



Proceedings of the
15th European Lisp Symposium

Porto, Portugal
April 21 — 22, 2022
In cooperation with ACM



ISBN-13: 978-2-9557474-6-9
ISSN: 2677-3465

Preface



Message from the Program Chair

Welcome to the 15th European Lisp Symposium!

It is my pleasure to off these proceedings to the community. Herein you will find descriptions of keynote presentations by Sam Ritchie and Robert Strand, to both of whom I hold the utmost appreciation for agreeing to present their work to this symposium. Additionally, you'll find the research papers and demo descriptions submitted by nine researchers. I would like to thank everyone who made a submission to this year's symposium.

A special thanks goes out to the chairing committee who had the task of reviewing the submissions, and giving feedback to the authors. This work is mostly done in silence and may not be appreciated by the symposium attendees. So again thank you for your work.

Thank you to the vitrualization team, Georgiy and Michał.

Thank you to the organizers of the <Programming> conference and to FEUP, Universidade do Porto who are hosting the venue.

Finally, thank you to all the attendees. I hope you enjoy the symposium, and that you find something helpful and inspiring.



Georgiy Tugai (left) and Michał Herda (right).

Post-Symposium Message from Virtualization Team

Let's start with bad news: COVID is still a thing in 2022, and I (Michał) was struck by it not even a week before the conference, which meant that I was stuck at home for the conference itself. That meant a change of plans, since I was supposed to do some on-site help!

The good news: the conference has nonetheless happened, and, for the first time ever, it has happened in a hybrid setting—with people participating both on-site in Porto and online via Twitch and IRC! This is thanks to the titanic work done mostly by Georgiy and the on-site technicians in Porto. I am greatly thankful to all of them and to all of the people who have made ELS possible - online and offline participants, technicians, and organizers of <Programming>.

Huge thanks to everyone, glad to see that we were able to make the hybrid ELS happen despite all the technical troubles—and, see you next year, hopefully in the flesh this time!

Organization

Symposium Organizer

- Didier Verna, EPITA, France

Programme Chair

- Jim Newton, EPITA, France

Virtualization Team

- Georgiy Tugai
- Michał Herda

Programme Committee

Philipp Meier	Nubank
Ioanna M. Dimitriou H.	Igalia
Mikhail Raskin	Technical University of Munich
Nick Levine	RavenPack
Adrien Pommellet	LRDE, EPITA
Marco Heisig	Friedrich–Alexander–Universität Erlangen
Alberto Riva	Bioinformatics Core, ICBR, University of Florida
Marco Antoniotti	DISCo, Università degli Studi di Milano-Bicocca
Nicolas Neuss	Friedrich–Alexander–Universität Erlangen
Christophe Rhodes	Google UK
Irène Anne Durand	LaBRI, University of Bordeaux
Ralf Moeller	Universität zu Lübeck
Breannán Ó Nualláin	University of Amsterdam
Marc Battyani	Fractal Concept
Pascal Costanza	Intel
Sky Hester	Private Consultant



Sponsors

We gratefully acknowledge the support given to the 15th European Lisp Symposium by the following sponsors:



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



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



SYSCOG
Campo Grande, 378 – 3
1700–097 Lisboa
Portugal



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

Invited Contributions

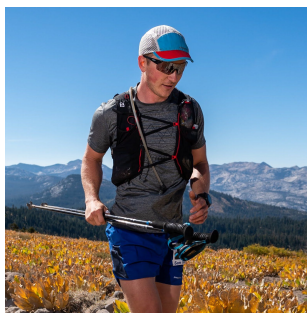
Lisp as Renaissance Workshop: A Lispy Tour through Mathematical Physics

Sam Ritchie, Mentat Collective, Bulder, Colorado, USA

Lisp is an exquisite medium for the communication of computational ideas. From our most accurate observations of physical reality up through chemistry, biology, and cognition, the universe seems to be computing itself; modeling and simulating these systems in machines has led to incredible technological wealth.

Deep principles and beautiful abstractions seem to drive these systems, but they have always been hard to discover; and we are floundering at the computational frontiers of intelligence, synthetic biology and control systems for our climate. The only way to push forward is to build powerful tools that can communicate and teach.

This talk will take a tour through SICMUtils, a Lisp system designed as a workshop for conducting serious work in mathematical physics and sharing those explorations in a deeply interactive, multiplayer way. The library's growth parallels our human scientific history; hopefully tools like this will help us write the next chapter.



Sam Ritchie is a researcher at the Mentat Collective, and currently working on a series of interactive, multiplayer computational textbooks for exploring mathematical physics and other forms of modeled reality. He has lived past work-lives at (Google) X, Stripe, Twitter, founded Paddleguru and Racehub; He is most well known in the software world as the author of Summingbird, Algebird, and SICMUtils, and as the maintainer of Cascalog. He even has a secret identity as a mountain athlete and amateur aircraft mechanic, and live with wife Jenna and daughter Juno in Boulder, Colorado.

Building SICMUtils, the Atelier of Abstractions

Sam Ritchie, Mentat Collective, Bulder, Colorado, USA

SICMUtils is a Clojure library designed for interactive exploration of mathematical physics. It is simultaneously a work of persuasive writing, a collection of essays on functional pearls and computational ideas, a stable of workhorse functional abstractions, and a practical place to work and visualize algorithms and physical systems, on a server or in the browser.

How do you build a library like this? This talk will go through the architecture of SICMUtils, based on many of the ideas of "additive programming" from Gerald Sussman and Chris Hanson's latest book, *Software Design for Flexibility*. We'll look at surprising examples of the system becoming easier to extend over time. Clojure's embrace of its host platform lets us use the best modern work in Javascript for visualization, while keeping the horsepower of our servers for real work. Lisp's particular elegance will shine throughout.

Creating a Common Lisp Implementation

Robert Strandh, Bordeaux, France

Being dissatisfied with the way current Common Lisp implementations are written, and with the duplication of system code between different implementations, we started the SICL project in 2008. The initial idea was to create modules that the creators of Common Lisp implementations could use to create a complete system from an initial minimal core. But this idea was unsatisfactory because it required each module to be written in a subset of Common Lisp. So instead, we decided to use the full language to implement these modules, effectively making them useless to an implementation using traditional bootstrapping techniques. We therefore decided to also create a new Common Lisp implementation (also named SICL), that could use those modules. A crucial element is a bootstrapping technique that can handle these modules. In this spirit, we have developed several modules, including an implementation of CLOS which is also an important element of bootstrapping. Lately, we have increased our level of ambition in that we want to extract those modules as separate (and separately maintained) repositories, which requires us to deal with code during bootstrapping that was not specifically written for SICL. In our talk, we describe this evolution of ambition, and its consequences to bootstrapping, in more detail. We also give an overview of several new techniques we created, some of which have been published (at ILC and ELS) and some of which have not. Finally, we discuss the future of the project, and other projects for which we imagine SICL to be a base.



Recently retired, Robert Strandh can look back at a lifelong experience in computer science and software development both in academia and industry, from 5 countries on 4 continents.

Currently, Strandh's projects are focused on the implementation of dynamic programming languages, as well as on operating-system technology in view of progress in computer and software technology during the past few decades.

Program overview

Monday Morning 21 March 2022

08:30–09:00		Registration, Badges, Meet and Greet
09:00–09:15		Welcome Message
09:30–10:00	Research Paper	Miguel Marcelino and António Leitão Transpiling Python to Julia using PyJL
10:00–10:30		Coffee break
10:30–11:30	Keynote	Sam Ritchie Lisp as Renaissance Workshop: A Lispy Tour through Mathematical Physics
11:30–12:00		Group Activity
12:00–13:30		Lunch

Monday Afternoon 21 March 2022

13:30–14:00	Research Paper	Michael Raskin QueryFS: compiling queries to define a filesystem
14:00–14:30	Research Paper	Robert Strandh and Irène Anne Durand A CLOS protocol for lexical environments
14:30–15:00	Demo	Max-Gerd Retzlaff IoT devices and embedded systems with uLisp
15:00–15:30		Coffee break
15:30–16:00	Remote Demo	Andrew Sengul April APL Compiler
16:00–16:30	Research Paper	Marco Heisig and Harald Koestler Closing the Performance Gap Between Lisp and C
16:30–17:00		Enlightening Lightning Talks

Tuesday Morning 22 March 2022

08:30–09:00		Meet and Greet
09:00–09:30	Research Paper	Stephan Monnier Open Closures: Disclosing lambda's inner monomaniac object!
09:30–10:00	Demo	Didier Verna ETAP: Experimental Typesetting Algorithms Platform
10:00–10:30		Coffee Break
10:30–11:30	Keynote	Robert Strandh Creating a Common Lisp Implementation
11:30–12:00	Demo	SICL
12:00–13:30		Lunch

Tuesday Afternoon 22 March 2022

13:30–14:00	Demo	Mermin Muñoz CEDAR
14:00–15:00	Keynote Demo	Sam Ritchie: Building SICMUtils, the Atelier of Abstractions
15:00–15:30		Coffee Break
15:30–16:00	Research Paper	Michael Wessel An Ontology-Based Dialogue Management Framework for Virtual Personal Assistants in Common Lisp
16:00–16:30	Research Paper	Turgut Reis Kursun, Jens Van der Plas, Quentin Stivenart, and Coen De Roover RacketLogger: Logging and Visualising Changes in DrRacket
16:30–17:00		Enlightening Lightning Talks
17:00–17:15		Closing Ceremony
17:15		Conference End

Monday, 21 March 2022



Open Closures

Disclosing lambda’s inner monomaniac object!

Stefan Monnier

monnier@iro.umontreal.ca

Université de Montréal

Département d’Informatique et Recherche Opérationnelle

Montréal, QC, Canada

ABSTRACT

While folklore teaches us that closures and objects are two sides of the same coin, they remain quite different in practice, most notably because closures are opaque, the only supported operation being to call them.

In this article we discuss a few cases where we need functions to be less opaque, and propose to satisfy this need by extending our beloved λ so as to expose as sorts of record fields some of the variables it captures. These *open closures* are close relatives of CLOS’s *funcallable objects* as well as of the *function objects* of traditional object-oriented languages like Java, except that they are functions made to behave like objects rather than the reverse.

We present the design and implementation of such a feature in the context of Emacs Lisp.

CCS CONCEPTS

• **Software and its engineering** → **Data types and structures; Procedures, functions and subroutines; Functional languages; Object oriented languages**; Integrated and visual development environments.

KEYWORDS

Functional programming, Function objects, Translucent functions, Emacs Lisp

ACM Reference Format:

Stefan Monnier. 2022. Open Closures: Disclosing lambda’s inner monomaniac object!. In *Proceedings of the 15th European Lisp Symposium (ELS’22)*. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.5281/zenodo.6228797>

1 INTRODUCTION

Undergraduate programming language courses will often point out that one can implement objects (in the *object-oriented* meaning of the term) as functions, e.g. by making them take a “method name” as a first argument and dispatching to different behaviors based on that argument. Yet if we try to take the idea seriously, one quickly encounters significant drawbacks, whether it’s because of efficiency concerns, or because of the difficulty to give static types to the resulting code, or the inability to determine if a function

obeys this convention before calling it, or any number of other issues that may come up.

The reverse is a somewhat simpler story: an object can be used to implement a function, by simply arranging for that object to have just one method, variously called `run`, `call`, `exec`, or `apply`. Depending on the language, this can be syntactically cumbersome and verbose, and may sometimes require to explicitly specify the captured variables, but in terms of efficiency at least not much is lost by treating a function as an object limited to a single method (often called a *function object*).

So while in the world of object-oriented languages, it is very common to add support for functions by encoding them as *function objects*, in the world of functional programming languages objects are usually not encoded as functions but as tuples. Disregarding issues of aesthetics, the result may appear to be just as good since we get both functions and objects in either case. Yet, *function objects* actually provide a bit more flexibility because they are *simultaneously* functions and objects, which has no equivalent in the world of functional programming languages.

A notable difference between functions and objects in this respect is that functions are opaque: the only non-trivial operation allowed on a function is to call it, but calling a function is a very risky business if we don’t know what kind of function we’re dealing with. This is usually not a problem because the responsibility is traditionally on the code that provides the function to provide one that works adequately, not on the code that calls it. But *function objects* can offer more flexibility since they may come with a type and may also expose object attributes that can be read via accessors, so while most attributes as well as the code of their sole method may be just as opaque as that of a function, the object itself can reveal extra information when desired.

In this article, we will discuss some situations where this kind of information is needed, and based on those we show the design of *open closures* which are an extension of the usual functions with extra information exposed in the form of a type and a set of *slots* that can be reached via accessors. Good old λ can then be redefined as a bare-bones open closure whose type is trivial, with an empty set of slots. Note that the types used to classify those open closures fundamentally constrain the set of slots exposed. These can be seen as a constraint on the captured environment of closures, and should not be confused with the notion of type used more traditionally to classify functions according to their signature, i.e. the set of arguments that the function accepts and the values it returns. Those two notions of type are orthogonal and in this article we will not discuss the types in the sense of function signatures.

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

ELS’22, March 21–22, 2022, Porto, Portugal

© 2022 Copyright held by the owner/author(s).

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

<https://doi.org/10.5281/zenodo.6228797>

```

(defun compose-function (function where orig-fun)
  (cond
   ((eq where :override) function)
   ((eq where :before)
    (lambda (&rest args)
      (apply function args)
      (apply orig-fun args)))
   ((eq where :after)
    (lambda (&rest args)
      (apply orig-fun args)
      (apply function args)))
   ((eq where :around)
    (lambda (&rest args)
      (apply function orig-fun args))))))

(defun add-function (function where var)
  (set var (compose-function function where
                            (symbol-value var))))

```

Figure 1: Adding functions to a variable

2 MOTIVATING EXAMPLE

In this section we will see the main example that will help explain the design of our open closures.

In Emacs, we have many variables holding functions that are called in various circumstances, in order to be able to customize the behavior of commands. We generally call them *hooks*, but you can just as well think of them as callbacks. They take various forms, but the form of interest here is when a global variable (or an object’s slot) holds a single function.

When a *package* (the name we give to plugins, in Emacs) wants to affect the corresponding behavior, it will want to modify the function stored in this variable by composing the old and the new function. We could provide that functionality as shown in Figure 1.

This would work fine but comes with an annoyance and a serious problem. The annoyance is that when we try to debug this code, the composed function will not show us what it is made of, even if the provided function and the original *orig-fun* are named functions. It requires trained eyes looking at the innards of the closure to decipher what it is made of and reverse engineer where it may come from.

But the more serious problem comes when the package decides it does not want to modify that variable any more and hence wants to undo its changes. The easy solution is to stash the old value somewhere so we can restore it afterwards, but that only works if all the packages add and remove their modifications in a properly nested order, which is neither enforced nor desirable. For example, after:

```

(defvar my-var #'A)
(add-function #'B :after 'my-var)
(add-function #'C :after 'my-var)

```

my-var will hold an anonymous function which first calls A, then B, then C. And if the package that added B has buyer’s remorse, it would like to be able to do:

```
(remove-function #'B 'my-var)
```

$$\begin{array}{l}
 (\textit{index}) \quad i \in \mathbb{N} \\
 (\textit{expressions}) \quad e ::= c \mid x \mid (\text{olam } \overrightarrow{(x \ e)} (x_a) e_b) \\
 \quad \quad \quad \quad \quad \mid (\text{oapp } e_1 e_2) \mid (\text{oref } e \ i)
 \end{array}$$

Figure 2: Syntax of the open lambda calculus

After this call, we would like *my-var* to hold a function which calls A and then C, but there is no mechanism which would let *remove-function* extract the necessary information from *my-var* to construct this new function, because the function stored there is opaque.

Ideally, we would like to be able to test whether a given function is one of the wrappers built by *add-function*, and if so, we would like to be able to extract the “function” and the “*orig-fun*” from which they were built, as well as “*where*” they were composed.

In current functional programming language, this can only be achieved with a significant amount of extra work, and often with additional runtime costs when calling the function, such as an additional indirection if one uses a CLOS-style *funcallable object* [Kiczales et al. 1991]. This is particularly frustrating considering that the most common internal representation of those closures makes the corresponding information readily available, if only one were given a way to access it.

3 OPEN CLOSURES

We propose to solve the previous problem by opening up our lambda abstractions such that some of the captured variables can also be accessed from outside, like the slots of a tuple. The result is fundamentally a combination of a tuple and a function. It can be seen as a function with slots, or as a tuple with code.

Figure 2 shows the syntax of an *open lambda calculus* which exposes the core idea in a minimalist way. We use the convention that \vec{m} is a shorthand for $(m_1 \dots m_n)$, and $\overrightarrow{(a \ b)}$ is a shorthand for $((a_1 \ b_1) \dots (a_n \ b_n))$.

- c stands for a builtin constant.
- x is the usual variable reference.
- $(\text{oapp } e_1 e_2)$ is your usual function application, limited to a single argument without loss of generality.
- $(\text{olam } \overrightarrow{(x \ e)} (x_a) e_b)$ constructs a function, where (x_a) is the list of arguments, again limited to a single argument, e_b is the body, and $\overrightarrow{(x \ e)}$ is the (ordered) list of slots. The slots are accessible both internally and externally. For internal access, e_b can refer to the value of those slots using the slot’s name as a variable.
- $(\text{oref } e \ i)$ fetches the value of the i^{th} slot of the function e . The index is an immediate value rather than an expression only for the purpose of simplifying the static semantics of the language: in a dynamically typed language, i can be generalized to an arbitrary expression evaluating to an integer.

This calculus is a superset of the standard λ -calculus since we can encode “ $\lambda x.e$ ” as $(\text{olam } () (x) e)$ which are those functions that expose no slots, and hence upon which we cannot apply any “*oref*”.

And while we can of course encode the usual tuples (e_1, \dots, e_n) using a Church-style encoding, we can also encode them more

(values) $v ::= c \mid x \mid (\text{olam } \overrightarrow{(x v)} (x_a) e)$
 (ctxts) $E ::= \bullet \mid (\text{oapp } E E) \mid (\text{oapp } v E) \mid (\text{oref } E i)$
 $\mid (\text{olam } ((x_1 v_1) \dots (x_i E) \dots (x_n e_n)) (x_a) e_b)$

$e \rightsquigarrow e'$ | Small-step reduction of e to e'

$$\frac{}{(\text{oapp } (\text{olam } \overrightarrow{(x v)} (x_a) e) v_a) \rightsquigarrow e[\overrightarrow{v}, v_a/\overrightarrow{x}, x_a]} (\beta)$$

$$\frac{}{(\text{oref } (\text{olam } \overrightarrow{(x v)} (x_a) e) i) \rightsquigarrow v_i} (\pi) \quad \frac{e \rightsquigarrow e'}{E[e] \rightsquigarrow E[e']} (\pi)$$

$$\frac{x' \notin \text{fv}(e)}{(\text{olam } \overrightarrow{(x e)} (x_a) e_b) \rightsquigarrow (\text{olam } \overrightarrow{(x e)} (x') e_b[x'/x_a])} (\alpha_1)$$

$$\frac{x' \notin \text{fv}(e) \quad x' \notin \overrightarrow{x}, x_a}{(\text{olam } ((x_1 e_1) \dots (x_i e_i) \dots (x_n e_n)) (x_a) e_b) \rightsquigarrow (\text{olam } ((x_1 e_1) \dots (x' e_i) \dots (x_n e_n)) (x_a) e_b[x'/x_i])} (\alpha_2)$$

Figure 3: Dynamic semantics of the open lambda calculus

directly as functions of the form $(\text{olam } ((_ e_1) \dots (_ e_n)) (x) x)$ where the traditional projection operation “ $e.i$ ” is just $(\text{oref } e i)$.

3.1 Dynamic semantics

Figure 3 shows the corresponding dynamic semantics, with a call-by-value reduction strategy. The top of the figure defines the syntax of values v , which are a subset of valid expressions, as well as the syntax of evaluation contexts E which define where evaluation can take place in an expression. The semantics is defined as the small step relation $e \rightsquigarrow e'$. Rule β shows how the slot values are substituted into the body of a function, making them available internally, while rule π shows how oref accesses a slot’s value from outside. The contexts E together with the congruence rule CONG show where primitive reductions can take place and define a left-to-right evaluation order. The two α renaming rules, where fv returns the free variables of a term, are only intended to give further details about the intended semantics.

Notice that the access to slots is done by position rather than by name. In other words, slot names are only meaningful internally when accessing them from within the body of the function and are not exposed outside of the function, so they obey the usual α -renaming of variable bindings as evidenced by the α_2 rule. This simplifies the metatheory and lets us rely on the usual conventions to avoid issues linked to name capture [Urban et al. 2007].

Note also the absence of an η rule $(\text{olam } ? (x) (\text{oapp } e x)) \rightsquigarrow e$ because what to put into “?” depends on the slots exposed by e . In other words, while olam encodes the usual λ , it does not enjoy the same η -reduction rule.

3.2 Typing rules

While open closures were developed in the context of a dynamically typed language, they would work just as well in a statically typed context.

$\tau ::= \text{Int} \mid \dots \mid (\text{oarw } \overrightarrow{\tau} \tau_a \tau_r)$
 $\Gamma ::= \bullet \mid \Gamma, x : \tau$

$\Gamma \vdash e : \tau$ | e has type τ in environment Γ

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} (\text{VAR}) \quad \frac{\Gamma \vdash e : (\text{oarw } \overrightarrow{\tau} \tau_a \tau_r)}{\Gamma \vdash (\text{oref } e i) : \tau_i} (\text{REF})$$

$$\frac{\Gamma \vdash e_1 : (\text{oarw } \overrightarrow{\tau} \tau_a \tau_r) \quad \Gamma \vdash e_2 : \tau_i}{\Gamma \vdash (\text{oapp } e_1 e_2) : \tau_r} (\text{APP})$$

$$\frac{\Gamma, \overrightarrow{x}, x_a : \tau_a \vdash e_b : \tau_r \quad \forall i. \Gamma \vdash e_i : \tau_i}{\Gamma \vdash (\text{olam } \overrightarrow{(x e)} (x_a) e_b) : (\text{oarw } \overrightarrow{\tau} \tau_a \tau_r)} (\text{LAM})$$

Figure 4: Static semantics of the open lambda calculus

To make that clear, Figure 4 shows a possible simple type system for our calculus. Even without a need for static types, those rules can be helpful to clarify the intended semantics. The top of the figure defines the syntax of type environments Γ and of types τ , which can include any number of builtin types plus the new type of open closures that we denote as $(\text{oarw } \overrightarrow{\tau} \tau_a \tau_r)$ which is the type of functions that take an argument of type τ_a , return a result of type τ_r , and expose slots of type $\overrightarrow{\tau}$. Just like open closures are a fusion of a function and a tuple, these function types are a fusion of the traditional function types and the traditional tuple types.

The typing judgment has the form $\Gamma \vdash e : \tau$. The typing rules reflect the dual nature of our open closures as both functions and tuples: the APP rule is the same as the corresponding rule in the simply typed λ -calculus, except for the extra $\overrightarrow{\tau}$ annotation in $(\text{oarw } \overrightarrow{\tau} \tau_a \tau_r)$ which is simply ignored, and the REF rule similarly matches the classic rule for the operation that projects a specific slot from a tuple, except for the extra τ_a and τ_r annotations on $(\text{oarw } \overrightarrow{\tau} \tau_a \tau_r)$ which are similarly ignored. The more interesting rule is LAM : the right part of the premises corresponds to the usual premise for the construction of tuples, but the left part does not quite match the premise for the typing rule of the usual λ constructor because the body e_b is now typed in an environment that includes not only the argument x_a but also all the open closure’s slots \overrightarrow{x} .

3.3 Compilation

Looking at the syntax and semantics of the open lambda calculus, one may wonder why it makes sense to introduce these open closure objects with their dual tuple/function nature, since both the dynamic and the static semantics suggest that the result is not much simpler than if we had introduced tuples and functions separately.

The real motivation becomes apparent only once we consider the usual implementation of closures via closure conversion. Closure conversion turns closures into tuples which contains a reference to the code of the function plus the values of all the variables captured by the function. Our open closures take advantage of this representation to store their extra slots alongside the values of the captured variables. This way, a degenerate open closure with zero slots ends up represented exactly as a normal λ would, and the only

$$\begin{array}{l}
C\llbracket x \rrbracket_\sigma = \sigma(x) \\
C\llbracket e_1 e_2 \rrbracket_\sigma = (\text{let } x \ C\llbracket e_1 \rrbracket_\sigma \ (\text{call } (\text{ref } x \ 0) \ x \ C\llbracket e_2 \rrbracket_\sigma)) \\
C\llbracket \lambda x_a. e_b \rrbracket_\sigma = (\text{tuple } (\text{code } (x_c \ x_a) \ C\llbracket e_b \rrbracket_{\sigma_e}) \\
\qquad \qquad \qquad \sigma(y_1) \dots \sigma(y_m)) \\
\text{where } \vec{y} = \text{fv}(e_b) - \{x_a\} \\
x_c \text{ is fresh} \\
\sigma_e = \{x_a \mapsto x_a, \\
\qquad y_1 \mapsto (\text{ref } x_c \ 1), \dots, y_m \mapsto (\text{ref } x_c \ m)\}
\end{array}
\qquad
\begin{array}{l}
C\llbracket x \rrbracket_\sigma = \sigma(x) \\
C\llbracket (\text{oref } e \ i) \rrbracket_\sigma = (\text{ref } C\llbracket e \rrbracket_\sigma \ i) \\
C\llbracket (\text{oapp } e_1 \ e_2) \rrbracket_\sigma = (\text{let } x \ C\llbracket e_1 \rrbracket_\sigma \ (\text{call } (\text{ref } x \ 0) \ x \ C\llbracket e_2 \rrbracket_\sigma)) \\
C\llbracket (\text{olam } \overrightarrow{(x \ e)} \ (x_a) \ e_b) \rrbracket_\sigma = (\text{tuple } (\text{code } (x_c \ x_a) \ C\llbracket e \rrbracket_{\sigma_e}) \\
\qquad \qquad \qquad e_1 \dots e_n \ \sigma(y_1) \dots \sigma(y_m)) \\
\text{where } \vec{y} = \text{fv}(e_b) - \{x_a, x_1, \dots, x_n\} \\
x_c \text{ is fresh} \\
\sigma_e = \{x_a \mapsto x_a, \\
\qquad x_1 \mapsto (\text{ref } x_c \ 1), \dots, x_n \mapsto (\text{ref } x_c \ n), \\
\qquad y_1 \mapsto (\text{ref } x_c \ n+1), \dots, y_m \mapsto (\text{ref } x_c \ n+m)\}
\end{array}$$

Figure 5: Example of closure conversion

On the left, the algorithm for a plain λ -calculus and on the right the algorithm for our *open lambda calculus*.

cost of adding slots to an open closure is to increase the size of the tuple. It does not introduce any extra indirection nor add any extra cost when the function is called.

Let's write $C\llbracket e \rrbracket_\sigma$ the closure conversion of expression e where σ is a substitution used to remember how to access the free variables of e . And let's assume the following lower-level language for the target of the closure conversion:

$$\begin{array}{l}
(\text{exps}) \quad e ::= c \mid x \mid (\text{call } e_1 \ e_2 \ e_3) \mid (\text{code } (x_1 \ x_2) \ e) \\
\qquad \qquad \mid (\text{let } x \ e_1 \ e_2) \mid (\text{tuple } \vec{e}) \mid (\text{ref } e \ i)
\end{array}$$

In this language $(\text{code } (x_1 \ x_2) \ e)$ denotes a chunk of closed code that could hence be represented as a pointer to piece of machine code. Without loss of generality, we limit this language to have only functions (and functions calls) of exactly two arguments.

Figure 5 shows what the closure conversion algorithm may look like, first for the plain λ -calculus, and then for open closures, where we highlighted the parts that are affected by the slots of open closures. As you can see, in both cases a function is converted to something of the form $(\text{tuple } (\text{code } \dots) \dots)$ and the only significant change is the addition of one extra value per slot into the tuple.

We place the extra slots of the open closure at the beginning of the tuple, which thus push the values of captured variables to later slots of the tuple. This is done to make it easy to find the exposed slots of the open closure since it is independent of the number of captured variables. This choice is not the only one, of course, but it is the simplest here and makes for an efficient implementation of $(\text{oref } e \ i)$. An alternative would be to place the captured variables first and the extra slots later, in which case $(\text{oref } e \ i)$ would need to know the number of captured variables of the closure e , which could be stored for example just before the beginning of the code, so as not to increase the size of the tuples.

More generally, there are many other ways to represent closures than the *flat closures* used in this algorithm, and open closures do not impose the use of flat closures. The only requirement is that the extra slots's values be stored somewhere and that $(\text{oref } e \ i)$ be able to find those values, potentially with the help of some extra information stored somewhere in the closure e , such as alongside its code.

4 OCLOSURES IN EMACS LISP

The calculus in the previous section shows the core idea of open closures in a minimalist setting, but we have designed and implemented open closures in the context of the Emacs Lisp language, so we discuss here what such a functionality can look like in a real-life setting.

Emacs Lisp is a programming language that lacks any namespace management features, so every globally-visible definition is instead given a name which includes a “package prefix”. In the case of open closures, we chose the prefix “oclosure-” for its definitions and we call its (open) closures “OClosures”.

The most important difference between the previous calculus and OClosures is that additionally to carrying values in slots, OClosures come with a type. This can be thought of as forcing every open closure to have an extra slot, placed first, which contains that runtime type information. In practice it is implemented differently, because for technical reasons we decided to store that type in a different place than the first slot.

So the constructor of OClosures has the following form:

$$(\text{oclosure-lambda } (\text{type } . \overrightarrow{(x \ e)}) \ \text{args } e_b)$$

Where *args* follows the usual format of Emacs Lisp formal arguments. The type can be retrieved with `oclosure-type`. We added this type information so as to be able to perform type tests and type-based dispatch, by integrating the feature with the rest of our CLOS-inspired object system.

For example, in the case of the composed functions presented in Section 2, we called the type of those OClosures `advic-e`. This is useful in `remove-function` where can now distinguish the case where `myvar` contains an `advic-e`, so we know we can look at its slots to find its component functions. It also lets us change the printer by defining a new method which dispatches on the specializer `advic-e` so as to print those functions in a more human-friendly way.

4.1 OClosure types

Of course, before using a new type, we need to define it. While the types of open closures in Section 3.2 constrain both the set of slots and the signature of the function, OClosure types leave the arity and return types unconstrained, and only specify their slots. While positional access to slots was convenient for our little calculus, it

```

(defun uncompose-function (function listfunc)
  (if (not (eql (oclosure-type listfunc) 'advice))
      listfunc
      ;; Nothing to remove.
      (let ((sva (slot-value listfunc 'car))
            (svd (slot-value listfunc 'cdr))
            (svw (slot-value listfunc 'where)))
          (if (eql sva function)
              svd
              ;; Found it!
              (compose-function
               sva svw
               (uncompose-function function svd))))))
)

(defun remove-function (function var)
  (set var (uncompose-function function
                                   (symbol-value var))))

```

Figure 6: Removing a function from a variable

is more convenient in real life to be able to access slots by name. Type definitions thus indicate the list of slots that are to be included for OClosures of that type. They work very much like Common Lisp’s `defstruct` and `defclass`. The syntax is loosely based on `defstruct`:

```
(oclosure-define (name . props) . slots)
```

Where *slots* is the list of slots included in this type, where each slots can come with some extra information, the only such extra information currently used is whether it’s mutable or not, the default being for slots to be immutable.

The type’s properties specified in *props* can include a list of parents, which allows subtyping, including multi-inheritance.

4.2 OClosure copies

Going back to our motivating example from Section 2, the function `add-function` can now create OClosures which work just as well as the old functions, but with extra information easily available. We can define the type `advice` of those OClosures as follows:

```
(oclosure-define (advice) car cdr where)
```

We chose `cdr` and `car` as the name of the slots holding resp. the original function and the added function because the repeated addition of functions creates a list structure.

The type information and the now exposed slots make it now possible for `remove-function` to do its job, by finding out the new set of functions to compose and reconstruct a new function after removing some element, as shown in Figure 6.

While this does work, we can do better if we consider that the last 4 lines of `uncompose-function` construct the same function as `listfunc`, except with a different `cdr`. For such use-cases, we have added the ability to perform functional updates of OClosures. We call them copiers. For example the previous type definition can be changed to:

```

(oclosure-define (advice
                 (:copier advice-with-cdr (cdr)))
  car cdr)

```

This defines a new function `advice-with-cdr` which will take an `advice` as first argument and any function as second argument and will return a new `advice` identical to the first except that its `cdr` slot will contain the function provided as second argument. With this function, we can simplify `uncompose-function` to:

```

(defun uncompose-function (function listfunc)
  (if (not (eql (oclosure-type listfunc) 'advice))
      listfunc
      ;; Nothing to remove.
      (let ((sva (slot-value listfunc 'car))
            (svd (slot-value listfunc 'cdr)))
          (if (eql sva function)
              svd
              ;; Found it!
              (advice-with-cdr
               listfunc
               (uncompose-function function svd))))))
)

```

Notice that we did not need the `where` slot any more nor did we have to call `compose-function` any more. A side effect is that this code is more efficient because it can blindly copy all the bits of `listfunc` and then just change the `cdr` slot, although this was not the motivation since speed of `remove-function` is not a concern.

OClosure copiers offer a second way to construct OClosures (besides `oclosure-lambda`) and they offer a limited way in which one can access the still opaque content of a closure, in the sense that they read the slots of the tuple containing the reference to the code and the values of captured variables that are not directly exposed as OClosure slots.

It should be noted that they impose an additional constraint on the system, in the sense that in order to be able to perform such a functional update, it is imperative that we be able to find all the places where the content of a slot are stored in the closure. In most closure representations, this is not a problem since the value of each captured variable is only stored in a single place, but there are exceptions such as when using run-time code generation to specialize the code of a closure to the particular values of the variables it captures [Lee and Leone 1996], or when the compiler notices that a captured variable always has the same value and decides to apply constant propagation to it.

4.3 Mutability

As mentioned earlier, when defining a type, each slot can be specified as being either mutable or immutable and that the default is for slots being immutable. Emacs Lisp is a language that is usually not in the business of preventing users from shooting themselves in the foot (preferring to merely try and make it easier for the users not to shoot themselves in the foot), so the choice of immutability deserves some explanation.

When a variable is both mutated and captured, the closure conversion will apply a *store conversion* to turn the variable into an immutable variable pointing to a “box” in which the real value is kept. This extra indirection can be avoided in some cases, but in the general case it is indispensable in order to handle a variable captured by several closures that need to share its state.

For this reason, when accessing the content of a slot, we need to know if that slot has been store-converted or not. One could store this auxiliary information alongside the code, inside a closure, but in order to make slot access more efficient, and to avoid having

to store that auxiliary information alongside the code, we decided instead to make this choice ahead of time in `oclosure-define`: if a slot is defined as mutable we simply force store-conversion on it.

Another, probably better option would be to never perform store-conversion on OClosure slots. Instead, such mutable slots would “live” in the OClosure object and any other closure that wants to refer to it will just have to keep a reference to the whole OClosure. This requires more changes in the closure conversion algorithm, so we decided to put it in the wishlist for now.

Another important reason to declare beforehand when a slot is mutable is that the evaluation of a λ usually does not guarantee it returns a fresh new object. This is a problem for example with Guile’s `set-procedure-property!` [Guile 2021] which may end up affecting more functions than intended. But if one of the slots is declared mutable, then `oclosure-lambda` will know that it needs to return a fresh new object, avoiding these unpredictable semantics.

4.4 Implementation

OClosures are currently implemented as a set of functions and macros that are loaded fairly early on during the bootstrap, but they are not implemented as a core data-structure. Most importantly, Emacs Lisp’s `lambda` is not defined as a special case of `oclosure-lambda` as it arguably should. It’s rather the reverse.

As far as we know, the only reliable way to implement something like `oclosure-lambda` involves defining it as a new *special form* of the language. Yet, introducing new special forms in Emacs Lisp is tricky because it can break existing packages which rely on code-walkers in their macros. So, instead we decided to implement it as a macro, and make it rely on cooperation from the closure conversion phase of the compiler.

