotation whenever possible and being able to reference symbols from external C files.

The style of meta programming we wanted to support and explore is strictly macro-based. That is, we want the language to be able to specify new syntax and cast semantics into it and not explicitly post-process a syntax tree (as possible, e.g., by working on our AST, as described in Section 3.4, or, for example when using C++ only, via Clang [23]). Note that the latter approach is in fact more powerful, but comes with a much larger overhead in engineering and might consequentially not pay off for projects of small to medium size [17].

3.2 Look and Feel

In the following we will provide a higher-level overview of a few aspects of C-Mera's internal workings, mainly aiming to provide a general outline of the intended look-and-feel we wanted to achieve for our meta language.

Symbols. The following toy example demonstrates the mapping of two addition expressions enclosed in a lexical environment that introduces a local variable in the Common Lisp context.

```
(let ((x  4))
  (set foo (+ 1 2 (cl:+ 1 2) x 'x y)))
```

As can be seen in the following line generated from this example, not all expressions appear in the resulting code, since the Common Lisp expressions are not part of the target language:

```
foo = 1 + 2 + 3 + 4 + x + y;
```

This example is comprised of the following components. The *set* form indicates an assignment for the target language and creates a syntax element that is carried over to the resulting code. Both the plain *+* and *cl:+* (from the *cl* package) denote addition, but the one from the *cl* package is evaluated directly within the host language and yields the number *3*, whereas the unqualified counterpart is kept as an expression of the target language. Common Lisp's special operator *let* defines a lexical scope and introduces the variable *x* in the example above. This operation is solely executed in the scope of the host language and does not contribute additional output to the resulting code. In contrast to *x*, the symbol *y* is undefined, thus it is taken to directly refer to a variable in C. Since there is obviously no useful application of an unquoted, undefined variable in the host language, the assumption that a symbol is designated to be used in the target language (and thus is undefined in the host's context) is justified. Therefore, there is no need to quote undefined symbols inside target language syntax elements. However, it is still required to quote symbols defined in the host language's context, to process them as variables for the target language and avoid value substitution. The implementation of this feature is described in Section 5.3.

Functions and Macros. Functions are managed in a similar fashion. This also holds for special forms to construct the syntax tree of the target language that are, naturally, defined in the host language's context (see Section 3.3). If the first element of a list is not defined during evaluation, it is taken to denote a function call in the target language. In the following example one function and two macros are defined (one function is commented out) and *foo* is assigned the result of calling those, in turn.

```
(defun bar (a b) (cl:+ a b))
; (defun baz (a b) (cl:+ a b))
(defmacro qux (a b) `(+ ,a ,b))
(defmacro qox (a b) `(cl:+ ,a ,b))

(set foo (bar 1 2))
(set foo (baz 1 2))
(set foo (qux 1 2))
(set foo (qox 1 2))
```

Here, the function *bar* returns the number *3*, which is used directly in the resulting code. The function *baz* is not defined (indicated by the line being commented out). Based on the aforementioned processing of unbound symbols, the undefined list head *baz* is treated as a function call in the target language. As can be seen in the generated code below, only one function call (for which there was no valid host-context function available) is generated in the target code:

```
foo = 3;
foo = baz(1, 2);
foo = 1 + 2;
foo = 3;
```

Also note how the expansion of the *qox* macro is evaluated in the host context.

As with symbols used for variable names, there is an ambiguity if the symbol is defined both in the host and target language. In these cases we opted to prefer the host language's version (as with variables), while a function call in the target language can be generated using the *funcall* form. Continuing the example above the following expressions do not trigger host-language function calls or macro invocations:

```
(set foo (funcall bar 1 2))    →   foo = bar(1, 2);
(set foo (funcall qux 1 2))    →   foo = qux(1, 2);
```

The implementation of this feature is also described in Section 5.3.

3.3 Evaluation

The evaluation scheme we apply in order to build an Abstract Syntax Tree (AST) may be one of the most simple approaches. Nevertheless, for our needs it is more than sufficient and, more importantly, very easy to implement, utilize, and extend. In the following we show how a target-language infix operator (e.g. *+*) can be defined.

```
(defclass infix-node ()
  ((operator :initarg :op)
   (member1  :initarg :lhs)
   (member2  :initarg :rhs)))
```

This specifies an infix expression as being comprised of an operation with a left- and right-hand side. Note that the representation is simplified at this point, a more detailed description of the AST nodes can be found in Section 3.4.

With this example AST, nodes for such expressions can be generated by calls to *make-instance*, building up a tree of node objects. As this would clearly not be very concise code, we wrap macros around each AST-node constructor. For the case of an addition expression this would be `(+ ...)`. The following macro suffices to wrap around the call to *make-instance*:

```
(defmacro + (lhs rhs)
  `(make-instance 'infix-node
                  :op '+
                  :lhs ,lhs
                  :rhs ,rhs))
```

Application of this macro yields the appropriate AST node:

```
(+ 1 2)   →   #<INFIX-NODE #x...>
```

Using this scheme we are not limited to one single level of evaluation, nor constrained to entirely stay in the target language. Nesting multiple target functions and mixing them with host code is supported and intended. The following example shows an evaluation process starting with:

```
(* (/ 1 2) (+ (cl:+ 1 2) 3))
```

Expressions are, as usual, evaluated from the inside (starting with the two leaf nodes of the AST to-be). In this case, one of those expressions generates a node object and the other evaluates as a build-in Common Lisp expression:

```
(* #<INFIX-NODE #x1...> (+ 3 3))
```

The next step is the evaluation of the remaining nested objects:

```
(* #<INFIX-NODE #x1...> #<INFIX-NODE #x2...>)
```

In the final step, the entire expression collapses to one single object:

```
#<INFIX-NODE #x3...>
```

As can be seen, every evaluation results in a node object that is thereafter used in the next evaluation step by its parent.

As described above, the full evaluation process also includes macro expansion that generates the instantiation calls. Thus, the evaluation first expands to the following *make-instance* form:

```
(make-instance 'infix-node
  :op '+
  :lhs (make-instance 'infix-node
         :op '/
         :lhs 1
         :rhs 2)
  :rhs (make-instance 'infix-node
         :op '+
         :lhs (cl:+ 1 2)
         :rhs 3))
```

According to this, the AST is built by expanding all macros and collapsing the individual calls to *make-instance* by evaluation:

```
(make-instance 'infix-node
  :op '+
  :lhs #<INFIX-NODE #x1...>
  :rhs #<INFIX-NODE #x2...>)
```

This shows that the evaluation scheme results in an implicitly self-managed construction of the AST. That is, we rely entirely on the standard Common Lisp reading and evaluation process. Therefore, there is no need for an extra implementation of a parser inside the host language, since every aspect is already handled by the Common Lisp implementation itself.

Note that this evaluation scheme seamlessly integrates with macro processing in general and thus facilitates the incorporation of new, user-defined syntax, even up to the scope of defining custom DSLs [17, 19] without any changes to the underlying AST.

3.4 Abstract Syntax Tree

The AST is the intermediate representation of the fully macro-expanded and evaluated input code. Every node type used for the AST is derived from one common class (*node*). Independent from additional information stored inside individual derived objects, every class instance contains the slots *values* and *subnodes* (inherited from *node*). This very simple setup renders the traversal of the AST almost trivial. Based on Common Lisp's multi-methods, we require only two methods to build various traversers. One of these methods handles the class *node*:

```
(defclass node ()
  ((values   :initarg :values)
   (subnodes :initarg :subnodes)))
```

For derived classes, the slot *values* contains a list of slot names, which are not processed further by the traverser (i.e. leaf nodes). The *subnodes* slot stores a list of slot names that the traverser descends into (i.e. internal nodes). The structure can be used to capture nodes where the sub-nodes have different semantics (e.g. conditional expressions). Nodes storing multiple objects with the same semantics (e.g. body forms) utilize *nodelist*:

```
(defclass nodelist (node)
  ((nodes :initarg :nodes)))
```

The *nodes* slot is a plain list containing the sub-node objects. All nodes in our AST are either a *nodelist* or derived from *node*, and most of the traversal is implemented in terms of them.

AST Traversal. Traversal then works as follows: The traverser starts at the root node and when it encounters an object of type *node* it calls itself recursively for slots of the current object listed in *subnodes*:

```
(defmethod traverser ((trav t) (node node))
  (with-slots (subnodes) node
    (loop for slot-names in subnodes do
      (let ((subnode (slot-value node slot-name)))
        (when subnode
          (traverser trav subnode))))))
```

A similar procedure is executed for *nodelist* nodes.

With the classes defined above and these methods we have a simple traversal scheme that can easily be extended for further tasks. Additionally, more specific traversal methods can implement mechanisms for processing the data of individual node types. As a result, building functionalities that require AST traversal becomes straightforward. The following traverser simply lists all infix expressions in the tree (continuing the example from Section 3.2):

```
(defclass debug-infix ())

(defmethod traverser ((_ debug-infix) (node infix-node))
  (format t "~a~%" (slot-value node 'op)
  (call-next-method)))
```

Note that *call-next-method* continues with the general tree traversal.

Before and After. With additional support from Common Lisp's *before* and *after* methods, the generation of syntactically faithful target code becomes even more comfortable. As an example, utilizing these features enables catching the beginning and end of an expression, which, for example, can easily be exploited for emitting parentheses:

```
(defmethod traverser :before ((pp pretty-printer) (_ infix-node))
  (format (stream pp) "("))

(defmethod traverser :after ((pp pretty-printer) (_ infix-node))
  (format (stream pp) ")")
```

With this approach we can easily ensure proper execution order for arithmetic expressions:

```
(/ (+ 1 3) (+ 2 5))   →   ((1 + 3) / (2 + 5))
```

Proxy-Node Extension. Although we are able to trigger traversal events when entering and leaving a node as described above, we cannot trigger them when descending and returning from specific child nodes while processing their parent node. Visitation of these nodes is implemented by methods on their respective node type, but this loses the context of their parent node (which might be required to generate inter-node output). This situation can easily be solved with *proxy nodes*. Nodes of this type are merely sentinels, without additional content, and are solely applied to identify transitions between nodes. They are usually inserted and removed by a node that needs control over the output between its child nodes.

One example would be placing the +-signs in an arithmetic expression such as `(+ a b)` to yield `a + b`, using the following proxy:

```
(defclass plus-proxy (node)
  ((subnode :initarg :subnode)))
```

Using such an object for the right-hand-side operand of the infix-node would then trigger the proper method with correct placement of the plus sign.

```
(defmethod traverser :before ((pp pretty-printer) (_ plus-proxy))
  (format (stream pp) " + "))
```

Note that this scheme is just an approach to keep traversal mechanics and output logic distinct. When mixing both, proxies will not be required, but then each method on a given node type will have to repeat the traversal logic.

Overall be believe that our AST scheme is very simple and consequently easy to use, while still being easily extensible.

4 PACKAGES

As seen in Section 3.2, we do not want to explicitly annotate symbols with their packages. However, starting from the default package (*cl-user*) the following attempt to define a macro fails:

```
(defmacro + (a b)
  `(make-instance 'infix-node ...))
```

This is due to the inherent *package-lock* on the user package's interned COMMON LISP functions and macros. Every symbol interned form *common-lisp* (or short, *cl*) is locked by default. This would prevent any of the redefinitions we have already used many times until now. A possible solution to allow modifications is unlocking packages, which generally is a poor approach, as it is not standardized and drops the overridden symbols' default implementation.

The lock, however, only affects the actual symbols in the *cl* package, not symbols of the same name from different packages. This fact is usually not obvious as virtually all COMMON LISP code uses the *cl* package, which then results in name conflicts that cannot be overridden due to the lock. The key to solve this issue is very simple: not using the *cl* package, or, when only few symbols are to be overridden, to not include those when using the *cl* package.

```
(defpackage :meta-lang
  (:use :common-lisp)
  (:shadow :+))
```

This package definition interns all symbols but + from the *cl* package. As a result the symbol + is unbound and can be used, e.g., for macro definitions:

```
(in-package :meta-lang)

(defmacro + (a b)
  `(cl:+ ,a ,b))
```

The example above defines a simple macro that maps the +-sign to its implementation from the *cl* package. As can be seen, access to the original implementation is still possible if the symbol is used with its package prefix. Such a package setup enables us do redefine symbols according to our needs:

```
(defpackage :cm-c
  (:use :common-lisp)
  (:shadow :+))

(in-package :cm-c)

(defmacro + (lhs rhs)
  `(make-instance 'infix-node
                  :op '+
                  :lhs ,lhs
                  :rhs ,rhs))
```

Using this package, superfluous prefixes can be omitted and are only required when accessing (overridden) symbols from *cl*:

```
(+ (cl:+ 1 2) (cl:+ 2 3))     →   3 + 5;
```

Since we also want to reduce the amount of explicitly qualified names in the input code we can utilize a simple *macrolet* to adjust the effects of those symbols in its lexical scope:

```
(defmacro lisp (&body body)
  `(macrolet ((+ (lhs rhs) `(cl:+ ,lhs ,rhs)))
     ,@body))
```

At this point we might end up at an impasse; once inside the lexical scope of the macrolet, globally defined functions redefined in the local scope are not accessible:

```
(+      1 2)      →   1 + 2
(cm-c:+ 1 2))     →   1 + 2
(cl:+   1 2)      →   3

(lisp
  (+      1 2)    →   3
  (cm-c:+ 1 2)    →   3
  (cl:+   1 2))   →   3
```

The obvious solution for this problem is to introduce a third variable that retains the initial functionality, as opposed to the locally used, volatile symbols. To keep the symbols' names, an additional package is required to place those symbols in:

```
(defpackage :swap (:use) (:export :+))
```

Macros plainly wrapping the original symbol or function, unfortunately, fail to provide the required behaviour. Such macros emit symbols that are then still bound in the lexical scope of the surrounding macrolet.

```
(defmacro swap:+ (lhs rhs)
  `(cm-c:+ ,lhs ,rhs))
```

To escape the lexical scope, we can access a symbol's original macro implementation with *macroexpand-1*:

```
(defmacro swap:+ (lhs rhs)
  (macroexpand-1 `(cm-c:+ ,a ,b)))
```

As long as *macroexpand-1* is called without an environment argument it returns the version of the macro defined in the global environment. With such *swap symbols* we are able to address the global implementation of our syntax, independent from the current lexical scope.

5 BRIDGING THE SYNTACTIC GAP

In this section we describe how certain details of C-MERA are laid out to strike a balance between the worlds of our host and target languages. The first part discussed is control over case. COMMON LISP converts symbols that it reads to upper case unless otherwise specified. With this default behaviour, users from C-family languages would be surprised to see how their code changed when printed out in the target language. Section 5.1 details why the built-in modes in COMMON LISP do not suffice and describes the compromise employed in our language.

A difficulty of a different kind is LISP's very uniform notation on the one hand and C's (and even more so its derived languages) extensive syntax on the other hand. Although every aspect of C-family languages can be modeled with S-Expressions, we doubt the benefit of having to formulate every little aspect of C's syntax this way. Arrays can be used as an example here: It is obvious that writing `(array (array (array foo 1) 2) 3)` is not as convenient, at least not as concise, as writing `foo[1][2][3]`. We aim for supporting as much as possible of C's handy syntax by exploiting the extensible COMMON LISP reader to parse special syntax. Details on our implementation of such shorthands are presented in Section 5.2.

The last aspect of our presentation is concerned with the input code's aesthetics and appeal. We want to write code as conveniently as possible and prevent exhaustive (and to part of our target audience, confusing) usage of quotes. Instead of writing `(funcall 'f 'a 'b (funcall 'g 1 'c))`, we simply want to allow (and do support) the call to be `(f a b (g 1 c))`, even if *f, a, b, g*, and *c* are unbound. Program code, even if it is target code, should appear as natural code in Lisp notation, and not require quotes when the situation can be uniquely resolved. Our implementation of adaptive quotation that tackles this issue is described in Section 5.3.

5.1 Preserving Case by Inversion

Code in Common Lisp is, unless explicitly avoided, converted to upper case, therefore it often appears that the case of the input code is not considered at all. It can be controlled on a per-symbol level via `(intern "foo")` and explicit literals such as `|foo|`. More general control is available via *readtable-case*, which can change the default behaviour. Even though there is the so called *modern style* for Common Lisp, which sets the *readtable-case* to *:preserve*, current implementations are usually compiled in *:upcase* (causing all *cl* symbols to be interned in upper case). Since our target language does not do any automatic case conversion, but keeps the input code's case as it is, we have been compelled to reproduce this behaviour as closely as possible in C-Mera.

The naïve approach is setting the *readtable-case* to *:preserve* when processing a source files. This is an inadequate solution, however, as it would require us to use upper-case representations of all the standard Common Lisp symbols. As a result, input code would have to be written in the following form:

```
(setf (readtable-case *readtable*) :preserve)

(DEFUN foo (a b) (+ b c))
(DEFUN bar (a b) (CL:+ a b))

(foo 1 (foo X y) (bar 1 2))
```

With *:preserve* we are free to use upper- and lower-case symbols for variables (*X* and *y* in this example), but also forced to write all existing Common Lisp symbols (such as *DEFUN*) in upper case.

Since *:downcase* would not work at all (does not keep case), the only option left to investigate is *:invert*. In fact, with *:invert*, input symbols in lower case are mapped to upper-case symbols (and vice versa). Therefore, input code in lower case can be mapped to existing functions and symbols. One problem remains: newly introduced functions and variables are also inverted. Luckily, Common Lisp processes symbols an additional time during printing, which is a natural part of source-to-source compilers such as we are targeting. Therefore, the desired functionality is available out of the box:

```
(format t "~a" 'foo)   →   FOO
(format t "~a" 'FOO)   →   FOO
(format t "~a" 'Foo)   →   FOO

(setf (readtable-case *readtable*) :invert)
(format t "~a" 'foo)   →   foo
(format t "~a" 'FOO)   →   FOO
(format t "~a" 'Foo)   →   Foo
```

This seems to be a reliable solution, but one detail should be kept in mind: The *intern* function now shows counter-intuitive behavior with inverted reading, since it does not use the reader and therefore is not affected by the *readtable*:

```
(setf (readtable-case *readtable*) :invert)
(format t "~a" (intern "foo"))   →   FOO
(format t "~a" (intern "FOO"))   →   foo
(format t "~a" (intern "Foo"))   →   Foo
```

Interning does not read symbols, but strings, and therefore it misses the initial inversion step. In the given situation, we have implemented and used our own *intern* function that inverts the read string in the same way as the reader does.

5.2 Universal Reader

As exemplified at the outset of of Section 5, forcing the use of S-Expressions for every minor syntactic detail to be generated can become a nuisance. Luckily, Common Lisp's flexible reader can be used to strike a balance between having a macro-processable S-Expression language and supporting idiomatic C-isms.

With `(set-macro-character #\& #\'&-processor)`, for example, the reader can be extended to process symbols starting with an ampersand by applying the function *&-processor* to such symbols. This particular reader function sets up a specific mechanism that covers one single macro character. Therefore, it is usually required to set up functions for individual syntax elements that differ from S-Expressions, but this scheme is limited to elements that can be captured by such a simple prefix.

According to that, it is easy to implement a function that handles C syntax for the address-of operator. In that case the reader simply consumes, e.g., `(+ &a &b)` and emits `(+ (addr-of a) (addr-of b))`. Unfortunately, it is not easily possible to identify C++ references, for example in `(decl ((int& a)))`, since they can occur at the end of the corresponding symbol. In addition to that, we are limited to one single character. Therefore, we are unable to use the reader in that fashion for neither prefix increments nor decrements: `(+ ++a --b)`.

Surprisingly, there is a very simple, general solution for processing almost every type of symbol: hooking the reader macro to whitespace.

```
(set-macro-character #\Space   #'pre-process)
(set-macro-character #\Tab     #'pre-process)
(set-macro-character #\Newline #'pre-process)
```

This setup configures the reader to utilize the function *pre-process* to handle every symbol with a leading whitespace character. The task of *pre-process* is parsing individual symbols, identifying non-Lisp syntax and emitting proper S-Expressions. By doing this, we can support very convenient, but tricky C syntax, as shown in the following examples:

```
(* ++a[4] --b[x++])
→ (* (aref (prefix++ a) 4) (aref (prefix-- b) (postfix++ x)))

(+ foo[baz[1]][2][3] &qox)
→ (+ (aref (aref (aref foo (aref baz 1)) 2) 3) (addr-of quox))

(set foo->bar->baz 5)
→ (set (pref (pref foo bar) baz) 5)
```

So far we are able to process symbols as long as they have leading whitespace. However, list heads usually do not have leading whitespace, but begin directly after the opening parenthesis. These situations should also be managed, for example in calls of type `(obj->func args...)`. Therefore, we need to register an additional macro character:

```
(set-macro-character #\( #'pre-process-heads)
```

Contrary to the previous symbol processing, where each symbol is handled separately, the macro-character setup above requires us to imitate Common Lisp's standard mode of reading lists, which is easily achieved using `(read-delimited-list #\))` in order to get all list elements. Thereafter, *pre-process-heads* emits a slightly adjusted list comprised of the altered list head and the (untouched) remaining list elements. Eventually, the list's head has been adapted to our needs by *pre-process-heads* and the remaining list elements will be modified later-on by *pre-process* (as described above), if necessary.