At its simplest that macro looks like:

```
(defmacro oclosure--lambda
  (type bindings args &rest body)
  `(let ,(reverse bindings)
     (lambda ,args
       (:documentation ,type)
       (if t nil ,@(mapcar #'car bindings)
           ,@body))))
```

The name has two hyphens, because this is an internal macro, used by the real `oclosure-lambda` macro, among other things because it takes its `args` in a slightly different form.

The way this macro works is as follows: it adds the desired slots as “normal” variables in the context of a normal `lambda` and then arranges two things: first it makes sure that those variables will be captured into the closure, and then it controls the placement of those variables into the closure.

For both of those, it relies on knowledge about the way the code will be compiled, so the macro itself does not tell the whole story, and it requires cooperation from the compiler. You can see that it arranges for the variables to be captured by adding a piece of dummy code (wrapped in an `if` test to make sure it’s never executed). To control the placement of the slots, it relies on the closure conversion which places the captured variables according to their position in the environment (one could say they are ordered by increasing de Bruijn index), which is why it uses `reverse` on the bindings so that the first slot gets added last to the environment.

The type information is handled specially, stashed as if it were the docstring of the function. A more obvious choice might have been to store that information in the first slot of the closure, except that we need to be able to distinguish reliably an OClosure from a normal closure that happens to have captured a variable holding a type information and placed it in its first slot.

The real macro is a bit more complex in order to handle the case of mutable slots, on which we want to force store conversion. This is obtained very simply by changing the dummy code that’s never executed so that instead of only referring to the variable it performs an update on it. This relies on the fact that the current closure conversion naively performs store conversion on any variable that is both captured and mutated.

Clearly, the current state of implementation is not ideal, but it works well enough for now. It will likely be replaced by something cleaner when (or if) OClosures are made into a core data structure such that `lambda` is defined as a special case of `oclosure-lambda`, but there are various backward compatibility hurdles along the way, which will take some years to iron out.

5 EXPERIENCE

OClosures were developed in response to a growing set of use cases collected over the years. Here are the highlights, showing cases where the alternatives had significant shortcomings.

5.1 Advice

While there are various ways to solve the problem presented in Section 2, we did not want to pay the corresponding run time price of incurring an additional indirection or storing the extra information in a separate eq-indexed hash table. So the preexisting implementation of those advice functions relied on manually constructed closures. It worked well enough but made for rather obscure code.

The use of OClosures made the code much cleaner, removing all the low-level implementation-dependent tricks from it. It also made it possible to implement the pretty printing with a normal `defmethod` rather than the previous ad-hoc test which intruded into the more generic part of the pretty printer. Other than that, the actual runtime representation of those objects ends up being virtually identical.

5.2 next-method-p

CLOS defines `next-method-p` to return a non-nil if there is a next method (which `call-next-method` will invoke when called) and nil otherwise. These two functions can only be called from within methods. Internally, the code of methods can be implemented in various ways, but as far as I can tell, they are usually implemented as in Figure 7 which shows the relevant code used in Closette. In that code, *form* is the actual body of the method received by the `defmethod` macro. As you can see, the method is compiled to a function that takes the actual arguments *args* that were passed to the generic function, of course, and it takes an additional argument *next-emfun* which holds the next method to call. This argument is nil when there is no next method, so `next-method-p` is trivial and efficient, but in return for that `call-next-method` has to test *next-emfun* with an `if` before it can call it. This is the

```

(lambda (args next-emfun)
  (flet ((call-next-method (&rest cnm-args)
        (if (null next-emfun)
            (error "No next method for the~@
                  generic function ~S."
                  (method-generic-function ',method))
            (funcall next-emfun (or cnm-args args))))
        (next-method-p ()
          (not (null next-emfun))))
    (apply #'(lambda ,(kludge-arglist lambda-list)
              ,form)
            args))))))

```

Figure 7: Implementation of a method in Closette

wrong trade-off since `next-method-p` is used much less often than `call-next-method`.

Now, arguably, this `if` test is fairly minor: `call-next-method` is not called very often and most of the performance issues have to do instead with the cost of creating the various closures and the layers of function calls. So more efficient implementations, such as PCL spend a fair deal of efforts optimizing this code but they still leave this `if` test untouched.

To remove this `if` we need to replace the `nil` representation of the error case with a function which will signal the error when called. This makes the `call-next-method` code simpler and more efficient, but it introduces a problem in `next-method-p`: how can we tell if `next-emfun` is one of those functions representing the “no next method” case?

In Emacs Lisp, we used to do just that with a really gross and brittle hack which dug into the innards of the closures to compare them against a sample. With OClosures we now simply defined a trivial type with no slot, (`oclosure-define cl-generic-nnm`), then use `oclosure-lambda` when building those functions, and finally replaced the 20 line monster of magic incantations with just:

```
(eq (oclosure-type cnm) 'cl--generic-nnm)
```

5.3 Keyboard macros

Emacs’s keyboard macros are not macros in Lisp’s sense but are simply a sequence of key presses recorded by the user so they can replay them later at will. Originally, they were represented as a simple vector of key presses and still several parts of Emacs support this form, but then the `kmacro` package extended that functionality and needed more info for that, making it unable to use the built in support to treat a mere vector as a kind of executable object. Instead it represented keyboard macros as a sort of object implemented as a list holding a vector of key presses, plus 2 other pieces of information, and in order to make it executable, it then wrapped it into a function.

The nasty part was when `kmacro` needed to look at such a function, in order to extract the 3-element list from it, either to print it in a human-friendly way, or even to let the user edit it. Contrary to the previous two examples, those functions constructed by `kmacro` did not need to run fast and could use more or less any calling convention they wanted, so were able to implement in a less hideous way, by arrange for the function to return its contents when called

with a special argument, and simply using a special docstring to recognize those functions (which was needed simply to know that it’s safe to call it with that special argument).

The new code uses an OClosure to replace both the list of 3 elements and the wrapper function, making most of the code significantly cleaner. Contrary to the previous two cases, this is a use case where something like *funcallable objects* would have worked almost as well since the extra indirection it would have imposed would be of no consequence.

5.4 Commands

Emacs Lisp functions are actually not quite as opaque as the λ -calculus wants them to be. We can not only get to know a function’s arity but we can also query a bit more information about it: Emacs Lisp functions can carry and expose a *docstring* as well as an *interactive form*. The first is used for documentation purposes only (except for exceptional cases as in `kmacro`), while the second makes it possible to use function names as interactive commands: an interactive form is a chunk of code which constructs the list of arguments to pass to the function when the user invokes the command.

These are basically ad-hoc forms of OClosure slots. Emacs also defines a subtype of functions, called *commands* which corresponds to those functions which have an interactive form.

The current OClosure code makes these ad-hoc forms of function slots and function subtypes obsolete, by defining the type of `oclosure-command` containing an *interactive-form* slot, and making it possible to use OClosure slots to carry a function’s docstring and interactive form. Nevertheless, the obsolete support is still in very heavy use because of the subtle incompatibilities that are introduced when using the new code.

5.5 Threesomes

Another circumstance where we have found a need to look inside a function is when trying to avoid accumulating function wrappers. These accumulations can typically occur for wrappers implementing coercions, as in type-directed unboxing [Leroy 1992] or in gradual typing [Siek and Taha 2006]. A solution to those accumulations consists in collapsing those wrappers by recognizing that some of them inevitably cancel others [Minamide and Garrigue 1998]. Siek and Wadler [2010] provide such a solution for the case of gradual typing. In the calculus they use to solve the problem, they introduce *threesomes* which are coercions written $\langle T \stackrel{R}{\Leftarrow} S \rangle s$, where a the term s of type S is coerced to type T via type R . The way they avoid accumulating coercions is by having a rule which reduces $\langle T \stackrel{R_1}{\Leftarrow} U \rangle \langle U \stackrel{R_2}{\Leftarrow} S \rangle s$ to $\langle T \stackrel{R_1 \& R_2}{\Leftarrow} S \rangle s$, so coercions can never accumulate.

In their calculus, those $\langle T \stackrel{R}{\Leftarrow} S \rangle s$ don’t reduce to functions when s itself is a function, instead they are part of the possible runtime values, which means that function calls have to handle the case of a λ differently than the case of a coercion. The other option when implementing such a system is to make those coercions (when applied to functions) reduce to functions implemented using wrappers. This can simplify and speed up the all important functions calls. But it is only an option if there is still a way to recognize those

wrapper functions so we can combine them when we try to apply a wrapper on top of another. If all you have is a plain λ , there is no other option than to get your hands dirty and look under the abstraction barrier. With OClosures instead, you can have your cake and eat it: simple and efficient function calls, with efficient wrappers, while still able to quickly recognize those wrappers and extract whichever information is needed in order to collapse them.

6 RELATED WORK

The idea of treating functions as objects is quite old.

As mentioned, the AMOP [Kiczales et al. 1991] uses *funcallable objects* which are somewhat like OClosures with a single slot which do nothing more than pass their arguments to that slot's value, which should be another function. They suffer from the fact that they tend to introduce an indirection between the *funcallable object* and the actual underlying function, and the fact that the code of the function cannot directly access the funcallable object's slots. In return for that, the contained function can be changed by side-effect, whereas it would be difficult to allow changing the code of an OClosure.

MIT Scheme [MIT-Scheme 2020] provides similar functionality under the name *application hooks*. The more interesting of them are called *entities* which contain a function and another object. When called, an entity calls its contained function, passing it the arguments it received *plus itself*. This somewhat reduces the problem mentioned above that the function cannot directly access the funcallable object's slots.

GNU Kawa [Kawa 2020] and GNU Guile [Guile 2021] allow functions to carry extra properties, called *procedure properties* that can be added via side effect and queried. Again, the function itself does not have any direct access to those properties, limiting their applicability.

The Lisp Machine Lisp [Stallman et al. 1984] did not really support lexical scoping like we now have in Scheme and Common Lisp, but it had a `closure` operator that took a list of (dynamically scoped) variables and a function and returned a new function which called its argument function with the vars temporarily re-bound to the value they had when you created that "closure". The relevant part here is that you could access the list of closed-over variables and extract their values, just as we do in OClosures. Going even further, Lisp Machine Lisp had the `entity` operator which worked almost identically, except that it made it possible to assign a type to the returned function, which was typically used to allow specialized pretty printing output for those *entities*.

More recently, Scheme's SRFI 229 suggests the notion of tagged procedure, which is a procedure that comes with one extra immutable slot (called its tag) holding an arbitrary value. Beside the fact that it is limited to a single slot, it is also more limited than open closures in the sense that the tagged procedure's body cannot directly refer to the tag, so when that is needed, the tag value will probably end up duplicated in the object: one copy in the tag slot and another among the captured variables.

Of course, OClosures correspond to objects limited to a single method, used quite widely in OO-style languages that do not have a separate notion of function. They differ a bit in the sense that they conflate `oclosure-define` and `oclosure-lambda` and force

every function with a different body to have a different type (since the method is associated with the class).

The function objects of Python are also similar: one can get the list of captured variables of a Python function as well as query (and modify) their values. But this is mostly a result of its introspection facilities, offering no way for the programmer to control which captured variables are exposed and which aren't.

Siskind and Pearlmutter [2007] propose to make closures more transparent by providing a `map-closure` function which is like `mapcar` but for closures, applying a given function to each of the values captured within the closure. The name of the captured variables is not made available, so this cannot be used to extract targeted information such as the value of a particular slot, and in this sense their functions remain quite opaque (in a sense analogous to security through obscurity, maybe).

7 CONCLUSION

We have presented the idea of making functions a bit less opaque in the form of open closures, then shown a design and implementation of this feature in Emacs Lisp under the name of OClosures, and given a sense of how they can be applied in a variety of circumstances where using either tuples or functions or a combination of both is not quite satisfactory.

ACKNOWLEDGMENTS

The author would like to thank the readers of `emacs-devel` for their naming suggestions and in particular Qiantan Hong who proposed the name of "open closures".

This work was supported by the Natural Sciences and Engineering Research Council of Canada grants № 298311/2012 and RGPIN-2018-06225. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the NSERC.

REFERENCES

- Guile 2021. *GNU Guile Reference Manual* (3.0.7 ed.). <https://www.gnu.org/software/guile/manual/>
- Kawa 2020. *The Kawa Scheme Language – Reference Documentation* (3.1.1 ed.). <https://www.gnu.org/software/kawa/pt01.html>
- Gregor Kiczales, Jim Des Rivières, and Daniel G. Bobrow. 1991. *The Art of the Metaobject Protocol*. MIT Press.
- Peter Lee and Mark Leone. 1996. Optimizing ML with Run-Time Code Generation. In *Programming Languages Design and Implementation*. ACM Press, Philadelphia, PA, 137–148.
- Xavier Leroy. 1992. Unboxed Objects and Polymorphic Typing. In *Symposium on Principles of Programming Languages*. 177–188.
- Yasuhiko Minamide and Jacques Garrigue. 1998. On the runtime complexity of type-directed unboxing. In *International Conference on Functional Programming*. ACM Press, 1–12.
- MIT-Scheme 2020. *MIT/GNU Scheme Reference* (11.2 ed.). <https://www.gnu.org/software/mit-scheme/documentation/stable/mit-scheme-ref/index.html>
- Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *Scheme Workshop*. 81–92.
- Jeremy G. Siek and Philip Wadler. 2010. Threesomes, with and without blame. In *Symposium on Principles of Programming Languages*. 365–376. <https://doi.org/10.1145/1707801.1706342>
- Jeffrey Mark Siskind and Barak A. Pearlmutter. 2007. First-class Nonstandard Interpretations by Opening Closures. In *Symposium on Principles of Programming Languages*. 71–76. <https://doi.org/10.1145/1190216.1190230>
- Richard Stallman, Daniel Weinreb, and David Moon. 1984. *Lisp Machine Manual* (6th ed.). MIT. <https://hanshuebner.github.io/lmman/frontpage.html>
- Christian Urban, Stefan Berghofer, and Michael Norrish. 2007. Barendregt's Variable Convention in Rule Inductions. In *International Conference on Automated Deduction*. 35–50. https://doi.org/10.1007/978-3-540-73595-3_4

QueryFS: compiling queries to define a filesystem

Michael Raskin*

raskin@mccme.ru

raskin@in.tum.de

Technical University of Munich
Garching bei München, Germany

ABSTRACT

Personal computing devices store more and more loosely arranged data. Each new method of keeping track of the data supposes that the user stops using the old methods on this data. One of the more stable interfaces for data access is the filesystem API. However, the standard filesystem semantic provides a fixed and limited set of ways to search for data.

QueryFS is a virtual filesystem for POSIX-like systems that compiles user-supplied queries in various DSLs via translation to Common Lisp code and represents the results as directories. The main current use-case is using it to navigate and process data stored or indexed in PostgreSQL with traditional tools (`grep`, `find`, `vim` etc.)

This paper describes what practical usage of QueryFS looks like and what lies behind this.

CCS CONCEPTS

- **Software and its engineering** → **Domain specific languages;**
- **Information systems** → **Query languages; Middleware for databases.**

KEYWORDS

filesystems, search, virtual directories, domain-specific languages

ACM Reference Format:

Michael Raskin. 2022. QueryFS: compiling queries to define a filesystem. In *Proceedings of the 15th European Lisp Symposium (ELS'22)*. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.5281/zenodo.6308532>

1 INTRODUCTION

Modern filesystems solve the problem of storing large volumes of data. When the data has some special structure, an SQL database may better suited for the task. In both cases there are many compatible implementations. Applications use the same interface to access multiple storage backends; and many applications developed before some technology improvement still benefit from it. For example, SBCL has no need to know about RAID0 to get improved write speed. Neither does it need to use its network code to access files over NFS.

There are also many tools to find data in the storage. Some of them traverse all the storage to find the needed piece of information,

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

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'22, March 21–22, 2022, Porto, Portugal
© 2022 Copyright held by the owner/author(s).
<https://doi.org/10.5281/zenodo.6308532>

some create and maintain indices, some expect user to explicitly add the data into indexed area.

Unfortunately, making these tools interact with unsuspecting applications is often hard and the query language may have limited expressive power.

This paper describes how QueryFS project tries to solve the problems of using query results in applications unaware of any special API, saving queries for future use and expressing complicated conditions with queries. Moreover, the interface for providing a virtual filesystem also remains stable (like other userspace interfaces provided by the Linux kernel), significantly reducing the typical foreign interface problem of regular breaking changes in the argument formats. The basic architecture of QueryFS is based on plugins that translate domain-specific query languages into Common Lisp.

The example queries are only intended as examples. The use cases are either contrived or specific to personal preferences. However, the intent of QueryFS is not to support specific workflows but to provide a foundation for writing queries supporting arbitrary workflows.

The rest of the paper is structured as follows. First there is a survey of the previous work in the area of augmenting the filesystem storage of data with additional metadata usable for search and filtering. Then the design goals are stated and an overview of the chosen structure of the filesystem implementation is provided. Some examples of plugins and queries follow. In the next sections the level of the filesystem API support, security considerations, and the impact of implementation language are described. Afterwards the experience of using QueryFS for daily tasks is summarised. The paper ends with a conclusion and some possible future directions.

2 RELATED WORK

The existing software that helps finding the files uses many different approaches. Often some parts of the interfaces intended for browsing directory structure are reused. Due to the fact that none of the previous work has achieved widespread popularity, we apologise in advance in case there are some projects that have been erroneously omitted.

An old example is feeding of search results to the UI element intended for directory view in many file managers. In current versions of Gnome Nautilus, Windows Explorer or MacOS X Finder user can save such a search query and interact with it as if it was a folder. The main problem is that applications unaware of this feature cannot use such directories. Even WinFS project by Microsoft (now long forgotten) was going to require applications to use special API to access such search folders.

Inability of some applications to access virtual directories and use plain text file lists can be mitigated by using FUSE [16]. It allows mounting special filesystems and processing of the filesystem operation in the userspace.

Naturally, the idea of using the universal filesystem interface for information search is older than FUSE. For instance, MIT Semantic File System [6] used Network File System protocol to allow queries by attribute; a path inside the virtual filesystem could include multiple conditions and provide files satisfying all of them. The system indexed the publicly visible files in advance to support e.g. search by file contents.

A more modern approach based on indexing the storage is represented by e.g. RecollFs [3], providing virtual directories corresponding to the results of Recoll desktop search engine [4] queries. This approach bears some similarity to feeding search results to a virtual folder in a file manager, but has the benefit of using a universal interface for result access.

Some of the filesystems emphasize user-entered metadata. For example, SemFS (previously TagFS) [1] and Tagsistant [17] support marking each file with tags (arbitrary strings) instead of building file hierarchies. User can then go into a virtual directory which contains only the files having all the tags from some list. The full path of the directory can easily be saved as a symbolic link (Tagsistant also has its own alias support essentially similar to symlinks). Unfortunately, file attributes such as the file size cannot be taken into account in such queries, at least in the available versions of the software.

Some tagging filesystems, like TMSU [15], support key-value tags and combining the conditions via «or» and «not» in addition to «and». Also of note is STUFFS [10], supporting WordNet-based fuzzy matching to help with the drift in the choice of words in tags, in addition to logical combinations (but not key-value tagging).

There are also some overviews of tagging filesystems [5, 7].

The libferris project [11] (which includes `ferris-fuse`) provides means to access many different types of metadata found inside common file types. Using `libferris` on its own requires use of special API or utilities to access the data, but allows complicated queries in query languages like XPath and SQL. The FUSE filesystem, `ferris-fuse` only allows browsing the data. Another project, BaseX [9], uses XQuery language and has GUI and command-line tools for browsing indexed data. BaseX lacks FUSE support and probably won't have it in foreseeable future.

The latter two projects can be of course compared with many available desktop file indexing and search engines, but differ from most in the supported level of query complexity.

The RelFS project [2] has its focus on representing SQL queries as directories. A RelFS filesystem can store files and symbolic links like an ordinary filesystem. It also allows going into a directory with name starting with # symbol, which is interpreted as an SQL query. Running `find` on such a directory returns approximately the same result as running the SQL query put into directory name. RelFS uses SQL query language, allows queries to return complicated directory trees, and allows saving queries as symbolic links. Unfortunately, RelFS queries process only files and symbolic links stored on the RelFS filesystem itself, and storing large files on RelFS causes performance problems.

Two projects with the most radical goals, DBFS[12] and Hypocampus [8], store all the files inside the DB and have no hierarchical structure. All available ways to access files search data by attributes via special SQL queries. Unfortunately, the implementations also require the use of a special API.

3 DESIGN GOALS OF QUERYFS

QueryFS aims to take a middle ground on the storage structure. The goal is being able to store some data as files on traditional filesystems, some data inside databases, and some data might be stored in some other way supported by some plugin, for example as data structures inside a Common Lisp image. Queries should be able both to generate plain files and symbolic links, in order to be able to combine data from the databases and other structured storage with efficient references to large files stored on some traditional filesystem.

Queries should be allowed to produce writeable files and directories. To provide a variety of query languages, QueryFS needs to support plugins defining the query languages.

QueryFS needs to be able to support experiments with idiosyncratic workflows and query structures and formats.

4 DEPENDENCIES AND ARCHITECTURE OVERVIEW

The main development platform for QueryFS is SBCL on Linux; however, any Common Lisp implementation with CFFI support for callbacks from C code into Lisp and 64-bit foreign structure fields should work. The OS needs to be supported by FUSE. To run any SQL-based queries using the current plugins one needs a database supported by CLSQL.

On the low level, QueryFS relies on CL-Fuse and CL-Meta-FS. CL-Fuse is a CFFI-based wrapper around `libfuse`. CL-Meta-FS is a Lisp-flavoured abstraction layer for a more declarative definition of virtual directory layouts. It includes both a set of functions to describe a layout and a set of macros for writing more succinct definitions.

QueryFS itself handles the interaction of a FUSE mount, plugins, and queries. In order to allow a wide range of queries to be used with good performance, QueryFS translates all queries to Common Lisp code. Of course for queries accessing e.g. an SQL database the performance of the query code is not the defining factor. To allow query languages suitable for different workflows, this translation is handled via plugins. The plugins define the query DSL parsing (using PEG definitions or the Common Lisp reader, but other parser libraries can trivially be added), and then they translate the S-expressions obtained as ASTs into Common Lisp code defining the virtual filesystem structure. The translated code usually uses the macros from CL-Meta-FS.

QueryFS core finds the specified plugins directory and starts by loading each of the plugins there. A plugin registers its parser functions in the hash-table of query loaders. Then each query is processed using the loader corresponding to its file extension. The loader is expected to return a Common Lisp form defining a virtual directory. This form is then evaluated by QueryFS. Each query defines a single virtual directory, and each defined directory gets the name of the query defining it.

In the following subsections the lifecycle of a QueryFS instance is described in more detail.

4.1 Plugin loading

First, the QueryFS process finds and loads the plugins in the specified plugin path. By default, QueryFS expects that a single directory contains the subdirectories for the filesystem mountpoint (attachment point), the plugin directory, and the query directory. It is expected that plugins use their corresponding supported query types as keys to set the values in the `query-fs:query-loaders*` and `query-fs:query-loader-types*` hashtables. The former value should be a function accepting an input stream; depending on the latter, it is expected either to parse and return a single top-level entry (:token-by-token) or a list of entries (:whole-file). The entries are supposed to be in the format expected by CL-FUSE directory listing functions. They are normally defined using the macros provided by CL-Meta-FS. The entries are pieces of code defining the filesystem objects, with directories, files, and symbolic links supported. Each entry can define one file, symlink, or directory at the top level of the query answer, or a generator returning zero or more filesystem objects. An example is provided in the following listing.

```
(progn
  (defparameter *sum-numbers-range* 10)

  (mk-splice
    (mk-file "README" "A POSIX interface to #'CL:+")
    (mk-pair-generator x
      (loop for k from 1 to *sum-numbers-range*
            collect (list (format nil "~a" k) k))
      (mk-dir (first x) :just
        (mk-pair-generator y
          (loop for k from 1 to *sum-numbers-range*
                collect (list (format nil "~a" k)
                              (+ k (second x))))
          (mk-file (first y)
            (format nil "~a" (second y))))))))
```

Here `mk-splice` is used to provide a fixed set of entries within a single one. The first one is a plain file with constant contents defined using `mk-file`. The second one is again producing multiple entries. However, this time the entries are not given as compile time parameters but are generated in runtime using `mk-pair-generator`. The parameters are the formal variable for enumeration, the generating code for the list of entry data, and the code that converts a datum from the list into an entry description (which is expected to be a single simple entry). Each element in the list of data should be a list, with the first element being used as the file or directory name. Here a directory is defined using `mk-dir` where the content is again generated. In this example, a file like `2/3` contains the sum of the two components of its name, 5.

Note that sometimes the filesystem knows what file the user cares about (e.g. in the case of `with-open-file` as opposed to `directory`). The generators may reference the formal variable in the data-generating form. This is useful both for performance (if the user only cares about one file, the filesystem can skip generating data for some of the others) and for providing some functionality

on a range where enumeration is infeasible. The following example computes the next integer for any file name that `parse-integer` accepts.

```
(mk-pair-generator x
  (let ((xn (ignore-errors (parse-integer (first x)))))
    (if xn `((, (first x) , (1+ xn))
            (loop for k from 1 to 10
                  collect `(,(format nil "~a" k) , (1+ k))))))
  (mk-file (first x) (format nil "~a" (second x))))
```

There is also support for generating symlinks, as well as for handling creation, removal, and modification of files.

For parsing there are currently two options in use. The plugins either use the standard Common Lisp reader (e.g. the above example is a valid query for the `literal-lisp` plugin), or use `Esrp-PEG` to read a PEG definition and build a `Esrp`-based `packrat` parser.

For example, the following PEG definition parses clauses like `on-write x "select 1"`.

```
WhiteSpace <- " " / "\r" / "\n" / "\t"
S <- WhiteSpace +
OnWrite <- "on-write" S Identifier S SQLCommand
```

And the following pattern-matching snippet is used for processing the ASTs.

```
...
(OnWrite
  ((_ _ ?var _ ?body)
   `(:on-write
     (, (! ?var)
      , (! ?body))))))
...
```

4.2 Query loading

After loading the plugins, the Query-FS process finds the query files in the specified place. The filename extension is used for determining which of the parsers registered by plugins need to be used. Each query is passed to the corresponding parser, which generates the Common Lisp code for handling the query. For instance, a query file with the name `email.sql2` will be parsed by the parser registered under the name `sql2`. In principle any plugin could register such a parser. In practice it makes sense to call such a plugin `sql2.lisp`.

A query fragment such as the following code

```
mkdir "fresh-no-deferred" do
  for msg in "select
    account || '-' || file_basename || '-' || id::text,
    file_path, account, id
  from emails
  where fresh and
    (${msg[0]} is null
    or
    id = regexp_replace(${msg[0]},'.*-', '::int)'"
  symlink $msg[0] $msg[1]
  on-remove "update emails
    set fresh = null, read_timestamp=now()
    where id = ${msg[3]};"
```

done

will get converted into a long piece of code starting with

```
(mk-dir (concatenate 'string "fresh-no-deferred")
 :just
 (mk-pair-generator msg
 . . .
```

defining a subdirectory based on the SQL request provided in the query fragment.

This code is wrapped so that a directory is defined for each query file. For instance, there will be a top-level email directory in the virtual filesystem. The entries provided by the code corresponding to the query describe the content of the directory. In particular, the email directory will contain a `fresh-no-deferred` subdirectory with the symlinks generated using SQL inside.

The combination of these directory definitions provides the code of the full definition of the content of the virtual filesystem. This code is evaluated.

4.3 Mounting and handling the filesystem

After the filesystem content definition is computed, the filesystem is mounted. FUSE callbacks are assigned so that filesystem operations traverse the content definition and use the corresponding value or handler for the specified filesystem path.

Whenever a client process attempts to access a path within the QueryFS mountpoint, QueryFS receives requests for all the components of the path. Parsing the queries and evaluating the definitions provides the toplevel definition of the filesystem contents; for each component of the path QueryFS takes the parent definition (starting at the top level) and uses it to compute the definition for the path up to the current component. Finally, it might use the definition corresponding to the entire path to compute the entry content (file content, symlink target, directory listing).

The definitions corresponding to intermediate paths are cached for some time.

Note that the handlers for some operations can modify the filesystem definition. In particular, QueryFS can be asked to reload some query without restarting the filesystem. Once the filesystem is unmounted, QueryFS instance exits.

5 PLUGIN AND QUERY EXAMPLE

The plugin most used in the author's workflows is the second version of an SQL-based DSL. Its syntax is described using a Parsing Expression Grammar (PEG) definition. The language includes the SQL queries almost verbatim, as well as additional instructions for arranging the layout of the virtual directory resulting from the query.

For example, the following query produces the list of the latest emails for each account that are marked as already read.

```
mkdir "latest-read-by-account" do
  grouped-for account in
    "select distinct account from emails" do
  grouped-for num_latest in
    "(select ${num_latest[0]}) union (select 5) union
     (select 10) union (select 100)" do
  for msg in
    "select header_date || '-' ||
     addr_from || '-' ||
     cast(id as text), file_path
```

```
from emails where
  ((deferred is null or deferred = ''))
  and
  (fresh is null or not fresh))
  and account = ${account[0]}
  order by id desc
  limit ${num_latest[0]}"
  symlink $msg[0] $msg[1]
done
done
done
```

This creates a subdirectory for each account in use, then creates subdirectories for different numbers of latest emails to select, then creates symlinks to the latest read messages in each such subdirectory (using the account and the message count to build the database query).

The filename is always the first element in the row returned by the database request; note that if a user tries to access a specific file, the filename is already known, and the generating code can use it (for optimisation or for creating new tags on the fly etc.), this is what happens with `num_latest` in this (part of the query).

For such a query, QueryFS produces a directory containing entries `raskin@mccme.ru`, `raskin@in.tum.de` etc. One can list `raskin@mccme.ru/7` and obtain 7 symbolic links like

```
2021-02-28T09:00:00Z-,els2021@easychair.org,-123456
2021-02-28T09:01:00Z-,els2021@easychair.org,-123457
```

Note that the `grouped-for` directive creates a level of filesystem hierarchy; the `for` directive does not, so the symlinks are all inside the same directory.

Some (sub)queries define mutable filesystem objects. For example, the following code for creating symbolic links to the files with unread emails defines the database request that marks the message read when the symlink is removed.

```
for msg in
  "select id, file_path from emails
   where account=${account[0]} and fresh
   and header_date = ${timestamp[0]}"
  symlink $msg[0] $msg[1] on-remove
    "update emails set fresh = 'f'
     where id = ${msg[0]}"
```

This query language also supports templates allowing creation of recursive data structures. For example, this is used to support hierarchical tag structures.

There are other query languages defined, such as the following primitive Lisp-based one used for browsing packages as directories.

```
with pkg from
  (mapcar (lambda (s) (list (package-name s) s))
         (list-all-packages))
file "::nicknames" (fmt "~{~a~%~}"
                     (package-nicknames pkg))
file "::uses" (fmt "~{~a~%~}"
                (package-use-list pkg))
file "::used-by" (fmt "~{~a~%~}"
                  (package-used-by-list pkg))
```

Currently the DSL for SQL-based queries and two versions of Lisp-based queries (the mixed format in the above example, and direct input of CL-Meta-FS based code) appear to be useful for some practical applications, with the other plugins either defining some helper functionality or being used for testing purposes.

The queries in use include:

- Commands for controlling QueryFS itself (reloading queries and plugins, requesting the error message from the last error, etc.)
- Various functionality for handling emails indexed in PostgreSQL
- Viewing the scraped web feeds
- Both plain and hierarchical (with unlimited depth) file tagging
- Lisp package and symbol browsing
- Simple SQL-backed notes storage and (encrypted) password storage

6 FILESYSTEM API SCOPE AND SECURITY CONSIDERATIONS

QueryFS uses the filesystem API to present data that is not expected to match the normal filesystem semantics perfectly. A strong security model would conflict with QueryFS design goals: the plugins should be allowed to define query languages manipulating arbitrary external data. Separating this from arbitrary code execution by plugins would require a lot of complexity. On the other hand, the default FUSE behaviour is to forbid any filesystem access by the other users. The current assumption is that plugins and queries are stored in a place writeable only by the user who launches QueryFS, access to the virtual filesystem by other users is not allowed, QueryFS can fully trust plugins and queries, and queries can safely assume that whoever accesses the filesystem is fully trusted.

The support of filesystem the features is driven by the needs of the workflows. Specifically, the file permissions and the file size reporting are necessary for comfortable interaction with a filesystem. The support is limited for simplicity; for example, the files are assumed to be writeable only by the owner (or by nobody at all). Timestamps are supported by CL-Fuse but not by CL-Meta-FS. Better support for these features can be easily added throughout the stack.

Advanced functionality such as `inotify` or `mmap` is not supported at all, and adding it would require significant work; in any case it is unclear whether there is any good way to define the semantics for SQL-based queries, or for browsing Common Lisp symbols.

From a practical point of view, Vim or Emacs can edit writeable virtual files provided by QueryFS.

7 IMPLEMENTATION LANGUAGE DEPENDENCE OF THE DESIGN

While QueryFS is implemented in Common Lisp, there are many FUSE filesystems implemented in various programming languages. The choice of the implementation language affects the available options in various parts of the CL-Fuse/CL-Meta-FS/QueryFS stack.

At the bottom level CL-Fuse cannot use the default event loop provided by FUSE because its thread management doesn't interact

well with SBCL expectations about the threads. This limitation is shared with many high-level languages with garbage collection. Single-request high-level FUSE API is used instead.

CL-Meta-FS design is based on the assumption that the programming language supports macros. In some languages the most natural replacement would be based on higher-order function, probably resulting in more indirection. On the other hand, for standalone use in Common Lisp a CLOS-based design could have been better. But a more primitive interface was chosen for use of CL-Meta-FS as a translation target.

Writing the plugin code for translating the queries into Lisp code after initial parsing benefits from availability of convenient pattern-matching facilities. In the case of languages with good macro support, one can expect multiple pattern matching libraries to arise. This is indeed the case with Common Lisp.

QueryFS compiles code in runtime and supports reloading single queries and recompiling their code without restarting the entire filesystem instance. This would require additional effort in the languages assuming ahead-of-time compilation (such as Pascal, C, or Rust).

In a language without macro support, one way to implement the QueryFS design would use a replacement for CL-Meta-FS in the form of an AST rewriting library.

Overall, the chosen architecture is best implemented in a just-in-time compiled language with macro support, which is a combination observed in the Lisp language family and in Julia.

8 EVALUATION

QueryFS has been in daily use for the past few years for a few different use cases.

One of the use cases is to index and track pending/dismissed state of some streams of files; this includes both emails and e.g. downloaded `planet.lisp.org` articles. In both cases the fetcher process puts each item into a separate file; a database entry is created with the item properties and the file path. Then QueryFS is used to both view the items matching some criterion, and to update the metadata related to the entry. Some of the workflows initially prototyped on QueryFS were eventually migrated to small helper tools doing the DB accesses directly.

An additional use case is storing some small snippets of text in a DB with transparent encryption.

I have tried a couple of file tagging solutions implemented as QueryFS queries, but ended up preferring the use of a hierarchical classification of the files. Sometimes tags get assigned to emails, though, as the current tooling expects the files with the emails to reside in a single directory.

From the point of view of performance, QueryFS seems to be quite acceptable in practice. It is of course much slower than a usual filesystem, but this is to be expected given the requests need to be handled in multiple userspace processes (QueryFS, PostgreSQL) instead of just the kernel. Single operations are fast enough to not be an issue. Large files are only referenced using symlinks, avoiding the main performance risk altogether. The most common slowdown is related to a specific SQL query that performs poorly when PostgreSQL caches are cold. Removal of too many entries one by one is also noticeably slow.

Slowdowns related to QueryFS itself are unnoticeable in practice. For instance, asking `cat` to open, read, and close a file containing the number 579 obtained as the sum of 123 and 456 many times in a row takes about $170\mu\text{s}$ per iteration. This is much more than around $1\mu\text{s}$ per iteration necessary for a normal file on a traditional filesystem. On the other hand, in the absolute terms this delay is quite small and it is rarely the main bottleneck. For comparison, PostgreSQL on the same computer needs more than $30\mu\text{s}$ within the engine (without the communication overhead included) to plan and execute a trivial query.

From the point of view of expressive power, QueryFS with the current set of plugins seems to be unmatched in the specific area the author cares about: defining virtual directories via completely arbitrary SQL queries and their subdirectories via SQL queries dependent on the data returned by the higher-level queries. The tagging query is behind some of the tagging filesystems due to the author only using it for a narrow set of tasks.

9 CONCLUSION AND FUTURE WORK

The present paper describes QueryFS, a tool allowing Common Lisp code to provide a filesystem interface for external tools, and its use to provide virtual filesystem structures defined using SQL queries.

We believe that the stability of the filesystem APIs is a desirable property when providing interface for foreign applications to interact with Common Lisp code. Another similar option is HTTP(S) API, but filesystem interface is convenient for integration with a different set of tools. The author will be glad to support any project that wishes to try CL-Meta-FS or QueryFS as a tool for providing an interface to the internal data of a Common Lisp based tool.

The rising use of virtual machines for isolation of software that would be previously considered a part of a single system motivates expanding QueryFS to also support the Plan9 resource sharing protocol 9p (also known as Styx) [13]. This will reduce the overhead when allowing a VM to access a QueryFS instance, as such access would otherwise happen over a 9p connection with just another layer of indirection.

An interest in supporting a specific use case could motivate development of new DSLs for QueryFS. A natural candidate is an integration with some desktop search engine.

AVAILABILITY

QueryFS and its supporting libraries are currently hosted as a Common-Lisp.net project [14].

ACKNOWLEDGEMENTS

The author is grateful to Nikita Mamardashvili for advice on the design of some of the SQL-based query DSL. The author is grateful to Lars Brinkhoff, the author of another CL-Fuse project (limited to browsing the packages and the symbols), for the idea of using FUSE to browse Common Lisp image contents and for the permission to take over the CL-Fuse name on the Common-Lisp.net for a more general Common Lisp FUSE library. The author would like to thank the anonymous reviewers for their feedback regarding the presentation.

REFERENCES

- [1] Stephan Bloehdorn, Olaf Görlitz, Simon Schenk, and Max Völkel. Tagfs - tag semantics for hierarchical file systems. In *6th International Conference on Knowledge Management (I-KNOW'06)*, 2006.
- [2] Vincenzo Ciancia. Relfs project, 2004–2005. URL <http://relfs.sf.net/>.
- [3] Piotr Dlugosz. Recollfs — fuse filesystem using recoll index, showing filtered files in directories, 2014. URL <https://github.com/pidlug/recollfs>.
- [4] Jean-François Dockes. Recoll project, 2012–2020. URL <https://www.lesbonscomptes.com/recoll/>.
- [5] Jean-François Dockes. Extended attributes and tag file systems, 2015. URL <https://www.lesbonscomptes.com/pages/tagfs.html>.
- [6] David Gifford, Pierre Jouvelot, Mark Sheldon, and James O'Toole. Semantic file systems. 1991.
- [7] goesZen.com user tengo. Tagfs, tracking progress in the field of semantic file systems, 2009. URL <https://linux.goeszen.com/tagfs-tracking-progress-in-the-field-of-semantic-file-systems.html>.
- [8] Roberto Guido. Hyppocampus project, 2005–2008. URL <https://sourceforge.net/projects/hyppocampus/>.
- [9] Alexander Holupirek, Christian Grün, and Marc H. Scholl. Basex and deepfs — joint storage for filesystem and database. In *EDBT (Demo Track)*, 2009.
- [10] Aaron Laursen. Stuffs: A novel tag-based file-system, 2014. URL https://digitalcommons.macalester.edu/cgi/viewcontent.cgi?article=1036&context=mathcs_honors.
- [11] Ben Martin. libferris project, 2001–2018. URL <http://www.libferris.com/>.
- [12] Lee Burton Onne Gorter. dbfs project, 2004. URL <http://dbfs.sourceforge.net/>.
- [13] Rob Pike and Dennis M. Ritchie. The styx architecture for distributed systems. 1999.
- [14] Michael Raskin. Queryfs project, 2010–2020. URL <https://gitlab.common-lisp.net/cl-fuse/>.
- [15] Paul Ruane. Tmsu project, 2011–2022. URL <https://tmsu.org/>.
- [16] Miklos Szeredi and Nikolaus Rath. Fuse project, 2001–2020. URL <https://github.com/libfuse/libfuse>.
- [17] Tx0. Tagsistant project, 2006–2017. URL <https://www.tagsistant.net/>.

A CLOS protocol for lexical environments

Robert Strandh
robert.strandh@gmail.com
Unaffiliated

Irène Durand
irene.durand@u-bordeaux.fr
LaBRI, University of Bordeaux
Talence, France

ABSTRACT

The concept of an *environment* is mentioned in many places in the Common Lisp standard, but the nature of the object is not specified. For the purpose of this paper, an environment is a mapping (or several such mappings when there is more than one namespace as is the case for Common Lisp) from *names* to *meanings*.

In this paper, we propose a replacement for the environment protocol documented in the book “Common Lisp the Language, second edition” by Guy Steele. Rather than returning multiple values as the functions in that protocol do, the protocol suggested in this paper is designed so that functions return instances of standard classes. Accessor functions on those instances supply the information needed by a compiler or any other *code walker* application.

The advantage of our approach is that a protocol based on generic functions and standard classes is easier to extend in backward-compatible ways than the previous protocol, so that implementations can define additional functionality on these objects. Furthermore, CLOS features such as auxiliary methods can be used on these objects, making it possible to extend or override functionality provided by the protocol, for implementation-specific purposes.

CCS CONCEPTS

• Software and its engineering → Compilers;

KEYWORDS

CLOS, Common Lisp, Environment, Compilation

ACM Reference Format:

Robert Strandh and Irène Durand . 2022. A CLOS protocol for lexical environments. In *Proceedings of the 15th European Lisp Symposium (ELS’22)*. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.5281/zenodo.6331519>

1 INTRODUCTION

The Common Lisp standard [1] contains many references to environments. Most of these references concern *lexical* environments at *compile time*, because they are needed in order to process forms in non-null lexical environments. The standard does not specify the nature of these objects, though

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS’22, March 21–22 2022, Porto, Portugal
© 2022 Copyright held by the owner/author(s).
<https://doi.org/10.5281/zenodo.6331519>

in the book “Common Lisp, the Language, second edition” [4] (henceforth referred to as “CLtL2”) there is a suggested protocol that is supplied by some existing Common Lisp implementations.

The protocol documented in CLtL2 has several problems. Functions in the protocol return multiple values, a fact that makes the protocol hard to extend. Furthermore, the protocol is incomplete. A typical compiler needs more information than the protocol provides, making implementation-specific extensions obligatory for the protocol to be useful. For that reason, although existing Common Lisp implementations often provide such extensions, the CLtL2 protocol is not what the native compiler of the implementation actually uses.

In this paper, we propose a modern alternative protocol based on CLOS. Rather than returning multiple values, our protocol functions return instances of standard classes. Accessors for those instances can be used by compilers and other *code walker* applications in order to obtain the information needed for the task to be accomplished. This protocol is defined and implemented in the Trucler library.¹

Two of the functions in the section about environments in CLtL2 are not discussed in this paper, namely `parse-macro` and `enclose`. These functions do not contribute any functionality to the protocol being described, and the interface provided by these functions does not require any modifications, which is why they are not discussed here. The function `enclose` requires an evaluator such as a compiler or an interpreter, and the evaluator will certainly *use* the functionality in the protocol, but not add to it. The function `parse-macro` does not seem to even use this functionality, and indeed the optional *env* parameter of this function is declared `ignore` both in SBCL and CCL.

Although `parse-macro` and `enclose` are essential for any code-walking application, the purpose of the current work is not to provide a complete implementation-independent library for code walking, but just to propose an alternative protocol for accessing lexical environments.

2 PREVIOUS WORK

In this section, we describe how different implementations of Common Lisp represent lexical environments, and whether these implementations include a version of the protocol described in CLtL2. For commercial implementations, we include only their documented version of the CLtL2 protocol. We start by presenting the details of the CLtL2 protocol as described in the book.

¹<https://github.com/s-expressionists/Trucler>

2.1 Common Lisp the Language, second edition

Section 8.5 of CLtL2 describes a set of functions for obtaining information from an environment object, for creating a new such object by augmenting an existing one, and two more operators related to environments that are outside the scope of this paper, i.e., `parse-macro` and `enclose`.

In this section, we provide an overview of that protocol, and we give an assessment of its usefulness in the context of a language processor.

2.1.1 Environment query. For environment query, the protocol defines three functions. We describe them briefly here.

The function `variable-information` takes a symbol and an optional environment object as arguments. It returns three values. The first value indicates the type of the binding (lexical variable, special variable, symbol macro, constant variable) or `nil` if there is no binding or definition in the environment for that symbol. The second value is a Boolean, indicating whether the binding is local or global. The third value is an association list containing declarations that apply to the binding.

The function `function-information` takes a function name and an optional environment as arguments. Again, three values are returned. The first value indicates the type of the binding (function, macro, special operator²) or `nil` if there is no binding or definition in the environment for that function name. As before, the second value indicates whether the definition is local or global, and the third value is an association list of declarations that apply.

The function `declaration-information` is used in order to query the environment for declarations that do not apply to any particular binding in the environment. It takes a *declaration identifier*³ and an optional environment as arguments. The declaration identifier can be the symbol `optimize`, the symbol `declaration`, or some implementation-defined declaration identifier. It returns a single value that contains information related to the corresponding declaration identifier.

To begin with, it is clear that this set of functions is insufficient to process all Common Lisp code, because no mechanism is described for querying the environment for information related to *blocks* and *go tags*. Functions for this purpose are provided as extensions by Allegro Common Lisp as described in Section 2.7, and by LispWorks as described in Section 2.8.

2.1.2 Environment augmentation. For augmenting an environment, i.e., creating a new, augmented, environment from an existing one, the same section describes the function `augment-environment`. This function has a keyword parameter for each type of object to be added to the current lexical environment: `:variable`, `:symbol-macro`, `:function`,

`:macro`, and `:declare`. Furthermore, each argument is a list of lexical definitions, thereby allowing an arbitrary number of mappings to be added to an environment in order to create an augmented environment.

2.1.3 Assessment of the protocol. In general, the protocol as described in the book is insufficient for use in any but the simplest kind of language processor. Even if query functions are added for tags and blocks, and additional keyword argument are added to the function `augment-environment` for tags and blocks, we argue that the protocol would still be insufficient.

Any non-trivial language processor would need for a function such as `function-information` to return information about the function, other than related declarations. At the very least, information such as the lambda list of the function, and information needed for inlining, would have to be included.

The protocol could obviously be extended to allow for such information, but such extensions would involve incompatible additions such as more return values. Furthermore, none of the Common Lisp implementations we investigated use this protocol internally, which is an indication that the compiler needs more information than the protocol provides. And none of the implementations we investigated provide extensions that would allow the use of the protocol in a non-trivial language processor.

2.2 SBCL

2.2.1 Native. SBCL⁴ defines a structure class `lexenv`. Instances of this class are passed as the `&environment` argument to macro expanders and other functions that take lexical environment objects as arguments.

This structure class contains several slots, and in particular:

- An association list of information about defined functions. The name of the function is used as a key.
- An association list of information about defined variables. The name of the variable is used as a key.
- An association list of information about blocks. The name of the block is used as a key.
- An association list of information about `tagbody` tags. The name of the tag is used as a key.

2.2.2 CLtL2. The distribution of SBCL contains a contribution that supplies some of the functionality described in the book CLtL2 but that was not included in the Common Lisp standard. Part of this contribution is an implementation of the environment protocol of CLtL2.

2.3 CCL

2.3.1 Native. CCL⁵ defines a class `lexical-environment` which is a special kind of class called an `istruct`. Classes of this type are represented as lists of slots rather than as

²The term used in the book is *special form*, but the terminology has been improved since then

³The term used in the book is *name* and the parameter is called *decl-name*, but the terminology has changed since then.

⁴<http://www.sbcl.org/>

⁵<https://ccl.closure.com/>

standard objects as would normally be the case, probably for reasons of bootstrapping.

2.3.2 CLtL2. CCL has implementations of the functions defined in CLtL2. These functions take a native lexical environment as an optional argument.

2.4 CMUCL

2.4.1 Native. A lexical environment is an instance of the structure class `lexenv`. There is a slot for each type of entry, i.e., `functions`, `variables`, `blocks`, `tags`, and some other slots for implementation-specific details. Each of the main slots contains an association list in which the name is the key and the value contains associated information for the name.

Access to the lexical environment is provided by the macro `lexenv-find` and the function `lexenv-find-function`. These operators do not take an environment object as an argument, and instead access this object as the value of the special variable `*lexical-environment*`. And `lexenv-find-function` is a wrapper for a call to `lexenv-find` with a specific `:test` function for the key of the association list containing functions.

2.4.2 CLtL2. CMUCL provides definitions of the functions defined in CLtL2. The code for these functions is defined in the package `ext`. No extensions are provided for tags or blocks.

2.5 ECL

2.5.1 Native. The native compilation environment of ECL⁶ is represented as a single `cons` cell where the `car` is a list of *variable records* and the `cdr` is a list of *macro records*. Information about blocks and tags is included in the list of *variable records*. With few exceptions, a record is a list with the name of the entity in the `car`. Records for blocks and tags are distinguished by having a keyword symbol `:block` or `:tag` in the `car` of the list representing the record.

2.5.2 CLtL2. Currently, ECL does not offer a CLtL2-compatible interface to its lexical environments. Some work has been done to create such an interface, but it is still work in progress.

2.6 Clasp

2.6.1 Native. The native compilation environment of Clasp⁷ is currently that used in early versions of the Cleavir⁸ compiler framework. Ultimately, Clasp will use Trucler as described in Section 3.

2.6.2 CLtL2. Clasp provides an implementation of the CLtL2 protocol. The code is present in the package named `clasp-clt12`. The function `augment-environment` has two additional keyword arguments, namely `tag` and `block`. However, no extension allows for client code to access information about blocks and tags.

⁶<https://common-lisp.net/project/ecl/>

⁷<https://github.com/clasp-developers/clasp>

⁸<https://github.com/s-expressionists/Cleavir>

2.7 Allegro

2.7.1 Support for CLtL2 protocol. The documentation for Allegro Common Lisp contains a separate document describing their protocol for environments in the spirit of CLtL2.⁹ We summarize the differences between the Allegro implementation and the CLtL2 protocol here.

- Information about blocks and tags have been added in the form of two new functions `block-information` and `tag-information`.
- The function `augment-environment` accepts additional keywords arguments such as `:block`, `:tag`, etc. in order to make it possible to augment an environment with all relevant information that the language processor may encounter.
- The function `augment-environment` accepts an additional keyword argument `:locative` that can be used by client code to supply additional information about the entity, for example the value of a constant variable. The query functions return an additional value which is the information supplied to `augment-environment`.
- The order and the number of the return values of the query functions have been modified, so as to allow for the additional *locative* value, and to have frequently used return values before the less frequently used.
- Several other features have been added to the protocol in order to make it a complete tool for a language processor, and for the purpose of minimizing memory allocation. These additional features are outside the scope of this paper.

2.8 LispWorks

2.8.1 Support for CLtL2 protocol. The documentation for LispWorks Common Lisp describes the operators that implement the CLtL2 protocol. These operators are available in the `hcl` package.

Like Allegro, LispWorks also provides the functionality for blocks and tags that is missing from the CLtL2 protocol, but instead of adding functions `block-information` and `tag-information`, LispWorks provides a single additional function named `map-environment`. This function has a single required parameter, namely an environment object. It has four keyword parameters: *variable*, *function*, *block*, and *tag*. Each corresponding argument is a designator for a function that can accept three arguments: *name*, *kind*, and *info* as follows:

- The function *variable* is called for each local variable binding in the environment. *name* is the name of the variable, *kind* is one of `:special`, `:symbol-macro` or `:lexical`, with the same meaning as for the function `variable-information` in the CLtL2 protocol. When *kind* is `:symbol-macro`, then *info* is the expansion; otherwise, *info* is unspecified.
- The function *function* is called for each local function in the environment. *name* is the name of the function,

⁹<https://franz.com/support/documentation/current/doc/environments.htm>

kind is one of `:macro` or `:function`, with the same meaning as for the function `function-information` in the CLtL2 protocol. When *kind* is `:macro`, then *info* is the macro-expansion function; otherwise, *info* is unspecified.

- The function `block` is called for each block in the environment. *name* is the name of the block, *kind* is the keyword symbol `:block`, and *info* is unspecified.
- The function `tag` is called for each tag in the environment. *name* is the name of the tag, *kind* is the keyword symbol `:tag`, and *info* is unspecified.

However, when `map-environment` calls the function in the keyword argument *function* and the name of the function is of the form `(setf symbol)`, then the argument is not the name of the function, but instead a symbol that is used internally in LispWorks to name the function. According to the maintainer of LispWorks, this restriction will be removed in future versions.

Similarly, the keyword argument `:function` of the function `augment-environment` must be a list of symbols. To represent a function with a name of the form `(setf symbol)`, the internal symbol used by LispWorks must be passed, rather than the true name of the function. Again, this restriction will be removed in future versions.

2.9 CLtL2 compatibility system

The system `cl-environments`¹⁰ provides a compatibility layer that allows client code to use the CLtL2 environment protocol independently of the Common Lisp implementation. Supported Common Lisp implementations are CLISP, CCL, ECL, ABCL, CMUCL, SBCL, Allegro, and LispWorks.

This library does not provide additional operators for querying the environment for tags or blocks, nor does it provide keyword arguments on `augment-environment` for augmenting an environment with such information.

2.10 Software including a code walker

In his paper presented at the European Lisp Symposium 2017 [3], Raskin gives an overview of various libraries that require code walking. In that paper, he also argues that it is impossible to write a completely portable code walker, although he addresses many of the difficulties in his own, mostly portable, code walker named Agnostic Lizard.

In particular, one of the libraries he mentions in his paper is `hu.dwim.walker`. This library provides a general-purpose configurable code walker. It uses its own protocol for accessing and augmenting the environment. This protocol resembles the one presented in this paper in some ways.

3 OUR TECHNIQUE

We define a CLOS-based protocol for accessing and augmenting a lexical environment. This protocol is defined and implemented in the Trucler library.

3.1 Querying the environment

A language processor calls one of the query functions in order to determine the nature of a language element, depending on the position in source code of that language element. All these functions are generic, and they all take a `client` parameter and an `environment` parameter. Methods defined by Trucler do not specialize to the `client` parameter. Client code should pass an object specific to the application as a value of that parameter, and it can supply methods specialized to the class of this object, for the purpose of extending or overriding default behavior. The `environment` parameter is an object of the type used by the implementation that Trucler is configured for. Functions that are used to query a particular *name* have an additional parameter for this purpose.

The following query functions are defined by Trucler. Each one returns an instance of a class that allows the language processor to determine the exact nature of the language element (`nil` is returned if there is no definition for the element), for example by using the instance in a call to a generic function:

- `describe-variable`. This function returns an instance of a class that distinguishes lexical variables, special variables, constant variables, and symbol macros.
- `describe-function`. This function returns an instance of a class that distinguishes global functions, local functions, and macros.
- `describe-block`.
- `describe-tag`.
- `describe-optimize`.
- `describe-declarations`. This function is called by the language processor in order to determine the declaration identifiers of `declaration` proclamations.

3.2 Augmenting the environment

A language processor calls one of the augmentation functions in order to define a lexical environment within the scope of a declaration or a definition encountered in source code. All these functions take at least a `client` parameter and an `environment` parameter just like the query functions, and they all return a new lexical environment, augmented according to the function being called.

The following functions are called by the language processor when a local definition is encountered, and they return a new environment that includes the new definition:

- `add-lexical-variable`.
- `add-special-variable`.
- `add-local-symbol-macro`.
- `add-local-function`.
- `add-local-macro`.
- `add-block`.
- `add-tag`.

The following functions are called by the language processor as the result of a local declaration that restricts an existing local function or variable:

- `add-variable-type`.

¹⁰<https://github.com/alex-gutsev/cl-environments>

- `add-variable-ignore`.
- `add-variable-dynamic-extent`.
- `add-function-type`.
- `add-function-ignore`.
- `add-function-dynamic-extent`.

The following functions are called by the language processor as the result of a local `optimize` declaration.

- `add-inline`.
- `add-speed`.
- `add-compilation-speed`.
- `add-debug`.
- `add-safety`.
- `add-space`.

3.3 Restricting the environment

Recall that the description of the function `enclose` in section 8.5 of CLtL2 mentions that the consequences are undefined if the *lambda-expression* argument contains references to entities in the environment that are not available at compile time, such as lexically visible bindings of variable and functions, go tags, or block names.

As a service to a robust implementation of the `enclose` function, the Trucler library provides a function named `restrict-for-macrolet-expander` that takes an environment as an argument and returns an environment that contains only entities available at compile time. Using this function, the implementation of `enclose` can return a function that will signal an error if the *lambda-expression* argument contains unavailable references.

3.4 The reference implementation

Trucler supports some existing Common Lisp implementations as described in Section 3.5, but it also comes with a *reference implementation* that can be used by a new Common Lisp implementation that does not have its own representation of lexical environments. The reference implementation is used by SICL¹¹ for instance.

In the reference implementation, a lexical environment is represented as a standard object containing a slot for each type of description to be returned by a query function as described in Section 3.1. Each slot contains a list of descriptions ordered from innermost to outermost. A query function merely returns the first item on the list that matches the name that was passed as an argument to the query function. As a direct consequence of this representation, there is no performance penalty in the query functions, due to the fact that a new environment is created for every call to an augmentation function.

In order to create new objects such as environments or descriptions, we use a technique that we call *quasi cloning*. A generic function named `cloning-information` is called with the original object as an argument. This function then returns a list of pairs. The first element of the pair is a slot initialization argument for the class of the object and

the second element of the pair is the name of a slot reader for the same slot. This information is then used to access slots in the original object and to pass that information as an initialization argument to `make-instance`. We call it quasi cloning, because some new value is prepended to the initialization arguments so that the copy is like the original, except for one slot.

The advantage of quasi cloning is that Trucler does not need to know the right class to instantiate. It creates an instance of the same class as the original object, and that class can be defined by client code. Client code must define a method on `cloning-information`, but this generic function uses the `append` method combination, so that only slots defined by client code need to be mentioned in that method.

Occasionally, an entirely new instance of some class must be created, rather than being obtained by quasi cloning an existing instance. This situation occurs when information about a new item such as a local variable or a local function must be used to augment an existing environment. To allow Trucler to create an instance of a class that has been determined by client code, Trucler first calls what we call a *factory* function. This function takes the `client` object and returns the class metaobject to instantiate. For example, to create an instance of a class that describes lexical variables, Trucler calls the function `lexical-variable-description-class`, passing it the `client` object supplied by client code. The default method on this generic function returns the default class used by the reference implementation, but client code that needs additional information about lexical variables may create a subclass of the default class, and a method on `lexical-variable-description-class` that returns this subclass.

3.5 Supported Common Lisp implementations

Trucler currently provides support for SBCL and CCL. Contributions for other Common Lisp implementations are welcome. With these implementations, it is possible to write code walkers that are portable across different Common Lisp implementations. In particular, a Cleavir-based compiler can compile source code for any of the supported implementations.

3.6 Examples

In this section, we show some examples of how Trucler can be used by a code walker. All examples are from Cleavir used in the SICL compiler. We have simplified the examples compared to the actual code, in order to avoid too much clutter. For example, we have omitted the handling of error situations and restarts.

The part of Cleavir that uses Trucler is the phase that converts a *concrete syntax tree* (CST) to an *abstract syntax tree* (AST). A concrete syntax tree can be thought of as a Common Lisp expression but where each sub-expression is wrapped in a standard object that holds additional information such as source location. At the core of this compilation

¹¹<https://github.com/robert-strandh/SICL>

phase is the generic function `convert-cst`. For each class of description objects that Trucler can return, this generic function has a method specialized to that class.

The function `convert-cst` is called by a top-level function `convert` that determines the structure of the expression to convert and calls the appropriate Trucler query function and then invokes `convert-cst` with the object returned by Trucler.

The method specialized to `local-macro-description` looks like this:

```
(defmethod convert-cst
  (client
   cst
   (info trucler:local-macro-description)
   environment)
  (let* ((expander (trucler:expander info))
        (expanded-form
         (expand-macro expander cst environment))
        (expanded-cst
         (cst:reconstruct expanded-form cst client)))
    (setf (cst:source expanded-cst) (cst:source cst))
    (with-preserved-toplevel-ness
      (convert client expanded-cst environment))))
```

As we can see, this method is specialized to the Trucler class `local-macro-description`, and no other parameter is specialized. The code calls the accessor `expander` on the `info` parameter, which returns the macro expander associated with the local macro.

The function `expand-macro` is responsible for taking into account `*macroexpand-hook*` as the Common Lisp standard requires. The call to `reconstruct` has to do with preserving source information in the expanded form. The essence of the body is the call to `convert` which converts the expanded form (wrapped in a concrete syntax tree).

The next example shows how the environment is augmented when a `block` special form is converted:

```
(defmethod convert-special
  (client
   (symbol (eql 'block))
   cst
   environment)
  (cst:db origin (block-cst name-cst . body-cst) cst
  (declare (ignore block-cst))
  (let ((name (cst:raw name-cst)))
    (let* ((ast (cleavir-ast:make-ast
                 'cleavir-ast:block-ast))
          (new-environment
           (trucler:add-block
            client environment name ast)))
      (setf (cleavir-ast:body-ast ast)
            (process-progn
             client
             (convert-sequence
              client body-cst new-environment)
             environment))
      ast))))
```

In the example above, `cst:db` is a version of the standard Common Lisp operator `destructuring-bind`, that is used to

destructure concrete syntax trees as opposed to ordinary Common Lisp source expressions. The last argument to `add-block` is an optional argument that Trucler calls `identity` and that Trucler stores in the environment, associated with the block information. The nature of the object supplied is entirely determined by client code. In our case, we supply an abstract syntax tree that represents the `block` special form so that when an associated `return-from` is found, the two abstract syntax trees can be connected.

The essence of the method body is the call to the function named `convert-sequence` which converts the body of the `block` form in the original environment augmented with information about the `block` form.

4 BENEFITS OF OUR TECHNIQUE

The query functions of our protocol are generic functions, allowing client code to define methods for overriding or extending default behavior. For this purpose, the query functions all have a `client` parameter. Default methods supplied by Trucler do not specialize to this parameter, but client code should supply a standard object as the corresponding argument when these functions are called. The class of this argument can then be used in primary or auxiliary methods defined by client code, thereby allowing arbitrary customization of the library.

Furthermore, each query function returns an instance of a standard class, rather than multiple values. Client code can define subclasses of the classes used by the query functions. In particular, for objects in the global environment, client code can return instances of classes containing arbitrary information that it finds useful for the language processor. For example, if a global function turns out to be a generic function, client code can then return a subclass of the Trucler class `global-function-description` that contains information such as the generic-function class, the method class, and the method combination, as we suggested in our paper about `make-method-lambda` [2].

5 DISADVANTAGES OF OUR TECHNIQUE

Compared to the protocol defined in CLtL2, our protocol probably involves more memory allocation, or “consing”. Multiple values are likely to be handled without memory allocation in most high-end Common Lisp implementations, whereas our query functions return standard objects which obviously need to be allocated. Initializing the slots of these standard objects also comes with an additional cost.

To make things worse, our protocol is able to add a single mapping for each call to a protocol function, whereas the CLtL2 protocol function `augment-environment` is able to add an arbitrary number of mappings with a single call.

Our protocol consists of generic functions, and in implementations with a mediocre implementation of generic dispatch, our protocol can require more resources for function calls. Furthermore, the multiple values returned by the CLtL2 protocol are likely transmitted to the caller in registers or

some other relatively direct location, whereas the information returned by our query functions is present in slots of the standard objects being returned. Accessing this information involves calling a slot reader, which involves another call to a generic function.

However, we believe that the work done by the code walker of a compiler is small compared to that of other compilation phases such as optimization of intermediate code.

6 CONCLUSIONS AND FUTURE WORK

We have defined a CLOS-based protocol for lexical environments. This protocol can be used by any code walker such as a compiler or a version of `macroexpand-all`. Compared to the protocol defined in CLtL2 [4], ours is complete in that it has operators for querying an environment for references to tags and blocks, and for augmenting environments with such entities.

Furthermore, our protocol is implemented in the Trucler library. Trucler supplies implementations for some existing Common Lisp implementations, currently SBCL and CCL. The library also contains a *reference implementation* that can be used in new Common Lisp implementations that do not have an existing native representation of lexical environments, such as SICL and Clasp.

Future work involves adding more supported existing Common Lisp implementations. Individual implementations may require additional protocol functions, but such functions will have names in a package that is specific to the implementation.

Future work also involves investigating what new functionality might be required in the reference implementation in order to support specific requirements of new Common Lisp implementations that choose to use an extended version of the Trucler reference implementation.

7 ACKNOWLEDGMENTS

We would like to thank Jan Morningen and Karsten Poeck for reading an early draft of the paper and for suggesting improvements. Furthermore, we would like to thank Martin Simmons for explaining the wordings in the documentation of the CLtL2 protocol for LispWorks.

REFERENCES

- [1] *INCITS 226-1994[S2008] Information Technology, Programming Language, Common Lisp*. American National Standards Institute, 1994.
- [2] Irène Durand and Robert Strandh. MAKE-METHOD-LAMBDA revisited. In Nicolas Neuss, editor, *Proceedings of the 12th European Lisp Symposium (ELS 2019), Genova, Italy, April 1-2, 2019*, pages 20–23. ELSAA, 2019. doi: 10.5281/zenodo.2634303. URL <https://doi.org/10.5281/zenodo.2634303>.
- [3] Michael Raskin. Writing a best-effort portable code walker in Common Lisp. In *Proceedings of the 10th European Lisp Symposium (ELS 2017), Brussels, Belgium, April 3-4, 2017*, pages 98 – 105. ELSAA, April 2017. doi: 10.5281/zenodo.3254669. URL <https://doi.org/10.5281/zenodo.3254669>.
- [4] Guy L. Steele, Jr. *Common LISP: The Language (2Nd Ed.)*. Digital Press, Newton, MA, USA, 1990. ISBN 1-55558-041-6.

Closing the Performance Gap Between Lisp and C

Marco Heisig
Chair for System Simulation
FAU Erlangen-Nürnberg
Erlangen, Germany
marco.heisig@fau.de

Harald Köstler
Chair for System Simulation
FAU Erlangen-Nürnberg
Erlangen, Germany
harald.koestler@fau.de

ABSTRACT

Lisp is the second oldest programming language, and the first one to value productivity more than raw execution speed. This initial disregard for performance has indeed led to some mind-bogglingly slow implementations, especially in the early days, but modern Lisp compilers such as SBCL have almost fully closed the performance gap to the fastest imperative programming languages. Almost, but not quite: Until now, many loop optimizations and support for single instruction, multiple data (SIMD) programming are still missing in Lisp.

We correct this deficiency with two libraries: The first one is `sb-simd`, an SBCL-specific library that provides convenient bindings for various SIMD instructions sets. The second one is `Loopus`, a portable loop optimization framework that works via macros and source to source transformations. The most prominent features of `Loopus` are its optimization of array accesses and that, on SBCL, it automatically applies SIMD vectorization to certain loops.

We conclude with a performance evaluation for several example programs, and show that Lisp code using our libraries can achieve up to 94% of the performance of highly optimized C code.

ACM Reference Format:

Marco Heisig and Harald Köstler. 2022. Closing the Performance Gap Between Lisp and C. In *Proceedings of the 15th European Lisp Symposium (ELS'22)*. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.5281/zenodo.6335627>

1 INTRODUCTION

A common misconception about Moore's law is that it promises a doubling of the performance of our computers roughly every two years. In fact, Gordon Moore was predicting that the number of transistors in an integrated circuit would double roughly every two years. This fine distinction between performance and number of transistors was hardly relevant for a long time, where hardware manufacturers managed to translate more transistors directly into more performance. Unfortunately, the last two decades show that this particular free lunch is now over, and that additional performance can only be gained in combination with additional complexity. We see chips having multiple cores, multiple levels of caches, and larger, more specialized instruction sets.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
ELS'22, March 21–22, 2022, Porto, Portugal
© 2022 Copyright held by the owner/author(s).
<https://doi.org/10.5281/zenodo.6335627>

In terms of programming models for multiple cores, pretty much all programming languages are sitting in the same boat. Programmers are compelled to use multiple threads, and some of the dozens of possible synchronization and communication primitives for distributing and coordinating work. The more interesting challenge is to increase the performance of a single core. In that case, and assuming memory bandwidth and latency is not an issue, the performance can only be improved by using more powerful instructions. The most important ones for doing so are SIMD instructions, where a single instruction can perform an operation simultaneously on all elements of a small, specialized vector. This paper is about using SIMD instructions in Common Lisp.

There are two challenges when incorporating SIMD instructions into a programming language. The first challenge is to seamlessly integrate the low-level functionality for manipulating the 128 bit, 256 bit, or even 512 bit wide SIMD packs offered by the hardware. The second challenge is to provide the programmer with tools that can automatically convert scalar code to efficient SIMD code, at least for the most frequently occurring cases. Without such a tool for automatic conversion, SIMD programming is unnecessarily error-prone and cumbersome, and will likely only be used by a small group of extremely dedicated programmers.

In this paper we present two libraries. The first one is `sb-simd`¹, a SBCL-specific library that provides convenient bindings for various SIMD instructions sets. The second one is `Loopus`², a portable loop optimization framework that works via macros and source to source transformations, and that automatically turns many kinds of scalar loops into more efficient versions using SIMD instructions. We conclude with a performance comparison of Common Lisp code using `Loopus`, and C code using the most recent version of GCC.

2 RELATED WORK

Programs written in Common Lisp are frequently among the fastest in scenarios that benefit from extensive metaprogramming, or from incremental compilation. Examples of this are Paul Graham's macro for generating Bézier curves[2], Breannán Ó Nualláin's DSL for graph algorithms[5], or Børge Svingen's use of on-the-fly compilation for genetic programming[3]. These projects achieve high performance by exploiting cases where Lisp has an inherent advantage. Our work differs from this in that we want to achieve high performance in cases where Lisp has no inherent advantage, such as image processing or number crunching.

The most recent publication comparing the performance of Lisp and C for a case where Lisp has no inherent advantage is from Didier Verna and was published in 2006[4]. In that paper, the author shows how Lisp can reach and sometimes even exceed the performance of

¹<https://github.com/marcoheisig/sb-simd>

²<https://github.com/marcoheisig/Loopus>

C for several simple image manipulation tasks. However, neither the C code nor the Lisp code in that paper use SIMD instructions. All recent C compilers will use SIMD instructions for such tasks, so the only way to catch up once more is to use SIMD instructions in Lisp, too.

Apart from the paper of Verna, there is also the seminal, although somewhat dated, book about the performance and evaluation of Lisp systems by Richard P. Gabriel [1], discussing the performance of various Lisp implementations in general. From today's perspective, the most important information therein is the discussion of the various tradeoffs being made when implementing Lisp and their implications for performance.

3 SB-SIMD

The library `sb-simd` allows Common Lisp code to utilize SIMD instructions. In contrast to most other SIMD interfaces, e.g., C intrinsics, `sb-simd` incorporates the usual conveniences of Common Lisp. Each instruction set has its own package, and instruction set inheritance is modeled by re-exporting the symbols of each parent instruction set. All SIMD data types have their own built-in classes that can be queried and specialized upon. All functions automatically coerce all supplied arguments to the correct type, broadcast them to the correct SIMD width, and, when the underlying operator allows it, accept any number of arguments.