Additionally, we might not want to process all list heads in general, but only those that are neither bound variables, nor functions or macros. This is due to the fact that valid Common Lisp macros, for example, can be named in a way that these reader macros would pick up on. In general, we opted to take the meaning defined in the host language for any ambiguous cases. Our approach to identify bound symbols is detailed in the next section.

One conflict that cannot be solved with the aforementioned reader still remains. Packages in Common Lisp are denoted similarly to namespaces in C++, but using the reader for this issue would break Common Lisp's package annotations. As an alternative, fully qualified symbols could be exploited for C++ namespaces:

```
(defpackage :N1)

(set N1::foo 4)
  →   N1::foo = 4;
```

This, however, does not support nesting of namespaces, since *nested packages* are not available in Common Lisp [21]. Naturally, the explicit form can always be utilized, but is very verbose:

```
(set (from-namespace N1 N2 foo) 4)
  →   N1::N2::foo = 4;
```

Therefore we apply *set-dispatch-macro-character* to introduce a specific annotation for C++ namespaces:

```
(set-dispatch-macro-character #\# #\: #'colon-reader)

(set #:N1::N2::var 4)
  →   (set (from-namespace N1 N2 var) 4)
  →   N1::N2::var = 4;
```

As a further convenience for Common Lisp users, our reader macro also supports the single-colon notation: `#:N1:N2:var`.

5.3 Adaptive Quotation

A crucial part of being able to write code as we claim in Section 3.2 is identifying which symbols are bound to host-language interpretations. One example is when it comes to using function calls for the target language in traditional Lisp notation: Instead of being forced to write `(funcall 'foo 1 2 3)` we want to support `(foo 1 2 3)`, even if the symbol *foo* is unbound and have it emit `foo(1, 2, 3)`.

The first attempt to realize the aforementioned notation has been the application of *boundp* and *fboundp*. Both functions work well for globally defined variables, functions, and macros:

```
(defvar foo 1)
(boundp 'foo) ;; -> T
```

However, they cannot be applied to symbols from lexical environments:

```
(let ((bar 1))
  (boundp 'bar)) ;; -> NIL
```

Since the naïve approach cannot handle such symbols, we had to look for an alternative. Every implementation of Common Lisp that supports lexical scoping has to keep track of bound symbols and their meaning. This information is stored in the *environment*, but not every implementation has a convenient method for accessing this data. In case of SBCL [15] and Clozure CL [3] we can exploit *function-information* to check whether a function is defined in the lexical or global scope. Similar to functions, we can use *variable-information* for symbols. For implementations that do not supply these functions, such as, for example, ECL [9], we have to implement a look-up in the environment object itself. The following example shows how one could utilize the listed functions and implement the missing look-up for ECL in order to retrieve information whether a symbols is bound or not.

```
(defun fboundp! (function &optional env)
  #+sbcl (sb-cltl2::function-information function env)
  #+clozure   (ccl::function-information function env)
  #+ecl   (or (fboundp function)
              (find function (rest env)
                :test #'(lambda (x y) (eql x (car y)))))
  #-(or sbcl clozure ecl) (error "..."))

(defun vboundp! (variable &optional env)
  #+sbcl (sb-cltl2::variable-information variable env)
  #+clozure   (ccl::variable-information variable env)
  #+ecl   (or (boundp variable)
              (find variable (first env)
                :test #'(lambda (x y) (eql x (car y)))))
  #-(or sbcl clozure ecl) (error "..."))
```

Due to the fact that these functions require access to the environment object, they can only be applied usefully inside macros. The following macro is a minimal example for a possible use of these functions:

```
(defmacro xboundp (item &environment env)
  (if (or (fboundp! item env)
          (vboundp! item env))
      t     ; item bound
      nil)) ; item unbound
```

With such functions at hand, we are now free to build a more flexible quotation scheme, specifically tailored to our meta language:

```
(defmacro quoty (item &environment env)
  (cond ((listp item)
         (if (fboundp! (first item) env)
             item
             `(function-call ...)))
        ((symbolp item)
         (if (vboundp! item env)
             item
             `',item))
        (t item)))
```

We can now add the *quoty* macro to the tree construction process (see also Section 3.3):

```
(defmacro + (lhs rhs)
  `(make-instance 'infix-node
                  :op '+
                  :lhs (quoty ,lhs)
                  :rhs (quoty ,rhs)))
```

This allows us to freely and seamlessly mix and match globally and lexically bound symbols and functions with unbound symbols taken to denote target-language functions and variables:

```
(labels ((foo (a b) (cl:+ a b)))
  (+ (foo 1 2) (bar 1 2)))

(labels ((foo (a b) (cl:+ a b)))
  (let ((x 5))
    (set A (+ (+ x y) (+ 'x (+ (foo 1 2) (bar 1 2)))))))
```

The arithmetic expression in the last line results in the following code for the target language:

```
A = 5 + y + x + 3 + bar(1, 2);
```

Quoty is most useful in special forms, where we do not want to quote every individual symbol, but still want to be flexible enough to call functions or use symbol values. Another example from C-Mera is that we utilize *quoty* in the variable-declaration macro, *decl*:

```
(decl ((const super_fancy_type x = 4)) ...)
→ const super_fancy_type x = 4; ...

(defmacro with-pointer (pointer &body body)
  `(decl (((postfix* ,pointer) x = (foo)))
     ,@body))

(with-pointer int ...)
→ int* x = foo(); ...
```

The flexible quotation allows us to use types (*super_fancy_type*) and functions (*foo*) that are not defined in the host language's context. Additionally we are now able to evaluate functions inside these *quasi-special* forms (*postfix** and *foo* in the example).

We have opted for this scheme to provide a simple syntax, even in the face of effects similar to *unwanted capture* (by definition of host functions).

6 CONCLUSION

In this paper we have presented many details on how we have constructed our meta language, ranging from Common Lisp implementation techniques to reader-macro hackery. Our pragmatic approach shows with how little effort Common Lisp can be bent toward our ends, resulting in an efficient meta-programming system for C-like languages.

We showed how our simple, Lisp-like notation can be evaluated to provide seamless integration with Common Lisp code during compilation, most notably with support for macros that are our primary vehicle for meta programming (this is also illustrated in our previous work on C-Mera). We also detailed the intricacies of our scheme, namely how to properly override built-in symbols while retaining their original interpretation in an accessible way, how to configure the Common Lisp system to keep our target language's case while not sacrificing a modern notation of the Common Lisp meta part of the language. We furthermore showed how we manage to provide many C-isms that programmers from that area would find awkward working without (and even seasoned Lisp users might miss for their conciseness), and how unnecessary quoting of unbound symbols can be avoided while keeping the Common Lisp interaction fully working.

Overall, none of these aspects are new findings. Our primary goal with this summary paper is to have all of this information collected in a single, clearly marked place. We hope this will help projects with similar demands to get up to speed more easily than when solutions to all of those issues have to found independently and without proper context.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the generous funding by the German Research Foundation (GRK 1773).

REFERENCES

[1] Andrei Alexandrescu. 2001. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley.
[2] Jonathan Carlos Baca. 2016. Lisp/c. https://github.com/eratosthenesia/lispc. (2016). GitHub Repository, Accessed Jan 2017, Active May 2016.
[3] Gary Byers. 2017. Clozure Common Lisp. http://ccl.clozure.com/. (2017). Accessed Jan 2017.
[4] Cristiano Calcagno, Walid Taha, Liwen Huang, and Xavier Leroy. 2003. Implementing Multi-stage Languages Using ASTs, Gensym, and Reflection. In *Proceedings of the 2nd International Conference on Generative Programming and Component Engineering (GPCE '03).* Springer-Verlag New York, Inc., New York, NY, USA, 57–76.
[5] Zachary DeVito, James Hegarty, Alex Aiken, Pat Hanrahan, and Jan Vitek. 2013. Terra: A Multi-stage Language for High-performance Computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13).* ACM, New York, NY, USA, 105–116.
[6] Andreas Fredriksson. 2010. Amplifying C. http://voodoo-slide.blogspot.de/-2010/01/amplifying-c.html and https://github.com/deplinenoise/c-amplify. (2010). Personal Blog & Github Report, Accessed Jan 2017, Repository active Feb 2010 – Mar 2010.
[7] de Guzman Joel, Kaiser Hartmut, and Nuffer Dan. 2016. Boost Spirit. http://www.boost.org/doc/libs/1_63_0/libs/spirit/doc/html/index.html. (2016). Accessed Jan 2017.
[8] S. C. Johnson. 1975. *YACC—yet another compiler-compiler.* Technical Report CS-32. AT&T Bell Laboratories, Murray Hill, N.J.
[9] Daniel Kochmański. 2017. Embeddable Common Lisp. https://common-lisp.net/project/ecl/main.html. (2017). Accessed Jan 2017.
[10] Jan Cornelis Willem Kroeze. 2010. *Tracing rays the past, present and future of ray tracing performance.* Ph.D. Dissertation. North-West University.
[11] M. E. Lesk and E. Schmidt. *Lex — A Lexical Analyzer Generator.* Technical Report. AT&T Bell Laboratories, Murray Hill, New Jersey 07974. PS1:16–1 – PS1:16–12 pages. http://kjellggu.myocard.net/misc/tutorials/lex.pdf
[12] Alexander Lier, Franke Linus, Marc Stamminger, and Kai Selgrad. 2016. A Case Study in Implementation-Space Exploration. In *Proceedings of ELS 2016 9th European Lisp Symposium.* 83–90.
[13] Nicholas D. Matsakis and Felix S. Klock, II. 2014. The Rust Language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology (HILT '14).* ACM, New York, NY, USA, 103–104.
[14] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.
[15] Christophe Rhodes. 2008. Self-Sustaining Systems. Springer-Verlag, Berlin, Heidelberg, Chapter SBCL: A Sanely-Bootstrappable Common Lisp, 74–86. DOI: http://dx.doi.org/10.1007/978-3-540-89275-5_5
[16] Vladimir Sedach. 2016. Parenscript. http://common-lisp.net/project/parenscript/. (2016). Accessed Jan 2017.
[17] Kai Selgrad, Alexander Lier, Jan Dörntlein, Oliver Reiche, and Marc Stamminger. 2016. A High-Performance Image Processing DSL for Heterogeneous Architectures. In *Proceedings of ELS 2016 9th European Lisp Symposium.* 39–46.
[18] Kai Selgrad, Alexander Lier, Franz Köferl, Marc Stamminger, and Daniel Lohmann. 2015. Lightweight, Generative Variant Exploration for High-Performance Graphics Applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2015).* ACM, New York, NY, USA, 141–150. DOI: http://dx.doi.org/10.1145/2814204.2814220
[19] Kai Selgrad, Alexander Lier, Markus Wittmann, Daniel Lohmann, and Marc Stamminger. 2014. Defmacro for C: Lightweight, Ad Hoc Code Generation. In *Proceedings of ELS 2014 7th European Lisp Symposium.* 80–87.
[20] Philipp Slusallek. 2015. Approaches for Real-Time Ray Tracing and Lighting Simulation. http://www.dreamspaceproject.eu/dyn/1429609964713/DREAMSPACE_-D4.1.1_Approaches_v1.3.pdf. (Jan. 2015).
[21] Alessio Stalla. 2017. Symbols as Namespaces. *ELS 2016 9th European Lisp Symposium*, Lightning Talks Session 1, https://www.european-lisp-symposium.org/editions/2016/lightning-talks-1.pdf. (May 2017).
[22] Masayuki Takagi. 2017. Cl-Cuda. https://github.com/takagi/cl-cuda. (2017). GitHub Repository, Accessed Jan 2017, Active Apr 2012 – Jan 2017.
[23] The Clang Developers. 2014. Clang: A C Language Family Frontend for LLVM. http://clang.llvm.org. (2014).
[24] David Vandevoorde and Nicolai M. Josuttis. 2002. *C++ Templates.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
[25] Todd Veldhuizen. 1995. Template Metaprograms. *C++ Report* (May 1995).

Static Taint Analysis of Event-driven Scheme Programs

Jonas De Bleser
Vrije Universiteit Brussel
jdeblese@vub.ac.be

Quentin Stiévenart
Vrije Universiteit Brussel
qstieven@vub.ac.be

Jens Nicolay
Vrije Universiteit Brussel
jnicolay@vub.ac.be

Coen De Roover
Vrije Universiteit Brussel
cderoove@vub.ac.be

ABSTRACT

Event-driven programs consist of event listeners that can be registered dynamically with different types of events. The order in which these events are triggered is, however, non-deterministic. This combination of dynamicity and non-determinism renders reasoning about event-driven applications difficult. For example, it is possible that only a particular sequence of events causes certain program behavior to occur. However, manually determining the event sequence from all possibilities is not a feasible solution. Tool support is in order.

We present a static analysis that computes a sound over-approximation of the behavior of an event-driven program. We use this analysis as the foundation for a tool that warns about potential leaks of sensitive information in event-driven Scheme programs. We innovate by presenting developers a regular expression that describes the sequence of events that must be triggered for the leak to occur. We assess precision, recall, and accuracy of the tool's results on a set of benchmark programs that model the essence of security vulnerabilities found in the literature.

CCS Concepts

•Theory of computation → Program analysis; •Security and privacy → *Software security engineering;*

Keywords

Taint Analysis, Abstract Interpretation, Static Program Analysis, Security Vulnerability, Event-driven Programs

1. INTRODUCTION

Event-driven programs are widely used on both client and server side where external events and their corresponding event listeners determine the program behavior. Analyzing such programs is hard because the order in which events occur is non-deterministic and control flow is not explicitly available. These problems negatively impact the ability of tools to detect security vulnerabilities. Among these vulnerabilities, leaks of confidential information and violations of program integrity remain a continuously growing problem [22].

Static taint analysis has been proposed to detect those vulnerabilities [3, 6, 8, 10, 18]. For *event-driven* programs, the state of the art in static taint analysis either completely ignores events or simulates them in every possible order. Another limitation of the state of the art is the lack of information about which event sequences cause a security vulnerability to occur. As a result, there is still little tool support available to precisely detect such defects.

In this work, we present a static taint analysis to compute the flow of values through event-driven programs in which program integrity or confidentiality of information is violated. We model a small event-driven Scheme language, Scheme$_E$, with an event model similar to JavaScript and support for dynamic prototype-based objects. Through *abstract interpretation* [4], we compute an over-approximation of the set of reachable program states. From this set, we identify security vulnerabilities and summarize event sequences causing these vulnerabilities. This paper makes the following contributions:

- We describe an approach to static taint analysis that is able to detect security vulnerabilities in higher-order, event-driven programs.
- We summarize event sequences leading to security vulnerabilities by means of regular expressions. This provides the developer with a description of the events that have to be triggered for a security vulnerability to occur, thereby facilitating the correction of the vulnerability.
- We investigate the use of k-CFA [15] as context sensitivity in event-driven programs. We measure the influence on the results of our analysis and how it affects the number of false positives.

2. TAINT ANALYSIS

A taint analysis is capable of detecting flows of values in a program that violate program integrity or confidentiality of information. Taint analysis defines such data-flow problems in terms of *sources*, *sinks* and *sanitizers*. A *source* is any origin of taint (e.g., user input or private information). A *sink* is any destination where taint is not allowed to flow to (e.g., database query or log utilities). A *sanitizer* converts taint in such a way that it is no longer considered to be

10th European Lisp Symposium ELS'17, Brussels, Belgium
© 2017 ACM. ISBN 123-4567-24-567/08/06.
DOI: 10.475/123_4

tainted (e.g., by stripping tags or by encryption). A security vulnerability occurs whenever taint flows from a source into a sink, without flowing through a sanitizer.

The program in Listing 1 contains a security vulnerability that leaks a password to the screen. The variable `password` is the source, as it is an origin of confidential information. The function `display` exposes its argument to the screen and is therefore a sink. The function `encrypt` is a sanitizer as it converts its argument to an encrypted equivalent that is allowed to be printed on screen. In this example, a leak of confidential information occurs whenever the second branch of the `if`-statement is executed. The goal of a static taint analysis is to detect this violation without executing the program.

```
(define password 'secret)
(define encrypt (lambda (x) (AES x)))
(if (> (length password) 10)
  (display (encrypt password))
  (display password))
```

Listing 1: Example of a security vulnerability that leaks the password to the screen.

2.1 Motivating example

To illustrate the problem we are addressing in this paper consider Listing 2, which exemplifies an event-driven program containing a security vulnerability. This example is written in Scheme$_E$ (Section 3) and represents a form that contains a text input listening to the `keypress` event, together with two buttons that respectively listen to the `clear` and `save` events.

```
1  (define o (object))
2  (define key #f)
3
4  (add-event-listener o 'keypress
5    (lambda (e) (set! key (source 'secret))))
6  (add-event-listener o 'clear
7    (lambda (e) (set! key #f)))
8  (add-event-listener o 'save
9    (lambda (e) (sink key)))
```

Listing 2: An event-driven program consisting of a security vulnerability that leaks confidential information.

We define three event listeners (lines 5, 7, and 9) that manipulate or access the variable `key` defined on line 2. The first event listener (line 5) sets the variable `key` to the value `secret`. This value represents confidential information and therefore is tainted. We indicate this by using the special form `source`, which returns its argument but annotates it with a taint flag behind the scenes. The second event listener (line 7) sets variable `key` to the untainted value `#f`. The third event listener (line 9) leaks the contents of `key` (e.g., prints it to the screen). This is indicated by means of the special form `sink`, which raises an error if its argument is tainted.

Each event listener is registered (lines 4, 6, and 8) on object `o` through the special form `add-event-listener`. This special form takes three arguments: the object on which the event listener is registered, the event that triggers the listener to be executed, and the function to execute when the event occurs.

Registration enables the event listeners to become executable, and in Listing 2 some execution orders may lead to security vulnerabilities. This is the case whenever a `keypress` event is immediately followed by a `save` event. By taking into account the execution of event listeners, there now exists a flow between the source (line 5) and the sink (line 9) through variable `key` (line 2). This flow causes the tainted value `secret` to be leaked.

However, information about the execution of event listeners is not explicitly available from the source code, making the flow between event listeners implicit. A naive abstract interpretation of event-driven programs that ignores the execution of event listeners will not detect that there is a flow from and to variable `key` from every event listener. As a result, such an analysis would not detect the leak of confidential information in Listing 2. We tackle this problem in the following sections.

3. AN EVENT-DRIVEN FLAVOR OF SCHEME

We start by introducing Scheme$_E$, the language on which we perform static taint analysis. Scheme$_E$ is a small Scheme language that supports higher-order functions, objects, events, and taint. The syntax of the language is shown in Figure 1.

$$
\begin{array}{rcl}
var \in Var & = & \text{a set of identifiers} \\
num \in \mathbb{N} & = & \text{a set of numbers} \\
str \in String & = & \text{a set of strings} \\
s \in Symbol & = & \text{a set of symbols} \\
b \in \mathbb{B} & ::= & \texttt{\#t} \mid \texttt{\#f} \\
l \in Lambda & = & (\texttt{lambda}\ (var)\ e_{body}) \\
e \in Exp & ::= & var \mid num \mid b \mid l \mid str \mid s \\
& \mid & (e_f\ e_{arg}) \\
& \mid & (\texttt{set!}\ var\ e_{val}) \\
& \mid & (\texttt{if}\ e_{cond}\ e_{cons}\ e_{alt}) \\
& \mid & (\texttt{letrec}\ ((var\ e))\ e_{body}) \\
& \mid & (\texttt{source}\ e_{val}) \\
& \mid & (\texttt{sanitizer}\ e_{val}) \\
& \mid & (\texttt{sink}\ e_{val}) \\
& \mid & (\texttt{object}) \\
& \mid & (\texttt{define-data-property}\ e_{obj}\ s_{\text{name}}\ e_{val}) \\
& \mid & (\texttt{define-accessor-property}\ e_{obj}\ s_{\text{name}}\ e_g\ e_s) \\
& \mid & (\texttt{get-property}\ e_{obj}\ s_{\text{name}}) \\
& \mid & (\texttt{set-property}\ e_{obj}\ s_{\text{name}}\ e_{val}) \\
& \mid & (\texttt{delete-property}\ e_{obj}\ s_{\text{name}}) \\
& \mid & (\texttt{event}\ s_{event}) \\
& \mid & (\texttt{add-event-listener}\ e_{obj}\ s_{event}\ e_{listener}) \\
& \mid & (\texttt{remove-event-listener}\ e_{obj}\ s_{event}\ e_{listener}) \\
& \mid & (\texttt{dispatch-event}\ e_{obj}\ e_{arg}) \\
& \mid & (\texttt{emit}\ e_{obj}\ e_{arg}) \\
& \mid & (\texttt{event-queue})
\end{array}
$$

Figure 1: Grammar of Scheme$_E$.