Luckily, we didn't have to trade convenience against speed when designing the public interface of `sb-simd`. SBCL's type inference and compiler are powerful enough to eliminate the overhead of dynamic typing, implicit conversions, broadcasts, and variadic arguments where necessary. Most calls to functions in `sb-simd` are compiled to a single machine instruction.

3.1 Software Architecture

The biggest challenge in writing this library was the sheer number of SIMD instructions available on a modern machine. The AVX instruction set alone provides almost 1000 instructions, which is more than the number of functions in the CL package!

The solution we came up with is, perhaps unsurprisingly, Lisp macros. We first create a table of metadata for each instruction set, which contains information about all data types, functions, corresponding mnemonics, mathematical properties, and so forth. Then we use the data in these tables to generate all the types, declarations, compiler transformations, machine code emitters, and functions, that are specified in each table. Not a single line of the code that is invoked when calling a function in `sb-simd` is written by hand. This is not just an eccentricity. By generating all code, we ensure that each bug breaks all functions simultaneously, which is much easier to detect and to fix.

All of the tables that are used to generate the functions and data structures in `sb-simd` can also be queried at run time. This makes it possible to look up the supported instruction sets, the functions exported by each instruction set, the arguments and return types of each function and much more. Our loop optimization library `Loopus` uses this metadata to perform automatic vectorization.

Something we didn't manage, unfortunately, is to make our SIMD interface portable across multiple Lisp implementations. The $k \times n$ interactions in code that runs on k implementations and n

architectures turned out to be too big of a headache. At some point we decided that it is better to have a high-quality SIMD interface for one Lisp implementation, than a mediocre one that is portable.

3.2 Data Types

The central data type in `sb-simd` is the SIMD pack. A SIMD pack is very similar to a specialized vector, except that its length must be a particular power of two that depends on its element type and the underlying hardware. The set of element types that are supported for SIMD packs is similar to that of SBCL's specialized array element types, except that there is currently no support for SIMD packs of complex numbers or characters.

The list of supported scalar types is shown in Figure 1. For each scalar data type X , there exists one or more SIMD data type $X.Y$ with Y elements. For example, in AVX there are two supported SIMD data types with element type `f64`, namely `f64.2` (128bit) and `f64.4` (256bit).

sb-simd	Common Lisp	
f32	single-float	
f64	double-float	
sN	(signed-byte N)	$N \in \{8, 16, 32, 64\}$
uN	(unsigned-byte N)	

Figure 1: Scalar data types in `sb-simd` and their corresponding Common Lisp type specifiers.

SIMD packs are regular Common Lisp objects that have a type, a class, and can be passed as function arguments. The price for this is that SIMD packs have both a boxed and an unboxed representation. The unboxed representation of a SIMD pack has zero overhead and fits into a CPU register, but can only be used within a function and when the compiler can statically determine the SIMD pack's type. Otherwise, the SIMD pack is boxed, i.e., spilled to the heap together with its type information. In practice, boxing of SIMD packs can usually be avoided via inlining, or by writing their values to specialized arrays (see section 3.11) instead of passing them around as function arguments.

3.3 Casts

For each scalar data type X , there is a function named X that is equivalent to $(\lambda (v) (\text{coerce } v 'X))$. For each SIMD data type $X.Y$, there is a function named $X.Y$ that ensures that its argument is of type $X.Y$, or, if the argument is a number, calls the cast function of X and broadcasts the result.

All functions provided by `sb-simd` (apart from the casts themselves) implicitly cast each argument to its expected type. So to add the number five to each single float in a SIMD pack x of type `f32.8`, it is sufficient to write $(\text{f32.8+ } x \ 5)$. We don't mention this implicit conversion explicitly in the following sections, so if any function description states that an argument must be of type $X.Y$, the argument can actually be of any type that is a suitable argument of the cast function named $X.Y$.

3.4 Constructors

For each SIMD data type $X.Y$, there is a constructor named `make- $X.Y$` that takes Y arguments of type X and returns a SIMD pack whose elements are the supplied values.

3.5 Unpackers

For each SIMD data type $X.Y$, there is a function named `$X.Y$ -values` that returns, as Y multiple values, the elements of the supplied SIMD pack of type $X.Y$.

3.6 Reinterpret Casts

For each SIMD data type $X.Y$, there is a function named `$X.Y$!` that takes any SIMD pack or scalar datum and interprets its bits as a SIMD pack of type $X.Y$. If the supplied datum has more bits than the resulting value, the excess bits are discarded. If the supplied datum has less bits than the resulting value, the missing bits are assumed to be zero.

3.7 Associatives

For each associative binary function, e.g., `two-arg- $X.Y$ -OP`, there is a function `$X.Y$ -OP` that takes any number of arguments and combines them with this binary function in a tree-like fashion. If the binary function has an identity element, it is possible to call the function with zero arguments, in which case the identity element is returned. If there is no identity element, the function must receive at least one argument.

Examples of associative functions are `f32.8+`, for summing any number of 256 bit packs of single floats, and `u8.32-max`, for computing the element-wise maximum of one or more 256 bit packs of 8 bit integers.

3.8 Reducers

For binary functions `two-arg- $X.Y$ -OP` that are not associative, but that have a neutral element, we provide functions `$X.Y$ -OP` that take any positive number of arguments and return the reduction of all arguments with the binary function. In the special case of a single supplied argument, the binary function is invoked on the neutral element and that argument. Reducers have been introduced to generate Lisp-style subtraction and division functions.

Examples of reducers are `f32.8/`, for successively dividing a pack of 32 bit single floats by all further supplied packs of 32 bit single floats, or `u32.8-` for subtracting any number of supplied packs of 32 bit unsigned integers from the first supplied one, except in the case of a single argument, where `u32.8-` simply negates all values in the pack.

3.9 Comparisons

For each SIMD data type $X.Y$, there exist conversion functions `$X.Y$ <`, `$X.Y$ <=`, `$X.Y$ >`, `$X.Y$ >=`, and `$X.Y$ =` that check whether the supplied arguments are strictly monotonically increasing, monotonically increasing, strictly monotonically decreasing, monotonically decreasing, equal, or nowhere equal, respectively. In contrast to the Common Lisp functions `<`, `<=`, `>`, `>=`, `=`, and `/=` the SIMD comparison functions don't return a generalized boolean, but a SIMD pack of unsigned integers with Y elements. The bits of each unsigned

integer are either all one, if the values of the arguments at that position satisfy the test, or all zero, if they don't. We call a SIMD packs of such unsigned integers a *mask*.

3.10 Conditionals

The SIMD paradigm is inherently incompatible with fine-grained control flow. A piece of code containing an `if` special form cannot be vectorized in a straightforward way, because doing so would require as many instruction pointers and processor states as there are values in the desired SIMD data type. Instead, most SIMD instruction sets provide an operator for selecting values from one of two supplied SIMD packs based on a mask. The mask is a SIMD pack with as many elements as the other two arguments, but whose elements are unsigned integers whose bits must be either all zeros or all ones. This selection mechanism can be used to emulate the effect of an `if` special form, at the price that both operands have to be computed each time.

In `sb-simd`, all conditional operations and comparisons emit suitable mask fields, and there is a `$X.Y$ -if` function for each SIMD data type with element type X and number of elements Y whose first arguments must be a suitable mask, whose second and third argument must be objects that can be converted to the SIMD data type $X.Y$, and that returns a value of type $X.Y$ where each element is from the second operand if the corresponding mask bits are set, and from the third operand if the corresponding mask bits are not set. An example of masks and conditionals is given in Figure 3.

3.11 Loads and Stores

In practice, a SIMD pack $X.Y$ is usually not constructed by calling its constructor, but by loading Y consecutive elements from a specialized array with element type X . The functions for doing so are called `$X.Y$ -aref` and `$X.Y$ -row-major-aref`, and have similar semantics as Common Lisp's `aref` and `row-major-aref`. In addition to that, some instruction sets provide the functions `$X.Y$ -non-temporal-aref` and `$X.Y$ -non-temporal-row-major-aref`, for accessing a memory location without loading the referenced values into the CPU's cache.

For each function `$X.Y$ -foo` for loading SIMD packs from an array, there also exists a corresponding function (`setf $X.Y$ -foo`) for storing a SIMD pack in the specified memory location. An exception to this rule is that some instruction sets (e.g., SSE) only provide functions for non-temporal stores, but not for the corresponding non-temporal loads.

One difficulty when treating the data of a Common Lisp array as a SIMD pack is that some hardware instructions require a particular alignment of the address being referenced. Luckily, most architectures provide instructions for unaligned loads and stores that are, at least on modern CPUs, not slower than their aligned equivalents. So by default we translate all array references as unaligned loads and stores. An exception are the instructions for non-temporal loads and stores, that always require a certain alignment. We do not handle this case specially, so without special handling by the user, non-temporal loads and stores will only work on certain array indices that depend on the actual placement of that array in memory. We'd be grateful if someone could point us to a mechanism for constraining the alignment of Common Lisp arrays in memory.

3.12 Specialized Scalar Operations

Finally, for each SIMD function $X.Y\text{-OP}$ that applies a certain operation OP element-wise to the Y elements of type X , there exists also a function $X\text{-OP}$ for applying that operation only to a single element. For example, the SIMD function $f64.4+$ has a corresponding function $f64+$ that differs from $cl:+$ in that it only accepts arguments of type double float, and that it adds its supplied arguments in a fixed order that is the same as the one used by $f64.4$.

There are good reasons for exporting scalar functions from a SIMD library, too. The most obvious one is that they obey the same naming convention and hence make it easier to locate the correct functions. Another benefit is that the semantics of each scalar operation is precisely the same as that of the corresponding SIMD function, so they can be used to write reference implementations for testing. A final reason is that scalar functions can be used to simplify the life of tools for automatic vectorization.

3.13 Instruction Set Dispatch

One challenge that is unique to image-based programming systems such as Lisp is that a program can run on one machine, be dumped as an image, and then resumed on another machine. While nobody expects this feature to work across machines with different architectures, it is quite likely that the machine where the image is dumped and the one where execution is resumed provide different instruction set extensions.

As a practical example, consider a game developer that develops software on an x86-64 machine with all SIMD extensions up to AVX2, but then dumps it as an image and ships it to a customer whose machine only supports SIMD extensions up to SSE2. Ideally, the image should contain multiple optimized versions of all crucial functions, and dynamically select the most appropriate version based on the instruction set extensions that are actually available.

This kind of run time instruction set dispatch is explicitly supported by means of the `instruction-set-case` macro. The code resulting from an invocation of this macro compiles to an efficient jump table whose index is recomputed on each startup of the Lisp image. An simple example of such an instruction set dispatch is given in Figure 2.

4 LOOPUS

Even though the interface provided by `sb-simdis` relatively convenient — at least when comparing it to similar libraries in other programming languages — there are certain repetitive patterns when writing vectorized code that almost beg for another layer of abstraction via macros. The most frequent repetitive pattern is that of using two loops to process a range of data: One with a step size that is the vectorization width, and one with a step size of one for handling the remainder. Figure 3 gives an example for this pattern. Further repetitive patterns are that of rewriting calls to `aref` as uses of `row-major-aref`, and hoisting all the loop invariant part of the index calculation outside of the loop.

After writing a variety of prototypes, we decided to create a portable loop optimization library for Common Lisp that can be used via macros. The library is invoked by using the `loopus:for` macro for looping over a range of integers. Once that macro is encountered, the whole form is turned into a tree of loops, where

```

1 (defpackage #:sb-simd-user
2   (:use #:common-lisp #:sb-simd)
3   (:local-nicknames
4     (#:sse2 #:sb-simd-sse2)
5     (#:avx #:sb-simd-avx2)))
6
7 (in-package #:sb-simd-user)
8
9 (defun quadruple4 (array)
10  (declare (type (simple-array f64 (4)) array))
11  (declare (optimize (speed 3) (safety 0)))
12  (progn array
13    (instruction-set-case
14      (:avx
15       (setf (avx:f64.4-aref array 0)
16             (avx:f64.4*
17              (avx:f64.4-aref array 0)
18              4)))
19      (:sse2
20       (setf (sse2:f64.2-aref array 0)
21             (sse2:f64.2*
22              (sse2:f64.2-aref array 0)
23              4)
24             (sse2:f64.2-aref array 2)
25             (sse2:f64.2*
26              (sse2:f64.2-aref array 2)
27              4))))))

```

Figure 2: A simple example for selecting the best available instruction set at run time: The eight elements of a supplied vector of double floats are quadrupled, using either AVX instructions, or, if those aren't available, SSE2 instructions.

each loop contains zero or more data flow graphs whose nodes are function calls, and whose roots are array store instructions or reduction statements. Each data flow graph may also reference nodes from any of the graphs of the surrounding loops. The leaves of each data flow graph are either constants or array load instructions.

Only a small subset of Common Lisp is allowed in the body of a loopus: for macro: functions, macros, and the special operators `let`, `let*`, `locally`, and `progn`. For now, this subset strikes the right balance between expressiveness and ease of optimization, but we may add support for further special operators in the future. The good news is that once a programmer obeys these restrictions, the entire loop nest and all expressions therein are subject to the following optimizations:

- Rewriting of multi-dimensional array references to references using only a single row-major index.
- Symbolic optimization of polynomials, and especially of the expressions for calculating array indices.
- Hoisting of loop invariant code.
- Automatic SIMD vectorization.

One may wonder why we use macros and didn't just add our loop optimizations to SBCL directly. The reason is that implementing loop optimizations for the entire Common Lisp language is a daunting task. The many possible interactions of language features would force us to be conservative in terms of optimization, or spend much more time on the development that we can currently spare. One advantage of providing optimizations as a macro, is that they are automatically available to all Lisp implementations.

5 EXAMPLES

5.1 Sum of Positive Numbers

This first example illustrates the various features provided by `sb-simd`. We deliberately don't utilize `Loopus` for this example to give a realistic impression of how programming with raw SIMD instructions looks like. The example problem is that of summing numbers in a supplied vector, with the additional constraint that numbers less than zero shall be ignored. The AVX2 vectorized code to perform this task is given in Figure 3.

```

1 (in-package #:sb-simd-avx2)
2
3 (defun sum-positive-numbers (vec)
4   (declare (type (simple-array f64 (*)) vec))
5   (let ((n (array-total-size vec))
6         (i 0)
7         (acc (f64.4 0))
8         (result 0d0))
9     (declare (f64.4 acc) (f64 result))
10    (loop while (<= i (- n 4)) do
11      (let ((v (f64.4-aref vec i)))
12        (f64.4-incf acc
13          (f64.4-if (f64.4> v 0) v 0))
14        (incf i 4)))
15    (f64-incf result (f64.4-hsum acc))
16    (loop while (< i n) do
17      (let ((v (f64-aref vec i)))
18        (f64-incf result
19          (f64-if (f64> v 0) v 0)))
20      (incf i))
21    result))
22

```

Figure 3: Summing all positive numbers in a vector, using AVX2 intrinsics. Two loops are needed to process any number of elements correctly: One vectorized loop with a step size of four (lines 10–14), and another one for handling the remainder (lines 16–20).

5.2 Jacobi

In this second example, we compare the performance of Common Lisp and C for the problem of applying Jacobi's method on a two-dimensional domain. For the C code, we took the best implementation we could find (Figure 4) and compiled it with GCC 9.2 and with

highest optimization settings (`-Ofast -march=native`). For the Lisp code in Figure 5 we used SBCL 2.2.0 and our loop optimization framework `Loopus`. The most critical part of assembler code of both versions is shown in Figures 6 and 7.

```

1 void jacobi(double* dst, double* src,
2           unsigned int rows,
3           unsigned int columns) {
4   double *C = dst + columns + 1;
5   double *N = src + 1;
6   double *W = src + columns ;
7   double *E = src + columns + 2;
8   double *S = src + 2*columns + 1;
9
10  for(size_t iy = 0; iy < rows - 2; ++iy) {
11    for(size_t ix = 0; ix < columns - 2; ++ix) {
12      size_t idx = iy * columns + ix;
13      C[idx] = 0.25 * (N[idx]+S[idx]+W[idx]+E[idx]);
14    }
15  }
16 }
17

```

Figure 4: An efficient C implementation of Jacobi's method.

```

1 (defun jacobi (dst src)
2   (declare (type (simple-array f64 2) dst src))
3   (loopus:for (i 1 (1- (array-dimension dst 0)))
4     (loopus:for (j 1 (1- (array-dimension dst 1)))
5       (setf (f64-aref dst i j)
6         (f64* 0.25d0
7           (f64+
8             (f64-aref src i (1+ j))
9             (f64-aref src i (1- j))
10            (f64-aref src (1+ i) j)
11            (f64-aref src (1- i) j)))))))
12

```

Figure 5: A Common Lisp implementation of Jacobi's method.

One can see that the assembler code produced by GCC (Figure 6) and SBCL (Figure 7) is extremely similar. Both versions use three 256 bit vector additions and one 256 bit vector multiplication. The only differences are that GCC's assembler code combines two loads directly with the subsequent addition, and manages to perform the loop test entirely in registers. In SBCL, the nature of how operations of its virtual machine are translated to assembler instructions makes it very hard to combine loads with subsequent instructions. We haven't yet investigated why SBCL decided to reference the stack for checking for termination of the innermost loop.

The reason that SIMD operations appear at all in the code by SBCL is that `Loopus` has automatically rewritten the scalar loop

```

1 L1: vmovupd ymm5, [rbx+rax*1]
2   vmovupd ymm6, [r11+rax*1]
3   vaddpd  ymm0, ymm5, [rcx+rax*1]
4   vaddpd  ymm1, ymm6, [r10+rax*1]
5   vaddpd  ymm0, ymm0, ymm1
6   vmulpd  ymm0, ymm0, ymm3
7   vmovupd [rdx+rax*1], ymm0
8   add rax,0x20
9   cmp [rsp+0x20], rax
10  jne L1
11

```

Figure 6: The assembler code of the innermost loop produced by GCC 9.2 for our C code.

```

1 L1: vmovupd ymm0, [rsi+rbx*4+8]
2   vmovupd ymm1, [rsi+rbx*4-8]
3   vmovupd ymm2, [r9+rbx*4]
4   vmovupd ymm3, [r8+rbx*4]
5   vaddpd  ymm0, ymm0, ymm1
6   vaddpd  ymm1, ymm2, ymm3
7   vaddpd  ymm0, ymm0, ymm1
8   vmulpd  ymm0, ymm4, ymm0
9   vmovupd [rcx+rbx*4], ymm0
10  add rbx, 8
11  cmp rbx, rdx
12  jl L1
13

```

Figure 7: The assembler code of the innermost loop produced by SBCL 2.2 for our Lisp code.

from Figure 5 line 4–11 as two loops, where the first loop has a step size of four and uses SIMD instructions and where the second loop has a step size of one and handles the remainder in case the loop length is not divisible by four. Furthermore, Loopus replaces each access to a Common Lisp array by direct pointer arithmetic, and hoists most of the loop index calculations outside of the innermost loop. This way, each array access can be encoded as a single load instruction whose address is the sum of two registers, where the value of the second register is scaled by a power of two, plus a small constant.

To compare the performance of our Lisp and C codes, we ran each Jacobi implementation for several minutes on a problem that fits well into the L1 cache of the target machine. In doing so, we ensure that the computation is not limited by memory throughput and thus accurately reflects how well the CPU can digest the generated machine code. Our results for a variety of x86-64 CPUs are shown in Figure 8.

CPU	Lisp	C	Ratio
AMD EPYC 7451 “Naples”	8.1	8.6	0.94
AMD EPYC 7452 “Rome”	8.2	10.0	0.82
Intel Xeon “Skylake” Gold 6148	10.8	13.9	0.78
Intel Xeon “Cascade Lake” Gold 6248	7.2	9.3	0.77
AMD EPYC 7543 “Milan”	12.8	18.1	0.71
Intel Xeon “Haswell” E5-2695 v3	6.9	9.8	0.70
Intel Xeon “Icelake” Platinum 8360Y	11.3	16.1	0.70
Intel Xeon “Icelake” Platinum 8358	7.9	11.3	0.70
Intel Xeon “Broadwell” E5-2697 v4	5.0	7.5	0.67

Figure 8: Performance in GFlop/s for Jacobi’s method on various CPU architectures, as well as the ratio of the performance of the Lisp version and the C version.

6 CONCLUSIONS

We have narrowed the performance gap between Common Lisp and C for number crunching to something between 6% and 33%, depending on the target hardware. We achieved this by developing a low-level SIMD library for SBCL, named *sb-simd*, and a portable library for loop optimization and automatic vectorization, named *Loopus*.

The interface provided by *sb-simd* is the most convenient way of using SIMD intrinsics among all programming languages known to us. It treats SIMD packs as regular, typed objects, allows the development of SIMD codes at the REPL, and even provides an introspection mechanism for querying the available instructions and data types at run time. We hope that the interface provided by *sb-simd* will eventually be supported by multiple Lisp implementations and turn into a de-facto standard similar to *bordeaux-threads*.

The loop optimization library *Loopus* is still in its infancy, but already powerful enough to vectorize inner loops written in a certain subset of Common Lisp. This makes it possible to harness SIMD instructions for a wide variety of loops using minimal effort. We hope that this paper will attract further contributors, and that *Loopus* will one day reach feature parity with the loop optimization machinery in GCC and Clang.

What makes us particularly excited about these new libraries is that they turn Common Lisp into a viable language for high performance computing. Programmers in that domain can now finally enjoy the convenience and flexibility of using Lisp, and, most importantly, harness the power of Lisp macros to develop lightweight, domain-specific optimizations. We are confident that in many cases, such domain-specific optimizations can outperform the general-purpose work that is normally done by a compiler. We are looking forward to working on such optimizing Lisp macros in the future.

7 ACKNOWLEDGMENTS

This work wouldn’t have been possible without the support of many other people and organizations. We express our heartfelt gratitude to:

- KONWIHR, the Competence Network for Scientific High Performance Computing in Bavaria, for funding the advancement of SIMD instructions in SBCL and our work on *sb-simd*.

- Paul Khuong, for writing the initial support for SIMD programming in SBCL.
- Stas Boukarev, for adding the low-level machinery required for AVX, AVX2, and FMA instructions to SBCL.
- All the other SBCL developers for kindly supporting this work and providing feedback, especially Douglas Katzmann, Charles Zhang, and Christophe Rhodes.
- Bela Pecsek for being an ardent user of sb-simd since the first day, for porting various benchmarks from C to SIMD-enhanced Lisp, and for his frequent feedback and bug reports.
- Nicolas Neuss, Hayley Patton, Michał Herda, Jan Münch, and Shubhamkar Ayare, for some valuable discussions.
- The group from the Erlangen National High Performance Computing Center (NHR@FAU) for giving us access to their benchmark systems, and for insistently questioning the performance of Lisp. The latter turned out to be a surprisingly good motivation.

REFERENCES

- [1] GABRIEL, R. P. *Performance and Evaluation of LISP Systems*. The MIT Press, 08 1985.
- [2] GRAHAM, P. *On LISP*. Pearson, Upper Saddle River, NJ, Sept. 1993.
- [3] SVINGEN, B. When lisp is faster than C. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation* (New York, NY, USA, 2006), GECCO '06, Association for Computing Machinery, p. 957–958.
- [4] VERNA, D. How to make lisp go faster than C. In *Proceedings of the International MultiConference of Engineers and Computer Scientists* (Hong Kong, June 2006), International Association of Engineers.
- [5] Ó NUALLÁIN, B. Executable pseudocode for graph algorithms. In *Proceedings of the 8th European Lisp Symposium* (Apr. 2015), ELS2015.

April: APL Compiling to Common Lisp

Andrew Sengul
asengul@fastmail.fm

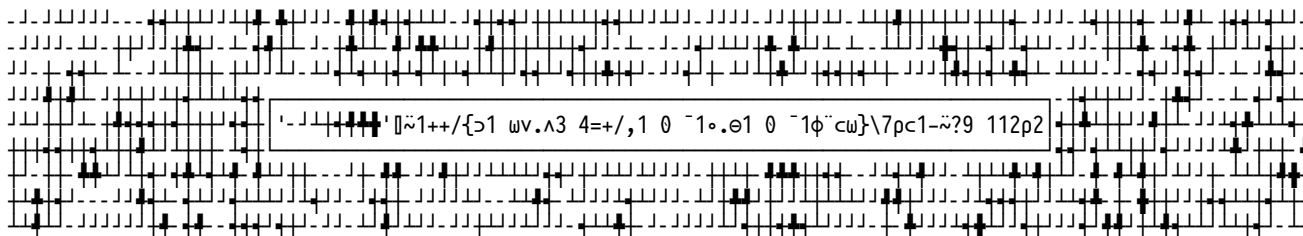


Figure 1: An APL expression framed by its output

ABSTRACT

This paper demonstrates the April APL compiler (code hosted at <https://github.com/phantomics/april>). April compiles a subset of the APL language into Common Lisp, allowing APL's terse, efficient syntax to be leveraged for array processing and mathematical operations within a Common Lisp program. Along with the compiler April includes a suite of specification tools making it easy to extend the language, allowing for a uniquely flexible development approach. Released under the permissive Apache 2.0 license, April has been leveraged in a graphical display hardware startup and a variety of applications including statistical analysis, vector graphics and terminal interfaces.

CCS CONCEPTS

- **Software and its engineering** → *Software design engineering*;
- **Computing methodologies** → **Computer algebra systems**;
- Representation of mathematical functions**.

KEYWORDS

Demonstration, Compiler, Array, DSL, APL, Lisp, Linear algebra, Vector languages, Interoperability

ACM Reference Format:

Andrew Sengul. March 2022. April: APL Compiling to Common Lisp. In *Proceedings of the 15th European Lisp Symposium (ELS'22)*. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.5281/zenodo.6381963>

1 INTRODUCTION

APL is known for its exotic character set and minimalist style. Like Lisp the language was originally designed as a mathematical notation [7] and creator Ken Iverson didn't anticipate that APL expressions could be evaluated by a computer. His colleagues built the first APL interpreter using a variant of Iverson's notation simplified for use with a teletype terminal [4], just as John McCarthy's students traded M-expressions for S-expressions to develop the

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

ELS'22, March 21–22, 2022, Porto, Portugal

© March 2022 Copyright held by the owner/author(s).

<https://doi.org/10.5281/zenodo.6381963>

original IBM 704 Lisp interpreter. APL followed an evolutionary path somewhat similar to that of Lisp as the language grew to prominence on mainframes, functioning as a complete operating system for the machines where it ran [5].

Work on April began in late 2017 and it has since gone through multiple development iterations of its core compiler, functions and specification macros. A previous talk I gave on April can be viewed at <https://youtube.com/watch?v=AUEIgf9koc>. Since then April has evolved considerably, incorporating tacit function composition, inline operators and multithreading support for almost all functions.

2 USING APRIL

The simplest way to use April is to pass APL strings to the (`april`) macro. An example is (`april "1+1 2 3"`), which returns the vector `⍋(2 3 4)` – APL composes addition and other scalar functions over arrays, so the 1 is added to each element of `⍋(1 2 3)`. APL's core functions are all just one character long, like `+`, `-`, `×` and `÷`. April can also take files of APL code as input and it has a wide variety of configuration options. April's parameters may be passed as the first argument to the (`april`) macro inside a (`with`) form.

A complete introduction to the APL language is far beyond the scope of this section but a good starting point is April's README file, located at the link in the abstract. The README has guidelines on ways of entering APL characters and links to resources including language references and interactive tutorials. April is included in `Quicklisp` and installing it is as simple as evaluating (`ql:quickload 'april'`).

April runs its character input through a lexer, converting the characters to tokens which are then fed to a compiler that generates Common Lisp code. The (`:print-tokens`) parameter prints tokens output by the lexer before they are passed to the compiler:

```
* (april (with (:print-tokens)) "1+1 2 3")
(3 2 1 (:FN #\+) 1)
⍋(2 3 4)
```

Note that the lexer accumulates the tokens in reverse order; this is natural since APL code is evaluated from right to left and the tokens are thus fed to the compiler starting from the end of each line read.

The (`:compile-only`) parameter causes April to print its compiled output instead of evaluating it:


```
* (april (with (:compile-only)) "1+1 2 3")
(IN-APRIL-WORKSPACE COMMON
 (LET ((OUTPUT-STREAM *STANDARD-OUTPUT*))
  (DECLARE (IGNORABLE OUTPUT-STREAM))
  (SYMBOL-MACROLET
   ((INDEX-ORIGIN  $\underline{e}$ *INDEX-ORIGIN*)
    (PRINT-PRECISION  $\underline{e}$ *PRINT-PRECISION*)
    (COMPARISON-TOLERANCE
      $\underline{e}$ *COMPARISON-TOLERANCE*)
    (DIVISION-METHOD  $\underline{e}$ *DIVISION-METHOD*)
    (RNGS  $\underline{e}$ *RNGS*))
   (A-OUT (A-CALL (APL-FN-S +)
                 (AVEC 1 2 3) 1)
          :PRINT-PRECISION
          PRINT-PRECISION))))
```

Note the \underline{e} reader macro. It works to intern symbols in the proper workspace packages in tandem with the `(in-april-workspace)` macro. Like other APLs April stores its functions and variables in named workspaces, which are implemented as Common Lisp packages. When the macro `(in-april-workspace common ...)` is expanded, an instance of `symbol` within is transformed into the symbol `april-workspace-common::symbol`. Considerable work has been done to make April's compiled output human-readable, with many macros abbreviating common structures that would otherwise bloat the code.

Compared to other APL implementations April stands out for its seamless interoperability with Common Lisp, and through CL other languages and systems. APLs have traditionally been implemented as monolithic interpreters whose communication with external APIs must be done through plugins to the executable. The most popular APL implementation, Dyalog APL¹, is proprietary and thus any such plugin must be created by Dyalog. Other free software APLs exist, but their implementation in Algol descendants like C++ and Java makes extension an ordeal.

The simplest way to pass values from CL into April is to use the `(april-c)` macro. Here, the number 10 is passed as the second argument to `(april-c)` and is represented by ω , which stands for the right argument, within the APL function.

```
* (april-c "{ $\omega$ +5}" 10)
15
```

April's `(:state)` parameter with the sub-parameters `(:in)` and `(:out)` can be used for more complex input and output.

```
* (april (with (:state :in ((a 3) (b 5))
               :out (a c)))
      "c+a+ $\omega$ ")
3
#(4 5 6 7 8)
```

Variables named `a` and `b` are passed in, and the variables named `a` and `c` are returned. The `[ω index]` function seen here produces a vector of numbers from 1 to its argument, and `←` assigns the result of the vector's addition to `a` to the variable `c`.

Passing functions into April is likewise simple:

```
* (april (with (:store-fun
              (add-ten (lambda (x)
                        (+ x 10))))))
      "")
NIL ;; nothing is evaluated, so nil is returned

* (april "addTen 20")
30
```

Dash-separated variable names are converted to camelCase within April, since the `-` character expresses the subtraction function in APL and so cannot be part of a variable name.

April does not have any stock functions for system interaction, but using the `(:store-fun)` parameter they can easily be added as required. Here is an example using the `uiop`² library:

```
* (april (with
         (:store-fun
          (sh (lambda (s)
                (uiop:run-program
                 (coerce s 'string)
                 :output :string))))))
      "")
NIL

* (april "' GOODBYE',~sh 'echo HELLO'")
"HELLO
GOODBYE"
```

In just a few lines, April can thus be extended to support running terminal commands. Recurring questions addressed to other vector language projects like “When will we get JSON support?” and “When will we be able to make HTTP requests?” can be addressed by April users within minutes.

3 IMPLEMENTATION

Common Lisp has powerful tools for working with arrays but their syntax is often cumbersome. APL can build and transform arrays with only a handful of characters, making tasks that take a large amount of code in Common Lisp much simpler to write. This led to my interest in leveraging APL within Common Lisp, and CL is one of the best choices of language to implement APL because it has almost all of the necessary array faculties inbuilt. With support for nested arrays, high-rank arrays and zero-rank arrays, it's easy to work with April's array output using standard CL code. This section outlines some of the more interesting challenges encountered in the course of developing April.

3.1 The Core Specification

Building a programming language is a complex task. I wrote the `(specify-vex-idiom)` macro to mitigate this complexity, implementing a core specification for April that can be seen in the source file `april/spec.lisp`³. This large macro specifies all of April's lexical functions and operators along with its language utilities, putting all the language's significant configuration in one central

¹<https://www.dyalog.com/>

²<https://gitlab.common-lisp.net/asdf/asdf/-/tree/master/uiop>

³<https://github.com/phantomics/april/blob/master/spec.lisp>

location. Information like the inverse forms of functions and their alternative character representations can be found here, along with all of their unit tests.

This centralized organization has made the development of the language significantly faster than would have been possible with a different style. For example, the recent addition of an inverse form for the [`⊔ where`] function required just one new line in the spec along with an 18-line function added to April's main library.

Moreover, April's specification macros can be used to augment the language with new functional characters in just a handful of lines.

```
(extend-vex-idiom
 april
 (functions
  (with (:name :extra-functions))
  (⊖ (has :title "Add3")
   (ambivalent
    (scalar-function
     (lambda (omega) (+ 3 omega)))
    (scalar-function
     (lambda (alpha omega) (+ 3 alpha omega))))))
 (tests (is "⊖77" 80)
        (is "⊖87" 18))))
```

In this code, functional character `⊖` is added to the April language, implementing a rather silly function called `Add3` that adds three to its argument (if given one argument) or to the sum of its arguments (if given two arguments). A pair of unit tests for this function are added to the main test sequence as well. The (`extend-vex-idiom`) macro can also be used to overload April's utilities, like the functions that strip comments from code and parse numeric strings.

With this macro skilled developers can patch the language for specific applications, creating custom variants of April with no need to fork its main codebase. The specification macros are implemented in April's sub-package `vex`⁴, which contains a set of general tools for implementing vector languages. In the future other vector languages may be implemented based on the `VEX` model.

3.2 Array Prototypes

The only significant array feature APL has that CL lacks is empty array prototypes. The prototype of an APL array is its first row-major element [1]. Prototypes are used by functions like [`↑ take`] and [`/ expand`] to fill the empty space resulting when an array is made larger. When an APL array is reduced to size 0, as with functions like [`ρ shape`] and [`↓ drop`], it retains the “memory” of its prototype so that if it is expanded to a nonzero size, the prototype will be used to populate the space in the new array. For a character array the prototype is a blank space, and for a numeric or mixed array the prototype is 0. For an array whose first row-major element is a nested array, the prototype is an array of the same shape whose elements are the prototype of the nested array. Thus if the first element in the array is the matrix `#2A((1 2)(3 4))`, the array's prototype will be `#2A((0 0)(0 0))`.

⁴<https://github.com/phantomics/april/blob/master/vex/vex.lisp>

The Common Lisp array model does not include a “prototype” value, but for non-zero-sized arrays it's unnecessary since the prototype is simply an “empty” version of the first element. Functions that output a zero-sized array will displace the array to a one-element vector containing a list of metadata with the prototype. The (`array-displacement`) function can be used to fetch the metadata from any context, making it straightforward to get the empty array's prototype for functions that use it.

3.3 Multithreading

One of April's recent development priorities has been to use multithreading wherever possible. April uses macros called (`xdotimes`) and (`ydotimes`) to accomplish this, leveraging the `lparallel`⁵ library. These macros' definitions can be found in the source file `april/aplesque/aplesque.lisp`⁶. The (`xdotimes`) macro is used for algorithms that iterate across the elements in a function's output array in row-major order. This macro splits an array processing task into appropriately-sized segments to divide between threads. Most CL implementations have been observed to use registers with sizes equal to the sizes of array elements when modifying arrays of elements 8 bits in size or larger. When dealing with arrays that have integer elements smaller than 8 bits, 64-bit registers are usually used to hold the values of elements being processed. This means that when operating on arrays with sub-8-bit integer elements, threads must work upon sub-vectors of elements with a length of (`/ 64 element-size`) to stop elements from being clobbered as multiple threads try to write to the same location in memory.

The (`ydotimes`) macro is like (`xdotimes`) but it doesn't support sub-8-bit elements; in the case of arrays with elements smaller than 8 bits, (`ydotimes`) will perform a task synchronously. April uses (`ydotimes`) in cases where it's impractical to iterate over an output array in row-major order and thus the operation can't be divided into 64-bit segments for small integer elements.

Most of April's array-transforming functions have a similar design pattern. Based on the dimensions of the input array and the arguments passed to the function, the shape of the output array is determined. Then, April iterates over the output or input array using (`xdotimes`) or (`ydotimes`) and performs arithmetic on the row-major index of each output element to determine the corresponding row-major element in the input, finally copying the elements from the input array to the output array.

4 APPLICATIONS

April has been used for image editing, statistical analysis, web development, terminal interfaces and more. In my experience, while Lisp is unmatched as a general-purpose language APL enables the most intuitive development within its domain of array processing and discrete algorithms. April shares in the interactive features of Common Lisp environments, enabling developers to re-evaluate individual closures in source code, which standard APL environments don't allow - their interactivity is limited to the REPL and to reinterpreting entire discrete functions.

⁵<https://lparallel.org/>

⁶<https://github.com/phantomics/april/blob/master/aplesque/aplesque.lisp>

```

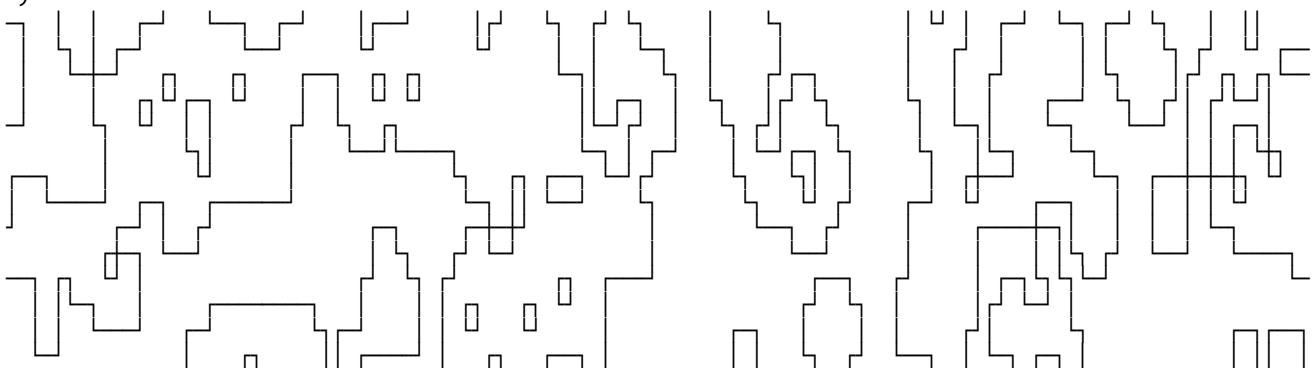
* (april "
random ← {⊖IO~?2ρ~|α ω}
  ⌘ create a randomized binary matrix
life ← {>1 ωv.Λ3 4=+/,1 0 ~1◦.⊖1 0 ~1φ~cω}
  ⌘ the classic Conway's Game of Life function
trace ← {α[;1]⊖~cα[;2]{1[α{ω×ω≠α}αωω]{2⊥,ω}⊖3 3~ω}
  ⌘ use [⊖ stencil] to outline cells according to matrix decoding maps

chars ← '- - | | r r L L 7 7 J J '
ints ← 48 384 144 288 16 416 128 304 32 400 256 176
xEncInts ← 68 69 257 261 321 324
decodings ← ints~{ωρ~1,ρω} chars~' '

H ← 14 ⌘ height
W ← 112 ⌘ width
I ← 5 ⌘ iterations of life function to perform before printing
M ← ' ' {(α,0)~ω[1;] {(~⊥×ω),~α⊖~cω~0} ω[2;] {α{ω×ω≠α}αωω} {2⊥,(2 2ρ0)⊖3 3ρ(9ρ2)τω}~⊥2*9} decodings
  ⌘ map of binary decodings of stencil matrices to box-drawing characters
M~← '+' ,xEncInts ⌘ add cross-line character values to decoding map

⊖+M trace life×I~H random W ◊ (⊖I), ' iterations'
")

```



"5 iterations"

Figure 2: APL evaluated via April implementing the Game of Life function with a convolution kernel to outline cells

4.1 Terminal Graphics

An example using April to generate text-based visuals is shown in Figure 2. This code includes a classic APL function used to implement mathematician John Conway’s Game of Life [2]. Rather than displaying the cells themselves, it uses the [⊖ stencil] operator to draw boxes around the locations of the cells. This operator can implement convolution kernels, a common technology in computer graphics [3][6]. Convolution kernels are used to blur and sharpen images, for pattern-matching (to detect faces, for example) and in this case to find edges.

The trace function uses [⊖ stencil] to process 3x3 submatrices of the binary matrix generated by the life function, producing 9-bit integers decoded from the binary vector displaced to each submatrix. In other words, matrix #2A((1 1 0)(1 0 0)(0 0 0)) is displaced to vector #(1 1 0 1 0 0 0 0 0) which decodes in binary to 416. This number corresponds to the box-drawing character r stored in the table M. For decoded values like 416 that indicate

the presence of adjacent cells while no cell is actually present at the position, box-drawing characters are placed in a character matrix of the same shape as the Game of Life matrix.

A more complex variant of this function is used in April’s ncurses demo application (april/demos/ncurses/⁷ in the repository). It’s integrated with the croatoan⁸ CL ncurses binding library to implement a terminal application displaying the cell outlines generated by the code in Figure 2. It also varies each character’s background color to reflect the presence or absence of cells in those spaces over time. Building these graphical algorithms in Common Lisp would have required much more code than could fit on one page.

Developing with ncurses has been regarded as a tedious and painful enterprise since using conventional languages means writing dozens of nested loops. April offers the potential to quickly and intuitively specify terminal interface elements and even add some animation and color to keep things fun.

⁷<https://github.com/phantomics/april/tree/master/demos/ncurses>

⁸<https://github.com/McParen/croatoan>

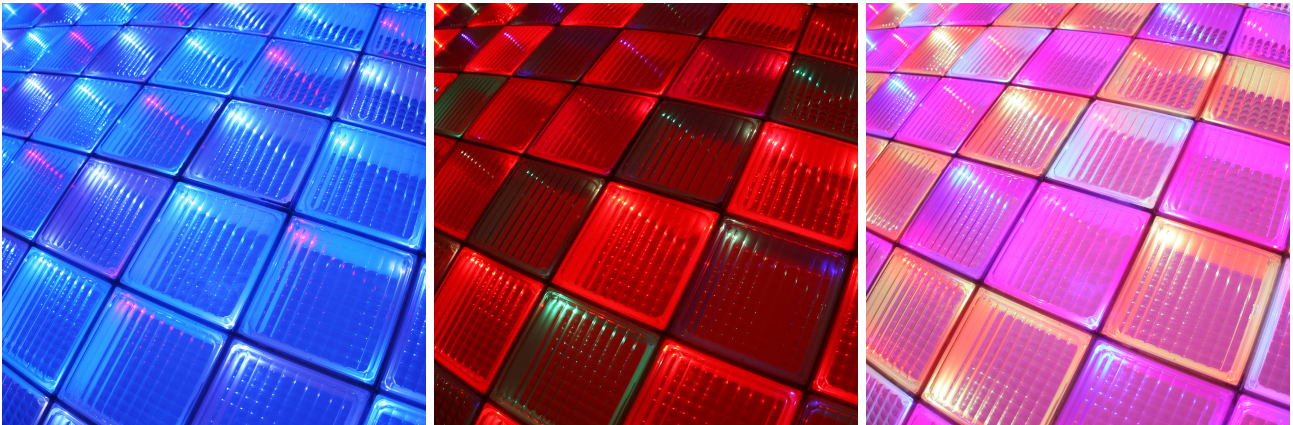


Figure 3: Three color combinations shown on the Bloxl display

4.2 Speaking of Color...

The April compiler's most prominent application is designing pixel animations for use with a custom LED display built by a hardware startup called Bloxl⁹. Raster graphics are a natural fit for APL; for instance, this code using the `opticl`¹⁰ image processing library is all that's needed to produce a palette matrix of the unique colors (one set of RGB values per row) in a PNG image:

```
(april-c "{ϕEτυ,(E+3ρ2*8)12 3 1ϕω}"
  (opticl:read-png-file "/path/to/image.png"))
```

While designing patterns for display on the LED device I experimented with different methods to generate appealing color combinations. This is one of the simpler algorithms I wrote:

```
(april-c "{(α×3)ρ1-~2*?ωρ8}" segment leds)
```

Figure 3 shows three of the resulting color schemes. A vector of random numbers between 1 and 8 of length `segment` is created, 2 is raised to the power of each element, 1 is subtracted from each result, and the output vector is repeated to fill a vector of length `leds` times 3 (3 RGB integer values for each LED). The resulting 8-bit integers will manifest a color series on an LED array. Varying the length of the `segment` will produce different patterns. In the span of about 10 minutes I wrote this code and used it to build a library of palettes for use with the Bloxl display, generating dozens of RGB vectors and saving the ones that looked good.

April has been a unique boon to the development of Bloxl since building animations often requires custom code for each animation that isn't used anywhere else. Using a more verbose language, I would be faced with the choice of either placing the custom code directly inside the spec for an individual animation and bloating it by many lines or collecting all custom animation functions in another part of the codebase, adding the cognitive overhead and technical debt of many more functions that are each only used for one task. April makes it possible to express sophisticated custom effects in just a line or two, negating complexity in a way that wouldn't otherwise be possible.

⁹<https://bloxl.co>

¹⁰<https://github.com/slyrus/opticl/>

5 ACKNOWLEDGEMENTS

Justin Dowdy and Nikolai Matiushev for many bug reports, Kevin Jones, Jérôme Ibanes and Nathan Rogers for steadfast support, Jan Münch, Elias Mårtenson, Marshall Lochbaum and many others on IRC, Matrix and Github for conversation and commentary.

6 CONCLUSIONS AND FUTURE WORK

I consider April to have substantially fulfilled my initial design goal: an alloy of two languages with complementary strengths. April has reduced the time needed to accomplish many tasks and made things possible that weren't before. At events featuring the Bloxl device I have live-coded effects that would have taken hours to assemble using conventional methods.

Upcoming design priorities for April include further speedups through the use of SIMD and even possible GPU acceleration through integration with ArrayFire¹¹. April remains slower than Dyalog APL but the compiler has a multifaceted framework for optimization, including its parallelizing macros and a pattern-matching system for code that can be implemented in a faster way than the compiler normally would (`april/patterns.lisp`¹²).

REFERENCES

- [1] APLWiki.com. Prototype, 2020. URL <https://aplwiki.com/wiki/Prototype>.
- [2] APLWiki.com. John scholes' conway's game of life, September 2021. URL https://aplwiki.com/wiki/John_Scholes%27_Conway%27s_Game_of_Life.
- [3] Thiago Carvalho. Basics of kernels and convolutions with opencv, 2020. URL <https://towardsdatascience.com/basics-of-kernels-and-convolutions-with-opencv-c15311ab8f55>.
- [4] Adin Falkoff. APL 360 history. In *Proceedings of the Conference on APL*, APL '69, page 8–15, New York, NY, USA, 1969. Association for Computing Machinery. ISBN 9781450373784. doi: 10.1145/800012.808128. URL <https://doi.org/10.1145/800012.808128>.
- [5] H. Hellerman. Experimental personalized array translator system. *Commun. ACM*, 7(7):433–438, Jul 1964. ISSN 0001-0782. doi: 10.1145/364520.364573. URL <https://doi.org/10.1145/364520.364573>.
- [6] Roger Hui. Towards improvements to stencil, 2020. URL <https://www.dyalog.com/blog/2020/06/towards-improvements-to-stencil/>.
- [7] Kenneth E. Iverson. Notation as a tool of thought. *Commun. ACM*, 23(8):444–465, Aug 1980. ISSN 0001-0782. doi: 10.1145/358896.358899. URL <https://doi.org/10.1145/358896.358899>.

¹¹<https://github.com/arrayfire/arrayfire>

¹²<https://github.com/phantomics/april/blob/master/patterns.lisp>

Tuesday, 22 March 2022



Transpiling Python to Julia using PyJL

Miguel Marcelino

INESC-ID/Instituto Superior Técnico, University of Lisbon
Lisbon, Portugal
miguel.marcelino@tecnico.ulisboa.pt

António Menezes Leitão

INESC-ID/Instituto Superior Técnico, University of Lisbon
Lisbon, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

ABSTRACT

Transpilers convert source code between programming languages. With the rise of new high-level programming languages, transpilers are ideal tools to speedup the conversion of libraries written in more established languages to newer and/or less popular ones, fostering their adoption.

Julia is a recently introduced programming language that targets various application areas of the widely popular Python language. Unfortunately, it still lacks many of the high-quality libraries found in Python. To speedup the development of libraries, we propose extending the PyJL transpiler to translate Python source code into human-readable, maintainable, and high-performance Julia source code.

Despite being at an early development stage, our preliminary results show that PyJL generates human-readable code that can achieve good performance with minor changes.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; • **General and reference** → **Cross-computing tools and techniques**.

KEYWORDS

Source-to-Source Compiler, Automatic Transpilation, Library Translation, Python, Julia

ACM Reference Format:

Miguel Marcelino and António Menezes Leitão. 2022. Transpiling Python to Julia using PyJL. In *Proceedings of the 15th European Lisp Symposium (ELS'22)*. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.5281/zenodo.6332890>

1 INTRODUCTION

A generic definition of a transpiler is a tool that transforms input source code into output source code, where the input and output source code can be written in the same or in different programming languages. The term *transformation* is relatively broad, and many solutions use more specific concepts to categorize different transpilation approaches. DMS [3], a tool that focuses on the automatic management of large software solutions, uses the concept of Design Maintenance. Other tools in the area of Safety-Critical Computing [28] use the concept of Source Code Manipulation to implement

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

ELS'22, March 21–22, 2022, Porto, Portugal

© 2022 Copyright held by the owner/author(s).

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

<https://doi.org/10.5281/zenodo.6332890>

fault-tolerance mechanisms. In the context of this research, we focus on the topic of Source-to-Source Translation, where transpilers translate source code from an input language to a target language.

The first transpiler was developed in 1978 to provide compatibility between an 8-bit and a 16-bit processor. It was called CONV86 [7] and was developed by Intel to translate assembly source code from the 8080/8085 to the 8086 processor. At the time, many other transpilers were developed with a similar purpose, such as TRANS86 and XLT86 [30]. Nowadays, with programmers developing software in higher-level programming languages, it makes sense to have transpilers operate at this level.

In recent years, we have seen the rise of many new high-level programming languages, such as Rust, Go, TypeScript and more. Among them is the Julia programming language, which claims to have the performance of C, the ease of use of Python, and the macro capabilities of Lisp, among others. However, Julia currently lacks the large library sets found in more established programming languages. Converting libraries from these languages to Julia would allow programmers to benefit from extended library support and from Julia's performance on modern hardware.

Manually translating large code-bases is a difficult task and requires substantial resources. For instance, the Commonwealth Bank of Australia converted its code-base from COBOL to Java, spending \$750 million over five years. In this regard, manually translating Python's large library set would be an enormous challenge. Using a transpiler to convert libraries automatically would benefit programmers. However, automatic translation is a challenge for a transpiler, which we will discuss in the following section.

2 AUTOMATIC TRANSLATION

Automating the translation between source and target languages is addressed with differentiating perspectives. LinJ [18] aims at a fully automatic translation of Common Lisp to Java source code. JSweet [24] translates Java to JavaScript and preserves JavaDoc documentation in JSDoc. Other tools, such as the Fortran-Python two-way transpiler [5], intentionally require manual intervention and request the programmer to annotate the input Python source code with type hints before translating it to Fortran.

We consider that automating the translation process is relevant, as the goal is to translate libraries. Furthermore, since the aim is to generate human-readable and maintainable code, the transpiler needs to translate language syntax and semantics as programmers would, by preserving the pragmatics of Julia.

To translate language syntax and semantics, we need to consider how different constructs map to the target language. As an example, consider the following code written in Python:

```
ls1 = [1, 2]
ls2 = [3, 4]
ls_sum = ls1 + ls2
```

Despite the fact that Julia has identical syntax for assignments, arrays, and arithmetic operators, executing this example in Julia would not yield the same results. In Julia, adding `ls1` and `ls2` results in an element-wise addition, producing `[4, 6]`. In Python, this results in the creation of a new list that contains the elements of both lists, i.e., `[1, 2, 3, 4]`. A correct translation to Julia would use the syntax:

```
ls_sum = [ls1;ls2]
```

Furthermore, if the source and target languages promote different programming paradigms, the transpilation process becomes even more difficult. For example, consider translating Object-Oriented (OO) Python code to Julia, where a transpiler would need to map functionalities such as multiple inheritance, which Julia currently does not support, and handle method overrides. Python classes also implement special methods, such as `__init__` and `__str__`. Additionally, Python also allows redefining built-in operators, such as arithmetic operators, for class instances. A transpiler that aims to preserve Julia's pragmatics would need to map all these functionalities while also generating a target program that is easy to understand and modify.

The mapping of library calls to the target language should also be considered. This can be done on a per-function basis, where functions in the input language are mapped to functions with equivalent behavior in the target language. For instance, calls to the function `np.argmax` of Python's NumPy [23] library, which retrieves the maximum value of a matrix, should be translated to calls to Julia's function `maximum`.

Type information is another important aspect of transpilation. In particular, the mapping of dynamically typed languages to statically typed ones presents a challenge, which was already addressed by some proposals ([18], [31]). Furthermore, languages such as Julia may benefit from type annotations to optimize code performance.

A transpiler can translate most use-cases automatically if the input and target languages have similar levels of abstraction. However, some cases present ambiguous translation scenarios, which we will discuss in the following section.

3 DISAMBIGUATE TRANSLATIONS

In the previous section, we discussed several conflicts that a transpiler should automatically resolve. However, there are some translation scenarios where automatic translation could result in the generation of convoluted code, which would make the mapping between the input and the generated source code fuzzy.

In particular, consider transpiling source code written in a dynamically typed language, such as Python. This scenario exposes the limitations of type inference, as we are bound to the information available at compile time. As an example, consider the following Python function and its translation to Julia:

```
def sum_two(x, y):
    return x + y
```

The function `sum_two` receives two inputs, `x` and `y` that can have arbitrary types. The main problem lies in Python's `+` operator, which applies different operations depending on the runtime types of its operands, such as integer addition or string concatenations, among other possibilities.

A possible solution to disambiguate such cases is to request the programmer to annotate the Python source code using type hints to assist the translation process. In the previous case, annotating the `x` and `y` arguments using type hints would result in a more accurate translation.

Generally, it is a good practice to annotate the arguments and return types of functions, as this conveys the programmer's intentions of the source code. Furthermore, functions such as `sum_two` are too generic to be able to infer any type information. In such cases, the transpiler requires type-hints to correctly map the operations to Julia.

4 JULIA AND PYTHON

After discussing several aspects of transpilation, it is important that we introduce the two languages that will be the focus of our project.

Python was introduced more than 30 years ago. Throughout the years, its popularity has increased among the scientific community for providing an easy learning curve and an extensive library set.

The Python programming language has many alternative implementations. Two of them are Jython [12] and IronPython [13], where the first approach compiles Python source code to Java bytecode that runs on the JVM and the latter compiles Python source code to IL bytecode for the .NET platform. However, these implementations lack support for Python 3. CPython, Python's reference implementation, is written in the C programming language and has support for Python's latest version.

However, CPython suffers from slow performance on modern hardware due to Python's implicit dynamism. Programmers who require highly efficient code usually implement a prototype in Python and then convert the kernel parts to C. Furthermore, Python's high-performance libraries, such as NumPy[23], are also highly optimized libraries written in C that provide a speedup when compared to Python's native implementations. This is commonly referred to as the two-language problem, where the prototyping language differs from the main implementation language.

On the other hand, we have the recently introduced Julia programming language, which has been proving to be a high-performance alternative to Python, aiming at solving the two-language problem. Julia's simple syntax combined with high performance on modern hardware makes it a great alternative to Python.

Nowadays, two critical factors for the success of programming languages are the quality and quantity of available libraries. This problem was acknowledged in the context of Common Lisp [19], which, despite being a high-performance and flexible language, did not become popular due to its absence of libraries and the difficult mechanisms used to integrate them. Julia has a good library integration mechanism, which has incentivized the development of many third-party libraries. However, the available library set is still small, which is an issue we plan to address.

5 PYJL

The development of this project is based on an existing solution called Py2Many [29], which includes the PyJL transpiler. Py2Many provides a generic architecture and implements the necessary transpilation mechanisms to transpile Python to many C-like languages.

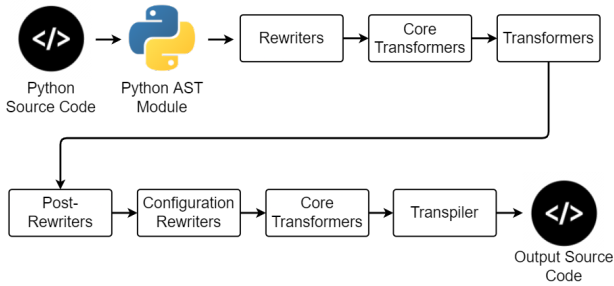


Figure 1: PyJL Architecture

PyJL builds upon that architecture and defines its transpiler implementation to translate Python to Julia. We opted to use Py2Many, since it has an active community updating it. Additionally, our preliminary analysis of the frameworks’ architecture shows that it is a good starting point to support this project.

Our implementation of PyJL [22] is still in its initial development stages and far away from our goal of automating the translation of Python libraries to Julia. The following section describes the current state of PyJL. We also analyze a performance scenario and detail our future plans for this project.

5.1 PyJL Architecture

In this section, we describe the stages of the transpilation pipeline used in PyJL. PyJL uses the same architecture as the Py2Many framework and adds the necessary functionality to transpile Python source code to Julia. The language-independent stages apply transformations common to all languages and are not extended by PyJL. The current pipeline can be seen in figure 1.

The input of this pipeline is Python source code that is parsed using Python’s `ast` module¹, which generates an Abstract Syntax Tree (AST). All the Phases in the Pipeline receive an AST as their input and use the visitor pattern to visit and modify nodes. We now describe each phase in the transpilation process:

- (1) *Rewriters* can be both language-specific or -independent and perform modifications in select nodes of the AST. A common use case of *Rewriters* is to change the structure of nodes to match the target language.
- (2) *Core Transformers* are language-independent transformers that modify the AST with relevant information for the translation process. The added information includes:
 - (a) Variable context: Adds all the variables to the node that represents their scope.
 - (b) Scope context: Adds a scope attribute to each node in the AST.
 - (c) Assignment context: Adds information to node assignments. An example is to annotate nodes that are on the left-hand side of an assignment, which is useful for operations that want to verify a node’s position in later phases.
 - (d) List call information: Adds all list transformation operations to the scope of the variable referencing the list.

- (e) Variable Mutability: Analyzes functions to detect mutable variables.
 - (f) Nesting levels: Annotates nodes with the respective nesting levels. This is important for languages sensitive to white spaces.
 - (g) Annotation flags: Differentiates type annotations and nested types
- (3) *Transformers* are language-specific and add complementary information to specific nodes of the AST. An example would be to add type information to nodes to help with type inference.
 - (4) *Post Rewriters* are rewriters that have dependencies on some previous phase. Their functionality is identical to that of *Rewriters*.
 - (5) *Configuration Rewriters* is an addition made to the Py2Many pipeline for PyJL, which supports configuration files in JSON and YAML format to modify the AST. This stage is language-specific.
 - (6) *Transpiler* translates language syntax and semantics and converts the AST to a string representation in the target language using the information provided by the previous phases. It is language-specific.

In the pipeline, the *Core Transformers* phase executes at two different stages. The first makes this information available for the stages that perform intermediary transformations. The second guarantees that no intermediary transformation overwrites the core functionalities of Py2Many and makes them available in the *Transpiler* phase.

After the pipeline has processed the Python source code, it generates the equivalent source code in the target language. The current implementation supports the transpilation of one or more files and performs the changes synchronously.

5.2 Code Annotations

The PyJL transpiler currently has a simple mechanism to allow programmers to specify code annotations separate from the Python source code. The goal is to support updates to the input source code while separately preserving the annotations that affect the transpilation process. This mechanism is integrated in the *Configuration Rewriters* phase of the Pipeline and reads YAML or JSON files that contain annotations, adding them to the corresponding AST nodes. This mechanism also supports the use of annotations in specific scopes, where a programmer can, for example, declare a decorator for a function within a specific class.

This is very beneficial when translating Python libraries to Julia. If a programmer annotates a Python library directly, this process will have to be repeated every time a new library version is released. By separating annotations from the source code, their application is ensured in subsequent translations.

An addition that is being considered is the integration of a Domain Specific Language (DSL) in PyJL, such as LARA [25], which was designed to be language-independent while offering more precise code annotations. Similar to the previous approach, the annotations are separate from the source code.

¹Abstract Syntax Tree - Python 3.10: <https://docs.python.org/3/library/ast.html> (Retrieved on January 27th, 2022)

5.3 Improving Compatibility

To maximize interoperability, programming languages commonly offer Foreign Function Interfaces (FFIs) to call externally developed modules or libraries. For instance, Julia's PyCall[15] provides an FFI to interoperate with Python. Tools such as PyonR [27] use Racket's FFI to call Python functions, and benchmarks reveal that it is a high-performance alternative to mapping Python's data model in Racket.

Despite FFI's presenting an alternative to translation, we argue that the translation process brings more benefits in the case of Julia. Using an FFI results in less maintainable source code, as external calls use a different language syntax. Furthermore, many of Python's highly-optimized libraries already have dedicated alternative implementations in Julia. Translated libraries would preserve Julia's syntax and benefit from Julia's performance on modern hardware.

Regarding the translation process, we found some cases that offer no direct mapping from Python to Julia. An example is the translation of Python's generator functions, which return a lazy iterator that implements the producer/consumer pattern. The producer generates a new value whenever `yield` is called and saves its execution state. When the consumer requests a value, the generator resumes its execution from the saved state. To demonstrate this use-case, we present an implementation of the Fibonacci sequence that returns an infinite iterator:

```
def fib():
    a = 0
    b = 1
    while True:
        yield a
        a, b = b, a + b
```

The producer/consumer pattern can be implemented in Julia using channels. The producer uses the `put!` function to add values to the channel while the consumer uses the `take!` function to retrieve them. We include a possible implementation below:

```
function fib()
    Channel() do ch
        a = 0
        b = 1
        while true
            put!(ch, a)
            a, b = b, a + b
        end
    end
end
```

Despite the syntactic similarities, there is an important difference. Even with the use of unbuffered Channels, the execution will only block at the first call to `put!`, allowing side effects in the producer to be executed before the consumer requests the first value.

A possible alternative that preserves Python's behavior is to use the third-party package `ResumableFunctions` [17]. This package defines a `@resumable` macro that is used to simulate the behavior of generator functions in Julia. A `@yield` macro is used to replace Python's `yield` keyword. Similar to Python, this implementation uses a Finite State Machine to save the execution state and resume it

in subsequent calls. An equivalent implementation of the Fibonacci sequence using this package is the following:

```
@resumable function fib()
    a = 0
    b = 1
    while true
        @yield a
        a, b = b, a + b
    end
end
```

Besides preserving Python's behavior, this approach also has the benefit of mapping more directly to its equivalent Python implementation, resulting in improved readability.

Another approach we found that helps the translation process is to use Julia's macro capabilities to map Python functionalities. We are experimenting with the development of a `dataclass` macro, that currently offers preliminary support for Python's `dataclass` decorator in Julia. However, we need to account for the fact that Julia's macros are expanded at macro-expansion time, which happens at compile time, while Python decorators operate dynamically at runtime. We are assessing the limitations of such implementations.

We considered both of these approaches in the development of our transpiler and found measurable improvements. Whenever possible, the transpiler should default to using Julia's native constructs. However, we recognize that translating Python's behavior may require the addition of new functionalities in Julia or the use of third-party packages. When the transpiler requires the use of these mechanisms to ensure correctness, it should always inform the programmer by producing a corresponding log message.

5.4 Object Mapping

Python allows programmers to use functional programming. However, it also supports the use of the OO Paradigm. Julia, on the other hand, is a mostly functional programming language, that does not fully support the OO paradigm. For the development of PyJL, we considered mapping Python's classes to Julia using native Julia constructs.

Translating Python's class model to Julia represents a tradeoff. A positive aspect is that we preserve the intended behavior of Python programs in Julia. However, this could potentially introduce a high overhead in computations, due to the added indirection of having class representations. We believe that a correct approach that also offers a choice to programmers is to support two alternative approaches:

- (1) The first uses Julia constructs to create a class hierarchy mechanism
- (2) The second relies on the use of a third party package called `Classes` [26].

The first approach converts classes into mutable structs, which have the corresponding fields of the class. It also creates abstract types for each class that are extended by each corresponding struct. The methods of each class are translated with a `self` field as their first parameter, which extends the abstract type mapped to the corresponding Python class.

The second approach uses the aforementioned `Classes` package. This package contains the `@class` macro, which defines a hierarchy

of abstract types and creates the necessary functions for each type, including a constructor function. This is a method of automating the previously introduced solution, with the benefit of having a simpler syntax.

To visualize both mechanisms, we provide a simple class inheritance example written in Python:

```
class Person:
    def __init__(self, name:str):
        self.name = name

    def get_id(self) -> str:
        return self.name

class Student(Person):
    def __init__(self, name:str, student_num:int):
        super().__init__(name)
        self.student_num = student_num

    def get_id(self) -> str:
        return f"{self.student_num} - {self.name}"
```

In this example, we define the `Person` and the `Student` classes. `Student` extends the `Person` class and adds the `student_num` field and a new definition of the `get_id` function. A possible translation to Julia using the first approach is the following:

```
abstract type AbstractPerson end
abstract type AbstractStudent <: AbstractPerson end

mutable struct Person <: AbstractPerson
    name::String
end
function get_id(self::AbstractPerson)
    return self.name
end

mutable struct Student <: AbstractStudent
    name::String
    student_num::Int
end
function get_id(self::AbstractStudent)
    return "$(self.student_num) - $(self.name)"
end
```

As was previously mentioned, this involves the creation of one abstract type for each Python class. In this case we have both the `AbstractPerson` and `AbstractStudent` abstract types which are inherited by the `Person` and `Student` structs respectively. The functions include a `self` parameter, which has the type of the corresponding abstract type. The argument types allow Julia's dispatch mechanism to select the correct function when performing calls.

The second approach uses `Classes.jl` to generate Julia source code that maps much more directly to Python. This package also uses abstract types to define its hierarchy, which are generated when using the `@class` macro. In the previous example, we chose the names of the abstract types to match the names of the generated abstract types used in the `Classes` package, to allow for an easier evaluation of the generated code. The following represents an equivalent translation using this package:

using Classes

```
@class mutable Person begin
    name::String
end
function get_id(self::AbstractPerson)
    return self.name
end

@class mutable Student <: Person begin
    student_num::Int
end
function get_id(self::AbstractStudent)
    return "$(self.student_num) - $(self.name)"
end
```

This approach offers a more direct mapping between the Python and the Julia source code. However, it still discloses some parts of the underlying Julia mechanism. For instance, notice how both `get_id` functions extend the types `AbstractPerson` and `AbstractStudent` to work in a class hierarchy. Still, it hides the creation of the abstract types and the creation of the structs to hold object fields, further blurring the lines between Python and Julia.

To choose between these two implementations, a programmer could use the provided annotation mechanism. The first approach would be the default implementation, as it does not require the use of a third-party library.

One important aspect that is not covered by both of these approaches is multiple inheritance. This would require implementing the C3 [2] algorithm in Julia to support Python's Method Resolution Order (MRO). For the first release of PyJL we are focused on supporting single inheritance, which already covers a large subset of Python implementations.

We also intend to cover the implementation of Python's special methods mentioned in section 2, such as `__init__` and `__str__`. These add necessary functionalities commonly used in Python. A possible solution is to extend the `Classes` package and create a new `PyClass` package that implements this functionality.

Furthermore, we also intend to map Python's default class field values. The current solution is to integrate the `Parameters` package [33], which defines a new constructor for each struct that includes default field values.

5.5 Mapping Dynamic Behavior

The mapping of operators from Python to Julia is frequently dependent on the types of its arguments. However, since Python is a dynamically typed language, this type information is only known at runtime.

A possible solution is to create new functions in Julia to simulate the behavior of Python operators. For instance, in the case of the `sum_two` function from section 3, we could map Python's `+` operator to a new `py_add` function in Julia. A similar approach was implemented in `PyonR` [27] to translate Python to Racket. Although this is a valid approach, the generated code would not preserve the pragmatics of Julia, which negatively affects maintainability.

An alternative solution is to use a type inference mechanism to help identify, at transpilation time, the most appropriate Julia

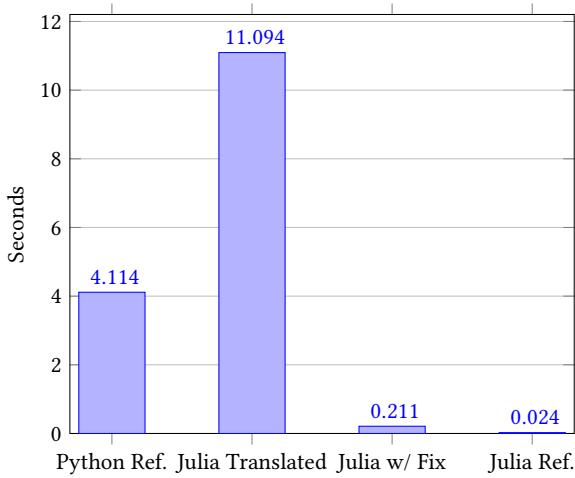


Figure 2: Performance of N-Body Implementations

operations. As these mechanisms are limited by the information available at compile time, it might be necessary to judiciously add type hints to the Python source code. This is also the case of MyPy [32], which requires type hints in function definitions to check for the soundness of the code.

A possible alternative is TYPPETE [11], which is an inference mechanism for Python based on the Z3 theorem prover [8] that uses a MaxSMT solver to define type constraints.

The approach that is currently being studied is the integration of the Pytype[10] type-inference mechanism. It generates separate .pyi stub files that contain type annotations. Optionally, we can merge these annotations with the original source code using the provided merge-py tool. This integrates newly added type information into the AST and makes the types available for the translation process.

5.6 Performance

To evaluate the current capabilities of the PyJL transpiler, an implementation of the N-Body Problem was translated to Julia. The chosen implementation predicts the gravitational interactions of planets in the solar system, and has both a Python [6] and a Julia [9] reference version. The results of the translation are publicly available². The benchmarks were executed on a machine with an Intel Core i7 4790K with 16GB of RAM under Windows 10. We compared the implementations with an input value of 500000 and chose an average of 10 runs for each test. Also, the obtained output generated by the program was verified to be identical in Julia and in Python.

The performance results shown in figure 2 reveal that the initial translation is not as high-performing as Python and is orders of magnitude slower than the reference Julia implementation.

After analyzing the generated source code, we discovered that the slowdown was caused by insufficient type information. The

²Transpiled N-Body Problem: https://github.com/MiguelMarcelino/translated_n_body_problem

Python function that resulted in the generation of generic Julia source code is shown below.

```
def combinations(l):
    result = []
    for x in range(len(l) - 1):
        ls = l[x+1:]
        for y in ls:
            result.append((l[x],y))
    return result
```

This function receives a list as its argument and generates combinations using the elements of that list. It then adds those combinations to a new list, which is returned by the function. The translation performed by the transpiler was the following:

```
function combinations(l)::Vector
    result = []
    for x in (0:length(l)-1-1)
        ls = l[(x+1+1):end]
        for y in ls
            push!(result, (l[x+1], y))
        end
    end
    return result
end
```

Note that the generated Julia function returns a generic array. In this case, we cannot infer the type of the returned array, as it is impossible to guarantee that the input list will always have the same type. This forces the translation to use generic containers that have considerable overheads when compared to type-specific ones. The result produced by the combinations function is not type stable, which impacts the performance of the generated source code.