3.1 Objects and Properties

Scheme$_E$ supports Javascript-like objects consisting of properties that are maintained in a map relating property names to their respective values. There exist two kinds of properties in JavaScript: data and accessor properties. The former associate property names with values, while the latter associate them with a getter and setter function (i.e., allowing side-effects). The special form `object` instantiates an object without any properties. Accessing a property which does not exist on an object yields `#f` as default value. Properties can be added (`define-(data|accessor)-property`), accessed (`get-property`), deleted (`delete-property`), or modified (`set-property`) at run time.

3.2 Events and Event Listeners

Event listeners (also referred to as event handlers or callbacks) are functions that are registered for a specific event on an object and are executed whenever such an event is dispatched as a result of an action (e.g., clicking a button or pressing a key). In general, event-driven programs do not terminate because they listen for events indefinitely (i.e., they enter an *event loop*). The behavior of event-driven programs is largely determined by the execution of event listeners.

In Scheme$_E$, event listeners are added and removed by the special forms `add-event-listener` and `remove-event-listener`, respectively. New events can be created through the special form `event`, which takes a symbol denoting the event type as argument.

Listing 3 illustrates Scheme$_E$'s support for objects and events. Two properties are defined on object `o`: data property `var` (line 2) and accessor property `result` (line 3) with its getter and setter functions (lines 4 and 5). Accessing property `result` will return the value of `var` multiplied by 2, while setting it will cause `var` to be changed to the given value.

Two event listeners for events `modify` and `resets` are registered on object `o` (lines 7 and 9). The former event listener (line 7) modifies the accessor property `result` to become 2. The latter resets the property `var` so that its value becomes 0. To simulate events directly, we use the special form `dispatch-event`. This special form dispatches events *synchronously* and represents the occurrence of a particular event on an object at that specific moment in time in the program. As a result, the corresponding event listeners on the target object are immediately executed.

First, a `modify` event is dispatched (line 12) and causes the value of `var` to become 2. Accessing `result` on the next line will therefore return 4. Second, a `reset` event is dispatched (line 15) and causes the value of `var` to become 0, as indicated on line 16. Third, the event listener registered for `modify` event is removed on line 18. Any event that occurs after the removal has no effect. This is reflected by the value of `result`, because accessing it on line 20 still results in the value 0 instead of 4. Finally, the property `var` is removed from the object on line 22, and accessing it returns the default value `#f`.

```
1   (define o (object))
2   (define-data-property o 'var 0)
3   (define-accessor-property o 'result
4     (lambda () (* 2 (get-property o 'var)))
5     (lambda (x) (set-property! o 'var x)))
6
7   (define modify (lambda () (set-property o 'result 2)))
8   (add-event-listener o 'modify modify)
9   (define reset (lambda () (set-property o 'var 0)))
10  (add-event-listener o 'reset reset)
11
12  (dispatch-event o (event 'modify)) ; var = 2
13  (get-property o 'result) ; 4
14
15  (dispatch-event o (event 'reset)) ; var = 0
16  (get-property o 'result) ; 0
17
18  (remove-event-listener o 'modify modify)
19  (dispatch-event o (event 'modify)) ; NOP
20  (get-property o 'result) ; 0
21
22  (delete-property o 'var)
23  (get-property o 'var) ; #f
```

Listing 3: Example program illustrating the features of Scheme$_E$.

3.3 Taint

Besides objects and events, Scheme$_E$ features special forms to explicitly define sources (`source`), sinks (`sink`), and sanitizers (`sanitizer`). Listing 4 shows how to use these to define that variable `v` is a source, function `display` is a sink, and `encrypt` is a sanitizer.

```
(define v (source 1))
(define display (lambda (x) (sink x)))
(define encrypt (lambda (x) (sanitizer x)))
```

Listing 4: Defining sources, sinks and sanitizers.

4. STATIC TAINT ANALYSIS OF EVENT-DRIVEN PROGRAMS

We explain how to detect vulnerabilities through abstract interpretation (Section 4.1), how to model such vulnerabilities in event-driven programs (Section 4.2), and how to report them in a user-friendly way through regular expressions describing event sequences (Section 4.3).

4.1 Abstract interpretation in the context of event-driven programs

To statically analyze event-driven programs, we perform abstract interpretation using the technique of Abstracting Abstract Machines (AAM) [21]. From the operational semantics of Scheme$_E$ defined as an abstract machine, we derive an *abstract* version of this semantics as an *abstract abstract machine*. This machine can then be used to perform abstract interpretation of event-driven programs. The result of an abstract interpretation is an *abstract state graph* in which nodes represent program states and edges represent transitions between program states. This graph contains every possible program behavior that can occur during concrete interpretation, but possibly also spurious behavior due to over-approximation. A single abstract state can represent multiple concrete states and is either the evaluation of an expression, the result of an evaluation, or an error state. Figure 2 depicts a fragment of an example abstract state graph.

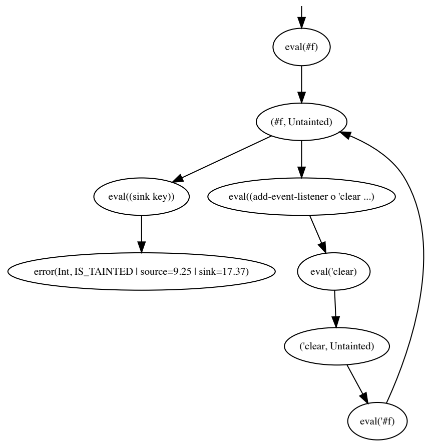

eval(#f)
(#f, Untainted)
eval((sink key))
eval((add-event-listener o 'clear ...)
error(Int, IS_TAINTED | source=9.25 | sink=17.37)
eval('clear)
('clear, Untainted)
eval('#f)

Figure 2: Abstract state graph resulting from abstract interpretation.

To detect security vulnerabilities, the abstract interpretation keeps track of the flow of tainted values. An error state is generated whenever a tainted value reaches a sink. For example, an error state is represented as the left leaf node in Figure 2. This state provides information about the tainted value such as the type of the value (i.e., `Int`), the line and column number of the source (i.e., `source=9.25`) and the sink (i.e., `sink=17.37`) involved in the vulnerability, together with the precision with which the vulnerability was detected. If the analysis detects a taint violation with full precision, the error `IS_TAINTED` is produced. If it detects that a taint violation *may* occur, the error `MAYBE_TAINTED` is produced instead.

As illustrated in Section 2.1, naively performing abstract interpretation of event-driven programs may miss security vulnerabilities if events are not taken into account. By ignoring events, the resulting abstract state graph does not contain behaviors related to the execution of event listeners. Such a graph is an *unsound* approximation of the program behavior and may not contain every security vulnerability present in the program.

In order to obtain a sound static analysis for event driven programs, it is crucial to execute event listeners. A sound static analysis ensures that a given event-driven program is free of vulnerabilities under *any* input and sequence of events if the analysis detects no possible vulnerability during abstract interpretation.

We describe our approach to simulate events in the next section.

4.2 Simulating events

Scheme$_E$ provides the special form `emit` to *asynchronously* trigger an event. As opposed to `dispatch-event`, any corresponding event listeners are not directly executed. Instead, the emitted event is scheduled in the global *event queue* and program execution continues right after the call to `emit`. At the end of the program, when the call stack is empty, `(event-queue)` is used to initiate the event loop. The event loop continually extracts a single event from the queue and executes its corresponding registered event listeners in registration order.

Because the registered event listeners are executed according to which event is consumed from the event queue during abstract interpretation, the final abstract state graph includes the behavior of the event listeners. This approach enables us to detect security vulnerabilities, including the ones that occur as a result of program flow through event listeners. To keep track of which event has been executed, we annotate the edges of the abstract graph with information about the triggered event. This information is used in a later stage (Section 4.3) to generate regular expressions that describe event sequences leading to a particular security vulnerability. We call the resulting graph an *abstract event graph*.

Because the order in which events are triggered is largely non-deterministic, a naive approach is to assume that every event can be triggered in any order at any time. However, it is computationally expensive to explore the whole event space for non-trivial programs in this manner, and it may result in many false positives. This can be mitigated partially with domain knowledge about the semantics of the program and how events occur. Madsen et al. [11] presents a modeling approach to support events and provide the abstract semantics of an event queue. We follow their approach in order to reduce the search space by explicitly emitting events in the program.

We extend our motivational example (Listing 2) with a model that indicates which events can occur and when they can occur. Listing 5 depicts such a model where the events `clear` and `save` can never occur at the start of the program. That is because buttons registered for these events are disabled as long as there has not been any `keypress` event. We specify this behavior by only emitting a `keypress` event at line 18, before calling `event-queue`. Whenever a `keypress` event has been triggered, any other type of event can occur. We model this by emitting every event (lines 6–8) in the event listener registered for `keypress`. We also model that a `save` event can never occur after a `clear` event. This is specified by only emitting `keypress` and `clear` at lines 12 and 13 in the event listener registered for `clear`. Finally, we model the fact that a `save` event implies termination of the program by not emitting any event from the listener at line 16. We consider termination to be the disappearance of the form once the save button is pressed. While explicit event modeling requires some effort, it reduces the search space and avoids exploring spurious event sequences.

```
1   (define o (object))
2   (define key #f)
3
4   (add-event-listener o 'keypress
5     (lambda ()
6       (emit o (event 'keypress))
7       (emit o (event 'clear))
8       (emit o (event 'save))
9       (set! key (source 'secret))))
10  (add-event-listener o 'clear
11    (lambda ()
12      (emit o (event 'keypress))
13      (emit o (event 'clear))
14      (set! key #f)))
15  (add-event-listener o 'save
16    (lambda () (sink key)))
17
18  (emit o (event 'keypress))
19  (event-queue)
```

Listing 5: An event-driven program consisting of explicitly modeled event sequences and a leak of confidential information.

While detecting security vulnerabilities in event-driven programs is important, knowing *why* and *how* they occur is at least as important. Manually inspecting the abstract event graph for event sequences of interest is not an option, given the complexity of event-driven programs and their resulting event graphs. We tackle this problem in the next section.

4.3 Computing event sequences

Manually deriving event sequences that lead to security vulnerabilities is not trivial. This is because programs typically consist of many events, as well as many sources, sanitizers, and sinks. We address this problem by automatically generating regular expressions describing the sequence of events required for a security vulnerability to occur. Event sequences provide valuable information to developers detecting and fixing these vulnerabilities.

We start from the observation that the abstract event graph is equivalent to a non-deterministic finite automaton with ϵ-transitions (ϵ-NFA). This because each state can have zero, one, or more successor states, and non-annotated edges in the graph (i.e., control-flow not induced by triggering of events) correspond to ϵ-transitions.

This automaton $(Q, \Sigma, \delta, q_0, F)$ consists of the set of abstract states Q, a transition function $\delta(q, a)$ where $q \in Q$ and $a \in \Sigma$ is either an event or ϵ. The initial state q_0 is the root state of the abstract state graph, and the set of final states F includes every error state.

This observation enables us to convert the abstract event graph to regular expressions in three steps:

1. Convert the ϵ-NFA to an NFA by calculating the ϵ-closure for each state.
2. Convert the NFA to a minimal deterministic finite automaton (DFA).
3. Convert the DFA to regular expressions for every combination of source and sink.

Conversion to NFA.

The function $ECLOSURE(Q) = \{s \mid q \in Q \wedge s \in \delta(q, \epsilon)\}$ calculates the ϵ-closure for each state of the automaton. Given this information, we can eliminate all ϵ-transitions because they do not contribute to the final regular expression. This step results in an ϵ-free NFA and reduces the number of states because most transitions are indeed ϵ-transitions.

Conversion to minimal DFA.

Any NFA can be converted into its corresponding unique minimal DFA [16]. We opt for Brzozowski's algorithm to perform this conversion because it outperforms other algorithms in many cases [2], despite its exponential character. This algorithm minimizes an ϵ-free NFA into a minimal DFA where both automatons accept the same language L. It does so by reversing the directions of the transitions in an NFA (*rev*), and then converting it into an equivalent DFA that accepts the reverse language L^R using the powerset construction method (*dfa*). The process is repeated a second time to obtain a minimalistic DFA that accepts language L. This algorithm is performed by the function *minimize*.

$$minimize(fa) = dfa(rev(dfa(rev(fa))))$$

Extracting regular expressions.

Given a minimal DFA, we can convert it into a regular expression using several methods [14]. We opt for the *transitive closure method* because of its systematic characteristic. First, an $n \times n \times n$ matrix from a given DFA $\langle Q, \Sigma, \delta, q_0, F\rangle$ with n states is built. We define $R^k_{i,j}$ as the regular expression for the words generated by traversing the DFA from state q_i to q_j while using intermediate states $\{q_1 \dots q_k\}$. We compute this regular expression in every iteration from 1 to k as follows:

$$R^k_{i,j} = R^{k-1}_{i,j} + R^{k-1}_{i,k} \;.\; R^{k-1^*}_{k,k} \;.\; R^{k-1}_{k,j}$$

The final outcome $R^k_{i,j}$ is the regular expression that describes all the event sequences that lead to a particular security vulnerability.

We apply these steps to the example described in Section 4.2 and show the resulting regular expressions below. The first regular expression (1) is generated from the program using the naive approach in which every possible event ordering is explored. The second regular expression (2) is generated by means of the model using explicitly modeled event sequences (Listing 5). We abbreviate the events to their first letter for brevity. A + indicates choice, . indicates concatenation, and * indicates repetition (Kleene star operator). From both expressions, it is clear that the event sequence leading to the leak *ends* with a `keypress` event followed by a `save` event.

Figure 3 depicts the second regular expression as an automaton.

$$(k + ((c + s).(c + s)^*.k)).(k + (c.((c + s)^*.k)))^*.s \quad (1)$$

$$(k.k^*.c.(c + (k.k^*.c + c^*.k.k^*.c))^*.(c^* + k)^*)^*.k.k^*.s \quad (2)$$

Even in this simple example the generated event sequences are rather long and complex. While all possibilities are important (e.g. when multiple unique event sequences may lead to a leak), we deem the shortest possible event sequence to be the most important one in order to patch the security vulnerability. In this example, this is the sequence `keypress.save`.

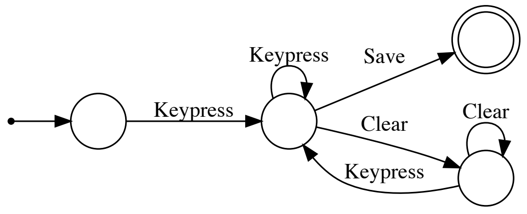


Figure 3: Finite state automaton of the regular expression that represents event sequences leading to the security vulnerability in Listing 5.

5. IMPLEMENTATION

We implemented the static taint analysis for event-driven programs discussed in this paper as a proof of concep[1]. We make use of the modular framework SCALA-AM [17] to perform static analysis based on systematic abstraction of abstract machines (AAM) [21]. The implementation supports Scheme$_E$, our small Scheme language with support for prototype-based objects, events, and taint that we described in Section 3. We incorporated an existing library for the manipulation of finite state automata [13] to obtain a minimal DFA from an abstract event graph by computing the ϵ-closure and applying the DFA minimization as described in Section 4.3.

Abstract counting [12] is enabled by default in our implementation. This to improve precision by avoiding unnecessary *joining* of values when it is safe to do so. Under abstraction, the abstract machine represents the *unbounded* heap memory by a map that relates a *finite* number of addresses to values. This results in possibly different values being allocated at the same *abstract* addresses. Such values are then *joined* in order to remain over-approximative in the interpretation. Suppose variable `x` is allocated at address a and represents a tainted value. Allocating a variable `y` representing an untainted value at the same address a will cause the values of `x` and `y` to join (i.e., to merge), so that the value at address a now *may* be tainted. This over-approximated

[1]https://github.com/jonas-db/aam-taint-analysis

value is then used in the remainder of the interpretation and may lead to false positives.

6. PRELIMINARY EXPERIMENTS

To measure the applicability of our approach, we extracted synthetic benchmarks from larger programs. These benchmarks are described in Table 1. We include multiple benchmarks that contain no security violation to assess to which extent our approach produces false positives. These benchmarks therefore enable us to determine the precision of our analysis. We investigate the increase of precision and accuracy that follows from the use of call-site-sensitivity (k-CFA) [15] as context sensitivity for the analysis. This context sensitivity is well-suited for programs with functions calls, and we observed that it is not uncommon for event listeners to call auxiliary functions.

The results of our experiments are shown in Table 1. The time it takes to produce these results is at most 1.3 seconds. Our analysis did not have any false negatives because it is sound, and thus has a recall of 100%. For each benchmark, we indicate whether the analysis detected a leak with maximal precision (*Must*) or not (*May*), or whether it detected no leak (/). We also indicate whether the results are correct (✓) or not (✗). For brevity, we only provide the shortest event sequence instead of the full regular expression. We conclude from this table that increasing the context sensitivity (i.e., increasing the value of k) results in less false positives in our experiments, while also increasing the precision and accuracy with which leaks are detected in the two first benchmarks. However, the analysis was unable to detect that benchmark programs `manylst` and `delprop` do *not* contain a leak. These false positives are a consequence of over-approximations due to abstract interpretation, and are a common side-effect of static analysis in general. However, the developer can be certain of the absence of security vulnerabilities by *only* verifying (e.g., simulating with increased polyvariance or manual inspection) the generated set of event sequences, as opposed to *every* possible combination of events. We discuss `lstfunc` and `rmlst` to understand how context sensitivity can contribute to less false positives and more precise results.

Context sensitivity and its influence on results.

Listing 6 shows the example `lstfunc` in which we model the scenario where event `a` is emitted, followed by event `b`. Each event listener calls the function `f`, which is a known sink. The first call to `f` on line 7 with untainted value 2 does not result in a security vulnerability. The second call on line 4, however, does result in a security leak.

```
1  (define window (object))
2  (define f (lambda (x) (sink x)))
3  (add-event-listener window 'b
4    (lambda (e) (f (source #f))))
5  (add-event-listener window 'a
6    (lambda (e)
7      (f 2)
8      (emit window (event 'b))))
9  (emit window (event 'a))
10 (event-queue)
```

Listing 6: `lstfunc` example, containing a security vulnerability.

Our analysis detects this leak but not with full precision. This because the parameter `x` is allocated twice to the same address, which causes the untainted value 2 and the tainted value `#f` to join. Hence, our monovariant analysis ($k = 0$) detects that `x` *may* be tainted. On the other hand, our polyvariant analysis with $k \geqslant 1$ is able to distinguish between the two calls to `f` and detects the security leak with full precision.

Listing 7 shows `rmlst` where a finite event sequence `a.b.c.d` is modeled. The example consists of multiple event listeners, each registered for one specific event. The problematic event listener defined on line 7 is registered for the event `d` on line 14 and could cause a leak if the property `p` of `oa` is tainted. However, this listener is removed on line 25 before any vulnerability can occur (i.e., this example does *not* contain a leak).

Without context sensitivity (0-CFA), a leak is detected by the analysis. The reason is related to the allocation and subsequent *joining* of objects, even though this occurs in two different event listeners (line 10 and 18). Two objects are joined whenever they are allocated at the same address, and a new abstract object is created that represents all properties and all registered event listeners of both objects.

Function `f` (line 4) calls function `g` (line 3) which allocates a new object. With 1-CFA, the analysis can differentiate between the two different calls to `f` on line 11 and 19, but not between the calls to `g` performed by `f`. Because of this, the allocation of the second object (by means of calling `f` on line 19) will join with the previously allocated object (by means of calling `f` on line 11). The definition of the tainted property `p` on line 12 will thus apply to both objects. Note that `o1` is aliased by `oa` which means that, due to joining, `oa` will point to *both* objects.

Whenever the event `c` occurs, the event listener is removed from the object `oa`. This is not the case when the object points to an address that represents multiple objects because removing it would be unsound since we do not know which event listener was registered to which object. As a result, emitting the event `d` on line 25 causes the event listener on line 7 to execute. the tainted property `p` of object `oa` reaches the sink.

```
1  (define oa #f)
2  (define window (object))
3  (define (g) (object))
4  (define (f) (g))
5
6  (define listener
7    (lambda (e) (sink (get-property oa 'p))))
8
9  (add-event-listener window 'a
10   (lambda (e)
11     (let ((o1 (f)))
12       (set-property o1 'p (source 1))
13       (add-event-listener o1 'd listener)
14       (set! oa o1)
15       (emit window (event 'b)))))
16
17 (add-event-listener window 'b
18   (lambda (e)
19     (f)
20     (emit window (event 'c))))
21
22 (add-event-listener window 'c
23   (lambda (e)
24     (remove-event-listener oa 'd listener)
25     (dispatch-event oa (event 'd))))
26
27 (emit window (event 'a))
28 (event-queue)
```

Listing 7: `rmlst` example, where no leak is present due to the removal of an event listener.