After manually modifying one line of code by specifying the necessary type information, we obtained a speedup of 52.6×, making the translated Julia code 19.5× faster than the original Python code. This result can be achieved in one of two ways. We can either annotate the result array with its corresponding type:

```
result::Vector{Tuple{Tuple{Vector{Float64},
    Vector{Float64},
    Float64},
    Tuple{Vector{Float64},
    Vector{Float64},
    Float64}}} = []
```

or convert the result array, changing the last line of the function to the following:

```
return typeof(result[1])[result...]
```

Regarding the reference Julia implementation, it is relevant to mention that it is highly optimized and takes advantage of Julia's performance characteristics.

5.7 Code Maintainability

The current status of PyJL does not allow us to make many claims on code maintainability. The generated code for the N-Body problem preserves Python's code structure and pragmatics. However, this example maps almost directly to Julia, only requiring minor syntax changes. More complex Python examples that use native Python

constructs with no direct translation to Julia or use Python's classes are required to analyze code readability.

We are also evaluating how the use of third party packages affects the readability of the generated source code. So far, they have shown measurable improvements and offer a better mapping between Python and Julia source code.

A proper evaluation would require user tests to determine if the generated code is intelligible by programmers. We are considering this evaluation method to assert that code generated by the PyJL transpiler is similar to human-written code.

6 EVALUATION

In the context of program translation, it is important to assess the limitations of transpilers, which will be covered in this section.

Throughout this work, we have acknowledged that there are translation cases that reach the limitations of type inference. We have previously shown two examples, the `sum_two` function in section 3 and the `combinations` function in section 5.6, where the lack of type information results in the generation of generic code. Attempting to infer types in these situations might be possible but only if bounded to a given scope, which does not guarantee overall correctness.

Another problem that we encountered was related to the reliability of type information, where some Python programs include type hints that do not match the correct attribute or variable types. One could use `Pytype` [10], the proposed inference tool, to perform these checks or even enforce the type annotations provided by programmers.

Regarding the mapping of Python's classes to Julia, it is important to note that translating Python's OO behavior to Julia will always inherit Julia's mechanisms. We are still relying on multiple dispatch to relate methods to object types, which implies that there is weaker coupling to objects in Julia. In the case of the translations shown in section 5.4, the Julia methods are only bound to the self argument that represents the equivalent Python Class.

7 FUTURE WORK

The PyJL transpiler is still a work in progress and far from our goal of converting Python libraries to Julia. In this section, we discuss the plans for the transpiler.

Regarding the mapping of Python's dynamic behavior to Julia, the integration of `Pytype`[10] should make the transpiler less dependent on type hints and help evaluate their soundness. Nonetheless, it is expected that type hints will still be necessary in function definitions due to their ambiguity.

The transpiler should also use Julia's functionalities to enhance the generated Julia source code. We have previously mentioned the creation of macros, which would result in the generation of more maintainable code. However, since macros are executed at compile-time, and due to Python's dynamism, this might only be achievable in some cases.

The performance of the generated source code is another facet of the translation process that can be optimized. Performing code optimizations is a topic which is more targeted at software restructuring tools, usually employed in software maintenance. These have

the ability to change the structure of a given program without modifying its behavior [1]. The generated source code would probably benefit mostly from perfective maintenance, an approach which focuses on improving program performance or maintainability [14].

A transpiler developed for code translation can have mechanisms that account for code restructuring. The programmer could use the annotation mechanism discussed in section 5.2 to annotate code segments to restructure. The transpiler could then apply the intended code transformations during the translation process.

The restructuring process could result in improved code performance. High performance in Julia is achievable through proper techniques. We present some that were considered:

- *Separating Kernel functions*: Separate source code into different functions to allow the compiler to generate type-specific code [4].
- *Devectorizing expressions*: In Julia, loops are very well optimized, making them as fast as loops written in C. We are currently analyzing the `Devectorize` [20] package used to devectorize expressions in favor of using loops.
- *Improving Cache hit rate*: Optimize the transpiler to rewrite loops over matrices in column major order to achieve even higher performance [4].

8 RELATED WORK

In the area of source-to-source translation, many transpilers have already used Python as their source language. `PyonR` [27] explores the use of two complementary solutions to use Python functionalities in Racket: (1) using an FFI to call Python's C functions, (2) translating Python's data model to Racket and use the Racket execution environment. In terms of performance, calls made to the FFI ended up taking a very similar time when compared to the calls made by Python to its C API. The implemented data model also managed good performance, sometimes outperforming the equivalent CPython implementations.

Additionally, some approaches that target the improvement of Python's performance. One of these [21] explores the use of Rust as an intermediary, high-performance, and high-level language to represent Python source code. The `PyRS` [16] transpiler, which is now also part of `Py2Many`, is used to translate Python to Rust. It is an experimental transpiler that requires manual intervention in some cases to generate running Rust source code. The Rust intermediary code can then be translated to a lower-level optimized target. The performance evaluation of the transpiled code shows that the transpiled Rust source code achieves better performance while using less memory than Python.

The two-way Fortran-Python transpiler [5] also aims at improving Python's performance by using Fortran as a high-performance target language. It offers two solutions, where the programmer can either transpile Fortran code to Python and improve its performance or improve the performance of an already existing Python program. The programmer annotates the kernel functions in Python with a decorator, which are translated back to Fortran at runtime to benefit from a high-performance execution environment. The performance results are similar to those obtained when manually translating Python to Fortran.

The context of library translation has also been discussed in the development of Jnil [19], that transpiles Java to Common Lisp. This approach also studied the challenges of preserving language pragmatics in automatic translation, an important aspect for the development of PyJL.

9 CONCLUSION

Throughout the years, transpilers have evolved to generate source code that is not only human-readable, but also hard to distinguish from human-written programs, which has allowed transpilers to become alternatives to manual translation.

This work extends the PyJL transpiler to convert Python libraries to human-readable and modifiable Julia source code. The process of automating the translation represents a challenge, but it can be achieved with high-levels of reliability if enough information is provided in the Python source code. The generated code should also respect the pragmatics of Julia.

We expect that PyJL further decreases the library gap between Python and Julia, speeding up library development. The conversion subset should cover widely used Python features, such as the aforementioned `yield` construct or the `dataclass` decorator. Python's Data Model should also be mapped to an equivalent Julia Data Model. This includes mapping Python's classes to Julia, with support for single inheritance.

Furthermore, this research also aims at improving the performance of the translated libraries. Preliminary results show that Julia's compilation strategy can lead to huge performance increases when some type hints are judiciously added to the generated code.

ACKNOWLEDGEMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) (references PTDC/ART-DAQ/31061/2017 and UIDB/50021/2020).

REFERENCES

- [1] R.S. Arnold. Software restructuring. *Proceedings of the IEEE*, 77(4):607–617, 1989. doi: 10.1109/5.24146.
- [2] Kim Barrett, Bob Cassels, Paul Haahr, David A. Moon, Keith Playford, and P. Tucker Withington. A monotonic superclass linearization for dylan. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, page 69–82, New York, NY, USA, 1996. Association for Computing Machinery. ISBN 089791788X. doi: 10.1145/236337.236343. URL <https://doi.org/10.1145/236337.236343>.
- [3] Ira D. Baxter, Christopher Pidgeon, and Michael Mehlich. Dms: Program transformations for practical scalable software evolution. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 625–634, Edinburgh International Conference Centre, Scotland, UK, 2004. Semantic Designs.
- [4] Jeff Bezanson, Stefan Karpinski, Viral Shah, Alan Edelman, et al. Julia Language Documentation - Performance Tips, 2014. Chapter 3, p.279-299.
- [5] Mateusz Bysiek, Aleksandr Drozd, and Satoshi Matsuo. Migrating Legacy Fortran to Python While Retaining Fortran-Level Performance through Transpilation and Type Hints. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*, pages 9–18, Salt Lake City, UT, USA, 2016. IEEE. doi: 10.1109/PyHPC.2016.006.
- [6] Kevin Carson, Fredrik Johansson, Tupteq, and Daniel Nanz. N-Body Problem, Python Implementation, Mar 2021. Retrieved January 4th, 2022 from: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/nbody-python3-1.html>.
- [7] Intel Corporation. Mcs-86 assembly language converter operating instructions for isis-ii users, 1979.
- [8] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.
- [9] Andrei Fomiga, Stefan Karpinski, Viral B. Shah, Jeff Bezanson, smallnamespaces, Adam Beckmeyer, and Vincent Yu. N-body problem, julia implementation, Apr 2021. Retrieved January 4th, 2022 from: <https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/nbody-julia-8.html>.
- [10] Google. Pytype: A static type analyzer for python code, March 2015. [Online. Retrieved February 25th, 2022 from: <https://github.com/google/pytype>].
- [11] Mostafa Hassan, Caterina Urban, Marco Eilers, and Peter Müller. Maxsmt-based type inference for python 3, 2018.
- [12] Jim Hugunin. Python and java: The best of both worlds. In *Proceedings of the 6th international Python conference*, volume 9, pages 2–18, Reston, VA, 1997. Citeseer.
- [13] Jim Hugunin. IronPython Home Page, 2013. [Online. Retrieved January 6th, 2022 from: <https://ironpython.net/>].
- [14] ISO/IEC/IEEE. International Standard for Software Engineering - Software Life Cycle Processes - Maintenance, 2006.
- [15] JuliaPy. PyCall - Calling Python functions from the Julia language, February 2013. [Online. Retrieved February 25th, 2022 from: <https://github.com/JuliaPy/PyCall.jl>].
- [16] Julian Konchunas. PyRS - A Python to Rust Transpiler, 2015. [Online. Retrieved January 18th, 2022 from: <https://github.com/konchunas/pyrs>].
- [17] Ben Lauwens. ResumableFunctions.jl, August 2017. Retrieved on January 29th, 2022 from: <https://github.com/BenLauwens/ResumableFunctions.jl>.
- [18] Antonio Menezes Leitao. Migration of common lisp programs to the java platform -the linj approach. In *11th European Conference on Software Maintenance and Reengineering (CSMR'07)*, pages 243–251, Amsterdam, Netherlands, 2007. IEEE. doi: 10.1109/CSMR.2007.34.
- [19] António Menezes Leitão. The next 700 programming libraries. In *Proceedings of the 2007 International Lisp Conference*, ILC '07, New York, NY, USA, 2007. Association for Computing Machinery. ISBN 9781595936189. doi: 10.1145/1622123.1622147.
- [20] Dahua Lin. Devectorize.jl, 2015. Retrieved January 2nd, 2022 from: <https://github.com/lindahua/Devectorize.jl>.
- [21] Henri Lunnikivi, Kai Jylkkä, and Timo Hämäläinen. Transpiling Python to Rust for Optimized Performance. In Alex Orailoglu, Matthias Jung, and Marc Reichenbach, editors, *Embedded Computer Systems: Architectures, Modeling, and Simulation*, pages 127–138, Cham, 2020. Springer International Publishing. ISBN 978-3-030-60939-9.
- [22] Miguel Marcelino and António Menezes Leitão. Pyjl implementation, 2021. Retrieved March 5th, 2022 from: <https://github.com/MiguelMarcelino/py2many>.
- [23] Travis Oliphant. NumPy, 2009. [Online. Retrieved November 18th, 2021 from: <https://numpy.org/>].
- [24] Renaud Pawlak. Jsweet: insights on motivations and design a transpiler from java to javascript, 2015.
- [25] Pedro Pinto, Tiago Carvalho, João Bispo, and João M. P. Cardoso. LARA as a Language-Independent Aspect-Oriented Programming Approach. In *Proceedings of the Symposium on Applied Computing*, SAC '17, page 1623–1630, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344869. doi: 10.1145/3019612.3019749. URL <https://doi.org/10.1145/3019612.3019749>.
- [26] Richard Plevin. Classes.jl: A simple, Julian approach to inheritance of structure and methods, November 2021. Retrieved November 22nd, 2021 from: <https://github.com/rjplevin/Classes.jl>.
- [27] Pedro Henriques Palma Ramos and António Menezes Leitão. PyonR: A Python Implementation for Racket. Master's thesis, Instituto Superior Técnico, 2014.
- [28] Maurizio Rebaudengo, Matteo Sonza Reorda, Massimo Violante, and Marco Torchiano. A source-to-source compiler for generating dependable software. In *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, pages 33–42, Florence, Italy, 2001. IEEE. doi: 10.1109/SCAM.2001.972664.
- [29] Arun Sharma, Lukas Martinelli, Julian Konchunas, and John Vandenberg. Py2many: Python to many clike languages transpiler, 2015. Retrieved November 18th, 2021 from: <https://github.com/adsharma/py2many>.
- [30] Roger Taylor and Phil Lemmons. Upward migration part 1: Translators using translation programs to move cp/m-86 programs to cp/m and ms-dos, 1982.
- [31] Tijs van der Storm. Nomen: A Dynamically Typed OO Programming Language, Transpiled to Java, 2016.
- [32] Guido van Rossum, Jukka Lehtosalo, Ivan Levkivskiy, and Michael J. Sullivan. Mypy, 2014. [Online. Retrieved February 29th, 2022 from: <http://mypy-lang.org/>].
- [33] Mauro Werder. Parameters.jl, 2015. [Online. Retrieved February 30th, 2022 from: <https://github.com/mauro3/Parameters.jl>].

ETAP: Experimental Typesetting Algorithms Platform

Didier Verna

EPITA

Research and Development Laboratory

Le Kremlin-Bicêtre, France

didier@lrde.epita.fr

ABSTRACT

We present the early development stages of ETAP, a platform for experimenting with typesetting algorithms. The purpose of this platform is twofold: while its primary objective is to provide building blocks for quickly and easily designing and testing new algorithms (or variations on existing ones), it can also be used as an interactive, real time demonstrator for many features of digital typography, such as kerning, hyphenation, or ligaturing.

CCS CONCEPTS

• **Software and its engineering** → **Application specific development environments**; • **Human-centered computing** → **Heuristic evaluations**; **Information visualization**; • **Applied computing** → *Document preparation*.

KEYWORDS

Typesetting, Paragraph Formatting, Real-Time Interactive Experimentation

ACM Reference Format:

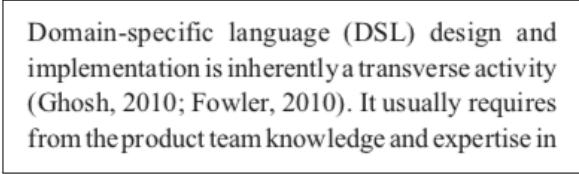
Didier Verna. 2022. ETAP: Experimental Typesetting Algorithms Platform. In *Proceedings of the 15th European Lisp Symposium (ELS'22)*. ACM, New York, NY, USA, 5 pages. <https://doi.org/10.5281/zenodo.6334248>

1 INTRODUCTION

The world of digital typography is a fascinating one. As an application domain, it combines a strong focus on aesthetics with many complicated technical challenges. On the outside, the concern for aesthetics is everywhere: from the shape of characters and the space between them, to the overall balance of lines, paragraphs, pages, complete documents. On the inside, any kind of formatting algorithm (for example, a paragraph justification one) risks exponential complexity as soon as *some* level of quality is expected.

Defining the notion of (good) typesetting quality is a very complicated and subtle problem, and is out of the scope of this paper. On the other hand, bad typesetting is easily perceived, and hence, rather easy to demonstrate, as it impacts readability and involves such notions as aesthetic disturbance.

Figure 1 exhibits the first four lines of a badly justified paragraph. Notice for example how different the inter-word spacing is between lines 1 (very large) and 4 (very small). Notice also that line 4 is so



Domain-specific language (DSL) design and implementation is inherently a transverse activity (Ghosh, 2010; Fowler, 2010). It usually requires from the product team knowledge and expertise in

Figure 1: Example of a badly justified paragraph

compressed that it becomes very difficult to separate the words from each other. Finally, within the particular context of the highly compressed fourth line, where the inter-word spacing becomes close to the inter-letter one, the word “knowledge” almost appears written in two words: “know ledge”. Even when the casual reader is unaware of all these problems, reading badly typeset documents results in fatigue.

This project originates in two connected, yet slightly different motivations. The first one is a general interest for the world of digital typography, and the will to “play” and experiment with typesetting algorithms (also in order to get a better understanding of how they work and what they actually do). The second one, more pragmatic, is the need to strengthen an existing lecture on typesetting, with striking illustrations for the various aesthetic challenges that the discipline faces. These two motivations have something in common: they both require a system which must be as real-time and interactive as possible. Suppose you are trying out a new paragraph justification algorithm. Rapid prototyping and experimentation would be made a lot easier with a direct visualization of the results on a sample text (actual contents not so important), and with the ability to interactively tweak such or such parameter from a Graphical User Interface (GUI), while observing the effects in real-time. In a similar vein, in order to illustrate the importance of, say, kerning (inter-letter spacing adjustment), there is nothing like having the ability to visualize a sample text (again, actual contents not so important), and turn kerning on or off by the click of a button.

In general, typesetting experimentation or demonstration is not a very practical thing to do. What You See Is What You Get (WYSIWYG) systems such as Word or Libre Office are usually quite reactive, but their algorithms are not necessarily of the highest quality, and neither are they easily configurable or extensible, let alone replaceable. On the other hand, \TeX [6, 7], the obvious competitor, and still a *de facto* standard in terms of typographic quality and customizability, is not a very interactive system. It works more like a compiled programming language with separate development, compilation, and visualization phases. There was one attempt at providing a GUI for controlling typesetting parameters[2], but it

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

ELS'22, March 21–22 2022, Porto, Portugal

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-2-9557474-6-9.

<https://doi.org/10.5281/zenodo.6334248>

was limited to global ones and didn't go very far. Many projects attempt to mitigate this by providing more or less interactive and real-time WYSIWYG layers on top of it. However, \TeX itself certainly doesn't make it easy to tweak or replace any of its internal typesetting algorithmic components. In fact, the "spaghetti code effect" is a well-known characteristic among the community of \TeX hackers.

Whatever the approach, all these systems have one thing in common: they are *production* systems. They target the feature-bloated generation of complete, actual documents. Interactive and real-time experimentation, testing, rapid prototyping, or demonstration is a different goal, and it is the niche that ETAP tries to occupy. It is not meant to become a complete typesetting system, although it could very well turn out to be a Petri dish for one [11, 12]. Rather, it attempts to provide low-level data structures and building blocks for experimenting with new typesetting algorithms (or variations on existing ones), with interactive and real-time parametrization and visualization, and hopefully in a near future, quality assertion and analysis (for some definition of "quality").

Section 2 describes the project and its current features. Section 3 gives a brief overview of the underlying implementation. Finally, Section 4 concludes and Section 5 details some general directions for future work.

2 CURRENT FEATURES

Figure 2 provides a screenshot of ETAP's GUI, which is currently implemented in LispWorks¹ CAPI². The platform currently focuses on paragraph formatting algorithms. The interface can be described as having four main areas.

2.1 Area 1: Text Editor

Area 1 is a simple text editor (a CAPI editor-pane) which lets you adjust the textual contents of the typeset paragraph. Any change in the text is automatically and continuously propagated to the paragraph view (area 4).

2.2 Area 2: Global Options and Features

Area 2 provides control over some global options, features, and visual clues, also tracked continuously and in real-time.

The paragraph disposition pane lets one choose between justification and various ragged formatting. Note that some combinations of algorithm / disposition don't actually make much sense, but still, these parameters are considered sufficiently orthogonal to be separate in the GUI.

The features pane lets one toggle kerning (inter-letter spacing), ligatures (character fusion, e.g. ff for ff), and hyphenation (word splitting) on or off. Kerning and ligature information is provided by the font in use. The hyphenation implementation is that of \TeX , itself based on Liang's thesis[9]. The language (hence the hyphenation patterns set) is currently hard-coded to English.

The clues pane allows one to choose what is actually displayed in the paragraph view (area 4): the characters themselves, but also different kinds of bounding boxes, plus the hyphenation points, and the underfull / overfull boxes. In Figure 2, the paragraph view

¹<http://www.lispworks.com/>

²<http://www.lispworks.com/products/capi.html>

exhibits individual characters, hyphenation points, and underfull boxes.

Finally, there is a slider to set the desired paragraph width.

2.3 Area 3: Algorithms

This is where you select a specific paragraph formatting algorithm. Depending on the chosen one, this area also displays algorithm-dependent variants, options, or other adjustable parameters. Again, any choice of algorithm or any modification of its parametrization is automatically and continuously reflected in the final paragraph view.

A complete description of the currently available algorithms is out of the scope of this paper, but each one is implemented in its own file, and there is always an explanatory comment at the top. Here, we only provide a quick overview of the five algorithms currently implemented.

2.3.1 Fixed. The "fixed" algorithm uses only the *natural*, constant, inter-word spacing (a value provided by the font in use). Hence, it can practically never justify properly. Lines are created sequentially, without look-ahead or backtracking: there are no paragraph-wide considerations.

2.3.2 Fit. As the name suggests, this is an implementation of the so-called First, Best, and Last Fit classical algorithms. Those ones make full use of elastic inter-word spacing ("glue" in \TeX terms) when attempting to justify lines. The acceptable range of inter-word spacing is also an information provided by the font in use. By nature of the *Fit algorithms, lines are also created sequentially here, without look-ahead or backtracking: there are no paragraph-wide considerations.

2.3.3 Barnett. This one is an implementation of a justification algorithm from Michael Barnett[1], originally published in 1965. In short, this algorithm behaves more or less as a combination of different *Fit policies, while favoring overfull lines when no perfect solution is found.

2.3.4 Duncan. This one is an implementation of a justification algorithm from C. J. Duncan [5], originally published in 1963. In short, this algorithm searches for an acceptable breaking solution while minimizing hyphenation.

2.3.5 Knuth-Plass. Finally, the fifth and last one is the \TeX one, *a.k.a.* the famous "Knuth-Plass" algorithm[8]. This is the one visible in Figure 2, and you can see that it has \TeX 's full set of adjustable parameters available.

2.4 Area 4: Paragraph View

Area 4 is where the typeset paragraph is eventually rendered, depending on the selected algorithm and options, and along with the various visual clues selected in area 2. In the screenshot from Figure 2, the orange triangles indicate the hyphenation points, and the rectangles at the end of lines 2 and 9 denote the underfull lines (that is, the lines that are too short to be justified). Overfull lines would be indicated, as in \TeX , by the same rectangles, only filled in with orange.

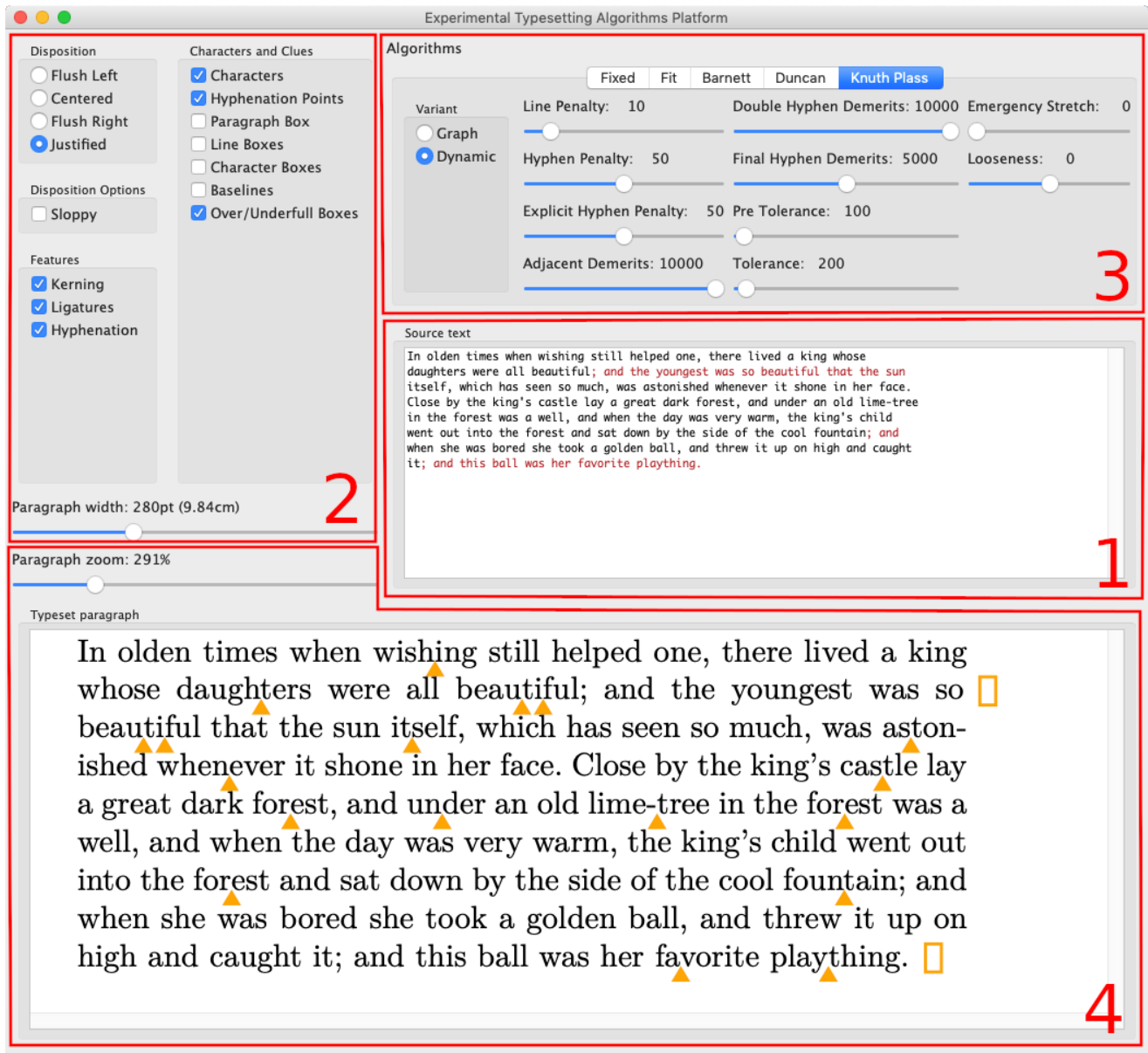


Figure 2: ETAP’s graphical user interface

There is a slider for zooming in / out the view. Note that the zooming facility is the only GUI component that doesn’t retrigger the typesetting engine. It operates at the window level.

The paragraph is currently rendered with a hard-wired font: Latin Modern Roman 10pt, Extended Cork encoding. The font and encoding descriptions have been copied from a MacTeX (TeXlive) distribution³. In particular, the font information (notably including kerning and ligatures) is read from its original TeX Font Metrics

(TFM) file thanks to a library also developed by the author of this paper⁴.

3 IMPLEMENTATION

In this section, we provide a brief overview of ETAP’s implementation. The design of the internals is heavily inspired from that of TeX, but it also fits a different set of objectives, most importantly being an experimentation platform rather than a production system.

³<https://www.tug.org/mactex/>

⁴<https://github.com/didierverna/tfm>

3.1 Basic Data Structures

ETAP provides five basic data structures. Characters are directly represented by their corresponding `character-metrics` structure from the TFM library. Then, there are classes for kerns (fixed, possibly negative, space between characters), and break points (discretionaries and glues). Discretionaries and glues follow \TeX 's jargon and design.

A *discretionary* represents a potential break point with different material to typeset, depending on whether the break actually occurs or not. For example, the sequence of characters `ffi` is reified as a discretionary specifying that without a break, the ligature `ffi` may be used, and in case of breaking the line, the first line ends with `f-` and the next one begins with `f i` (or the ligature `fi`). A simple hyphenation point simply states that in case of breaking a line, the first one ends with a dash, and nothing else happens otherwise.

A *glue* represents an elastic space between words, with a natural width, plus specific amounts of shrinkability and stretchability (again, those values are provided by the font in use).

3.2 The Lineup

The paragraph text retrieved from Area 1 of the GUI is processed into a so-called *lineup*. A lineup is essentially a vector of objects to typeset. The paragraph text is trimmed from consecutive blanks. It is then sliced into words, possibly hyphenated (in which case discretionaries are added). After that, ligatures are handled if requested (which may lead to the creation of new discretionaries, or the modification of existing ones). Kerns are then inserted at the appropriate places, again, if requested. Finally, an infinitely stretchable glue is appended at the end of the lineup.

3.3 The Lines

Each algorithm's entry point is implemented as a method on a generic function called `create-lines`. The algorithms receive a lineup, a paragraph width, a disposition, and set of algorithm-specific options specified in the GUI. They compute their own view on where exactly the lineup should be broken into lines, and they return the lines in question.

A *line* is essentially a sequence of characters, each one with a specific horizontal placement with respect to the beginning of the line. This placement is computed out of the characters widths, the kerns, and the glue present in the lineup, and of course, the desired line's length. Characters placed at a specific horizontal position are called *pinned characters*.

3.4 The Paragraph

Finally, the resulting *paragraph* is created and passed to the GUI for rendering. There is in fact not much left to do to generate it. Each line computed by the selected algorithm is positioned both horizontally and vertically, relative to the paragraph's top-left corner. Such a fully placed line is called a *pinned line*. The horizontal position of each line depends on the selected paragraph disposition (centered, flushed, or justified). Vertically, the lines are simply spaced by a currently hard-wired constant (the "line skip" in \TeX 's jargon).

4 CONCLUSION

ETAP is currently in an "early prototype" development state⁵. The internals are not stabilized, there is no decent documentation, the code has *not* been carefully crafted, and no concern for optimization or general performance has entered the picture yet.

Despite all this, the project already works surprisingly well. The GUI runs very smoothly in real-time, and it has been used successfully several times already to support lectures or conferences on typesetting. The observable reactions in the audience, facing the real-time effects of kerning, hyphenation, or ligaturing, for example, is a testimony to the pertinence of this approach for increasing the general awareness of the technical challenges involved in digital typography.

One of the most important advantages in using Common Lisp [10] for this project is the ease of development and the concision of the resulting code. The program (excluding the TFM library and a large font description file) is currently just under 3000 lines of code. The GUI code and the typesetting building blocks take around 25% of that each, and the other half of the code is devoted to the algorithms implementations. The Knuth-Plass algorithm itself, for which we actually provide two different implementations (see Section 5.2), takes less than 500 lines (granted, the whole of \TeX isn't there obviously; user-level macros, mathematics, *etc.*).

5 FUTURE WORK

In addition to improving the general state of the project (essentially meaning stabilizing the internals and providing accurate and up to date documentation), we currently envision two major directions for future work.

5.1 Direction 1: Experimentation

One of the very first, and already achieved goal of this project was to make it easy to experiment with typesetting algorithms, by either creating new ones, extending or modifying existing ones, and quickly visualizing the results. In a near future, we intend to use ETAP to do research on known typesetting problems such as rivers detection, or experiment with new features or extensions, notably to the Knuth-Plass algorithm. Some people, for instance, prefer different kinds of placement for end-of-line hyphens in justified paragraphs.

5.2 Direction 2: Analysis

Because typography is not a technical question only, but also an aesthetic one, a very difficult problem, when experimenting with typesetting algorithms, is how to evaluate the quality of the results. Of course, the ability to directly visualize a typeset paragraph, as in this project, is a tremendous help, but it is surely not enough.

In fact, we can come up with mathematical formulas representing some measure of typesetting quality (for example, taking into account the amount of stretching or shrinking of lines, compared to their natural width), and this is in fact precisely what the Knuth-Plass algorithm attempts to optimize, paragraph-wide (the so-called *badness*).

⁵<https://github.com/didierverna/etap>

With a platform such as ETAP, it becomes very easy to instrument the underlying data structures to keep track of quality measurement (badness, demerits from \TeX , or anything else one may think of), and perform statistical analysis afterward.

Here lies the second most important advantage in using Common Lisp for this project. Its interactive nature makes it effortless to bypass the GUI altogether, and run the typesetting algorithms, without visualization, from the Read-Eval-Print Loop (REPL) or through a batch script.

In a near future, it is hence also our intention to collect large empirical data on the quality of typesetting (for example, using \TeX 's notion of quality) and perform statistical and comparative analysis between different algorithms, or algorithm implementations. For example, we can easily run the five algorithms currently implemented on the same paragraph, for many different widths, and compare the resulting data. The original Knuth-Plass algorithm uses a *dynamic programming*[3, 4] optimization technique for cutting through the (potentially very large) graph of break possibilities. ETAP already provides an unoptimized (and slow) variant implementation of it, working on the full graph. It would be interesting to collect statistical data from both these implementations, in order to get a concrete idea of the impact of \TeX 's optimization on the actual quality of the typesetting.

REFERENCES

- [1] Michael P. Barnett. *Computer Typesetting: Experiments and Prospects*. MIT Press, January 2000.
- [2] Kaveh Bazargan. Batch commander: a graphical user interface for \TeX . *TUGboat*, 26(1):74–80, 2005.
- [3] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–516, 1954. doi: 10.1090/S0002-9904-1954-09848-8.
- [4] Richard Bellman. *Dynamic Programming*. Princeton University Press, 2003.
- [5] C.J. Duncan, J. Eve, L. Molyneux, E.S. Page, and M.G. Robson. Computer typesetting: an evaluation of the problems. *Printing Technology*, 7:133–151, 1963.
- [6] Donald E. Knuth. *The \TeX book*. Addison-Wesley, 1984.
- [7] Donald E. Knuth. *\TeX : the Program*, volume B of *Computers and Typesetting*. Addison-Wesley, January 1986.
- [8] Donald E. Knuth and Michael F. Plass. Breaking paragraphs into lines. *Software: Practice and Experience*, 11(11):1119–1184, 1981. doi: 10.1002/sp.4380111102.
- [9] Franklin Mark Liang. *Word Hy-Phen-a-Tion by Com-Put-Er*. PhD thesis, Stanford, CA, USA, 1983.
- [10] ANSI. American National Standard: Programming Language – Common Lisp. ANSI X3.226:1994 (R1999), 1994.
- [11] Didier Verna. Star \TeX : the next generation. In Barbara Beeton and Karl Berry, editors, *TUGboat*, volume 33. \TeX Users Group, 2012.
- [12] Didier Verna. TiCL: the prototype (Star \TeX : the next generation, season 2). In Barbara Beeton and Karl Berry, editors, *TUGboat*, volume 34. \TeX Users Group, 2013.

An Ontology-Based Dialogue Management Framework for Virtual Personal Assistants in Common Lisp

Michael Wessel
michael.wessel@sri.com
SRI International
Menlo Park, California, USA

ABSTRACT

We present a new approach to *dialogue specification for Virtual Personal Assistants (VPAs)* based on so-called *dialogue workflow graphs*. Our approach relies on Semantic Web technology (OWL), implemented in Common Lisp with the help of the Racer reasoner. Our new *dialogue specification language (DSL)* is a set of Common Lisp macros, a Domain Specific Language, which facilitates customer participation in the modeling process. The resulting dialogue models are also very concise. The DSL is a new modeling layer on top of our *ontology-based Dialogue Management (DM) framework OntoVPA*. We explain the rationale and benefits behind the new language, and support our claims with concrete reduced Level-of-Effort (LOE) numbers from two recent OntoVPA projects.

CCS CONCEPTS

• **Software and its engineering** → **Model-driven software engineering**; **Very high level languages**; **Domain specific languages**; **State based definitions**; **Application specific development environments**; **Software design tradeoffs**; **Rapid application development**; • **Information systems** → **Web Ontology Language (OWL)**; **Ontologies**; **Business intelligence**; • **Human-centered computing** → **Participatory design**; • **Computing methodologies** → **Discourse, dialogue and pragmatics**; **Description logics**; **Ontology engineering**.

KEYWORDS

Knowledge-Based Dialogue Modeling, Knowledge-Based Workflow Modeling, Ontology-Based Dialogue Management, Semantic Natural Language Processing, Domain Specific Modeling Languages, Semantic Web, OWL, SPARQL, Common Lisp, Macros

ACM Reference Format:

Michael Wessel. 2022. An Ontology-Based Dialogue Management Framework for Virtual Personal Assistants in Common Lisp. In *Proceedings of the 15th European Lisp Symposium (ELS'22)*. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.5281/zenodo.6335631>

1 INTRODUCTION & MOTIVATION

In 2021, *Virtual Personal Assistants (VPAs)* have become commonplace on our smartphones and smart speakers.

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

ELS'22, March 21–22, 2022, Porto, Portugal

© 2022 Copyright held by the owner/author(s).

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

<https://doi.org/10.5281/zenodo.6335631>

The VPA landscape ranges from simple “one-shot request-response” systems to control appliances and lights in an IoT home automation network, over more sophisticated *task-oriented virtual specialists* à la KASISTO [14] that help with complex tasks such as online banking, to freeform conversational chatbots that aim at passing the Turing Test.

With growing demands and user’s expectations regarding the “intelligence” and “human-likeness” of VPAs, there is clearly a need for DM systems that go beyond the simple “one-shot request-response” model.

A recent Gartner blogpost reads as follows [8]:

Gartner predicts that by 2025, 50 % of knowledge workers will use a virtual assistant on a daily basis, up from 2 % in 2019.

In 2022, users expect systems that are capable of supporting a) elements of freeform chat with contextual memory (e.g., learn about the user and current situation), b) are capable of supporting complex workflows and multiple turns, and c) can combine both in a non-rigid, non-linear, naturally flowing conversation that doesn’t feel like a telephone hotline support system. We have embraced and addressed some of these challenges in our *OntoVPA framework* [33, 35].

Most VPAs rely on one central component – the Dialogue Manager. The Dialogue Manager is typically concerned with *Dialogue Management (DM)*, which encompasses *Dialogue State Tracking* [37], and it also implements the *Dialogue Policy*: the computation of the system response (“system turn”) based on the current state of the world and dialogue / discourse (including the previous and current “user turns”).

A commonly used *conceptual DM model* is the *Dialogue Flow Graph*. We are referring to this graph as a *conceptual model* because the actual dialogue model representation might be different from a graph (e.g., in case of a rule-based manager). Still, the flow graph is a useful conceptual notion of high utility for developers and customers, given its comprehensibility. Unsurprisingly, various commercial solutions for “visual drag & drop” chatbot construction exist.

In the following, we prefer the term *Dialogue Workflow Graph (DWG)* because it emphasizes that this graph may also contain elements (“nodes and edges”) of *workflow management*, i.e., elements that describe the workflow actions that need to be carried out behind the scenes of the dialogue.

In this paper, we are presenting a novel DWG Specification Language in Common Lisp, *Dialogue Specification Language (DSL)* for short, and demonstrate its potential for significantly reducing the VPA modeling effort. Our DSL is also a *Domain Specific Language* in the conventional sense. It is built on top of OntoVPA [33, 35].

OntoVPA is a declarative, knowledge-based dialogue manager, in which new VPA domains can be implemented with very little to no conventional programming effort. OntoVPA is highly expressive and comes with built-in solutions to standard DM problems such as state tracking, anaphora resolution, contextual intent slot-filling, intent management, stack-based sub-dialogue management, etc. The rules that implement these common DM capabilities are *generic* and *cross-domain*. They are typically expensive to implement from scratch. As we will illustrate in Section 2.2, by relying on expressive ontology reasoning and forms of higher-order logic quantifications, these rules can *succinctly cover large regions in the DM problem space*, hence greatly reducing the modeling effort, yet achieving generality and hence cross-domain reusability.

Unfortunately, a much bigger *Level-of-Effort (LOE)* is required to model the dialogue workflows by means of rules. These are obviously VPA domain-specific and hence cannot be transferred cross-domain easily. From our experience in developing 5 VPAs with OntoVPA, this LOE can easily account for more than 70 % of the overall LOE.

Our DSL has shown great potential to reduce and simplify this LOE. It also has the benefit of being more intuitive and much less technically involved: unlike plain OntoVPA, developers no longer need to use OWL and SPARQL exclusively. *Visual* workflow dialogue graphs can be generated automatically from the textual DSL specifications, which facilitates customer participation, transparency, and comprehensibility. *The DSL hence has the potential to significantly reduce the modeling effort, thus widening the dialogue workflow modeling bottleneck.*

The remainder of this paper is structured as follows. We first describe the OntoVPA framework, providing the basis for this work. The essence of ontology-based DM is illustrated by means of typical DM problems in the "Restaurant Recommendation" domain [13]. OntoVPA relies on Common Lisp and Racer [11] for the OWL DM modeling and reasoning at *development time* - we hence describe the modeling environment and OntoVPA compiler. At *runtime*, the compiled (target) ontology and a set of ontology-based SPARQL rules are being executed by the JENA reasoner [6, 19] to implement the runtime dynamics. We then subsequently describe three different workflow representation options, corresponding to abstraction layers. The lowest layer (Level 1) corresponds to plain OntoVPA dialogue workflow modeling, and the final (Level 3) abstraction layer corresponds to the new DSL. Next we give a concrete illustration of the DSL, utilizing Common Lisp macros. We then quantitatively demonstrate the benefits of the DSL in terms of reduced modeling LOE. We conclude with a discussion of related work and a summary.

2 ONTOLOGY-BASED DIALOGUE MANAGEMENT

The original OntoVPA is described elsewhere in full detail [33, 35]. It realizes many of the ideas first described by Milward and Beveridge [21, 22], using modern Semantic Web languages and frameworks.

Let us first summarize OntoVPA's distinguished features and architecture so that we can describe the DSL improvements "on top" of the OntoVPA model.

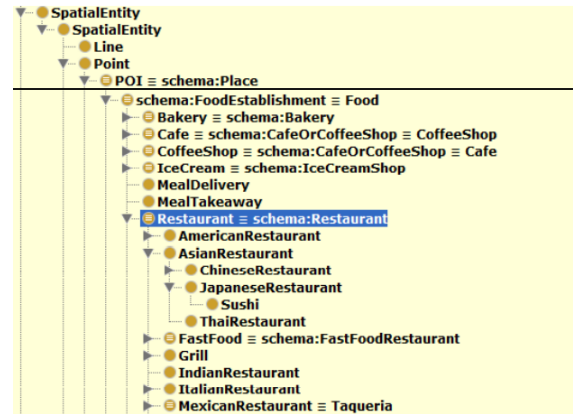


Figure 1: Domain-Specific Ontology

2.1 The Original OntoVPA Model & Architecture

OntoVPA employs the *Web Ontology Language (OWL)* for knowledge representation and reasoning [27]. Additionally, it uses *ontology-based SPARQL* [28] rules for DM which are executed in a custom VPA-specific rule engine built on top of the JENA reasoner [6, 19]. In what follows, we assume basic familiarity with OWL notions such as class, instance, and property, and likewise for SPARQL [3].

These SPARQL rules are SPARQL queries extended with some extra-annotations: rules have IDs, priorities for conflict resolution, can pass control to other rules, etc. This results in a custom, DM-specific ontology-based highly expressive rule language and engine: the *OntoVPA rule engine*.

An OntoVPA model has the following main components:

- An OWL ontology for *background and domain knowledge* – e.g., for the Restaurant Recommendation VPA it will have classes (concepts), relations (OWL object and datatype properties) and instances (individuals) for representing (types of) restaurants, cities, different cuisines, an actual database of restaurant instances, etc. Frequently, we extend and reuse Schema.org [24]; see Figure 1.
- An OWL ontology of *classes and relations for dialogue / discourse representation* – this includes classes for user and system turns, intents and their slot values, and system response classes. *Speech act theory* [25] provides us with a coarse but useful upper ontology, for Requests, Response, etc.; also compare [5] for a similar upper-level ontology. A portion of OntoVPA's upper-level ontology is shown in Figure 2. *At runtime*, these classes and relations are instantiated in an *OWL ABox*, which is a set of class instances and relationships, representing the actual dialogue, i.e., the history of user and system dialogue turns. For example, a restaurant recommendation VPA will have a user intent class `FindRestaurantIntent`, and a corresponding system response class `SuggestRestaurantResponse`, and relations (object properties) for "slots" such as `cuisine` and `location`.
- A layer of *Generic Dialogue Management Rules* – these rules implement core DM capabilities, like slot-value filling, anaphora resolution, context-based disambiguation, details of

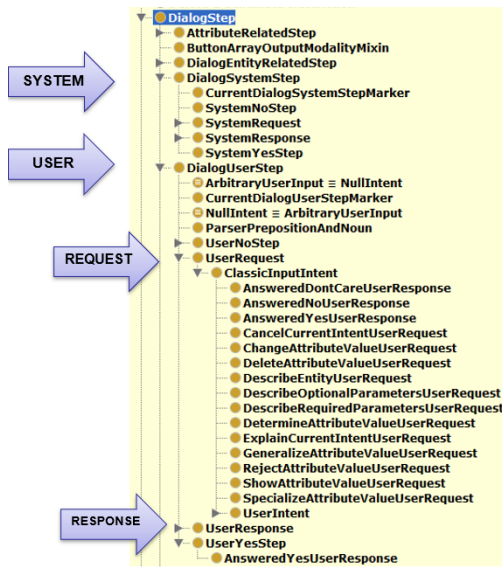


Figure 2: OntoVPA Upper Ontology - Subclasses of *DialogStep*

turn taking, etc. Not every domain requires all these capabilities; they can be disabled individually, but also refined as needed.

- VPA *domain-specific rules that implement the dialogue workflow and intent processing*. In a Restaurant Recommendation VPA, there will be rules that trigger if certain intents are instantiated, e.g., a `RecommendRestaurant` rule triggers if the current intent is a `FindRestaurantIntent`. The rule checks for the presence of the requested location and cuisine, then employs ontology-based query answering to retrieve matching restaurant instances and presents the result.

We heavily rely on *upper-level ontologies and inheritance*, and on generic and reusable behavior specified in the generic DM rules. The combination of ontology reasoning with expressive, succinct higher-order ontology-based rules is the defining feature of OntoVPA.

2.2 Ontology-Based Dialogue Management Example

Let us illustrate the *interplay of ontology reasoning, generic DM rules, and domain-specific rules that implement the dialogue workflow* by means of the following dialogue in the Restaurant Recommendation Domain:

User I am looking for a restaurant!
VPA In what city?
User In Palo Alto.
VPA How about McDonalds?
User Chinese please.
VPA Got it – Su Hong on 4256 El Camino Real?

The first utterance can clearly be classified as a `FindRestaurantIntent` by the statistical intent classifier – let us assume this intent class has a required slot `location` (of range `City`), and an optional slot `cuisine` (of range `Cuisine`). This intent is instantiated in the dialogue representation `ABox`, but without any slot values.

A generic DM rule now inspects the current intent and the definition of its associated OWL intent class and determines that a required slot value (the `location` slot) is missing. To check for the presence or absence of slot values, this generic SPARQL rule uses *existential quantification over slots / predicates*; consequently, we consider this a *higher-order rule*. It hence triggers a follow-up question to the user, who subsequently answered "Palo Alto".

The string "PaloAlto" is automatically mapped to the corresponding OWL `City` instance of the same name. Next, another generic DM rule determines that the `PaloAlto` individual is substitutable for the inquired `location` slot value of the previous intent – the user has answered the question. Hence, `PaloAlto` is filled in as a slot value of the previous `FindRestaurantIntent`, and the intent is now marked *completely specified* by another DM rule. This subsequently triggers the domain-specific workflow `RecommendRestaurant` rule, which proceeds as already described. It retrieves and presents a single restaurant to the user, who subsequently requests "Chinese" instead of the presented option.

Since no intent was recognized from the word "Chinese" in isolation, the system relies again on the dialogue history for sense making / understanding in context. The ontology-based parser has mapped "Chinese" to the class `ChineseCuisine`, which is a subclass of `Cuisine`. From the dialogue history it determines that `ChineseCuisine` is a potential slot-filler for the `cuisine` slot on the previous `FindRestaurantIntent`, and it is hence augmented with the optional slot filler. Given that the intent has changed, it is then re-executed by a rule that detects the change and which then triggers re-execution of the `RecommendRestaurant` rule.

Many of OntoVPA's generic DM rules are specified in a similar flavor. Central expressive means of these higher-order generic SPARQL rules are the ability to

- introspect the OWL class definitions (of intents),
- traverse and assess the full dialogue history,
- perform ontology reasoning (e.g., sub-class and sub-property inferences) in the rule itself,
- existentially and universally quantify "in a second-order fashion" over arbitrary classes and slots (properties) rather than having to codify individual rules for individual classes and slots,
- create and update arbitrarily nested and complex structured graph representations (in the `ABox`); SPARQL allows for the creation of Skolem instances in the right-hand side of rules, and of arbitrary atoms / structure,
- specify defeasibility and priorities on rules.

These DM rules are supplied in an upper level, reusable rule layer applicable to any domain – they provide generic DM capabilities.

In addition, OntoVPA modelers typically still must write domain specific rules like the `RecommendRestaurant` rule, which is the highest LOE activity that we are seeking to replace with the DSL here.

2.3 Rationale for Using a Lisp Language Layer

The OntoVPA compiler is written in Common Lisp with Racer [11]. Given that the runtime engine of OntoVPA is implemented in Java with JENA [6, 19], why have Racer and Common Lisp in the loop at all? We could have just used Protégé [23] for OWL modeling (and

```
(implies FindRestaurantIntent
  (and SchemaOrgSearchIntent
    (some location City)
    (all cuisine Cuisine)))

(implies City Location)
(instance PaloAlto City)
(implies ChineseCuisine Cuisine)
```

Figure 3: Source Ontology Fragment in Racer Syntax

SPARQL rules are written by hand anyway). We are using the Racer & Common Lisp-empowered tool chain for the following reasons.

Common Lisp provides an ideal framework for implementing the DSL. Its macros, and most importantly, the ability to flexibly adjust, extent, and change the DSL for the current VPA project under development makes Common Lisp ideal for resource-constrained research-oriented prototyping projects in which high agility and flexibility is required. In fact, there never was a single DSL, and it was never “designed” - the DSL evolved dynamically over and in different projects, sometimes even incorporating customer-specific terminology to increase model comprehensibility for the customer and hence customer participation. It is likely that these projects would have failed or would have required substantially more development time if a more formal, less agile, “plan and implement” waterfall modeling approach had been taken. After all, implementing a fresh DSL in more traditional DSL environments usually requires specifications of meta-models (MOFs), grammars, parsers, etc., greatly adding to the development time and ultimately, project development costs.

Racer is implemented in Common Lisp, providing convenient and mature Lisp-based OWL modeling and reasoning. OWL modeling in Lisp has many advantages, is concise and readable, and Common Lisp macros can reduce the amount of boiler-plate modeling representations to a minimum. Racer’s built-in OWL parsers enable use to reuse existing OWL and RDFs ontologies (such as Schema.org [24]) by importing these, combining, extending and blending them with our dialogue model, post-processing the unified combined model, and then exporting it in a standard OWL syntax (OWL RDF XML, OWL Functional) utilizing one of Racer’s OWL renderers. The source ontology definition of the just discussed FindRestaurantIntent is illustrated in Figure 3.

Racer is used to perform (static) inferences about the domain and dialogue model at development time for quality assurance and ultimately, trust; e.g., for identifying inconsistencies, redundancies, and implied relationships between classes and relations. Racer is mature and offers reliable, high performance for non-trivial medium-sized ontologies.

Racer then transforms (“compiles”) the development time (source) ontology into a runtime (target) ontology that is loaded into JENA together with the (hand-authored) ontology-based SPARQL rules.

The main rationale for the compilation process is to *avoid most (if not all) potentially expensive JENA OWL inferences at runtime*, as VPAs must be very reactive. We achieve this by compiling the source ontology into a light-weight version of itself, and by making most relevant inferences explicit at compile time (anticipating them) via *inference materialization*. At runtime, OWL inference for

```
<RequiredParameterSpec rdf:about="#param21">
  <functional rdf:datatype="XSD#string">true</functional>
  <range rdf:resource="#City"/>
  <parameter rdf:resource="#location"/>
  <assertedType rdf:resource="#RequiredParameterSpec"/>
  <assertedType rdf:resource="#ParameterSpec"/>
</RequiredParameterSpec>

<OptionalParameterSpec rdf:about="#param66">
  <functional rdf:datatype="XSD#string">true</functional>
  <range rdf:resource="#Cuisine"/>
  <parameter rdf:resource="#cuisine"/>
  <assertedType rdf:resource="#OptionalParameterSpec"/>
  <assertedType rdf:resource="#ParameterSpec"/>
</OptionalParameterSpec>

<owl:Thing rdf:about="#FindRestaurantIntent">
  <representativeConcept
    rdf:resource="#FindRestaurantIntent"/>
  <finalExpectedSystemResponse
    rdf:resource="#RecommendRestaurantResponse"/>
  <requiredParameterSpec rdf:resource="#param21"/>
  <optionalParameterSpec rdf:resource="#param66"/>
  <assertedType rdf:resource="#DialogUserStep"/>
  <assertedType rdf:resource="#UserRequest"/>
  <assertedType rdf:resource="#UserIntent"/>
  ...
  <assertedType rdf:resource="#SchemaOrgSearchIntent"/>
</owl:Thing>
```

Figure 4: Target Ontology Fragment in OWL RDF/XML

JENA is hence then mostly reduced to slot-value lookups, or simple taxonomic query answering, which is extremely fast.

As illustrated in Figure 4 using the target ontology definition of the FindRestaurantIntent, this inference materialization process has computed the deductive closure of the taxonomy via the assertedType property, and has also pre-computed the required and optional slot values of intents. In general (but not in the FindRestaurantIntent), intent superclasses will inherit required property values to their subclasses, and these will need to be specified in a dialogue with the user. The ranges of property values (e.g., Ci ty) might as well enforce yet additional necessary property values (e.g., Ci ty might have required name and geoLocati on properties). The materialization process pre-computes required and optional properties as well as their characteristics (e.g., their ranges and cardinalities). Note the optional/requiredParameterSpec properties and corresponding specification instances in Figure 4. The deductive closure is a finite subset of the full deductive closure of the ontology (which might be infinite). In general, we *avoid disjunctions in the ontology*. Disjunctions can be dealt with *at runtime* in the dialogue via SPARQL rules, but don’t need to be present in the OWL ontology.

The OntoVPA compiler also facilitates the generation of visual representation; we are using DOT and Graphviz [7, 9].

3 MODELING DIALOGUE WORKFLOWS

Our underlying conceptual model is the

- *Dialogue Workflow Graph (DWG)*. Like a (*labeled*) *transition system* [36], this graph represents the space of possible dialogues and workflows, and hence the policy of the system.

This is a development time (static) representation. At runtime, this conceptual graph is interpreted and operationalized by the Dialogue

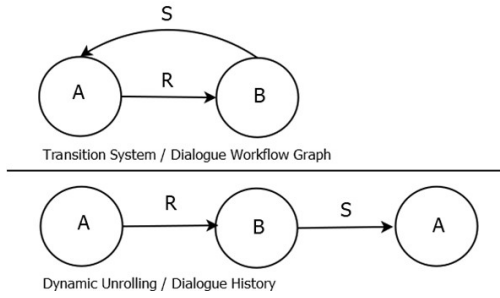


Figure 5: Illustration Transition System & Unrolling

```

CONSTRUCT {
  _:s vpa:assertedType vpa:B .
  _:s vpa:message "Transitioned to B!" .
  _:s vpa:assertedType vpa:CurrentSystemStepMarker .
  ?i vpa:computedSystemResponse _:s }
WHERE
{ ?i vpa:assertedType vpa:CurrentUserStepMarker .
  ?i vpa:assertedType vpa:R .
  ?i vpa:previousUserStep ?p .
  ?p vpa:computedSystemResponse ?pr .
  ?pr vpa:assertedType vpa:A }

```

Figure 6: Simple SPARQL Transition Rule

Manager to compute the system answers / turns. This results in the dynamic

- *Dialogue (Discourse) History*, which is an OWL ABox in OntoVPA. OWL instances are representing user and system turns (nodes), which are asserted and computed, with their causal, temporal, and thematic attributes and relationships.

The dialogue history can be considered an unrolling of the conceptual DWG, analog to the unrolling of a transition system into a trace of dynamic system behavior, see Figure 5.

In this abstract example, A might represent a `ConfirmRestaurantLocation` state, R a corresponding user utterance ("In Palo Alto!"), and B a `SuggestRestaurant` system response.

The dialogue history employs a notion of *current user turn* and *current (last) system turn*. User turns are simply asserted in the dialogue history (and made current) by the intent classifier and ontology-based slot-filler, whereas the system turns are computed by the Dialogue Manager.

In the following, we describe three *concrete* representation options for the conceptual DWG, corresponding to *layers of representations* in OntoVPA. Level 1 is the original OntoVPA representation, Level 3 the new DSL, and Level 2 an intermediate representation.

3.1 Level 1 – Rule-Based Workflow Graph Modeling

It is straightforward to operationalize a conceptual DWG, or transition system, via rules:

For each labeled edge $A \xrightarrow{R} B$ in the DWG, create a corresponding rule that, given current state A , creates a new successor node of state B if condition R holds.

A corresponding OntoVPA SPARQL rule is shown in Figure 6. If, in the WHERE antecedent, the current user step satisfies R and the previous system response was of type A , then the CONSTRUCT consequent of the SPARQL rule creates a new node of type B via the $_:s$ Skolem constant constructor. The freshly constructed node will be asserted as current system step into the dialogue history, and the asserted message property will cause a system utterance. Note that A , B , and R , are OWL ontology classes and properties, and that the SPARQL engine is aware of the deductive closure of the OWL ontology (i.e., its inferences).

The type and number of *conditions* (R in the example) can be arbitrarily complex and may involve full OWL / Description Logic reasoning over the current dialogue, workflow completion, and world state.

There is no limit on what can be asserted with a CONSTRUCT consequent of a SPARQL rule at dialogue runtime – not only can we update the dialogue history, but also other parts of the ABox. For example, personal information learned about the user could be stored on the user's profile information in the ABox.

Let us discuss two obvious drawbacks of the just discussed rule-based modeling style are: a) the large number of rules required, and b) the amount of boilerplate code needed (e.g., each SPARQL rule must determine the `CurrentUserStep` in the history, construct the follow-up node, mark it as `CurrentSystemStep`, etc.)

Regarding a), the number of rules is roughly given by

$$\#nodes * average_node_out_degree$$

This number is *quickly in the three-figure range* for non-trivial dialogue models, see Table 1 in Section 4.

Problem b) can be alleviated by using our *SPARQL macros*. However, the "graph as a set of rules" representation is also unwieldy from a modeling perspective, and a direct graph representation would be preferable. Moreover, SPARQL modeling is intellectually demanding and requires expert knowledge.

3.2 Level 2 – Graph-Based Workflow Graph Modeling

Our first step of remediation is hence to represent the workflow graph *directly as a graph* in the OWL ABox, in terms of *instances (nodes) and relationships (edges)*.

Unlike a set of SPARQL rules, an ABox graph representation is not an executable specification – hence, a *workflow graph interpreter* is required that operationalizes the graph. We chose to implement the interpreter (in a meta-circular way) *itself* in terms of OntoVPA's SPARQL rules, instead of extending the JENA-based rule engine.

The ABox-based, explicit and direct graph representation has some modeling benefits – SPARQL domain rules no longer need to be authored by the domain developer. The interpreter rules are part of the OntoVPA framework already and usually do not need to be altered, just like the generic DM rules already discussed. Moreover, DWGs can now be inspected and, in principle, also be edited visually, by means of OWL tools such as Protégé [23] and RacerPorter [32, 34].

One drawback of this representation though is that it requires *additional modeling vocabulary*, e.g., classes and relationships for representing logical conditions and control structure, as we can

```
(node "In node n1"
  n1
  (:condition A)
  (:transition (R n2)))

(node "Transitioned to node n2"
  n2
  (:condition B)
  (:transition (S n1)))
```

Figure 7: DSL-Based Transition Specification

no longer rely on SPARQL specifications. The representation is *considerably less succinct* – a graph of hundred nodes might require a few thousand ABox assertions (see Table 1 in Section 4 for concrete numbers). The direct representation hence suffers from a *boilerplate representation problem* and complicates modeling due to a lack of tool support for our extra- and control-vocabulary.

Hence, a final abstraction layer added on top, *Level 3*. This layer provides the high-level DSL which is *translated (compiled) into the Level 2 representation* just discussed. We can consider Level 2 as semantic virtual machine code produced from Level 3 DSL high-level specifications, whereas Level 1 provides the generic implementation of the virtual machine that implements the DM / VPA instruction set in OntoVPA.

3.3 Level 3 – DSL-Based Workflow Graph Modeling

In our DSL, a workflow graph node is described together with its outgoing transitions and conditions. In Figure 7, the abstract transition system example from Figure 5 is specified.

Transitions are activated by *edge conditions*, and in addition, there is a *notion of active and disabled nodes*, based on the truth values of their corresponding *node conditions*. The interpreter will never transition into a disabled node.

A variety of *condition types* exist. These condition types are specified via different clauses using the node macro. A more complete illustration of the DSL is given in Figures 8 & 9, using the MEDIC VPA domain (see Section 4). Here, the `:next` clause specifies an unconditional transition to the successor node, and the `:utterance-domain` clauses correspond to the discussed `:transition` clauses.

Node activation and transition conditions can be specified based on:

- the *current intent and its slot values*. This information is usually asserted by a statistical intent classifier and the ontology-based slot filler. However, training statistical intent classifiers can be costly (in terms of training time and data requirements), and we hence support building VPAs without them, by offering ontology-based parsing of raw textual input, which can be used to implement intent classification or information extraction of slot-values or similar “text snippets” from the raw input text.
- parsing raw textual user input (e.g., from ASR) via *ontology-based grammar expressions*. The domain ontology then also plays the role of a *lexicon or thesaurus*, providing domain terms, and its *hyponyms, hypernyms, synonyms, and antonyms*. We support *regular expressions over ontology terms* – for example, the expression (Neg Amp1 PosDesc) is satisfied by the utterance “not very good”, or “not so well”, and might be used to transition to some other node. Sometimes, simple

```
(node "Massive Hemorrhage Control."
  node_mhc1
  (:immediate-transition t)
  (:next node_mhc2))

(node "Where is the bleeding?"
  node_mhc2
  (:utterance-domain
    ("Limb" node_mhc_limb)
    ("HeadOrNeck" node_mhc_head_or_neck)
    ("BodyPart" node_mhc_head_or_neck))
  (:extract-and-store
    ("BodyPart" "currentUser" "hemorrhageLocation")))

(node "Ok, $[currentUser.hemorrhageLocation] hemorrhage."
  node_mhc_head_or_neck
  (:immediate-transition t)
  (:next node_mhc_check_if_clamp_applicable
    node_mhc_dont_apply_clamp))

(node "Can the wound edges be easily re-approximated?"
  node_mhc_check_if_clamp_applicable
  (:utterance-domain
    ("!vpa:AnsweredYesUserResponse" node_mhc_apply_clamp)
    ("!vpa:AnsweredYesUserResponse" node_mhc_apply_dressings)
    ("!vpa:AnsweredNoUserResponse" node_mhc_apply_dressings))
  (:negated-condition
    ("currentUser"
      ("hemorrhageLocation"
        ("nearTo" (:individual "Eye"))))))
```

Figure 8: DSL Illustration from the MEDIC VPA

ontology-based keyword spotting is sufficient as well. We usually accept a hyponym for the specified ontology term in the expression, but this is not a strict requirement.

- complex (negated) *logical conditions that are evaluated over the ABox*, having access to the dialogue history and (ABox) domain model. Logical conditions are specified as *path expressions of OWL properties and classes*, starting from either an ABox individual (that then has to satisfy the expression), or from a class name (then there needs to be some instance that satisfies the expression). For example, in Figure 8 & 9, two possible `:next` transitions are specified on the `node_mhc_head_or_neck` node (yellow node in Figure 9): `node_mhc_check_if_clamp_applicable` and `node_mhc_dont_apply_clamp`. The former is only *active* (and a transition into it possible) if its `:negated-condition` evaluates to false, i.e., the `hemorrhageLocation` must not be `nearTo` the `Eye` (the haemostatic clamp cannot be applied to stop the bleeding in this case).

It is not always desirable or feasible to anticipate all potential dialogue transitions in terms of outgoing conditioned edges on nodes at development time. To allow a more naturally flowing and less linear dialogue, we allow transitions into different dialogue graph regions, e.g., corresponding to different conversation topics, *without requiring explicit outgoing edges on nodes leading into these regions*.

To facilitate these transitions into different regions, we employ the notion of a *trigger* for a node. If the trigger on a node is satisfied, control can transition into the trigger-activated node *in a non-local way, i.e., no edge needs to be transitioned to arrive at it*.

Additional annotations are specified on the node-level, determining aspects of control and thus direct the interpreter

- whether the node is a *start or end node* of a “topic”, i.e., a *sub-graph* that initiates a dialogue / conversation about a specific topic,

- if it *can be triggered*, and hence gain control, if a certain *trigger condition* is satisfied (i.e., an intent or special event has been recognized and been asserted as current),
- whether the node allows to *relinquish control* to another node via trigger-based / non-local transitions,
- whether the dialogue should return and resume at that point after the node lost control in a non-local way (i.e., after the triggered "sub-dialogue" has completed),
- if, in case of a topic end node, the control should *return* to the previously active node, or simply continue;
- if the node is *modal*, i.e., waiting for input before continuing / transitioning to the next node, or if the transition is :immediate; immediate nodes are useful for breaking up the workflow into smaller chunks, among other things.

Finally, the last group two groups of node annotations allow us to a) generate output and b), to update the ABox.

For a), each node can be annotated with a multitude of messages using the :message field. We support *template-based NLG* – rather than using variables for the template "holes", we are yet again using ontology-based path expressions that "peek into the ABox" to fill the template holes; see Figure 9.

Regarding b), we can assert and retract arbitrary ABox assertions. An example from a OntoVPA Medical Decision Support System is shown in Figure 8: the medic is asked for the location of a bleeding. Any answer subsumed by the BodyPart class is then extracted from the utterance and stored as a slot value on the currentUser individual’s hemorrhageLocation slot, using the :extract-and-store clause. Depending on the specific BodyPart, the system then transitions into different branches (for Limb, HeadOrNeck).

4 QUANTIFYING THE DSL BENEFITS

Table 1 shows the *number of nodes*, *number of rules that did not have to be modeled*, *number of ABox graph assertion generated by the DSL compiler*, *the average number of Rules per Nodes (RpN)*, and *the number of Assertion per Node (ApN)* for 2 domains, ChatPal and MEDIC:

Domain	#Nodes	#Rules saved	#Assertions	RpN	ApN
ChatPal	109	291	2520	2.7	23
MEDIC	29	26	374	0.9	11

Table 1: Quantifying the DSL Benefits

ChatPal: A virtual personal companion for the elderly that aims to overcome loneliness and social isolation. ChatPal flexibly engages the senior in templated conversations about specific topics, such as hobbies, the senior’s life history, family, relationships, etc., and it learns about the users’ interests, hobbies, and relationships and is able to leverage the learned knowledge in subsequent sessions. The generated visual DWG can be found online; follow the URL under [30].

MEDIC: A simple Medical Decision Support System prototype. It implements the workflows from a standard medical procedure handbook. Using the DSL, its workflow dialogue graph was developed in ≈ 20 hours only. Again, the visual DWG can be found online; follow the URL under [31].

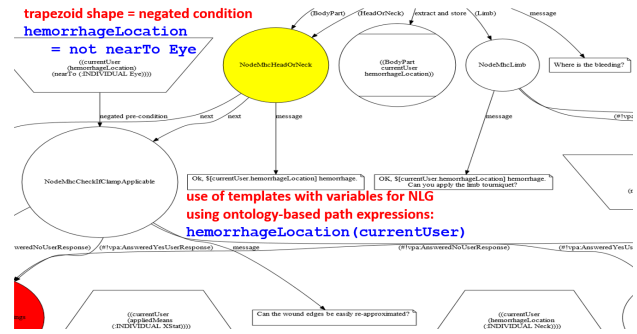


Figure 9: Crop of Generated MEDIC Visual DWG

From our experience, each Level 1 OntoVPA rule modeled by an OntoVPA expert accounts for ≈ 1.5 hours LOE. In contrast, MEDIC was modeled using DSL with 0.68 hours LOE per DSL node; modeled on Level 1, the corresponding 26 rules would have required $26 * 1.5 = 39$ hours – a reduction by 48.7 %.

For ChatPal, the situation is even more significant: $1.5 * 291 = 436.5$ vs. $109 * 0.68 = 74.12$ hours – a reduction by 83 %.

5 RELATED WORK

Knowledge bases and ontologies have been used since the early days of the LUNAR system [38] for Natural Language Understanding, Question Answering and Dialogue Systems, and, more recently, in HALO [10]. Frequently, (OWL) ontologies are being used by the Dialogue Manager for ontology-driven question answering relying on domain ontologies and external knowledge sources [2], [29].

Other case studies focused on ontology modeling and dialogue design based on task structures represented in OWL [4]. OntoDM [1] uses OWL ontologies for domain representation and partially for the dialogue history and NLU tasks such as anaphora resolution [22], but unlike OntoVPA it relies on special purpose algorithms that are informed by the ontologies, and is thus not fully declarative.

OWLSpeak [12] is based on Information State Theory [17], but the state is not implemented in OWL at runtime; hence, no ontology-based rules are being used. VONa [15] is similar to OntoVPA in that it employs OWL and uses a proprietary RDFs/OWL-inference aware "reactive rules" language for policy specification, whereas OntoVPA relies on extended SPARQL over an ABox.

The Convergence system [20] is a hybrid system that uses OWL2 for most aspects of dialogue and domain representation, SPARQL endpoints for question answering, and defeasibility rules for context-aware reasoning [18]. The rules support disambiguation and conflict resolution over the dialogue discourse. Stoyanchev and Johnston [26] implement the Information State Approach [17], using Knowledge Graphs, also drawing inspiration from OntoVPA [33, 35].

An early approach to a generic, visual dialogue flow specification language is given by Kölzer [16]: a compiler translates visual dialogue flow graphs into Prolog knowledge bases.

The dozens of contemporary commercial visual flow builders for chat bot development (Google’s Dialogflow, bots, Visual Chatbot Builder, etc.) usually require some form of programming for customization once the boundaries of the narrowly defined standard domains are reached (e.g., pre-defined models are supplied that

don't generalize well). We believe that OntoVPA's DSL provides a more general and more productive modeling environment.

6 CONCLUSION

We have made steps towards improving dialogue modeling efficiency and customer participation for ontology-based VPAs with our new dialogue modeling language, DSL. It corresponds to a new abstraction layer on top of OntoVPA, comes with a less technical DWG modeling syntax, a compiler and visualizer, and was successfully applied in 2 projects, reducing dialogue modeling effort significantly (by an estimated 49 % and 83 %, respectively).

Our solution is not based on visual programming or modeling; we believe that the throughput, modeling efficacy and efficiency of textual modeling languages is higher and leads to more general solutions.

ACKNOWLEDGMENTS

I am grateful for the fruitful collaborations, support, ideas, and leadership provided by the following (ex-) SRI colleagues over the last 6 years: Girish Acharya, David Berends, Edgar Kalns, Min Yin, James Carpenter, Andreas Kathol, and Theodore Camus.

We would also like to thank Karen Myers and the anonymous reviewers for feedback and suggestions that significantly helped to improve the paper.

The work on MEDIC was supported by the US Army Medical Research and Materiel Command under Contract No. W81XWH-19-C-0096. The views, opinions and/or findings contained in this report are those of the author(s) and should not be construed as an official Department of the Army position, policy or decision unless so designated by other documentation.

The development of OntoVPA and ChatPal was supported and funded by SRI.