						0-CFA		1-CFA		2-CFA	
Name	Description	LOC	Leaks	Listeners	Emits	Result	Regex	Result	Regex	Result	Regex
`lstfunc`	Event listeners call the same function	9	1	2	2	May ✓	ba	Must ✓	ba	Must ✓	ba
`samelst`	Same event listener for different events	22	1	2	2	May ✓	ba	Must ✓	ba	Must ✓	ba
`nestedlst`	Event listener calls nested function	14	0	2	2	May ✗	ba	/ ✓	/	/ ✓	/
`objjoin`	Event listeners call factory method	28	0	4	4	May ✗	abcd	May ✗	abcd	/ ✓	/
`delprop`	Event listeners delete object property	21	0	2	2	May ✗	aa	May ✗	aa	May ✗	aa
`funccalls`	Nested registration of event listeners	11	0	2	2	May ✗	ab	/ ✓	/	/ ✓	/
`rmlst`	Removing an event listener	26	0	4	4	May ✗	abcd	May ✗	abcd	/ ✓	/
`manylst`	Multiple event listeners for an event	16	0	2	3	May ✗	ac	May ✗	ac	May ✗	ac
					Precision	25%		33%		50%	
					Recall	100%		100%		100%	
					Accuracy	25%		50%		75%	

Table 1: Precision, recall, and accuracy with k-CFA. *Result* describes whether the result is detected with full precision (*Must*) or not (*May*). It also indicates whether the result is correct (✓) or not (✗). *Regex* is the shortest event sequence that leads to the security vulnerability. The use of / means that no regular expressions are generated and thus no vulnerabilities were found. The gray areas indicate correct results and shows how precision increases with increased context sensitivity.

With 2-CFA, the analysis can distinguish between the two calls to `g`, and it correctly detects that there are no leaks.

7. RELATED WORK

To the best of our knowledge there exists no precise static taint analysis to detect security vulnerabilities in the context of higher-order event-driven programs written in a dynamically-typed language. Arzt et al. [3] present FlowDroid, a static taint analysis for Android based on the static analysis framework SOOT [20]. It is aware of the event-driven lifecycle of Android and user-defined event listeners. However, their approach does not support higher-order functions. Jovanovic et al. [8] introduce Pixy which aims to detect cross-site scripting vulnerabilities (XSS) in PHP 4. However, they do not support objects, events or higher-order functions. Guarnieri et al. [6] present Actarus, a blended (i.e., a combination of static and dynamic) taint analysis for JavaScript to detect client-side vulnerabilities. Their approach is based on the static analysis framework WALA [5]. However, being a blended analysis, it depends on run-time information. Tripp et al. [18] present TAJ, a static taint analysis for Java 6 without support for higher-order functions. It targets four security vulnerabilities in web applications, including cross-site scripting (XSS), command injection, malicious file executions and information leakage.

There is existing work related to static analysis of event-driven JavaScript programs using abstract interpretation. Liang and Might [10] present a static taint analysis for Python using abstract interpretation. However, their analysis does not support event-driven programs. Another difference is that we opt to maintain a product of abstract values and taint in a single abstract store instead of using two separate stores. Tripp et al. [19] present Andromeda, a demand-driven analysis tool for Java, JavaScript and .NET that has been successfully used in a commercial product, but has no support for event-driven programs. Jensen et al. [7] present TAJS which is a tool to detect type-related and data-flow related programming errors in event-driven JavaScript web applications. It is capable of detecting the absence of object properties or unreachable code and has support for the HTML DOM. While the tool has support for events, it does not track each individual event separately. Their approach consists of merging events in several categories such as load, mouse, keyboard, etc. This decision is a trade-off in terms of performance but leads to less precise event information. Kashyap et al. [9] present JSAI, a tool with support for the HTML DOM and events. We notice that both TAJS and JSAI simulate an event queue where event listeners are executed in every possible order. To avoid exploring the complete search space of events, Madsen et al. [11] present a modeling approach to support events. They do not implement a taint analysis but rather focus on detecting dead event listeners, dead emits and mismatched synchronous and asynchronous calls in Node.js. To model event-driven programs, they require the developer to explicitly place emit statements in the program. The proposed abstract event queue will then be filled with these events and enables the tool to explore the flow of events. While this approach leads to a smaller search space, it requires some knowledge about the semantics of the program. Nevertheless, this work inspired our implementation of an event queue used in our abstract machine.

8. CONCLUSION AND FUTURE WORK

In this work, we outline an approach to statically detect security vulnerabilities in event-driven Scheme programs. We propose the event-driven language Scheme$_E$ and use abstract interpretation as a technique to compute an over-approximation of the program's behavior. We use a three-step process to generate regular expressions that describe event sequences that lead to a particular security vulnerability. Event sequences provide valuable information to developers detecting and fixing these vulnerabilities. We also investigate the effect of context sensitivity, more precisely k-CFA, on the results of the analysis. Our results show that our technique can detect security vulnerabilities in event-driven programs and that higher precision can be achieved with increased context sensitivity.

As future work, we envision to investigate techniques that are able to avoid exploration of spurious event sequences. We also want to implement our technique for JavaScript. We deem this language support to be a continuation of our work because we closely followed the semantics of objects and events in JavaScript. For larger programs, we foresee that the size of the abstract state graph grows rapidly because many event sequences have to be explored. The size could be reduced by applying a macro-step evaluation [1] (i.e., a single node per event listener that may consist of multiple states) instead of a small-step evaluation (i.e., a single node per state). Another improvement can be to avoid the exploration of all permutations of event listeners. However, according to Madsen et al. [11] it is uncommon for a single object to have multiple event listeners registered for the same event. Madsen et al. [11] also proposes two context sensitivities specific to event-driven programs. We will implement these as future work and investigate whether one of the context sensitivities (or a combination thereof) can further improve the results of the analysis. The concept of event bubbling and event capturing is another problem that affects program security, but requires a hierarchical relationship between objects.

Although our approach is able to detect security vulnerabilities in event-driven programs, the non-deterministic behavior of events remains a computational challenge that influences the ability to detect vulnerabilities. Techniques are needed to reduce the search space and to further improve the precision of taint analysis in event-driven programs. Our work provides foundations toward this goal.

Acknowledgements

Quentin Stiévenart is funded by the *GRAVE* project of the Research Foundation - Flanders (FWO). Jens Nicolay is funded by the SeCloud project sponsored by Innoviris, the Brussels Institute for Research and Innovation.

References

[1] Gul A Agha, Ian A Mason, Scott F Smith, and Carolyn L Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(01):1–72, 1997.

[2] Marco Almeida, Nelma Moreira, and Rogério Reis. On the performance of automata minimization algorithms. In *Proceedings of the 4th Conference on Computation in Europe: Logic and Theory of Algorithms*, pages 3–14, 2007.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *ACM SIGPLAN Notices*, volume 49, pages 259–269. ACM, 2014.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[5] Stephen Fink and Julian Dolby. WALA – T.J. Watson libraries for analysis., 2006. `http://wala.sourceforge.net`.

[6] Salvatore Guarnieri, Marco Pistoia, Omer Tripp, Julian Dolby, Stephen Teilhet, and Ryan Berg. Saving the world wide web from vulnerable javascript. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 177–187. ACM, 2011.

[7] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *Proceedings of 16th International Static Analysis Symposium (SAS)*, volume 5673 of *LNCS*. Springer-Verlag, August 2009.

[8] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[9] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. Jsai: A static analysis platform for javascript. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 121–132. ACM, 2014.

[10] Shuying Liang and Matthew Might. Hash-flow taint analysis of higher-order programs. In *Proceedings of the 7th Workshop on Programming Languages and Analysis for Security*, page 8. ACM, 2012.

[11] Magnus Madsen, Frank Tip, and Ondřej Lhoták. Static analysis of event-driven node. js javascript applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 505–519. ACM, 2015.

[12] Matthew Might and Olin Shivers. Improving flow analyses via γcfa: abstract garbage collection and counting. In *ACM SIGPLAN Notices*, volume 41, pages 13–25. ACM, 2006.

[13] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.

[14] Christoph Neumann. Converting deterministic finite automata to regular expressions, 2005. `http://liacs.leidenuniv.nl/ bonsanguem-m/FI2/DFA_to_RE.pdf`.

[15] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie-Mellon University, 1991.

[16] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[17] Quentin Stiévenart, Maarten Vandercammen, J Nicolay, W De Meuter, and C De Roover. Scala-am: A modular static analysis framework. In *Proceedings of the 16th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM*, volume 16, 2016.

[18] Omer Tripp, Marco Pistoia, Stephen J Fink, Manu Sridharan, and Omri Weisman. Taj: effective taint analysis of web applications. *ACM Sigplan Notices*, 44 (6):87–97, 2009.

[19] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. Andromeda: Accurate and scalable security analysis of web applications. In *International Conference on Fundamental Approaches to Software Engineering*, pages 210–225. Springer, 2013.

[20] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot – a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[21] David Van Horn and Matthew Might. Abstracting abstract machines. In *ACM Sigplan Notices*, volume 45, pages 51–62. ACM, 2010.

[22] Dave Wichers. OWASP top ten project – a list of the 10 most critical web application security risks, 2013. `https://www.owasp.org/index.php`.

Session VI: Languages and meta-languages (2)

+

on the lambda way

Alain Marty Engineer Architect
Villeneuve de la Raho, France
marty.alain@free.fr

ABSTRACT

The {lambda way} project is a web application built on two engines:

- {lambda tank}, a tiny wiki built as a thin overlay on top of any web browser,
- {lambda talk}, a purely functional language unifying writing, styling and scripting in a single and coherent Lisp-like syntax.

In this document we progressively introduce the language beginning with {lambda word} built on a minimal set of three rules, then we add a bit of arithmetic with {lambda calc} and finally call the browsers' functionalities leading to a programmable programming language, {lambda talk}. As a guilding line, we present how can be computed the factorial of any natural number, at each level, regardless of its size and with a total precision, for instance:

```
5!  = 120
50! = 304140932017133803983857288678599
      00744706627746248466026044200000
```

KEYWORDS

- Information systems~Wikis
- Theory of computation~Regular languages
- Theory of computation~Lambda calculus
- Software and its engineering~Functional languages
- Software and its engineering~Extensible Markup Language (XML)

INTRODUCTION

« *But there are hundred of wiki engines and hundred of languages! Why yet another wiki and another language nobody will ever want to use?* » Let's talk about it!

Web browsers give everybody an easy access to a plethora of rich documents created by people mastering HTML, CSS, JS, PHP... Web browser can also host web applications allowing everybody to write HTML/CSS/JS code and so add informations to web pages. **Wikis** belong to this category: « *A wiki is a web application which allows collaborative modification, extension, or deletion of its content and structure.*[1] »

Writing HTML/CSS/JS code being rather complex and at least tiresome, intermediate syntaxes, for instance **WikiText**[2], have been created to make enriching and structuring text a little bit easier. And it's exactly what people use in blogs and wikis. The best known of wikis is Wikipedia, full of rich documented pages written by people supposed to be neither web designers nor coders.

Everything works well but the underlying code is a very obfuscated text, difficult to write, read, edit and maintain. In fact, the WikiText syntax is not intended for writing rich documents, not to speak of coding. Works have been done to build enhanced syntaxes in order to unify writing, styling and coding, for instance, not to mention desktop tools like **LaTeX**[3], web tools like **CURL**[4], **LML**[5], **Skribe**[6], **Scribble**[7], **SXML**[8], **LAML**[9], **Pollen**[10] ... But these tools are definitively devoted to coders, not to web designers and even less to beginners. Hence the **{lambda way}** project ...

We will forget the PHP engine,{lambda tank}, whose task is to manage the text files on the server side, and we progressively introduce the Javascript engine, {lambda talk}, working on the client side. Using exclusively the {lambda way} environment, 1) we begin to build {lambda word} on a minimal set of three rules, 2) we add a bit of arithmetic with {lambda calc} and finally 3) we use the browser's functionalities leading to {lambda talk}.

1. {LAMBDA WORD}

We present the structure and evaluation of a {lambda word} expression, then the implementation of the evaluator built on the underlying engine, Javascript.

1.1. Structure & Evaluation

{lambda word} is built on three rules freely inspired by the **lambda calculus** [11]. An expression is defined recursively as follows:

```
expression := [word|abstraction|application]*
where
  - word:        [^\s{}]*
  - abstraction: {lambda {word*} expression}
  - application: {expression expression}
```

A {lambda word} expression is a tree structure made of `words`, `abstractions` and `applications` where 1) a `word` is any character except spaces "\s" and curly braces "{}", 2) an `abstraction` is the "process" (called a *function*) selecting a sequence of `words` (called *arguments*) in an `expression` (called *body*), 3) an `application` is the "process" calling an `abstraction` to replace selected `words` by some other `words` (called *values*). The evaluation of an expression follows these rules:

- 1) a word is not evaluated,
- 2) an abstraction is evaluated to a single word, as a reference stored in a global dictionary,
- 3) an application is evaluated to a sequence of words,
- 4) abstractions, called "special forms", are evaluated before applications, called "forms".

Examples:

```
1) Hello World -> Hello World
2) {lambda {o a} oh happy day!} -> lambda_5
3) {{lambda {o a} oh happy day!}
 oOOOo aaAAaa} -> oOOOoh haaAAaappy daaAAaay!
```

In the last example the `abstraction` is first evaluated, defining "`o`" and "`a`" as characters whose occurences in the expression "`oh happy day!`" will be replaced by some future values, and returning a reference, "`lambda_5`" ; the `application` gets the awaited values "`oOOOo`" and "`aaAAaa`", calls the `abstraction` which makes the substitution and returns the result, "`oOOOoh haaAAaappy daaAAaay!`". Let's look at a more interesting example:

```
4) {{lambda {z} {z {lambda {x y} x}}}
    {{lambda {x y z} {z x y}} Hello World}}
-> Hello
   {{lambda {z} {z {lambda {x y} y}}}
    {{lambda {x y z} {z x y}} Hello World}}
-> World
```

Let's trace the first line returning "Hello":

```
1: {{lambda {z} {z {lambda {x y} x}}}
    {{lambda {x y z} {z x y}} Hello World}}
2: {{lambda {z} {z {lambda {x y} x}}}
    {lambda {z} {z Hello World}}}
3: {{lambda {z} {z Hello World}}
    {lambda {x y} x}}
4: {{lambda {x y} x} Hello World}
5: Hello
```

In fact, without naming them, we just have built and used a set of useful functions: `[CONS CAR CDR]`. This is a last example:

```
5) {{lambda {:n} {{lambda {:p} {:p {lambda {:x
:y} :y}}} {{:n {lambda {:p} {{lambda {:a :b :m}
{{:m :a} :b}} {{lambda {:n :f :x} {:f {{:n :f}
:x}}} {{lambda {:p} {:p {lambda {:x :y} :x}}}
:p}} {{lambda {:n :m :f} {:m {:n :f}}} {{lambda
{:p} {:p {lambda {:x :y} :x}}} :p} {{lambda {:p}
{:p {lambda {:x :y} :y}}} :p}}}}} {{lambda {:a
:b :m} {{:m :a} :b}} {lambda {:f :x} {:f :x}}
{lambda {:f :x} {:f :x}}}}}}} {lambda {:f :x} {:f
{:f {:f {:f {:f :x}}}}}}} -> lambda_200
```

Let's look at the end part of the expression, `{:f {:f {:f {:f {:f :x}}}}}`. We notice that `:f` is applied 5 times to `:x`. We will show later that the resulting word, `lambda_200`, can be associated to the number **120**, which is the factorial of **5**: `5! = 1*2*3*4*5`. Writing the same expression where `:f` is applied 50 times to `:x` would lead to the exact 65 digits of 50! ... provided we had thousands years before us!

Anyway, it happens that with nothing but three rules we can do maths with absolute precision, at least theoretically! More generally, these three rules make {lambda word} a Turing complete [12] programmable programming language. Even if, at this point, this language is practically unusable!

1.2. Names

In order to make life easier, we introduce a second special form `{def NAME expression}` to populate the dictionary with global constants and give names to lambdas. For instance:

```
{def MY_PI 3.1416} -> MY_PI
{MY_PI} -> 3.1416  // it's not a number
```

Note that, contrary to languages like Lisp[13] or Scheme[14], the name of a constant is NOT evaluated, it's a reference *pointing to some value*. Bracketing the name between {} returns the pointed value, {MY_PI} is evaluated to **3.1416**. A similar example is given in any spreadsheet where **PI** stays **PI** and **=PI()** is evaluated to 3.141592653589793.

```
{def GOOD_DAY
 {lambda {:o :a} :oh h:appy day!}}
-> GOOD_DAY
{GOOD_DAY oOOOo aaAAaa}
-> oOOOoh haaAAaappy day!
```

Note that arguments and their occurences in the function's body have been prefixed with a colon ":". It's easy to understand that doing that prevents the word **day** to be unintentionally changed into **daaAAaay**. Escaping arguments - for instance prefixing them with a colon ":" - is highly recommended if not always mandatory.

```
{def CONS {lambda {:x :y :z}
 {:z :x :y}}} -> CONS
{def CAR {lambda {:z}
 {:z {lambda {:x :y} :x}}}} -> CAR
{def CDR {lambda {:z}
 {:z {lambda {:x :y} :y}}}} -> CDR
{CAR {CONS Hello World}} -> Hello
{CDR {CONS Hello World}} -> World
```

This example not only makes more readable the fourth examples of the previous section evaluated to "Hello" and "World", but it opens the way to powerful structures.

1.3. Implementation

Working on the client side the {lambda word} evaluator is a Javascript **IIFE** (Immediately Invoked Function Expression), `LAMBDAWORD`, returning a set of functions, the main one being `eval()`, called at every keyboard entry:

```
var LAMBDAWORD = (function() {
var eval = function(str) {
  str = pre_processing(str);
  str = eval_lambdas(str);
  str = eval_defs(str);
  str = eval_forms(str);
  str = post_processing(str);
  return str;
};
return {eval:eval}
})();
```

We note that, in its main part, the `eval()` function follows strictly the definition of the language, evaluating abstractions *before* applications.

1.3.1. Simple Forms

Simple forms `{first rest}` are nested *evaluable expressions* caught recursively from the leaves to the root and replaced by words. The evaluation stops when the expression is reduced to words in a tree structure sent to the browser's engine for the final evaluation and display.

```
var eval_forms = function( str ) {
  var leaf =
   /\{([^\s{}]*)(?:[\s]*)([^{}]*)\}/g;
  while (str !=
   (str = str.replace( leaf, eval_leaf ))) ;
  return str
};
var eval_leaf = function(_,f,r) {
  return (DICT.hasOwnProperty(f))?
          DICT[f].apply(null,[r]) :
         '('+f+' '+r+')';
};
var DICT = {}; // initially empty
```

We note that {lambda word} is built on a **regular expressions based evaluator**. Contrary to, for instance, Lisp[13]] or Scheme[14]], the evaluator doesn't follow the standard **AST** process. It literally scans the code string, skips the words and progressively replaces *in situ* nested forms by words. Even if this choice is considered by some people as *evil*, it does work, at least in a wiki context, allowing in most cases realtime editing. The reason is that Regular Expressions [15] are powerful and fast - . This is what Ward Cunningham [16] wrote about that: « *I was surprised that the technique worked so well in so many cases. I knew that regex are highly optimized and the cpus themselves optimize sequential access to memory which the regex must have at its core. [..] Yes, this has at its heart the repeated application of a text transformation. The fact that it is repeated application of the same transformation makes it exceptional. [..] Repeated application of Regular Expressions can perform Touring Complete computations. This works because the needed "state" is in the partially evaluated text itself.* » All is said!

The special forms `{lambda {arg*} body}` and `{def name body}` are evaluated **before** simple forms. They are matched and evaluated in a similar way, according to specific patterns, and return words as references of functions added to the dictionary.

1.3.2. Lambdas

```
var eval_lambdas = function(str) {
 while ( str !== ( str =
  form_replace(str,'{lambda', eval_lambda)));
  return str
};
var eval_lambda = function(s){
 s = eval_lambdas( s );  // nested lambdas
 var index = s.indexOf('}'),
      args = supertrim(s.substring(1, index))
             .split(' '),
      body = s.substring(index+2).trim(),
      name = 'lambda_' + g_lambda_num++,
      reg_args = [];
 for (var i=0; i < args.length; i++)
  reg_args[i] = RegExp( args[i], 'g');

 DICT[name] = function() {
  var vals =
      supertrim(arguments[0]).split(' ');
  return function(bod) {
   if (vals.length < args.length) {
    for (var i=0; i <  vals.length; i++)
     bod = bod.replace(reg_args[i],vals[i]);
    var _args=args.slice(vals.length)
                .join(' ');
    bod = '{' + _args + '} ' + bod;
    bod = eval_lambda(bod); // -> a lambda
   } else {                 // -> a form
    for (var i=0; i <  args.length; i++)
     bod = bod.replace(reg_args[i],vals[i]);
   }
   return bod;
  }(body);
 };
 return name;
};
```

lambdas are **first class** functions. We note that they can be nested but that they don't create **closures**: *inside functions have no access to outside functions' arguments*, functions can't have free variables getting their value from the outside environment. But in lambdatalk **partial function application** is trivial: *not giving a multi-argument function all of its arguments will simply return a function that takes the remaining arguments*. We have seen a first useful application of these capabilities while building **pairs**, `[CONS, CAR, CDR]`. In fact any user defined **agregate data** can be built, for instance `lists, Btrees, Rational and Complex numbers, 2D/3D vectors, ...`, even if for a matter of efficiency it's better to implement them in the underlying language as primitive functions.

1.3.3. Defs

```
var eval_defs = function(str, flag) {
  while ( str !== ( str =
   form_replace(str,'{def',eval_def,flag)));
  return str
};
var eval_def = function (s, flag) {
  flag = (flag === undefined)? true : false;
  s = eval_defs( s, false ); // nested defs
  var index = s.search(/\s/),
      name  = s.substring(0, index).trim(),
      body  = s.substring(index).trim();
  if (body.substring(0,7) === 'lambda_') {
    DICT[name] = DICT[body];
    delete DICT[body];
  } else {
     body = eval_forms(body);
     DICT[name] = function() { return body };
  }
  return (flag)? name : '';
};
```

We note that **defs** can be nested but that inner definitions are added to the global dictionary, there is no local scope. In order to prevent names conflicts, inner names should be prefixed by the outer function names, as objects' methods in OOP.

It's worth noting that this implementation buit on two functions `eval_lambdas(), eval_forms()`, an optional one, `eval_defs`, and an initially empty dictionary, `DICT = {}`, is sufficient to make {lambda word} a programmable programming language. At this point {lambda word} knows nothing but text substitutions. Surprisingly, this is enough to introduce the concept of number and the associated operators.

2. {LAMBDA CALC}

After Alonzo Church, we define the so-called **Church numbers** like this [#]: *a Church number N iterates N times the application of a function f on a variable x*. For instance:

```
{def ZERO {lambda {:f :x} :x}} -> ZERO
{def ONE {lambda {:f :x} {:f :x}}} -> ONE
{def TWO {lambda {:f :x}
   {:f {:f :x}}}} -> TWO
{def THREE {lambda {:f :x}
   {:f {:f {:f :x}}}}} -> THREE
{def FOUR {lambda {:f :x}
   {:f {:f {:f {:f :x}}}}}} -> FOUR
{def FIVE {lambda {:f :x}
   {:f {:f {:f {:f {:f :x}}}}}}} -> FIVE
```

Applied to a couple of any words, we get for instance: `{THREE . .} -> (. (. (. .)))`. It's easy to count three couple of parenthesis and we are led to define a function `CHURCH` which returns their number:

```
{def CHURCH {lambda {:n}
 {{:n {lambda {:x} {+ :x 1}}} 0}}} -> CHURCH
{CHURCH ZERO} -> 0
{CHURCH ONE}  -> 1
{CHURCH FIVE} -> 5
```

Note that the `CHURCH` function calls a primitive function, `'+'`, which is not supposed to exist in {lambda calc}. Consider that it's only for readability and that does not invalidate the demonstration. The definition of Church number gives the basis of a first set of operators, [SUCC, ADD, MUL, POWER]

```
{def SUCC {lambda {:n :f :x}
  {:f {{:n :f} :x}}}} -> SUCC
{def ADD {lambda {:n :m :f :x}
  {{:n :f} {{:m :f} :x}}}} -> ADD
{def MUL {lambda {:n :m :f}
  {:m {:n :f}}}} -> MUL
{def POWER {lambda {:n :m}
  {:m :n}}} -> POWER

{CHURCH {SUCC ZERO}} -> 1
{CHURCH {SUCC ONE}} -> 2
{CHURCH {ADD TWO THREE}} -> 5
{CHURCH {MUL TWO THREE}} -> 6
{CHURCH {POWER TWO THREE}} -> 8

5! = 1*2*3*4*5 =
 {CHURCH {MUL ONE {MUL TWO {MUL THREE
           {MUL FOUR FIVE}}}}} -> 120
```

Computing **50!** the way we did with **5!** could be boring and at least wouldn't have any interest. We are going to build a better algorithm, usable for any number `N`, using the previously defined `[CONS, CAR, CDR]` constants. This is how: we define a function `FAC.PAIR` which gets a pair `[a,b]` and returns a pair `[a+1,a*b]`. `FAC` computes `n` iterations of `FAC.PAIR` starting on the pair `[1,1]`, leading to the pair `[n,n!]` and returns the second, `n!`

```
{def FAC.PAIR {lambda {:p}
  {CONS {SUCC {CAR :p}}
        {MUL {CAR :p} {CDR :p}}}}}
-> FAC.PAIR
{def FAC {lambda {:n}
  {CDR {{:n FAC.PAIR} {CONS ONE ONE}}}}}
-> FAC

3! = {CHURCH {FAC THREE}} -> 6
4! = {CHURCH {FAC FOUR}}  -> 24
5! = {CHURCH {FAC FIVE}}  -> 120
```

Replacing the names `[FAC, FIVE]` by their associated lambda expressions in the last line, `{FAC FIVE}` displays exactly the unreadable fifth expression of the previous section! And just as {lambda word} could compute **50!**, {lambda calc} could do it, at least theoretically! And following, for instance, "Collected Lambda Calculus Functions" [17], we could go on and define some other operators, `[PRED, SUBTRACT, DIV, TRUE, FALSE, NOT, AND, OR, LT, LEQ, ...]` and even the `Y combinator` allowing *almost-recursive* functions to become recursive. But the underlying browser's capabilities will help us to go further more effectively!

3. {LAMBDA TALK}

{lambda word} was built on a single special form, `{lambda {args} body}` and a hidden dictionary containing annonymous functions. Then with a second special form, `{def name expression}`, constants could be added to the dictionary, `[MY_PI, GOOD_DAY, CONS, CAR, CDR`, leading to {lambda calc} and its set of numerical constants, `[CHURCH, ZERO, ONE? TWO, ..., SUCC, ADD, MUL, POWER, FAC]`. It's time to use the power of web browsers, coming with a powerful language, Javascript, and a complete "Document Object Model" on which can be built a true usable programmable programming language, {lambda talk}.

A new set of special forms, `[if, let, quote|', macros]` is added to the primitive set, `[lambda, def]`. Note that there is no `set!` function, {lambda talk} is **purely functional**. This is the current content of the lambdatalk's dictionary:

```
DICTionary: (220) [ debug, lib, eval, force,
apply, when, <, >, <=, >=, =, not, or, and, +,
-, *, /, %, abs, acos, asin, atan, ceil, cos,
exp, floor, pow, log, random, round, sin, sqrt,
tan, min, max, PI, E, date, serie, map, reduce,
equal?, empty?, chars, charAt, substring,
length, first, rest, last, nth, replace,
array.new, array, array?, array.disp,
array.length, array.nth, array.first,
array.rest, array.last, array.slice, array.push,
array.pop, array.set!, cons, cons?, car, cdr,
cons.disp, list.new, list, list.disp,
list.length, list.reverse, list.2array,
list.first, list.butfirst, list.last,
list.butlast, @, div, span, a, ul, ol, li, dl,
dt, dd, table, tr, td, h1, h2, h3, h4, h5, h6,
p, b, i, u, center, hr, blockquote, sup, sub,
del, code, img, pre, textarea, canvas, audio,
video, source, select, option, svg, line, rect,
circle, ellipse, polygon, polyline, path, text,
g, mpath, use, textPath, pattern, image,
clipPath, defs, animate, set, animateMotion,
animateTransform, br, input, script, style,
iframe, mailto, back, hide, long_mult, drag,
note, note_start, note_end, show, lightbox,
minibox, lisp, forum, editable, sheet, lc,
turtle, MY_PI, GOOD_DAY, CONS, CAR, CDR, ZERO,
ONE, TWO, THREE, FOUR, FIVE, CHURCH, SUCC, ADD,
MUL, POWER, FAC.PAIR, FAC, fac, tfac_r, tfac, Y,
almost_fac, yfac, bigfac, D, log', log'',
log''', q0, q1, q2, q3, q4, quotient, sigma,
paren, mul, variadic, PDF, COLUMN_COUNT, space,
ref, back_ref, castel.interpol, castel.sub,
castel.point, split_gd, castel.split,
castel.build, svg.frame, svg.dot, svg.poly,
spreadsheet, sheet.new, sheet.input,
sheet.output ]
```

where can be seen after the constant `turtle` the user defined constants specifically built for this page. We are going to give examples illustrating some of the capabilities of {lambda talk}.

3.1. *Recursion and the Y-Combinator*

With {lambda word} and {lambda calc} it was possible to compute `5!` defined as a product `1*2*3*4*5`. Adding to {lambda talk} the special form `{if bool then one else two}` and its **lazy evaluation** opens the way to recursive algorithms. It's now possible to write the factorial function following its mathematical definition:

```
{def fac
 {lambda {:n}
  {if {< :n 0} then {b n must be positive!}
   else {if {= :n 0} then 1
   else {* :n {fac {- :n 1}}}}}}} -> fac

{fac -1} -> n must be positive!
{fac 0} -> 1
{fac 5} -> 120
{fac 50} -> 3.0414093201713376e+64
```

Let's write the tail-recursive version:

```
{def tfac
 {def tfac_r
  {lambda {:a :n}
   {if {< :n 0} then {b n must be positive!}
    else {if {= :n 0} then :a
    else {tfac_r {* :a :n} {- :n 1}}}}}}
 {lambda {:n} {tfac_r 1 :n}}} -> tfac

{tfac 5} -> 120
{tfac 50} -> 3.0414093201713376e+64
```

The recursive part is called by a "helper function" introducing the accumulator `:a`. Because {lambda talk} doesn't know lexical scoping, this leads to some pollution of the dictionary. The `Y combinator` mentionned above in {lambda calc}, *making recursive an almost-recursive function*, will help us to discard this helper function. The `Y combinator` and the almost-recursive function can be defined and used like this:

```
{def Y {lambda {:f :a :n} {:f :f :a :n}}}
-> Y

{def almost_fac
 {lambda {:f :a :n}
  {if {<  :n 0} then {b n must be positive!}
   else {if {= :n 0} then :a
   else {:f :f {* :a :n} {- :n 1}}}}}}
-> almost_fac

{Y almost_fac 1 5} -> 120
```

Because the `Y combinator` can be applied to any other almost-recursive function (sharing the same signature, for instance fibonacci) we have reduced the pollution but we can do better. Instead of applying the Y combinator to the almost recursive function we can define a function merging the both:

```
{def yfac {lambda {:n}
  {{lambda {:f :a :n}
    {:f :f :a :n}}
     {lambda {:f :a :n}
      {if {<  :n 0}
       then {b n must be positive!}
       else {if {= :n 0} then :a
       else {:f :f {* :a :n} {- :n 1}}}}} 1
:n}}}
-> yfac

{yfac 5} -> 120
{yfac 50} -> 3.0414093201713376e+64
{map yfac {serie 0 20}}
-> 1 2 6 24 120 720 5040 40320 362880 3628800
39916800 479001600 6227020800 87178291200
1307674368000 20922789888000 355687428096000
6402373705728000 121645100408832000
2432902008176640000
```

The last point to fix is that `{fac 50}`, `{tfac 50}` and `{yfac 50}` return a rounded value **3.0414093201713376e+64** which is obvioulsly not the exact value. We must go a little further and build some tools capable of processing big numbers.

3.2. Big Numbers

The way the **javascript Math object** is implemented puts the limit of natural numbers to $\mathbf{2^{54}}$. Beyond this limit last digits are rounded to zeroes, for instance the four last digits of 2^{64} = {pow 2 64} = 18446744073709552000 should be **1616** and are rounded to **2000**. And beyond $\mathbf{2^{69}}$ natural numbers are replaced by float numbers with a maximum of 15 valid digits. Let's come back to the definition of a natural number: *A natural number* $a_0a_1\ldots a_n$ *is the value of a polynomial* $\Sigma_{i=0}{}^{n}a_ix^i$ *for* $x=10$. For instance $12345 = 1*10^4+2*10^3+3*10^2+4*10^1+5*10^0$. {lambda talk} knowing `lists`, we represent a natural number as a list on which we define a set of functions:

```
1) k*p
{def BN.k {lambda {:k :p}
  {if {equal? :p nil}
   then nil
   else {cons {* :k {car :p}}
              {BN.pk :k {cdr :p}}}}}}
2) p1+p2
{def BN.+ {lambda {:p1 :p2}
  {if {and {equal? :p1 nil} {equal? :p2 nil}}
   then nil
   else {if {equal? :p1 nil} then :p2
   else {if {equal? :p2 nil} then :p1
   else {cons {+ {car :p1} {car :p2}}
              {BN.p+ {cdr :p1} {cdr :p2} }}}}
}}}
3) p1*p2
{def BN.* {lambda {:p1 :p2}
  {if {or {equal? :p1 nil} {equal? :p2 nil}}
   then nil
   else {if {not {cons? :p1}}
   then {BN.pk :p1 :p2}
   else {BN.p+ {BN.pk {car :p1} :p2}
               {cons 0 {BN.p* {cdr :p1} :p2
}}}}}}}
4) helper functions
{def BN.bignum2pol {lambda {:n} .. }}
{def BN.pol2bignum {lambda {:p} .. }}
{def BN.normalize {lambda {:p :n} .. }}
```

We can now define a function `bigfac` working on big numbers:

```
{def bigfac {lambda {:n}
  {if {<  :n 1} then {BN.bignum2pol 1}
   else {BN.* {BN.bignum2pol :n}
              {bigfac {- :n 1}}}}}}
-> bigfac

5! = {BN.pol2bignum
            {BN.normalize {bigfac 5} 1}} -> 120
50! = {BN.pol2bignum
            {BN.normalize {bigfac 50} 50}} ->
30414093201713380398385728867859 9
00744706627746248466026044200000
```

To sum up, exclusively working on words made of chars, we could exceed the limits of the Javascript Math Object and work with "numbers" of any size with exact precision. At least theoretically. When numbers become too big the evaluation is considerably slowed down and it's better to forget pure user defined {lambda talk} function and add to the dictionary some javascript primitive function, `long_mult`:

```
DICT['long_mult'] = function () {
  var args =
      supertrim(arguments[0]).split(' ');
  var n1 = args[0], n2 = args[1];
  var a1 = n1.split("").reverse();
  var a2 = n2.split("").reverse();
  var a3 = [];
  for (var i1 = 0; i1 < a1.length; i1++) {
   for (var i2 = 0; i2 <  a2.length; i2++) {
    var id = i1 + i2;
    var foo = (id >= a3.length)?0:a3[id];
    a3[id] = a1[i1] * a2[i2] + foo;
    if ( a3[id] > 9 ) {
     var carry =
         (id+1 >= a3.length)?0:a3[id+1];
     a3[id+1] = Math.floor(a3[id]/10)+carry;
     a3[id] -= Math.floor(a3[id]/10)*10;
    }
   }
  }
  return a3.reverse().join("");
};
```

We just redefine the previous factorial **!** as **!!** replacing the primitive `*` by the primitive `long_mult`:

```
{def !! {lambda {:n}
 {if {< :n 1} then 1
  else {long_mult :n {!! {- :n 1}}}}}}

{!! 100} ->
933262154439441526816992388562667004907159682
643816214685929638952175999932299156089414639
761565182862536979208272237582511852109168640
00000000000000000000000
```

In {lambda word} knowing nothing but three rules working on text substitutions, it was theoretically possible to compute **5!** and even **50!**. In {lambda calc} results became readable, without calling any other user defined function than the `CHURCH` function. In {lambda talk} using the Math Object, things became easy and for big numbers a specific primitive `long_mult` opened an effective window in the world of big numbers.

We will now show other applications of {lambda talk}, built on the powerful functionalities of the browsers' engines.

3.3. Derivatives

Because functions can be partially called, derivatives of any function **f(x)** can be defined as functions of **x** and NOT as values at a given **x, f'(x), f''(x), f'''(x)**:

- 1) we define the derivative function:

```
{def D {lambda {:f :h :x}
  {/ {- {:f {+ :x :h}} {:f {- :x :h}} }
     {* 2 :h}}}} -> D
```

- 2) we create the 1st, 2nd and 3rd derivatives of the function log for a given value of :h and as functions of :x:

```
{def log' {lambda {:x} {D log   0.01 :x}}}
-> log'
{def log'' {lambda {:x} {D log'  0.01 :x}}}
-> log''
{def log''' {lambda {:x} {D log'' 0.01 :x}}}
-> log'''
```

- 3) we can now call the 1st, 2nd and 3rd derivatives of log on a given value of x:

```
{log'   1} -> 1.0000333353334772  // 1
{log''  1} -> -1.0002000533493427 // -1
{log''' 1} -> 2.0012007805464416  // 2
```

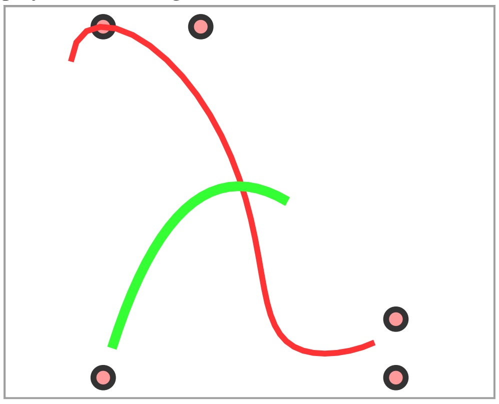
3.4. de Casteljau

{lambda talk} can call the set of SVG functions implemented in web browsers. The **de Casteljau** recursive algorithm [18] allows drawing Bezier curves of any degree. For instance writing:

```
{{svg.frame 250px 200px}
  {svg.dot {{def q0 {cons 50 10}}}}
  {svg.dot {{def q1 {cons 100 10}}}}
  {svg.dot {{def q2 {cons 200 160}}}}
  {svg.dot {{def q3 {cons 50 190}}}}
  {svg.dot {{def q4 {cons 200 190}}}}

 {polyline {@ points="{castel.build
  {list.new {q0} {q1} {q2} {q3} {q4} {q2}}
  -0.1 0.9 {pow 2 -5}}"
  stroke="#f00" fill="transparent"
  stroke-width="3"}}
 {polyline {@ points="{castel.build
  {list.new {q2} {q1} {q3}}
  0.3 0.98 {pow 2 -5}}"
  stroke="#0f0" fill="transparent"
  stroke-width="5"}}
}
```

displays the following colored λ:



We have defined 2D points as **pairs** and polylines as **lists** and built a set of user functions in another wiki page, `lib_decasteljau` called via a `(require lib_decasteljau)`:

```
{def castel.interpol {lambda {:p0 :p1 :t}
 {cons
  {+ {* {car :p0} {- 1 :t}} {* {car :p1} :t}}
  {+ {* {cdr :p0} {- 1 :t}} {* {cdr :p1} :t}}
}}}
{def castel.sub {lambda {:l :t}
 {if {equal? {cdr :l} nil} then nil
  else {cons {castel.interpol
                 {car :l} {car {cdr :l}} :t}
             {castel.sub {cdr :l} :t}}
}}}
{def castel.point {lambda {:l :t}
 {if {equal? {cdr :l} nil}
  then {car {car :l}} {cdr {car :l}}
  else {castel.point {castel.sub :l :t} :t}
}}}
{def castel.build {lambda {:l :a :b :d}
 {map {castel.point :l} {serie :a :b :d}}}}
{def svg.frame {lambda {:w :h}
  svg {@ width=":w" height=":h"
         style="border:1px solid #888;
         box-shadow:0 0 8px;"}}}
{def svg.dot {lambda {:p}
 {circle {@ cx="{car :p}" cy="{cdr :p}" r="5"
            stroke="black" stroke-width="3"
            fill="rgba(255,0,0,0.5)"}} }}
{def svg.poly {lambda {:l}
 {if {equal? :l nil}
  then else {car {car :l}} {cdr {car :l}}
            {svg.poly {cdr :l}}}}}
```

3.5. Spreadsheet

A spreadsheet is an interactive computer application for organization, analysis and storage of data in tabular form. A spreadsheet is a good illustration of functional languages, (Simon Peyton-Jones [19]) and thereby rather easy to implement in {lambda talk}. The basic idea is that each cell **contains the input** - *words and expressions* - and **displays the output**. Writing:

```
{require_ lib_spreadsheet}
{center {spreadsheet 4 5}}
```

displays a table of 5 rows and 4 columns of editable cells in which **almost all** {lambda talk} functions can be used:

Editing cell L5C4:

{+ {IJ -3 0} {IJ -2 0} {IJ -1 0}}

NAME	QUANT	UNIT PRICE	PRICE
Item 1	10	2.1	21
Item 2	20	3.2	64
Item 3	30	4.3	129
.	.	TOTAL PRICE	214

[local storage]

Writing `(require lib_spreadsheet)` calls a set of {lambda talk} and javascript functions written in another wiki page, `lib_spreadsheet`. Two functions are added by this library for linking cells:

- `{LC i j}` returns the value of the cell L_iC_j as an **absolute** reference,
- `{IJ i j}` returns the value of the cell $L_{[i]}C_{[j]}$ as a **relative reference**. For instance writing `{IJ -1 -1}` in `L2C2` will return the value of `L1C1`.

Let's test: click on the **[local storage]** button, copy the code in the frame below, paste it in the local storage frame, click on the **editor -> storage** button, confirm the action and analyze the spreadsheet's cells.

```
["{b NAME}","{b QUANT}","{b UNIT PRICE}","{b
PRICE}","Item 1","10","2.1","{* {IJ 0 -2} {IJ 0
-1}}","Item 2","20","3.2","{* {IJ 0 -2} {IJ 0
-1}}","Item 3","30","4.3","{* {IJ 0 -2} {IJ 0
-1}}","","","{b TOTAL PRICE}","{+ {IJ -3 0} {IJ
-2 0} {IJ -1 0}}","4"]
```

3.6. *MathML*

{lambda talk} forgets the **MathML** markup set which is not implemented in Google Chrome [20]. A set of functions, `[quotient, paren, sigma]`, can be defined and used to render Mathematical Symbols:

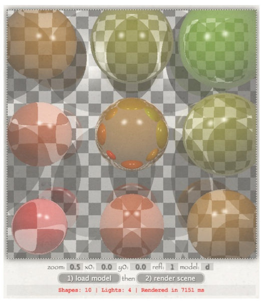
```
i{del h}{quotient 20 ∂ψ ∂t}(x,t) = {paren 3 (}
mc{sup 2}α{sub 0} - i{del h}c {sigma 20 j=1 3}
α{sub j}{quotient 20 ∂ ∂x{sub j}} {paren 3 )}
ψ(x,t) ->
```

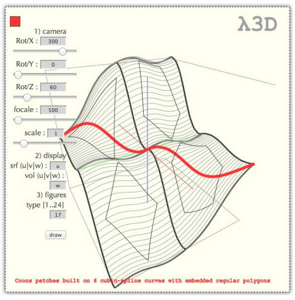
$$i\hbar\frac{\partial\psi}{\partial t}(x,t) = \left(mc^2\alpha_0 - i\hbar c\sum_{j=1}^{3}\alpha_j\frac{\partial}{\partial x_j}\right)\psi(x,t)$$

No, it's not a picture!

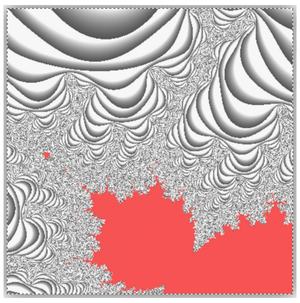
3.7. *Scripts*

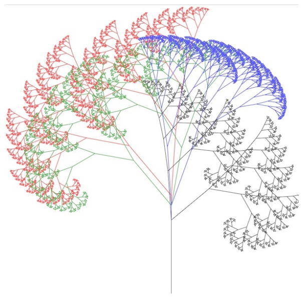
{lambda talk} code can be interfaced with Javascript code written in any wiki page via a `{script ...}` form, allowing the exploration of intensive computing. For instance ray-tracing [21], curved shapes modeling [22], fractal [23] and turtle graphics drawing [24]:





λ3D
1) camera
Rot/X :
Rot/Y :
Rot/Z :
focale :
scale :
2) display
3) figures
draw
Coons patches built on 4 cubic spline curves with embedded regular polygons





3.8. *Macros*

{lambda talk} macros bring the power of regular expressions directly in the language. As a first application, the expression `{def name {lambda {args} body}}` could be replaced by the **syntaxic sugar** `{defun {name args} body}`

```
{macro
 {defun {(\w*?) ([^{}]*?)}(?:[\s]*?)(.*)}
     to {def €1 {lambda {€2} €3}}}

{defun {mul :x :y} {* :x :y}} -> mul
{mul 3 4} -> 12
```

As a second example, {lambda talk} comes with some variadic primitives, for instance [`+,-,*,/,list`]. At first sight, user functions can't be defined variadic, for instance:

```
{* 1 2 3 4 5}    -> 120      // * is variadic
{mul 1 2 3 4 5} -> 2  // 3, 4, 5 are ignored
```

In order to make `mul` variadic we glue values in a list and use a user defined helper function:

```
{def variadic
 {lambda {:f :args}
  {if   {equal? {cdr :args} nil}
   then {car :args}
   else {:f {car :args}
            {variadic :f {cdr :args}}}}}}
-> variadic

{variadic mul {list 1 2 3 4 5}} -> 120
```

But it's ugly and doesn't follow a standard call. We can do better using a macro:

```
1) defining:
{macro {mul* (.*?)}
    to {variadic mul {list €1}}}

2) using:
(mul* 1 2 3 4 5) -> 120
```

Now `mul*` is a variadic function which can be used as any other primitive or user function, except that it's not a first class function, as in most Lisps. As a last example, {lambda talk} comes with a predefined small set of macros allowing writing without curly braces titles, paragraphs, list items, links:

```
_h1 TITLE ¬
  stands for {h1 TITLE}
_p Some paragraph ... ¬
  stands for {p Some paragraph ...}
[[PIXAR|http://www.pixar.com/]]
  stands for
    {a {@ href="http://www.pixar.com/"}PIXAR}
[[sandbox]]
  stands for
    {a {@ href="?view=sandbox"}sandbox}
```

These simplified alternatives, avoiding curly braces as much as possible, are fully used in the current document.

CONCLUSION

To sum up, {lambda talk} takes benefit from the extraordinary power of modern web browsers, simply adding a coherent and unique notation without re-inventing the wheel, just using the existing foundations of HTML/CSS, the DOM and Javascript. It's probably why the implementation of {lambda talk} is so easy and short, as we have seen before. Three rules and an empty dictionary built the foundations of a programmable programming language, {lambda word}, at least in theory. Adding a few ones and populating the dictionary with primitives built on the browsers functionalities led to an effective one, {lambda talk}.

The {lambda way} project is « *a dwarf on the shoulders of giants* »[25], a thin overlay built upon any modern browser, proposing a small interactive development environment and a coherent language *without any external dependencies* and thereby easy to download and install [26] on a web account provider running PHP. From any web browser on any system, complex web pages can be created, enriched, structured and tested in real time and directly published on the web. It's exactly how the current document was created: entirely created and tested in the {lambda way} environment [27] it was directly printed as a PDF document [28], 8 pages in two columns following the ACM format specifications.

Villeneuve de la Raho, 2017/03/19

REFERENCES

[1] Wiki: https://en.wikipedia.org/wiki/Wiki
[2] Wiki_markup: https://en.wikipedia.org/wiki/Wiki_markup
[3] LaTeX: http://fr.wikipedia.org/wiki/LaTeX
[4] Curl: https://en.wikibooks.org/wiki/Curl
[5] lml: http://lml.b9.com/
[6] The-Skribe-evaluator: http://www-sop.inria.fr/members/Manuel.Serrano/publi/jfp05/ar Skribe-evaluator
[7] scribble: http://docs.racket-lang.org/scribble/
[8] SXML: https://en.wikipedia.org/wiki/SXML
[9] LAML: http://people.cs.aau.dk/~normark/laml/papers/web-programming-laml.pdf
[10] Pollen: http://docs.racket-lang.org/pollen/
[11] A Tutorial Introduction to the Lambda Calculus (Raul Rojas): http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf
[12] Turing http://epsilonwiki.free.fr/lambdaway/?view=turing
[13] Lisp: http://www.cs.utexas.edu/~cannata/cs345/Class%20No
[14] Scheme: https://mitpress.mit.edu/sicp/full-text/book/book.html
[15] Regular Expressions: http://blog.stevenlevithan.com/archives/reverse-recursive-pattern
[16] Ward_Cunningham: https://en.wikipedia.org/wiki/Ward_Cunningham
[17] Collected Lambda Calculus Functions: http://jwodder.freeshell.org/lambda.html
[18] De_Casteljau's_algorithm: https://en.wikipedia.org/wiki/De_Casteljau's_algor
[19] Simon_Peyton_Jones: https://en.wikipedia.org/wiki/Simon_Peyton_Jones
[20] google-subtracts-mathml-from-chrome: https://www.cnet.com/news/google-subtracts-mathml-from-chrome-and-anger-multiplies/
[21] raytracing: http://epsilonwiki.free.fr/lambdaway/?view=raytracing
[22] pForms: http://epsilonwiki.free.fr/lambdaway/?view=pforms
[23] fractal: http://epsilonwiki.free.fr/lambdaway/?view=mandel
[24] turtle: http://epsilonwiki.free.fr/lambdaway/?view=lambdatree
[25] dwarf: http://www.phrases.org.uk/meanings/268025.html
[26] download: http://epsilonwiki.free.fr/lambdaway/?view=download
[27] wiki page: http://epsilonwiki.free.fr/lambdaway/?view=brussels
[28] PDF document: http://epsilonwiki.free.fr/lambdaway/data/lambdawa (2.8Mb)

{λ way} v.20160608

Writing a best-effort portable code walker in Common Lisp

Michael Raskin[*]
LaBRI, University of Bordeaux
raskin@mccme.ru

ABSTRACT

One of the powerful features of the Lisp language family is possibility to extend the language using macros. Some of possible extensions would benefit from a code walker, i.e. a library for processing code that keeps track of the status of different part of code, for their implementation. But in practice code walking is generally avoided.

In this paper, we study facilities useful to code walkers provided by "Common Lisp: the Language" (2nd edition) and the Common Lisp standard. We will show that the features described in the standard are not sufficient to write a fully portable code walker.

One of the problems is related to a powerful but rarely discussed feature. The `macrolet` special form allows a macro function to pass information easily to other macro invocations inside the lexical scope of the expansion.

Another problem for code analysis is related to the usage of non-standard special forms in expansions of standard macros. We review the handling of `defun` by popular free software Common Lisp implementations.

We also survey the abilities and limitations of the available code walking and recursive macro expansion libraries. Some examples of apparently-conforming code that exhibit avoidable limitations of the portable code walking tools are provided.

We present a new attempt to implement a portable best-effort code walker for Common Lisp called Agnostic Lizard.

CCS CONCEPTS

•Software and its engineering →Macro languages; Software testing and debugging;

KEYWORDS

code walker, macro expansion, code transformation

[*]The author acknowledges support from the Danish National Research Foundation and The National Science Foundation of China (under the grant 61361136003) for the Sino-Danish Center for the Theory of Interactive Computation. This work was supported by the French National Research Agency (ANR project GraphEn / ANR-15-CE40-0009).

ACM Reference format:
Michael Raskin. 2017. Writing a best-effort portable code walker in Common Lisp. In *Proceedings of 10th European Lisp Simposium, Vrije Universiteit Brussel, Belgium, April 2017 (ELS2017),* 8 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Much of the power of Common Lisp comes from its extensibility. Abstractions that cannot be expressed by functions can still be expressed by macros; actually, many of the features described in the standard must be implemented as macros.

Whilst macros are powerful on their own, some ways of extending Lisp would benefit from using higher-level code transformation abstractions. For many such abstractions the most natural way to implement them includes code walking, i.e. enumeration of all the subforms of a piece of code, identification of function calls, variable bindings etc. among these subforms, and application of some code transformations. Unfortunately, code walking is complicated and the libraries to perform it are non-portable. Even portable implementations of recursive macro expansion, also known as `macroexpand-all`, are missing.

Guy Steele writes (in the second edition of the book "Common Lisp: the Language" [3]) that Common Lisp implementations are expected to provide all the functionality needed for code walking code analysis tools. The version of the language described in the book includes support for changing and inspecting the lexical environment objects; the book explicitly recommends to use `macroexpand` on the macros in the language core.

Lexical environment objects are defined by the Common Lisp standard [2] are more opaque. Many Common Lisp implementations do include some functions for environment inspection and manipulation, and even some code walking function. Unfortunately, both the naming and the feature set vary between implementations.

We will show that (unlike CL:tL2) the Common Lisp standard does not allow to implement a portable code walker correctly. We suggest an approach that approximates the desired functionality fairly well and remains implementation-agnostic even when doing the implementation-specific workarounds. We present an implementation of this approach, a library called Agnostic Lizard.

2 RELATED WORK

2.1 Portable tools

The `iterate` library [4] provides an alternative iteration construct also called `iterate`. It is more flexible than the standard `loop` macro, and it uses code walking for implementation. It doesn't work exactly as expected for some code, though. The code snippet

```
(iterate:iterate
  (for x :from 1 :to 3)
    (flet
      ((f (y) (collect y)))
      (f x)))
```

works the same as

```
(iterate:iterate
  (for x :from 1 :to 3)
  (collect x))
```

But the following version with a local macro definition will not work

```
(iterate:iterate
  (for x :from 1 :to 3)
    (macrolet
       ((f (y) `(collect ,y)))
       (f x)))
```

The `macroexpand-dammit` library [7] is an attempt to provide implementation of the full recursive macro expansion functionality (`macroexpand-all`) including an optional lexical environment parameter, but it has some bugs. It is not clear if these bugs can be fixed without major changes. For example, both the original version and the freshest known fork [8] return `1` instead of `2` in the following case:

```
(defmacro f () 1)

(defmacro macroexpand-dammit-here
  (form &environment env)
  `(quote
     ,(macroexpand-dammit:macroexpand-dammit
        form env)))

(macrolet ((f () 2))
  (macroexpand-dammit-here
    (macrolet () (f))))
```

Additionally, the `macroexpand-dammit` library includes the `macroexpand-dammit-as-macro` macro that the environment handling system of the Common Lisp implementation. Both the function and the macro versions of the recursive macro expander remove `macrolet` and `symbol-macrolet` forms from the code (replacing them with `progn` if necessary).

As `SICL` [14] aims to be a modular and portable conforming implementation of Common Lisp in Common Lisp, and the standard requires `compile` to do an equivalent of `macroexpand-all`, there probably will be a usable code walker in `SICL` at some point; to the best of our knowledge, there is currently none.

2.2 Implementation-specific tools

Unfortunately, implementation-specific tools often check the name of the Common Lisp implementation to choose the code path (for example, using `#+` reader conditionals to check for the implementation name). Even if two Common Lisp implementations are closely related and have the same names for the environment processing functions, support for these two implementations has to be added separately. This limitation makes some of the implementation-dependent tools not work on some newer implementations (such as Clasp [15]) even when the tool contains all the code needed for supporting the implementation.

On the other hand, if different versions of the same Common Lisp implementation behave in a different way, such checks can lead to breakage on some of the relatively recent versions of a supported Common Lisp implementation.

Richards Waters has described [5, 6] a clean, almost portable implementation of `macroexpand-all` that needs only a single environment-related function to be implemented separately for each of the Common Lisp implementations.

`CLWEB` [9] by Alex Plotnick follows a similar approach for its custom code walker, and the same approach is described as a "proper code walker" in an essay [10] by Christophe Rhodes.

`hu.dwim.walker` [11] is a comprehensive code walking library that uses a lot of implementation-specific functions for inspecting lexical environments etc. Unfortunately, the current versions of some previously supported Common Lisp implementations are not supported because of relatively recent changes in environment handling. Also `macroexpand-dammit` is implemented in such a way that it removes `macrolet` and `symbol-macrolet` from the code completely after expanding the local macros and the local symbol macros.

The `trivial-macroexpand-all` library [13] provides the `macroexpand-all` functionality by wrapping the best function provided by each Common Lisp implementation. Unfortunately, some implementations don't support the lexical environment argument (for example, CLISP [16]). The same approach is used by SLIME (Common Lisp editing and debugging support for Emacs) [12]. Apparently, no more generic code walking functionality is provided in a consistent way by multiple implementations.

3 PROBLEMS

In this section we present a brief overview of the problems that the portable code walkers face.

3.1 Interpretations and violations of the standard

The Common Lisp standard allows Common Lisp implementations to implement some standard-defined macros as additional special operators. But all special operators added instead of macros must also have a macro definition available. This requirement seems to imply that the standard expects the code walkers to succeed if they implement special handling only for the special operators listed in the standard.

In practice many Common Lisp implementations violate this expectation and implement standard macros using non-standard implementation-specific special operators. For example, the `iterate` library contains workarounds for such macros as `handler-bind` and `cond`.

Fortunately, for most macros the current versions of the major implementations provide usable expansions.

3.1.1 Named lambda expressions. A particular macro that is almost always expanded to code with non-standard special forms is `defun`. We have checked six free software Common Lisp implementations (SBCL, ECL, CCL, ABCL, GCL, CLISP); we have found that only GCL expands the `defun` macro into portable code. SBCL and ABCL pass a form starting with `named-lambda` to `function`, ECL does the same but calls the special symbol `lambda-block`, CLISP passes two arguments (the name and the definition) to `function` and CCL does the same but replaces `function` with special form `nfunction`. For example, the expansion produced by SBCL is as follows.

```
(progn
 (eval-when (:compile-toplevel)
   (sb-c:%compiler-defun 'f nil t))
 (sb-impl::%defun 'f
                  (sb-int:named-lambda f
                      (x y)
                    (block f (* x (1+ y))))
                  (sb-c:source-location)))
```

3.1.2 Theoretical worst-case **defun** *implementation.* It seems that an implementation of the `defun` macro that compiles the code at expansion time and puts a literal compiled function object into the expansion does not violate neither the Common Lisp standard nor the description in the "Common Lisp: the Language". While such an implementation could be compliant, it would make code-walking of entire file meaningless without handling the `defun` macro in a special way.

3.2 Environment handling

The Common Lisp standard provides no functions for inspecting or modifying environments. On the other hand, macro functions can receive environment parameters and request macro expansion of arbitrary code using the environment they receive. Unfortunately, sometimes there is no way to construct an environment that would make the macro expansion would work exactly as desired.

Although `*macroexpand-hook*` is described as an expansion interface hook to `macroexpand-1`, the retrieval of the macro function from the environment is done by the standard rules and cannot be overridden.

4 PASSING INFORMATION VIA MACROLET AND MACROEXPAND-1, AND TWO KINDS OF THE LEXICAL SCOPE FOR MACROS

In this section we will discuss the available options for passing information between the macro expansion functions, the scopes and extents relevant to the different ways of passing information and implications for portable code walkers. The aim of this section is to provide some context and explanation for the technique used in the next section. This technique will be used to construct an example of impossibility of a correct portable recursive macro expansion function that accepts a lexical environment parameter.

The environment handling issue is related to a useful feature of Common Lisp macros. Let's consider a macro function that needs to pass some information to another macro function or another invocation of the same macro function.

Ordinary functions can pass information via the arguments and the return values, or via global variables. A macro function has an extra option. A macro function can wrap its expansion in a `macrolet` form or a `symbol-macrolet` form that defines a macro (symbol macro) not intended to be used directly. This definition will be accessible to all the macro function invocations corresponding to macro invocations inside the lexical scope of the definition. Such a macro function can use the `macroexpand-1` function to access the definition.

It is a bit awkward to describe the scope of such a definition because there are two kinds of lexical scope relevant for macros. There is the normal lexical scope of the macro function when it is defined and there is the lexical scope of the expansion output in terms of expanded code.

Such approach allows, for example, to define a macro that can be nested but limits the depth of its own nesting without code walking:

```
(defmacro depth-limited-macro
  (n-max &body body &environment env)
  (let*
    ((depth-value
       (macroexpand-1
         (quote (depth-counter-virtual-macro))
           env))
     (depth (if (numberp depth-value)
              depth-value 0)))
    (if
      (> depth n-max)
      (progn
        (format *error-output*
          "Maximum macro depth reached.~%")
        nil)
      (progn
        `(macrolet
           ((depth-counter-virtual-macro ()
              ,(1+ depth)))
           ,@body)))))
```

The following code will expand fine:

```
(depth-limited-macro 0
  (depth-limited-macro 1
    :test))
```

but after a small change it will print a warning and expand to nil:

```
(depth-limited-macro 0
  (depth-limited-macro 0
    :test))
```

This example is probably not very useful on its own except as an illustration and as a test case for code walkers.

5 IMPOSSIBILITY OF A GENERAL SOLUTION

In this chapter we present an example where a portable recursive macro expansion cannot guarantee correct expansion and explain why this code is problematic for macro expansion.

Consider the following code

```
(macrolet
  ((with-gensym ((name) &body body)
    `(macrolet ((,name () '',(gensym))) ,@body)))
  (with-gensym (f1)
    (with-gensym (f2)
      (defmacro set-x1 (value &body body)
        `(macrolet ((,(f1) () ,value))
           ,@body))
      (defmacro set-x2 (value &body body)
        `(macrolet ((,(f2) () ,value))
           ,@body))
      (defmacro read-x1-x2 (&environment env)
        `(list ',(macroexpand-1 `(,(f1)) env)
               ',(macroexpand-1 `(,(f2)) env))))))

(defmacro expand-via-function
  (form &environment e)
  `',(macroexpand-all (quote ,form) ,e))

(set-x1 1
  (set-x2 2
    (expand-via-function
      (set-x2 3
        (read-x1-x2)))))
```

If we replace the `expand-via-function` invocation with an `identity` function call, this code will return `(1 3)`. If `macroexpand-all` worked correctly, the unmodified code snippet would return `(list 1 3)`, because the macros named by the symbols returned by `(f1)` and `(f2)` would expand to `1` as defined by `(set-x1 1 ...)` and `3` as defined by the innermost `(set-x2 3 ...)`. Evaluating this expansion will provide `(1 3)`, as expected. Unfortunately, a portable implementation of a `macroxepand-all` function cannot expand such code correctly.

Note that the symbol naming the local macros defined by `set-x1` is not accessible to the `macroexpand-all` function. The `do-all-symbols` function iterates only on the symbols in the registered packages, and `gensym` produces symbols not accessible in any package; and the scope where the symbol was available does not contain the `macroexpand-all` function or even its call. It is also impossible to observe the internals of execution of the macro expansion function for the `read-x1-x2` macro.

But the `macroexpand-all` function needs to call the macro expansion function for `(read-x1-x2)` and pass it some environment. The Common Lisp standard does not provide any way of obtaining the environment except getting the current environment at the call position. We need the lexical environment to depend on the run-time input of the `macroexpand-all` function, so we need to use `eval`. The `eval` function evaluates in the null lexical environment, so the new environment will contain only the entries that the `macroexpand-all` function can name explicitly while constructing the form to evaluate.

Therefore we can either pass the initial environment (which contains the definition set by `set-x1` but also contains an obsolete definition set by `set-x2`), or construct a new environment, which can take into account the innermost `set-x2` invocation but cannot include any macro definition for the macro name defined by `set-x1`. Both options will cause `(read-x1-x2)` to expand to a wrong result.

6 PARTIAL SOLUTIONS

In this section we describe partial solutions that allow to provide `macroexpand-all` and code walking functionality for the simple cases and many of the most complicated ones.

6.1 Hardwiring specific macros

Given that `defun` is expanded into something non-walkable in most Common Lisp implementations, the walker can treat this macro as a special form and implement special logic to handle it. The same approach can be applied to all the macros as soon as an unwalkable expansion is observed in some Common Lisp implementation.

Note that while hardwiring macros can make some applications of a code walker less convenient, it doesn't sacrifice portability. The resulting code will still be legal even on the implementations where the workaround was not needed.

This workaround doesn't fully solve the problem of implementations expanding standard macros to non-compliant code, because a portable program could ask for an expansion of a standard macro and use it in a macro function for some user-defined macro. The following code fragment illustrates the problem:

```
(defmacro my-defun (&rest args)
  (macroexpand `(defun &rest args)))

(macroexpand-all `(my-defun f (x) x))
```

There is also a risk that an implementation would expand a standard macro to some code including a non-standard invocation of another standard macro; if only the second macro is hardcoded, the code walker will fail.

6.2 Recognizing named lambda by name

The most popular approach to the expansion of `defun` includes passing a non-standard argument to `function`. As a list starting with anything but `lambda` and `setf` is a non-standard argument, trying to interpret the symbol name won't break compatibility with an implementation that implements `function` without extensions. Apparently the forms with the first symbol being `named-lambda` or `lambda-block` are handled by the `function` special form in the same way in all the Common Lisp implementations using them.

This workaround relies on extrapolating behaviour of non-standardized forms based on unwritten traditions. The code walker has no way to know whether

```
(function (named-lambda f (x) y))
```

is a function called `f` returning `y` or a function called `y` returning `f`, but there are reasons to believe that the latter interpretation doesn't occur in practice. Here we have an inherent conflict between detecting the walker's failure and portability.

6.3 Macro-only expansion

In most cases the lexical environment for code walking is the lexical environment in the place of the call to the code walker. This situation allows the code walker to be implemented as a macro that does a single step of the expansion and leaves the recursive calls to itself in the expansion. The lexical environment will be handled by the Common Lisp implementation.

A minor drawback of this approach is a necessity for additional processing when the result of code walking should be put into a run-time variable. In other words, this approach requires additional processing to produce a quoted result.

A more significant limitation is related to the use of code walking with callbacks. Macro-only code walking is done in a top down manner, so the callbacks don't have access to the result of processing the subforms.

If the implementation provides a `macroexpand-all` function with an environment parameter, combining a code transformation implemented by a macro-based code walker and the `macroexpand-all` function yields a code walker that can be called as a function.

This approach fully solves the environment handling problem by asking the Common Lisp implementation to handle the environment, but shares the problems related to non-standard code in the expansions of standard macros.

In the following subsections we describe what can be done in order to construct a portable code walking function.

6.4 Start with the top level

Whilst it is impossible to augment a lexical environment in a portable way, it is easy to construct a lexical environment with given entries. So when a form is walked in the null lexical environment, the code walker can guarantee correct environment arguments for all macro expansions.

The limitations related to the expansions of the standard macros still apply in this case, but the environment handling can be made fully correct.

6.5 Guessing which environment to pass

Despite the fact that it is sometimes impossible to construct a correct environment for the call to `macroexpand-1`, there are cases where it is clear which environment to use. In other cases we can try to make a good guess.

If we have started from the null lexical environment, we can always create the environment from scratch. If we haven't yet collected any lexical environment entries that add or shadow any macro (or symbol macro) definitions, we can use the lexical environment initially provided by the caller of the code walker.

In the remaining case any guess can be wrong. We can try to improve our track record by using the initial environment if and only if we expand a macro defined locally in the initial environment. Otherwise we construct a lexical environment using the entries collected while processing the containing forms.

There is no way to check whether a guess is correct, and in some cases like the example presented in the section 5 both options are wrong. Moreover, there is no way for a code walker to check whether a macro expansion function defined outside of the form being walked uses macroexpand.

7 POSSIBLE APPLICATIONS OF CURRENTLY IMPLEMENTATION-SPECIFIC FUNCTIONALITY

7.1 Environment-augmenting functions

Having a macro `with-augmented-environment` that creates a lexical-scope dynamic-extent variable with an environment with specified additions with respect to an initial one would be enough for implementing a code walker following the standard approach currently taken by the implementation-specific walkers.

7.2 Using a recursive macroexpander to build a code walker

If a code walker has access to the `macroexpand-all` function which accepts an environment argument, there are two natural strategies. The first one is to call `macroexpand-all` before code walking, and code walk the expanded code without needing to expand any macros. The second one is to build `with-augmented-environment` using `macroexpand-all`. It can be done using code similar to the following example.

```
(defmacro with-current-environment
    (f &environment env)
  (funcall f env))

(macroexpand-all
  `(let ((new-x nil))
     (macrolet ((new-f (x) `(1+ ,x)))
       (with-current-environment ,(lambda (e) ))))
  env)
```

7.3 Using environment inspection for constructing augmented environments

Just having a list of all locally defined and shadowed names is enough to construct an augmented environment in a way that indistinguishable using only the standard facilities. The idea of the construction is to use a macro that obtains its environment, passes it to the function we want to call, quotes the result and returns the quoted result as the result of the macro expansion. An invocation of such a macro can be wrapped in to set up the correct environment, and the wrapped code can be evaluated using `eval`. We do the wrapping related to the desired changes first so that these forms are the innermost forms altering the environment. The evaluated code would be similar to the following example.

```
(defmacro with-current-environment
    (f &environment env)
  (funcall f env))

(eval
  `(let ((y-from-environment nil))
     (let ((new-x nil))
       (macrolet ((new-f (x) `(1+ ,x)))
         (with-current-environment
           ,(lambda (e) ...)))))
```

For each symbol we can query whether it defines a local macro (using `macro-function`) or a local symbol-macro (using `macroexpand` as the macro expansion of a symbol macro does not depend on anything). Having a name and a macro expansion function is enough to construct a wrapping `macrolet` form; having a name and an expansion is enough to construct a wrapping `symbol-macrolet` form. If it doesn't define a local macro or a local symbol macro we can use `let` or `flet` to ensure that global definitions are shadowed. Tags and block names cannot be inspected by macros. The list of names in the lexical environment will also be preserved.

Of course, it would be even better to have access to listing all the variable-like names, all the function-like (operator) names, all the tag names and all the block names. Distinguishing variables and symbol macros, and functions and macros can be done in the same way as before.

8 THE AGNOSTIC LIZARD LIBRARY

We present a new library for code walking, Agnostic Lizard. It implements all of the described workarounds.

For code walking and macro expansion it exports two functions and two macros.

The `macroexpand-all` function accepts two arguments, a form and an optional lexical environment object. It tries to perform recursive macroe xpansions and usually succeeds. The `macro-macroexpand-all` macro accepts a form, and expands it in the current lexical environment. The expansion is quoted, i.e. the runtime value of the generated code is the expansion of the initial form.

The `walk-form` function accepts a form, a lexical environment object (required argument, can be `nil`), and the callbacks as the keywords arguments. The callbacks can be:
`:on-every-form-pre` — called before processing each form in an executable position;
`:on-macroexpanded-form` — called for each form after possibly expanding its top operation, the hardwired macros are passed unexpanded;
`:on-special-form-pre` — called before processing a special form or a hardwired macro;
`:on-function-form-pre` — called before processing a function call;
`:on-special-form` — called after processing a special form or a hardwired macro;
`:on-function-form` — called after processing a function call;
`:on-every-atom` — called after processing a form expanded to an atom;
`:on-every-form` — called after expanding each form in an executable position.

The `macro-walk-form` macro accepts the form as a required argument, and the callbacks as keyword arguments. The callbacks have the same semantics as for `walk-form`. The expansion is the result of walking the form in the current lexical environment with the specified callbacks.

8.1 Implementation details

Agnostic Lizard mostly follows the design of the walkers using the implementation-specific environment inspection and manipulation functions. In other words, it starts at the top and recursively calls itself for the subforms, passing the updated environment information. It keeps track of the lexical environment changes in an object, and uses the initial environment only as a fallback. Agnostic Lizard always identifies macro invocations correctly, but it has to use heuristics for choosing what environment to pass. The code walker also tries to guess how to handle non-standard special forms in the expansions.

Agnostic Lizard defines three classes to handle code walking. The `metaenv` class contains the data directly describing the current walking context. It contains the list of defined functions and macros, variables and symbol-macros, blocks, and tags. It also keeps a reference to the Common Lisp implementation specific lexical environment object which has been initially passed by the caller.

This class stores just enough data to implement the basic recursive macro expansion. The `metaenv-macroexpand-all` generic function is used for dispatching the expansion logic. For the forms that are not `cons` forms with a hardwired operator the function calls `metaenv-macroexpand` first. The `metaenv-macroexpand` generic function contains all the decisions about the environment construction used at that step. Then the `metaenv-macroexpand-all` generic function passes the result of macro expansion or the original form to the `metaenv-macroexpand-all-special-form` generic function with the operator of the form as the first parameter. The methods of this generic function contain all the handling of the special operators and the hardwired macros These methods call `metaenv-macroexpand-all` for recursive processing of the subforms. Actually, the methods do little else: they clone create a new `metaenv` object with extra entries for the child forms (if needed), call `metaenv-macroexpand-all` and build the expanded form out of expansions of the subforms.

The class implementing the callbacks for code walking is `walker-metaenv`. It is a subclass of `metaenv`. Objects of the `walker-metaenv` class additionally store the callbacks to allow replacing the form in various stages of expansion. The only non-trivial method defined for this class is for the `metaenv-macroexpand-all` generic function. The method does the same operations as the method for the `metaenv` class, but optionally invokes the callback functions.

The class for macro-only walking is `macro-walker-metaenv`. It extends `walker-metaenv` with the data needed for macro-based walking: a boolean to alternate between creating a macro wrapper for capturing the updated environment and actually expanding the form using the captured environment; a callback for the macro creation step; and a label for letting the callbacks distinguish the parts of the code wrapped for further walking.

The macro-based recursive macro expansion that returns the code evaluated to the expanded form is done using the callbacks provided by the walkers. The callbacks walk the already walked part of the code and perform a transform similar to quasiquotation; they also make sure that the code that has not yet been expanded will be expanded in the correct environment despite such rewrite of the containing code.

8.2 Portability testing

We have tested Agnostic Lizard by checking that various forms return the same value before and after the expansion. We have written a small test suite to check the handling of local macros, and we have used the `iterate` library test suite to get special form coverage. Agnostic Lizard passed these tests when loaded under SBCL, ECL, CCL, ABCL and CLISP.

8.3 Reimplementing access to local variables

As a demonstration of the code walking interface we provide a partial portable reimplementation of the wrappers ensuring access to local variables during debugging [17]. Currently Agnostic Lizard does not provide an interface that would allow a callback to check if the processed form has the same environment as the parent one, so the wrapper saves the references to the lexical environment entries for every form. On the other hand, the macro code walker in the Agnostic Lizard library allows to get the same code-walking based functionality on a wider range of implementations.

The implementation of the wrapper presented in [17] uses `hu.dwim.walker` [11], which is the only implementation of a generic code walker we could find that was compatible with at least two implementations. Unfortunately, it has limited portability because of changes in some of the previously supported Common Lisp implementations, and it removes macro definitions completely. Agnostic Lizard lacks environment object handling features relying on implementation specific functions, but has better portability and preserves macro definitions.

9 CONCLUSION

We have shown that although a portable code walker for Common Lisp is impossible, it is possible to create an incorrect code walker that is wrong less often than the currently available ones.

9.1 Benefits for portable code walkers from hypothetical consensus changes in implementations

Portable code walkers suffer from two issues: non-standard expansions of standard macros and opaqueness of the lexical environment objects.

For most macros we hope that all the implementations with a release in the last couple of years already expand them to code using only standard invocations of macros and special forms. The only currently exceptional macros are `defun`, `defmacro` and `defmethod` that typically use non-standard `named-lambda` forms as arguments to `function` (or something very similar). There are benefits for storing the name of the function, and adopting Alexandria solution of using `labels` and saving the form definition would probably have some implementation cost. It is possible that the current practice goes against the intent of some parts of the Common Lisp standard, but there are non-trivial costs to changing it. It would be convenient to have at least a consensus symbol name and package name for `named-lambda` and `nfunction` without requiring neither to be present. It would also be nice to have an agreement that the expansions of standard macros should only use the special forms that have consensus names.

Opaqueness of lexical environments can be solved by having a consensus name for either augmenting the lexical environments or listing their entries. Listing the entries seems preferable because of the additional applications of such functionality, but using environment modification functions would probably provide better performance.

The `macroexpand-all` function has a recognizable name (although some Common Lisp implementations use different names, and of course there is no consensus package name), and its interface is clear and natural. It also allows implementing environment augmentation. It would still be useful to support listing the names in the lexical environment, but having portable access to a `macroexpand-all` implementation does allow implementing a portable generic code walker.

We believe that a good interface for a more general code walker requires some period of experimentation and evolution of alternative implementations. Therefore it is too early to promote a consensus name and interface for a generic code walker.

9.2 Further related issues that could benefit from consensus naming

In general it would be nice to have all a consensus package name for CLtL2 functionality not included in the ANSI Common Lisp standard in order to have a portable way to check which parts of this functionality are provided by an implementation. Many implementations de facto provide a large part of the difference, but they use different package names and sometimes also change function names.

Having a consensus name for a type for the lexical environment objects would make using generic functions more comfortable. This would allow writing portable generic functions handling both implementation defined lexical environment objects and library-defined enriched environments. This type can coincide with some other predefined type.

9.3 Future directions

Some interesting uses of code walking require processing large amounts of code. For such applications, performance of a code walker can be important. We haven't benchmarked Agnostic Lizard. It seems likely that some optimisations may be needed.

We have mentioned that most implementations expand `defun` to code containing non-standard special operators or non-standard uses of the `function` special form. In most cases such code is easy to analyze by using a few predetermined heuristics. It may be possible to further reduce the impact of this problem by expanding a `defun` form with the function name and argument names obtained by `gensym`, and analyzing where these names get mentioned.

It would be interesting to see how much of the code available via QuickLisp breaks after expansion by Agnostic Lizard. We haven't tried doing it yet.

Different tasks solved by code walking require different interfaces to be provided by the code walker. The callback interface of Agnostic Lizard is currently pretty minimalistic and should be expanded. Any advice and feedback are very welcome.

10 ACKNOWLEDGEMENTS

We would like to thank Irène Durand and the anonymous reviewers for their comments and advice.

REFERENCES

[1] Agnostic Lizard homepage. Retrieved on 30 January 2017 https://gitlab.common-lisp.net/mraskin/agnostic-lizard
[2] American National Standards Institute, 1994. ANSI Common Lisp Specification, ANSI/X3.226-1994. A hypertext version (converted by Kent Pitman for LispWorks) retrieved on 24 January 2017 from http://www.lispworks.com/documentation/HyperSpec/Front/index.htm
[3] Guy L. Steele. 1990. Common Lisp the Language, 2nd Edition. Also retrieved on 24 January 2017 from https://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html
[4] Jonathan Amsterdam. Don't Loop, Iterate. Working Paper 324, MIT AI Lab. Also retrieved on 24 January 2017 from https://common-lisp.net/project/iterate/doc/Don_0027t-Loop-Iterate.html Project homepage: https://common-lisp.net/project/iterate/
[5] Richard C. Waters. 1993. Some Useful Lisp Algorithms: Part 2. Tech. Rep. TR93-17, Mitsubishi Electric Research Laboratories. Also retrieved on 24 January 2017 from http://www.merl.com/publications/TR93-17
[6] Richard C. Waters. 1993. Macroexpand-All: an example of a simple lisp code walker. Newsletter ACM SIGPLAN Lisp Pointers. Volume VI Issue 1, Jan.-March 1993.
[7] John Fremlin. 2009. Macroexpand-dammit. Web Archive copy. Retrieved on 24 January 2017. https://web.archive.org/web/20160309032415/http://john.freml.in/macroexpand-dammit
[8] The freshest macroexpand-dammit fork repository. Retrieved on 24 January 2017. https://github.com/guicho271828/macroexpand-dammit
[9] Alex Plotnick. 2013. CLWEB homepage. Retrieved on 24 January 2017. http://www.cs.brandeis.edu/~plotnick/clweb/
[10] Christophe Rhodes. 2014. Naive vs proper code-walking. Retrieved on 24 January 2017. http://christophe.rhodes.io/notes/blog/posts/2014/naive_vs_proper_code-walking/
[11] hu.dwim.walker repository. Retrieved on 24 January 2017. http://dwim.hu/darcsweb/darcsweb.cgi?r=LIVE%20hu.dwim.walker;a=summary
[12] The Superior Lisp Interaction Mode for Emacs (SLIME) project repository. Retrieved on 24 January 2017. https://github.com/slime/slime
[13] trivial-macroexpand-all repository. Retrieved on 24 January 2017. https://github.com/cbaggers/trivial-macroexpand-all
[14] SICL homepage. Retrieved on 24 January 2017. https://github.com/robert-strandh/SICL
[15] Christian E. Schafmeister. 2015. Clasp - A Common Lisp that Interoperates with C++ and Uses the LLVM Backend. In proceedings of ELS2015. Retrieved on 24 January 2017. http://european-lisp-symposium.org/editions/2015/ELS2015.pdf Project repository: https://github.com/drmeister/clasp
[16] GNU CLISP homepage. Retrieved on 24 January 2017. http://www.clisp.org/
[17] Michael Raskin, Nikita Mamardashvili. 2016. Accessing local variables during debugging. In proceedings of ELS2016. Retrieved on 30 January 2017. http://european-lisp-symposium.org/editions/2016/ELS2016.pdf

Removing redundant tests by replicating control paths [*]

Irène Durand
Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
irene.durand@u-bordeaux.fr
robert.strandh@u-bordeaux.fr

ABSTRACT

We describe a technique for removing redundant tests in intermediate code by replicating the control paths between two identical tests, the second of which is dominated by the first. The two replicas encode different outcomes of the test, making it possible to remove the second of the two. Our technique uses local graph rewriting, making its correctness easy to prove. We also present a proof that the rewriting always terminates. This technique can be used to eliminate multiple tests that occur naturally such as the test for `cons`-ness when both `car` and `cdr` are applied to the same object, but we also show how this technique can be used to automatically create specialized versions of general code, for example in order to create fast specialized versions of sequence functions such as `find` depending on the type of the sequence and the values of the keyword arguments supplied.

CCS Concepts

•Theory of computation → Rewrite systems; •Software and its engineering → Compilers;

Keywords

Intermediate code, compiler optimization, local graph rewriting

1. INTRODUCTION

In a language such as Common Lisp [1], it is hard to avoid redundant tests, even if the programmer goes to great lengths to avoid such redundancies. The reason is that even the lowest-level operators in Common Lisp require *type checks* to determine the exact way to accomplish the operation, so that two or more calls to similar operators may introduce redundant tests that are impossible to eliminate manually.

[*]This work was supported by the French National Research Agency (ANR project GraphEn / ANR-15-CE40-0009).

As an example of such an introduction of redundant tests, consider the basic list operators `car` and `cdr`. We can think of these operators to be defined[1] in a way similar to the code below:

```
(defun car (x)
  (cond ((consp x) (cons-car x))
        ((null x) nil)
        (t (error 'type-error ...))))

(defun cdr (x)
  (cond ((consp x) (cons-cdr x))
        ((null x) nil)
        (t (error 'type-error ...))))
```

where `cons-car` and `cons-cdr` are operations that assume that the argument is of type `cons`. These operations are implementation defined and not available to the application programmer.

Now consider some typical[2] use of `car` and `cdr` such as in the following code:

```
(let ((a (car x))
      (b (some-function)
      (c (cdr x)))
  ...)
```

After the inlining of the `car` and `cdr` operations, the code looks like this:

```
(let ((a (cond ((consp x) (cons-car x))
               ((null x) nil)
               (t (error 'type-error ...)))
      (b (some-function)
      (c (cond ((consp x) (cons-cdr x))
               ((null x) nil)
               (t (error 'type-error ...)))
  ...)
```

We notice that the test for `consp` occurs twice, and that the second occurrence is *dominated by* the first one, i.e.,

[1]For these particular operators, an implementation may use some tricks to avoid some of these tests, but such tricks are not generally usable for other operators. We still prefer to use `car` and `cdr` as examples, because their definitions are easy to understand.
[2]In this case, the programmer might use the standard macro `destructuring-bind`, but for reasons of simplicity, that macro will very likely expand to calls to `car` and `cdr`, rather than to some implementation-specific code that avoids the redundant tests.

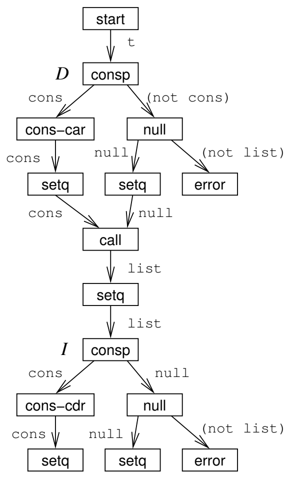


Figure 1: Control flow generated by a typical compiler.

every control path leading to the second occurrence must pass by the first occurrence as well.

As a consequence, the outcome of the second test for `consp` is always the exact same as the outcome of the first one. Unfortunately, this fact can not be easily exploited. To see why, we need to study the intermediate code generated by a typical compiler as a result of compiling the example program. The result is shown in Figure 1.

As Figure 1, shows, the intermediate code takes the form of a *control-flow graph* in which the nodes are *instructions* and the arcs represent the control flow. When we use the terms *predecessor* and *successor*, they refer to the relationship between two instructions in the control-flow graph, as defined by the control arcs.

In Figure 1, we have omitted references to data so as to simplify the presentation. In this intermediate representation, we have also eliminated scoping constructs, so that liveness of a variable is defined to be between its first assignment and its last use. Each control arc is annotated by a type descriptor, indicating the type of the variable `x` at that point in the execution of the program.

After the first `let` binding has been executed, the control arc with the type `cons` and that with the type `null` both arrive at the instruction that calls `some-function` which is the start of the second `let` binding. As a consequence, after the second `let` binding has been established, the type information available for `x` is `(or cons null)`, which is the same as `list`.

In order to avoid the second test for `consp`, we need to *replicate* the instructions corresponding to the establishment of the second `let` binding. In this paper, we introduce a technique for accomplishing this replication using *local graph rewriting*. The advantage of this technique is that it is very simple to implement, and that its semantic soundness is trivial to prove. We also prove that the technique always terminates, no matter how complicated the intermediate computation between the two tests.

2. PREVIOUS WORK

Mueller and Whalley [4] describe a technique for avoiding conditional branches by path replication. Their work includes heuristics for determining whether such replication is worthwhile. However, their technique for replicating the paths is not based on graph rewriting, and they do not supply a proof that their technique is correct.

To our knowledge, no existing research uses our technique based on local graph rewriting, and we are unaware of any existing Common Lisp compilers that implement it.

3. OUR TECHNIQUE

3.1 General description

Our technique consists of applying *local graph rewriting* to the graph of instructions in intermediate code. Local graph rewriting has the advantage of being simple, both to implement and when it comes to proving correctness.

For the purpose of this paper, we assume that some initial phase has determined that the following conditions are respected:

1. there are two instructions, D and I, in the program that are identical tests,
2. the variable being tested is the same in D and I,
3. D dominates I, and
4. the variable being tested is not assigned to in any path from D to I.

In a real compiler, such a phase probably does not exist. Some conditions are easier to verify if the compiler translates the intermediate code to SSA form [2, 3], and some conditions can be verified during the execution of our technique, avoiding the need to include them in a separate phase. The last condition may require an analysis for detecting variables that are assigned to in nested functions. Such a phase is present in most optimizing compilers in order to determine where to allocate space for such variables.

In Figure 1, the two test instructions labeled D and I respectively verify the conditions listed above, and we will use these two instructions to illustrate our technique.

During the execution of our algorithm, the instruction I will be *replicated*, so that it is part of some set S in which every replica remains dominated by D. Initially, S contains I as its only element.

In our technique, we keep track of the outcome of the test in the *control arcs* of the control-flow graph. We can think of this information as being represented as *labels* associated with control arcs:

- An arc is unlabeled if we have no information concerning the outcome of the test at that point in the program.
- An arc is labeled *true* if the outcome of the test at that point in the program is known to be true.
- An arc is labeled *false* if the outcome of the test at that point in the program is known to be false.

Initially, only the outgoing arcs of D and I have a label.

Our technique involves the repeated application of the first applicable rewrite rule in the following list to some arbitrary element of S, say s, that does not itself have an immediate predecessor in the control-flow graph that is also an element of S:

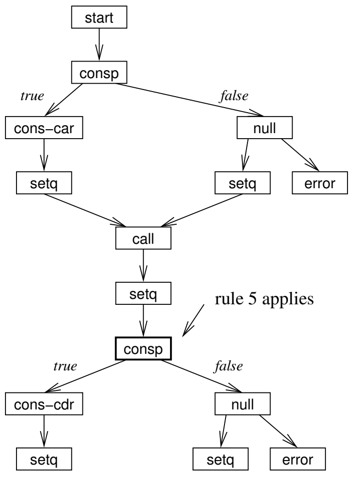


Figure 2: Initial instruction graph.

1. If s has no predecessors, then remove it from S.
2. If s has an incoming arc labeled *true*, then change the head of that arc so that it refers to the successor of s referred to by the outgoing arc of s labeled *true*.
3. If s has an incoming arc labeled *false*, then change the head of that arc so that it refers to the successor of s referred to by the outgoing arc of s labeled *false*.
4. If s has $n > 1$ predecessors, then replicate s n times; once for each predecessor. Every replica is inserted into S. Labels of outgoing control arcs are preserved in the replicas.
5. Let p be the (unique) predecessor of s. Remove p as a predecessor of s so that existing immediate predecessors of p instead become immediate predecessors of s. Insert a replica of p in each outgoing control arc of s, preserving the label of each arc.

Rewrite rules are applied until the set S is empty, or until each element of S has an immediate predecessor in the control-flow graph that is also a member of S. An element of S could have an immediate predecessor like that if the dominated instruction I were part of a loop. We need to exclude such elements, or else our technique might not terminate in all cases.

3.2 A simple example

Let us see how our technique works on the example in Figure 1. The initial situation is shown in Figure 2. The instructions that are members of S are drawn with a slightly thicker box.

As Figure 2 shows, the second `consp` is dominated by the first, so it becomes the only member of the set S. The last rewrite rule applies to the second `consp` so that the `setq` is replicated as its successors. The result of this first rewrite is shown in Figure 3.

As we can see in Figure 3, the last rewrite rule applies again resulting in the replication of the `call`. The result after the second rewrite is shown in Figure 4.

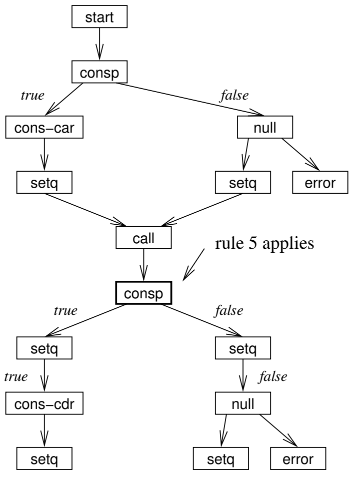


Figure 3: Result after one rewrite.

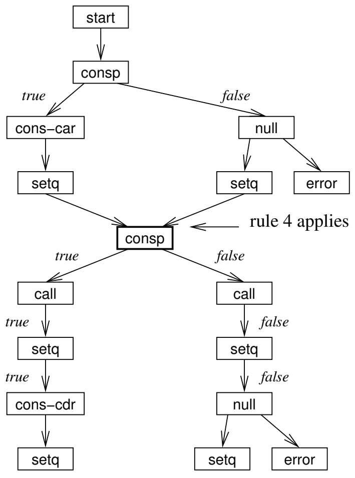


Figure 4: Result after two rewrites.

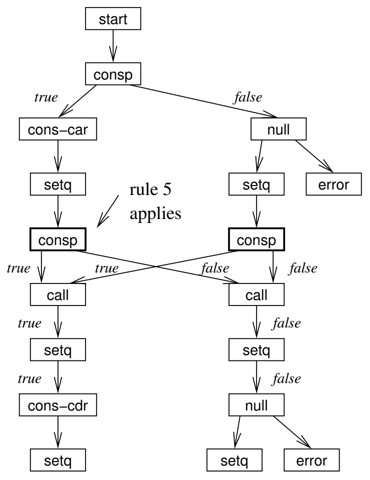


Figure 5: Result after replicating the test.

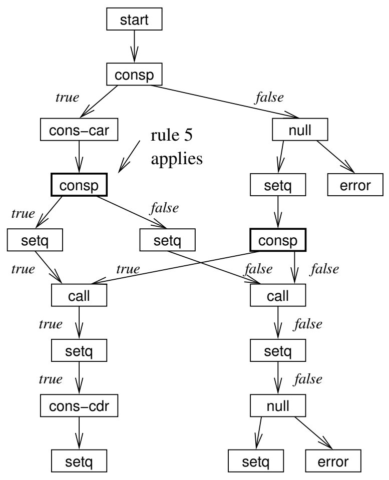


Figure 6: Result after replicating `setq`.

As we can see in Figure 4, the second `consp` now has two predecessors, and both incoming arcs are unlabeled. Therefore, rewrite rule number 4 applies and the `consp` is replicated. As a result, S now has two members. The result of applying this rule is shown in Figure 5.

We now choose the leftmost replica of the second `consp` to apply our rules to. It has a single predecessor with an unlabeled incoming control arc, so the last rewrite rule applies. We replicate the `setq` in both branches of the test, giving us the result shown in Figure 6.

In Figure 6, the last rewrite rule applies again, and we replicate the `cons-car`, giving us the situation shown in in Figure 7.

As Figure 7 shows, the `consp` instruction now has a single predecessor, but the incoming arc has a known outcome of the test, namely *true*. Therefore, rewrite rule number 2 applies. The left outgoing arc of the first `consp` is redirected to go directly to the `cons-car` instruction. The result of applying this rule is shown in Figure 8.

At this point, the `consp` that we have been processing

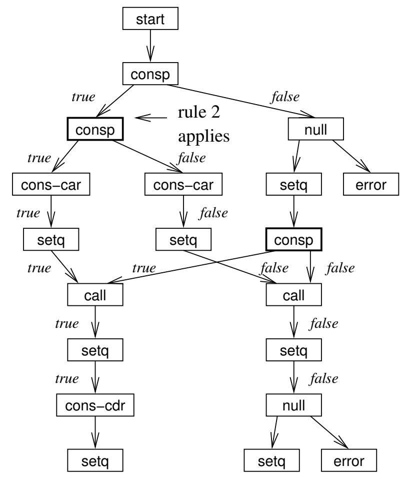


Figure 7: Result after replicating `cons-car`.

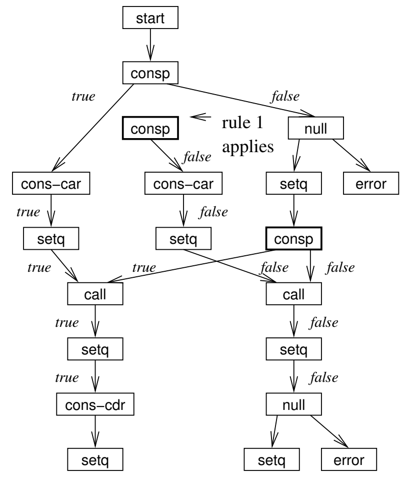


Figure 8: Result after short-circuit `consp`.

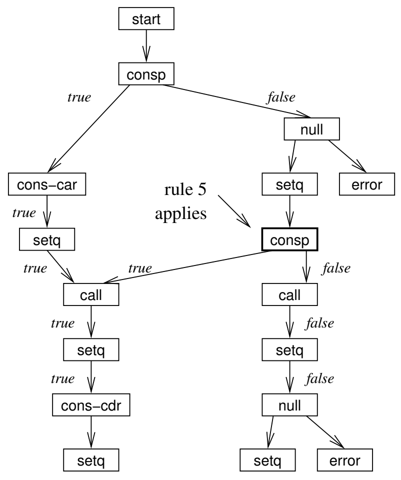


Figure 9: Result after removing unreachable instructions.

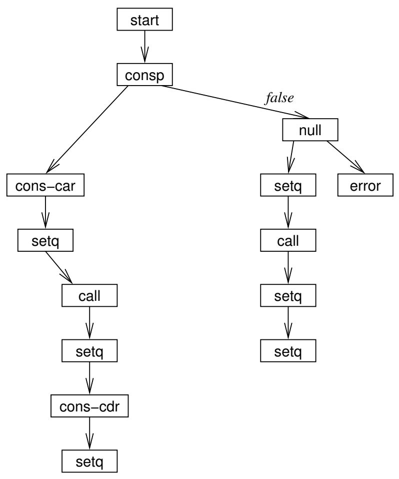


Figure 10: Final result.

has no predecessor. Therefore we apply rule number 1 and remove it from S. Removing all instructions that can not be reached from the start instruction gives the situation shown in Figure 9.

Analyzing Figure 9, we can see that if the result of the first `consp` yields *true*, then no second test is performed. Instead, the variable `a` is set to the result of the instruction `cons-car`, the variable `b` is set to the result of the call, and the variable `c` is set to the result of the instruction `cons-cdr`. Applying the same rules to the remaining `consp` instruction in S and then to the second `null` instruction (which is now dominated by the first), yields the final result shown in Figure 10.

This example represents a control graph that is particularly simple, in that there are no loops between the first and the second `consp` instructions. Our technique must obviously work no matter the complexity of the control graph, as long as the first test dominates the second.

3.3 Proof of correctness and termination

The correctness of our technique is easy to prove, simply because each rewrite rule preserves the semantics of the program. The last rewrite rule preserves the semantics only under certain circumstances which are easy to verify:

- The predecessor does not assign to a lexical variable that is read by the test instruction. This condition is respected because we have assumed that the variable being tested is not assigned to in any path between the first and the second occurrences of the test, as condition number 4 in Section 3.1 requires.
- The predecessor must not have any other side effect that may alter the outcome of the test. By restricting the test to lexical variables, this restriction is also respected.

Termination is a bit harder to prove. One way is to find some non-negative *metric* that can be shown to strictly decrease as a result of the application of each rewrite rule. We have not found any such metric. However, this conundrum can be avoided by a simple *grouping* of the rewrite rules. This grouping is not required to be present in the implementation of our technique, only in the termination proof.

To see how the rewrite rules can be grouped, consider a general case where the test instruction has some arbitrary number of labeled or unlabeled incoming control arcs. Rules number 2 and 3 are first applied a finite number of times. What happens next depends on the number n of unlabeled incoming control arcs:

- If $n = 0$ the first rewrite rule applies, in which case the instruction is removed from the set S.
- If $n = 1$, the last rewrite rule is applied. The crucial characteristic of this rewrite rule is that the total number of unlabeled control arcs decreases by one.
- If $n > 1$, rewrite rule number 4 is applied. Notice that the number of unlabeled control arcs is not modified by the application of this rule.

For the purpose of this proof, we assume that the individual rewrite steps in a group happen immediately after each other, so that for a particular instruction, the labeled incoming control arcs are first eliminated, the same instruction is then potentially replicated, and finally, the last rewrite rule is applied to one of the replicas. However, the implementation does not have to work that way in order for termination to be certain.

In other words, we can create groups of rewrite steps, where a group can be formed according to one of the following *group types*:

A. A group in this type has a finite number of applications of rewrite rules number 2 and 3, followed by a single application of rewrite rule number 1.

B. A group in this type has a finite number of applications of rewrite rules number 2 and 3, followed by a single application of rewrite rule number 5.

C. A group in this type has a finite number of applications of rewrite rules number 2 and 3, followed by a single application of rewrite rule number 4, followed by a single application of rewrite rule number 5.

With this information, we can create a metric consisting of a pair (U, N), where U is the total number of unlabeled control arcs of the program and N is the number of elements of the set S. Two pairs can now be compared using a *lexicographic order*, so that for two pairs (U_1, N_1) and (U_2, N_2), (U_1, N_1) is *strictly smaller than* (U_2, N_2), written $(U_1, N_1) < (U_2, N_2)$, if and only if either $U_1 < U_2$ or $U_1 = U_2$ and $N_1 < N_2$.

THEOREM 1. *The rewrite algorithm terminates.*

PROOF. As a result of a rewrite according to a group of type A, U remains the same, but N decreases by 1. As a result of a rewrite according to a group of type B or C, U decreases by 1 (but N may increase). Since U and N are both non-negative integers, we must reach a normal form after a finite number of rewrites. □

The following table illustrates how the grouping technique applies to our example:

Initial	U	N	Group	Final	U	N
Figure 2	11	2	B	Figure 3	10	2
Figure 3	10	2	B	Figure 4	9	2
Figure 4	9	2	C	Figure 6	8	3
Figure 6	8	3	B	Figure 7	7	3
Figure 7	7	3	A	Figure 9	7	2

Each row in the table represents a group of rewrites. For each rewrite group, we give the figure representing the initial state with the values of U and N for that figure, followed by the figure representing the final state with the values of U and N after the application of the rewrite steps in the group.

4. BENEFITS OF OUR TECHNIQUE

The main benefit of our technique is its simplicity. This simplicity is important both in terms of its implementation and in terms of ensuring termination for all possible control graphs.

As the example in Section 3.2 shows, redundant tests can be avoided in cases where it is not possible to express this redundancy by the use of any portable lower-level constructs. Situations similar to the one in the example occur naturally in many programs:

- If the same variable is used in more than one consecutive numeric operation, then there will be redundant tests to determine the exact numeric subtype of the contents of that variable. An important special case is to determine whether a particular value is of type `fixnum`.

- If a variable holding an array is used in more than one consecutive operation to access some element, then there will be redundant tests to determine various aspects of the array that influence the way the indices and the elements are handled, such as the upgraded element type and whether the array is simple or not.

A particularly interesting special case of these situations occurs when the dominated test is part of a loop. It is particularly interesting, because our technique will then eliminate a test that would otherwise potentially be executed a large number of times. This feature can be taken advantage of in a highly portable version of some of the Common Lisp *sequence functions*. By duplicating a very general loop in every branch of a multiway test for keyword arguments such as `test`, `test-not`, and `end`, each copy of the loop will automatically be simplified differently according to the particular branch it occurs in.

Our technique has some disadvantages as well. First of all, the size of the code may increase, which can have a negative influence on cache performance, especially when different invocations of the code result in different results of the test. In fact, if several variables with overlapping regions of liveness are processed by our technique, the result may be an *exponential blowup* of the size of the code in the overlapping region. It is hard to quantify the increase in code size, because it requires precise definitions of overlapping regions of liveness, and we have not yet defined such metrics. For that reason, it is outside the scope of this paper to discuss heuristics that will determine the conditions for applying our technique, but such conditions are required to avoid such problematic effects.

The increase of the size of the code automatically means longer compilation times as well. Techniques that work on global information about the program can avoid some of these disadvantages, at the cost of increased complexity compared to our simple local rewrite technique.

5. CONCLUSIONS AND FUTURE WORK

We have defined a technique for eliminating redundant tests in intermediate code. The technique relies on replication of code paths between two identical tests. So far, our technique only defines a *mechanism* for achieving the result. It does not yet define a *policy* stating when the technique should be applied.

The question of policy is an important one, because with a large number of redundant tests in the intermediate code, there is a possibility for *exponential blowup* of the code size. Future work involves defining a reasonable policy to avoid such pathological cases.

The technique described in this paper will become available as one of the optimization techniques provided by the Cleavir compiler framework that is currently part of the SICL project.[3] Only then will it be possible to determine the exact characteristics of our technique in terms of applicability, computational cost, performance gain of compiled code, and size increase of typical programs.

6. ACKNOWLEDGMENTS

We would like to thank Philipp Marek for providing valuable feedback on early versions of this paper.

7. REFERENCES

[1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp.* American National Standards Institute, 1994.

[2] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 25–35, New York, NY, USA, 1989. ACM.

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

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct. 1991.

[4] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 56–66, New York, NY, USA, 1995. ACM.