REFERENCES

- [1] Duygu Altinok. An Ontology-Based Dialogue Management System for Banking and Finance Dialogue Systems, 2018.
- [2] G. Amores, G. Pérez, P. Manchón, F. Gómez, and J. González. Integrating OWL Ontologies with a Dialogue Manager. *Proces. del Leng. Natural*, 37, 2006. URL http://grupo.us.es/julietta/publications/2006/pdf/Integrating_OWL.pdf.
- [3] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logic Handbook – Theory, Implementation and Applications*. Cambridge University Press, 2003.
- [4] Vinay K. Chaudhri, Adam Cheyer, Richard Guili, Bill Jarrold, Karen L. Myers, and John Niekraz. A case study in engineering a knowledge base for an intelligent personal assistant. In *In the Proc. of the 2006 Semantic Desktop Workshop*, 2006.
- [5] Enrique Fernández-Rodicio, Á. González, F. Alonso-Martín, Marcos Maroto-Gómez, and M. Salichs. Modelling Multimodal Dialogues for Social Robots Using Communicative Acts. *Sensors (Basel, Switzerland)*, 20, 2020.
- [6] Apache Software Foundation. Apache Jena Apache Jena – A free and open source Java framework for building Semantic Web and Linked Data applications., Accessed: 9-14-2021. URL <https://jena.apache.org/>.
- [7] Emden R. Gansner and Stephen C. North. An Open Graph Visualization System and its Applications to Software Engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.
- [8] Anthony J. Bradley / Gartner. Brace Yourself for an Explosion of Virtual Assistants, 8-10-2020. URL https://blogs.gartner.com/anthony_bradley/2020/08/10/brace-yourself-for-an-explosion-of-virtual-assistants/.
- [9] Graphviz. Graphviz & DOT Project Page, Accessed: 9-14-2021. URL <https://graphviz.org/>.
- [10] David Gunning, Vinay K. Chaudhri, Peter E. Clark, Ken Barker, Shaw-Yi Chaw, Mark Greaves, Benjamin Grosf, Alice Leung, David D. McDonald, Sunil Mishra, John Pacheco, Bruce Porter, Aaron Spaulding, Dan Tecuci, and Jing Tien. Project Halo Update—Progress Toward Digital Aristotle. *AI Magazine*, 31(3):33–58, Jul. 2010. doi: 10.1609/aimag.v31i3.2302. URL <https://ojs.aaai.org/index.php/aimagazine/article/view/2302>.
- [11] Volker Haarslev, Kay Hidde, Ralf Möller, and Michael Wessel. The RacerPro Knowledge Representation and Reasoning System. *Semantic Web Journal*, 3(3): 267–277, 2012.
- [12] Tobias Heinroth, Dan Denich, and Alexander Schmitt. OwlSpeak - Adaptive Spoken Dialogue within Intelligent Environments. In *2010 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops)*, pages 666–671, 2010. doi: 10.1109/PERCOMW.2010.5470518.
- [13] Matthew Henderson, Blaise Thomson, and Jason Williams. The Second Dialog State Tracking Challenge. In *In Proceedings of the SIGdial 2014 Conference*, 2014.
- [14] Kasisto. Kasisto company website, Accessed: 9-14-2021. URL <https://kasisto.com/kai/>.
- [15] Bernd Kiefer, Anna Welker, and Christophe Biwer. VONDA: A Framework for Ontology-Based Dialogue Management, 2019.
- [16] A. Kölzer. Universal dialogue specification for conversational systems. *Electronic Transactions Artificial Intelligence (ETAI)*, 3:33–52, 1999.
- [17] S. Larsson and D. Traum. Information State and Dialogue Management in the TRINDI Dialogue Move Engine Toolkit. *Natural Language Engineering*, 6:323–340, 2000.
- [18] Thanassis Mavropoulos, Georgios Meditskos, Spyridon Symeonidis, Eleni Kamateri, Maria Rousi, Dimitris Tzimikas, Lefteris Papageorgiou, Christos Eleftheriadis, George Adamopoulos, Stefanos Vrochidis, and Ioannis Kompatsiaris. A Context-Aware Conversational Agent in the Rehabilitation Domain. *Future Internet*, 11(11), 2019. ISSN 1999-5903. doi: 10.3390/fi11110231. URL <https://www.mdpi.com/1999-5903/11/11/231>.
- [19] B. McBride. Jena: A Semantic Web Toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.
- [20] Georgios Meditskos, Efstratios Kontopoulos, Stefanos Vrochidis, and Ioannis Kompatsiaris. Conversiveness: Ontology-driven conversational awareness and context understanding in multimodal dialogue systems. *Expert Systems*, 37(1), 2020. doi: 10.1111/exsy.12378. URL <https://doi.org/10.1111/exsy.12378>.
- [21] David Milward. Ontologies and the structure of dialogue. In *Proceedings of the 8th Workshop on the Semantics and Pragmatics of Dialogue (Catalog)*, pages 69–77, 2004.
- [22] David Milward and Martin Beveridge. Ontology-based Dialogue Systems. In *Proceedings of the 3rd IJCAI Workshop on Knowledge and Reasoning in Practical Dialogue Systems, Acapulco, Mexico*, pages 9–18, 2003. URL <http://www.ida.liu.se/labs/nlp/lab/ijcai-ws-03/papers/milward.pdf>.
- [23] Mark A. Musen. The Protégé Project: A Look Back and a Look Forward. *AI Matters*, 1(4):4–12, 2015. doi: 10.1145/2757001.2757003. URL <https://doi.org/10.1145/2757001.2757003>.
- [24] Schema.org. Schema.org Project Page, Accessed: 9-14-2021. URL <https://schema.org/>.
- [25] John R. Searle. *Speech Acts: An Essay in the Philosophy of Language*. Cambridge University Press, 1969. doi: 10.1017/CBO9781139173438.
- [26] Svetlana Stoyanchev and Michael Johnston. Knowledge-Graph Driven Information State Approach to Dialog. In *AAAI Workshops*, 2018.
- [27] W3C. OWL 2 Web Ontology Language Document Overview (Second Edition), Accessed: 9-14-2021. URL <https://www.w3.org/TR/owl-overview/>.
- [28] W3C. SPARQL 1.1 Query Language, W3C Recommendation 21 March 2013, Accessed: 9-14-2021. URL <https://www.w3.org/TR/sparql11-query/>.
- [29] E. Wantroba and R. Romero. A Method for Designing Dialogue Systems by Using Ontologies. In *Standardized Knowledge Representation and Ontologies for Robotics and Automation*, volume 58, pages 89–123, September 2014.
- [30] M. Wessel. Auto-Generated Visual Dialogue Workflow Graph for ChatPal, Accessed: 9-14-2021. URL <https://www.michael-wessel.info/downloads/chatpal.pdf>.
- [31] M. Wessel. Auto-Generated Visual Dialogue Workflow Graph for Medic, Accessed: 9-14-2021. URL <https://www.michael-wessel.info/downloads/medic.pdf>.
- [32] M. Wessel. RacerPorter Project Page, Accessed: 9-14-2021. URL <https://github.com/lambdamikel/RacerPorter>.
- [33] M. Wessel, G. Acharya, J. Carpenter, and M. Yin. OntoVPA - An Ontology-Based Dialogue Management System for Virtual Personal Assistants. In M.; Devillers L.; Mariani J. Eskenazi, editor, *Advanced Social Interaction with Agents*, volume 510 of *Lecture Notes in Electrical Engineering*, pages 219–233. Springer, 2019.
- [34] Michael Wessel and Ralf Möller. Design Principles and Realization Techniques for User Friendly, Interactive, and Scalable Ontology Browsing and Inspection Tools. In *Proceedings of the OWLED 2007 Workshop on OWL: Experiences and Directions*, volume 258, 01 2007.
- [35] Michael Wessel, Girish Acharya, James Carpenter, and Min Yin. OntoVPA - An Ontology-Based Dialogue Management System for Virtual Personal Assistants. In *International Workshop on Spoken Dialogue Systems, IWSDS 2017*, 2017.
- [36] Wikipedia. Transition system, Accessed: 9-14-2021. URL https://en.wikipedia.org/wiki/Transition_system.
- [37] Jason Williams, Antoine Raux, and Matthew Henderson. The Dialog State Tracking Challenge Series: A Review. *Dialogue & Discourse*, 7:4–33, 04 2016. doi: 10.5087/dad.2016.301.
- [38] William Woods. *Lunar Rocks in Natural English: Explorations in Natural Language Question Answering*, volume 5, pages 521–569. 01 1977.

RacketLogger: Logging and Visualising Changes in DrRacket

Turgut Reis Kursun
turgut.reis.kursun@vub.be
Vrije Universiteit Brussel
Brussels, Belgium

Quentin Stiévenart
quentin.stievenart@vub.be
Software Languages Lab, Vrije Universiteit Brussel
Brussels, Belgium

Jens Van der Plas
jens.van.der.plas@vub.be
Software Languages Lab, Vrije Universiteit Brussel
Brussels, Belgium

Coen De Roover
coen.de.roover@vub.be
Software Languages Lab, Vrije Universiteit Brussel
Brussels, Belgium

ABSTRACT

Developers frequently make code changes while programming, such as deleting a line of code and renaming or introducing a variable. These changes can be detected and logged, for example by the IDE used by the developer. Logging changes is possible at two levels: at the textual level or at the level of the abstract syntax tree (AST) of the program. The logged changes, in both forms, are useful because they can be used to build new software engineering tools, such as static code analysers.

Plugins that log changes have already been developed for some IDEs. However, so far, no change-logging plugin has been developed for the DrRacket IDE, which supports the development of programs written in Scheme-like languages such as R5RS Scheme and Racket. To fill this gap, we have developed RacketLogger, a change-logging plugin for DrRacket. RacketLogger logs changes both at the textual level and at the AST level. To determine changes at the level of the AST, we have adapted Negara et al.'s algorithm to support Scheme syntax. We have evaluated our plugin by creating a visualisation for the logged changes to measure how well RacketLogger can be used as a building block, and conducted a small-scale user study to measure its usability.

CCS CONCEPTS

• **Software and its engineering** → **Integrated and visual development environments.**

KEYWORDS

Racket, change logging, IDEs

ACM Reference Format:

Turgut Reis Kursun, Jens Van der Plas, Quentin Stiévenart, and Coen De Roover. 2022. RacketLogger: Logging and Visualising Changes in DrRacket. In *Proceedings of the 15th European Lisp Symposium (ELS'21)*. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.5281/zenodo.6326894>

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

ELS'21, March 21–22, 2022, Genova, Italy

© 2022 Copyright held by the owner/author(s).

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

<https://doi.org/10.5281/zenodo.6326894>

1 INTRODUCTION

During development, developers make multiple changes to their program. Many types of changes can occur: inserting new code, deleting a variable, applying a refactoring, and so on. To support development, usually, changes are tracked with version control systems, such as git. The changes logged by version control systems are unsatisfactory for certain applications, as they only provide snapshots of committed code. This raises the need for a more immediate tracking of changes, allowing changes *between* commits to also be stored. As a solution, a *change logger* can be used: a change logger logs all changes that happen during development. Having a lower-level view on the changes enables new applications such as programming pattern detection [1, 5, 9, 10, 14], and tools for collaborative software development [3].

Different granularities and representations can be used for the logged changes. A change logger is called fine-grained if it logs changes at a detailed level, so that almost every interaction is logged. Such fine-grained change loggers can reconstruct every intermediary state of the source code using the logged data [6, 14]. On the other hand, change loggers that only log certain interactions are called coarse-grained.

Changes can be logged at two different levels: at the textual level, and at the level of the abstract syntax tree (AST). Logging changes at the textual level means a change denotes how the *text* of the program has changed. Such a logging mechanism may for example track every character insertion. Logging changes at the AST level means that changes are represented as operations on the nodes of the AST (e.g., inserting, deleting, or updating a node) that connect the AST at a given state of the program to the AST of the subsequent state of the program. Changes logged at the AST level are a rich source of information. They represent which subtrees of the AST have been subject to change, information that can for example be used by incremental program analyses [11, Ch. 7],[12, 13]. Of course, changes at the textual level and changes at the AST level are related, as one often needs the textual changes in order to compute the AST changes. However, AST changes can be obtained only when an AST can be constructed, that is, at points during the development of the program where the program can correctly be parsed.

Multiple change-logging plugins have already been developed. For example, efforts have been made by the research community to provide change loggers for IDEs such as IntelliJ [1] and Eclipse [14]. However, to the best of our knowledge, there does not yet exist a change-logging plugin for the DrRacket IDE.

In this paper, we present RacketLogger, the first change-logging plugin for the DrRacket IDE. RacketLogger –developed as part of a Bachelor Thesis [8]– is implemented in the Racket language and can log changes for any language that uses s-expression syntax –in the remainder of this paper, we will focus on the Scheme language. RacketLogger is a fine-grained change-logging plugin which logs changes textually, representing the actual edits made to the source code, and uses these changes to also infer changes at the AST level, which are persisted as well. To this end, we have adapted the change-inferencing algorithm of Negara et al. [6] to Scheme.

Section 2 discusses the two change representations adopted by RacketLogger in more detail. We then show how RacketLogger can be used as a building block to develop new tools by building a change visualiser (Section 3) which we evaluate by means of a user study (Section 4). Other potential usages of RacketLogger are discussed in Section 5. In Section 6, we review related work on change-logging. We conclude in Section 7.

2 REPRESENTING PROGRAM CHANGES

In this section, we discuss how changes to Scheme programs can be represented. RacketLogger stores the changes at a textual level but is also able to infer the AST node operations from these textual changes. First, in Section 2.1, changes at the textual level are described, before presenting the changes at the level of the AST, which we refer to as *AST node operations*, in Section 2.2.

2.1 Textual Changes

At the lowest level, RacketLogger logs all meaningful interactions with the DrRacket IDE. For that reason, and similar to other change-logging plugins [14], RacketLogger relies on a hierarchical representation of textual changes. The use of a hierarchy enables reasoning about the changes at different levels of abstraction. The hierarchy of changes used by RacketLogger is represented in Figure 1.

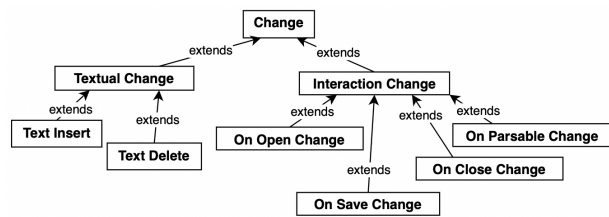


Figure 1: The change hierarchy used by RacketLogger

Changes are divided into two broad categories: *textual changes* and *interaction changes*. Textual changes impact the contents of the source code file, i.e., these are text inserts and text deletions. Interaction changes are non-textual changes, and represent interactions between the programmer and the IDE, such as opening, saving, and closing a file. They provide a context for the textual changes, such as in which file the textual changes were made. An *on parsable* change is logged whenever the code reaches a parsable state. This type of change is useful to trigger the inferencing of AST node operations, which relies on the code being in a parsable state.

2.1.1 Persisting changes. The textual changes logged by RacketLogger are persisted to a file, where each line corresponds to one change. An example log file is given in Listing 1. Each change is represented as a tagged list, where the first element, the tag, indicates the type of the represented change, and the remaining elements contain information about the change itself. For example, text insert changes contain the text that has been inserted and the offset at which the text was inserted. Storing changes as a tagged list is particularly useful in Racket, as this representation can easily be parsed and manipulated in Racket itself.

Listing 1: Example of persisted textual changes.

```

(text-insert ") " 39)
(on-parsable)
(on-close)
    
```

2.1.2 Merging changes. Logging changes at each keystroke is likely to result in large log files. When multiple characters are inserted or deleted at the same place in the source code, it is possible to merge these changes into a single change [6, 14]. Consider the addition of the character 'a' in a program represented by (text-insert "a" 0), followed by the addition of the character 'n' (text-insert "n" 1). These two changes can be merged into a single change, (text-insert "an" 0).

RacketLogger automatically merges changes when possible, that is, when the following conditions are met:

- The textual changes must be of the same type, e.g., a text insertion change cannot be merged with a text deletion change.
- The changes have to be made consecutively in time, to avoid losing information about how changes were interleaved, e.g., two changes cannot be merged if another change has been made in between.
- Changes need to be made consecutively in space. This is the case if the second change starts at the offset where the first change has stopped. This means that if a character is added somewhere in the file, and then a character is added somewhere unrelated in the file, these two changes should not be merged.

Note that merging is performed transitively: a change that is the result of a previous merge can become part of a merge again, and hence, any number of changes can become merged into a single change as long as the above conditions are fulfilled.

As an example, consider Figure 2 which represents two textual changes, (text-insert "ab" x) and (text-insert "c" x+2). Note that when we add the offset of the first change, x, to the length of its text, then the offset of the second change is obtained. As the second change starts at the offset where the first change has stopped, RacketLogger merges them. Similarly, two consecutive textual deletions can also be merged. In this case, the first change needs to be (text-delete "c" x+2) and the second change needs to be (text-delete "ab" x).

2.2 AST Changes

Storing changes at the textual level may be too fine-grained or impractical for applications to work with. For this reason, RacketLogger is able to infer AST changes from the textual changes when

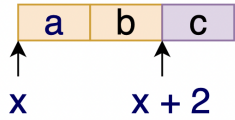


Figure 2: Two textual changes to be merged

the program is in a parsable state (so that an AST can be obtained). AST changes describe how the AST has changed from one parsable state to the next, and are computed as soon as the AST comes to a new parsable state. To see this, consider Figure 3, which exemplifies the derivation of AST changes from textual changes. Blue nodes denote node update operations, meaning that the contents of the node has been updated, and green nodes denote node insert operations, meaning that nodes are inserted. When no node operations are present in a subtree, this means that the subtree has remained unchanged between two parsable states. For example, in Figure 3, consider the subtree encircled in orange. Clearly, the AST changes are a more rich and interesting source of information – which can be used by program analyses run by the IDE for example – than the corresponding textual changes – which give no information on what nodes have been updated, inserted, deleted, or have remained unchanged.

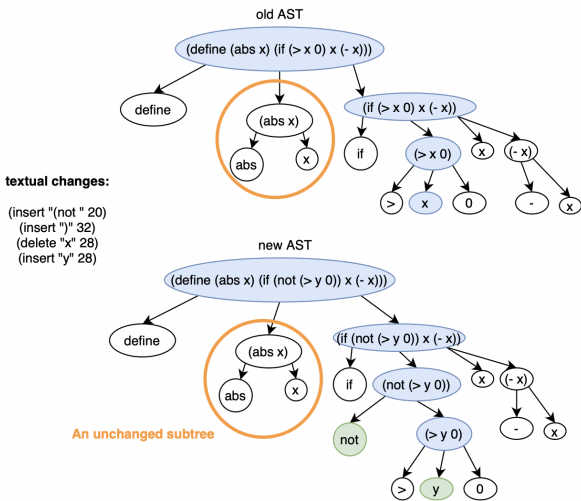


Figure 3: AST changes between two parsable states, showing node updates (blue) and node inserts (green). An unchanged subtree has been encircled in orange.

2.2.1 *An Analogy for AST Change Inferencing Algorithms.* AST changes are inferred from the last parsed AST before the changes, the current parsed AST, and the textual changes that connect them. To understand how the change-inferencing algorithm for AST operations works, we now first provide an intuitive analogy: the game of spot-the-difference, often played by children, and exemplified in Figure 4. The goal of the game is to find differences between two images. Analogously to the game, a change-inferencing algorithm

finds the differences between two ASTs, in terms of node operations. More similarities emerge between inferencing algorithms and the game if we assume that the right image follows from the left image. This assumption allows the left image to play the role of an old AST, whereas the right image can play the role of a new AST. The deletion of an item in the left image corresponds to a node deletion operation in the old AST, and is indicated in black in the figure. Notice that the deleted item is present only in the first image, and similarly, a node deletion occurs only at a node of the old AST as it cannot be shown in the new AST. The insertion of an item into the second image corresponds to a node insertion operation into the new AST, and is indicated in blue. The inserted item is present only at a node of the new AST. Finally, some elements that are present in both images are updated, which corresponds to node update operations. The updated elements are present in both figures, and similarly, updated nodes come in pairs: one in the old AST, one in the new AST.



Figure 4: Example of a spot-the-difference game. Blue circles mark updates to a part of the figure, green circles mark additions, and black circles mark deletions.

2.2.2 *High-level overview of the Change-Inferencing Algorithm used by RacketLogger.* To derive AST changes made to a Scheme program from textual changes, we have adapted the algorithm for the inferencing of AST node operations of Negara et al. [6]. This algorithm returns node operations (insertions, deletions and updates), given the old AST, new AST, and the corresponding textual changes that represent the changes to code when going from the old AST to the new AST. We first give an overview of the algorithm developed by Negara et al. Afterwards, we discuss how it was adapted for Scheme.

To generate node changes, the algorithm first establishes the root of the changed subtree, called the *common covering node*, which is present in both the old AST and the new AST. Finding it is of interest, since the rest of the algorithm can then operate on this subtree, saving computational efforts. Since such a common covering node represents the root of the changed subtree, it encloses all changes. In general, nodes are found by finding the traversal path from the root

of the AST to that node. Hence, finding the common covering node boils down to finding its path. The path is found in two steps. First, the algorithm looks for local covering nodes in both ASTs, these are the innermost nodes that enclose all textual changes. Second, the path to the common covering node is found by taking the common part of the paths to local covering nodes.

When the algorithm has found the common covering node, it starts matching descendants of the common covering node. The matching of nodes denotes the fact that the new AST node was already present in the old AST. Nodes are matched in two ways:

- The algorithm matches outliers, i.e., nodes that have not been affected by any changes. Hence, outliers remain unchanged and no node operations need to be generated.
- The algorithm matches yet unmatched nodes that have the same traversal path from their respective roots. For these nodes, update operations need to be generated.

Lastly, the algorithm generates node insert, node delete, and node update operations. For every unmatched descendant of the common covering node in the old AST, a delete operation is generated. For every unmatched descendant of the common covering node in the new AST, an insert operation is generated. For every pair of matched nodes whose content has changed, an update operation is generated. Notice how this generation of operations is consistent with our previous analogy.

2.2.3 Application to Scheme. Now that we have given an overview of the original algorithm developed by Negara et al. [6], we explain how we have adapted it to Scheme. First, we have noticed that the algorithm must generate more update operations. More precisely, an update operation must be generated for every pair of ancestors of the common covering node. The original algorithm generates operations only for nodes below the common covering node. Since the common covering node in the old AST and the new AST share their traversal paths, this means that they have an equal number of ancestors, who match with each other. These nodes enclose all changes since they enclose the common covering node, so they must have changed. Thus, an update operation must be generated for each pair of nodes on the path from the root of the AST to the common covering node.

Second, we have implemented low-level logic that enables the original algorithm to decide when a node is affected by a change. Note that when the algorithm matches outliers, it must be able to tell if a change affects a node. We have implemented this check for Scheme, and have noticed some peculiarities that relate to the syntax of the language. Consider Figure 5, where the code at the top corresponds to an identifier enclosed within parentheses and the code at the bottom corresponds to a simple identifier. Now, consider a textual insertion that occurs to the right of these nodes, indicated by the arrow. If a change starts at this offset, we see that the top node is untouched. However, the node representing the identifier node might be touched, as the name of the identifier may be made longer: when the added code does not start with a space or a parenthesis, the identifier is affected. Similar rules are required for insertions that occur at the beginning of the code represented by a node.

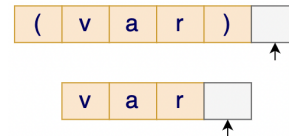


Figure 5: A bracketed S-exp, and an identifier S-exp. The arrow indicates the offset of a textual insertion

2.2.4 RacketLogger's Inferencing Algorithm in Pseudocode. Algorithm 1 shows the pseudocode for RacketLogger's AST node operations inferencing algorithm. The input to the algorithm is the old AST, the new AST, and the textual changes. These are the textual changes that took the code from the oldAST to the newAST. The output to the algorithm is a set of AST node operations, ASTops. These operations are update, insert, and delete operations. Recall that our algorithm is based on the state-of-the-art algorithm used by Negara et al. [6]. We have highlighted the parts of the algorithm which we adapted to Scheme. We will now discuss the algorithm in more detail.

First, two variables are initialised to the empty set, ASTops, which will store the inferred AST node operations (line 3), and matches, which the algorithm uses to store pairs of matched nodes between the old and new AST (line 4). Then, the traversal path to the common covering nodes is found, and, using this path, the common covering nodes are obtained from both ASTs (lines 5-7).

Next, the algorithm matches outliers, i.e., the nodes that have not been affected by any change. Every descendant node of the common covering node in the old AST is checked against the changes. A change does not affect a node if the code that the node represents is completely before the change, or completely after it. If the offset of the node is before the change, the change does not impact the offset of the node either. However, if the offset of a node is after the offset of the change, it alters the offset of the node. Changing the offset of a node can also be seen as shifting the node to the left or to the right. Text insertions shift the offset by the length of the inserted text, whereas text deletions shift the offset by the opposite number (- length of deleted text). These offset shifts are computed by the function `getChangeOffset` and accumulated in a variable `deltaOffset` (line 12). If no changes affect the old AST node, then the algorithm looks for its matching node in the new AST by using `deltaOffset` (line 14), and the matched pair of nodes is added to matches (line 15).

In the next step, the algorithm matches nodes that are still unmatched but have the same traversal path starting at the root of their ASTs. The algorithm first loops over all the old AST nodes that are descendants of the `oldCoveringNode` and that have not yet been matched (line 18). For each such node, the traversal path to this node is computed and the algorithm attempts to find the node in the new AST on that path (lines 20-21). This pair of nodes is then matched together if the new AST node is also not yet matched (line 22). In case no node can be found, then there cannot be a match.

Next, the algorithm starts generating node operations. For each matched node, the algorithm generates an update operation if the

code contained in the old AST node is different from the code contained in the matching new AST node (lines 25-27). Then, the algorithm generates an update operation for all pairs of corresponding ancestors of the common covering nodes, as explained in Section 2.2.3. Lastly, a delete operation is generated for each unmatched descendant of the common covering node in the old AST (lines 31-33), and an insertion operation is generated for each unmatched descendant of the common covering node in the new AST (lines 34-36).

3 EVALUATION: VISUALISING CHANGES WITH RACKETVIZ

We have evaluated RacketLogger by using the AST changes it captures to create an interactive visualisation of the changes it has logged. To this end, we have built a second plugin for DrRacket, RacketViz. RacketLogger supports registering a callback function which will be called each time a set of node operations is inferred. RacketViz plugs into RacketLogger through this callback. Listing 2 shows how such a callback can be registered. RacketLogger provides five arguments to the callback: the old AST, the new AST, the inferred node operations, the changes connecting both ASTs, and an object which indicates in which tab the changes occurred (in DrRacket, a developer can use multiple tabs simultaneously). Hence, the first four arguments provide all information on the changes, both textually and at the level of AST nodes. We refer back to Figure 3 for illustrative values of the first four arguments.

Listing 2: Registration of the callback function of RacketLogger.

```
(set-AST-inference-callback!
  (lambda (old-ast new-ast inferred-node-ops
           changes-obj defs-text)
    ...))
```

RacketViz implements a visualisation of the current AST of the program, which is updated according to the changes made by the developer whenever the program reaches a parsable state. To this end, RacketViz uses the information on AST node changes provided by RacketLogger, and creates a visualisation that is encoded in the dot language, so that it can be converted into an image by GraphViz [2]. This image is shown to the user in the a frame within the DrRacket editor, and the image is updated every time when AST node operations are inferred. An example image is shown in Figure 6.

Since RacketViz is provided with the inferred AST node operations, it can colour the nodes of the new AST according to these operations: update operations are coloured blue, and insert operations are coloured green. RacketViz does not show the old AST, because this would be cumbersome to do within the small DrRacket frame. Hence, delete operations cannot be visualised, as the deleted nodes are no longer present in the new AST. However, it is entirely possible to also generate images for the old AST, where delete operations could be shown.

Finally, when the programmer changes to another tab, RacketViz loads the image for the corresponding tab. This can simply be retrieved from memory, where RacketViz stores the last generated image for every tab.

Algorithm 1: Inferencing algorithm for AST node operations.

```
1 affects(change, node, offset) returns true if a change affects a node,
   i.e., if the change is made within the boundary of the node.
2 getCommonCoveringPath(oldAST, newAST, changes) returns a list
   describing the path to the common covering node, by finding a local
   covering node in both oldAST and newAST, and extracting their
   common path.
input : The old AST, oldAST, the new AST, newAST, and the
   textual changes, changes.
output: A set of AST node operations, ASTops.
3 ASTops := ∅; // AST node operations.
4 matches := ∅; // Pairs of matched nodes.
5 coveringPath :=
   getCommonCoveringPath(oldAST, newAST, changes);
6 oldCoveringNode := getNode(oldAST, coveringPath);
7 newCoveringNode := getNode(newAST, coveringPath);
   // Match outliers.
8 foreach oldNode ∈ getDescendants(oldCoveringNode) do
9   deltaOffset := 0;
10  foreach change ∈ changes do
11    if affects(change, oldNode, deltaOffset) then
12      Continue foreach line 8;
13    else deltaOffset := deltaOffset +
14      getChangeOffset(change, oldNode, deltaOffset);
15  end
16  if ∃ newNode ∈ getDescendants(newCoveringNode) :
17    getOffset(oldNode) + deltaOffset = getOffset(newNode) then
18    matches := matches ∪ (oldNode, newNode);
19  end
20 end
   // Match same-path nodes.
21 foreach oldNode ∈ getDescendants(oldCoveringNode) do
22  if oldNode ∉ getOldNodes(matches) then
23    oldPath := getNodePath(oldNode, oldAST);
24    newNode := getNode(newAST, oldPath);
25    if newNode ≠ null and
26      newNode ∉ getNewNodes(matches) then
27      matches := matches ∪ (oldNode, newNode);
28  end
29 end
   // Infer node operations.
30 foreach (oldNode, newNode) ∈ matches do
31  if getText(oldNode) ≠ getText(newNode) then
32    ASTops := ASTops ∪ makeUpdateOp(oldNode, newNode);
33 end
34 foreach (oldParent, newParent) ∈
   parentsOnPath(coveringPath, oldAST, newAST) do
35  ASTops := ASTops ∪ makeUpdateOp(oldParent, newParent);
36 end
37 foreach oldNode ∈ getDescendants(oldCoveringNode) do
38  if oldNode ∉ getOldNodes(matches) then
39    ASTops := ASTops ∪ makeDeleteOp(oldNode);
40 end
41 foreach newNode ∈ getDescendants(newCoveringNode) do
42  if newNode ∉ getNewNodes(matches) then
43    ASTops := ASTops ∪ makeInsertOp(newNode);
44 end
```

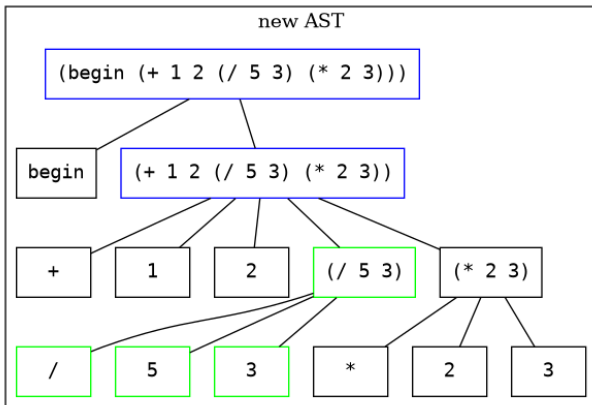


Figure 6: AST visualisation by RacketViz. Node updates are shown in blue, whereas node inserts are shown in green.

4 USER STUDY

To evaluate the usefulness of RacketLogger as a building block for any application that requires information about changes, we have conducted a user study. RacketViz, a tool enabled by RacketLogger, was installed on the computers of five participants, all students in computer science at the bachelor level, using DrRacket daily. Then, they were given about a week to use RacketViz, after which we asked them a series of closed and open questions regarding their experience using RacketViz.

Table 1 lists the closed questions of our user study and the responses of the five participants. The first two questions were used to evaluate the past experience of the participants with DrRacket and Scheme and to know for how much time the participants have used the IDE with RacketViz installed. Next, participants were given a series of statements, for which they had to indicate how much they agreed with each on a scale from 0 to 10.

The three last columns of the table in Table 1 summarise the results of our user study. We see that the participants were already familiar with the DrRacket IDE, having 2 to 4 years of experience using it. The second question asked how intensively they used DrRacket whilst the plugins (RacketViz and RacketLogger as its dependency) were installed. In total, the users have reported using it for 9 hours.

The remaining questions asked about their user experience with RacketViz, where participants rated propositions on a scale from 0 to 10, where 0 indicates a negative user experience, whereas 10 indicates a positive user experience. By looking at all the answers, we see that the users had a positive experience with the plugins. In short, the participants found that DrRacket kept working smoothly, they managed to easily inspect the AST, they found the provided information quite useful and easy to understand, and that the shown AST was quickly updated after changes.

Alongside our closed questions, we also asked the participants some open questions. First, we asked whether any errors occurred whilst using the plugins. Two errors were reported, explaining that the AST is not shown when multiple frames (not tabs) of the DrRacket IDE are open. Second, we asked what else our participants

would like to see in the visualisation. We explicitly encouraged wild ideas from our participants. Three participants came up with an idea:

- To add zoom buttons to the AST visualisation.
- To highlight the AST nodes corresponding to run-time errors.
- To highlight the code that was added from one parsable state to the next using a different colour in the editor.

We find that the third idea is particularly interesting since it could make great use of the detailed data about changes provided by RacketLogger. As a third open question, we asked participants whether they had any additional remarks. However, no participant had additional remarks. We plan in the future to extend this preliminary evaluation to evaluate in details the performance, correctness, and usability of RacketLogger and RacketViz.

5 OTHER POTENTIAL APPLICATIONS OF RACKETLOGGER

In this Section, we discuss other applications that could be built on top of the change information provided by RacketLogger. Recall that RacketLogger provides information about textual changes as well as AST changes.

Empirical studies. Developers might be changing complex Scheme expressions more often than straightforward ones. It could also be the case that developers change procedure declarations more often than class declarations. By using the information provided by RacketLogger, and gathering a large and diverse sample of programmers, one could shed light into these and related matters. Declarative change query languages [9, 10] have been developed to facilitate such empirical studies. One could also mine for patterns in the captured changes [5], which can be indicative of refactoring operations for which automated tool support ought to be provided.

Incremental program analysis. Many IDEs already have some form of built-in program analysis to support software development. For example, IntelliJ employs a data flow analysis [4]. When software changes, these analyses have to be rerun to update their results. Clearly, this is a frequent event within an IDE, therefore making it impractical to run a full software analysis upon every change to the code base. As a remedy, incremental program analyses can be used, which update the analysis results based on the changes made to the code [11, Ch. 7],[12, 13]. This makes employing static analysis practical, as an incremental update of the analysis results takes less time than a full analysis of the codebase.

For an incremental analysis to be efficient however, it must have an overview of the changes, allowing the analysis to find the parts of its result that need invalidation and recomputation. In light of this, RacketLogger could be used to provide these changes, and therefore to bring incremental static analysis to DrRacket in the future.

6 RELATED WORK

In this section, we discuss some related work on change logging.

Yoon et al. developed Fluorite [14], an event-logging (or change-logging) plugin for the Eclipse IDE. It logs all low-level events (or changes) that occur in the editor using an XML format. This format

Question	Mean	Min	Max
How familiar are you with DrRacket/Scheme (in years)	3	2	4
How intensively have you used DrRacket after installing RacketViz (in minutes)	108	60	120
<i>Answers on a scale of agreement from 1 to 10</i>			
Does DrRacket works as smoothly as usual when running RacketViz?	9	8	10
Could you easily inspect the AST?	9.6	9	10
Do you find the provided AST information useful?	8.4	8	10
Did the AST shown in DrRacket update quickly when the code was parsable?	9	8	10
Are the inferred node operations clear to you?	8.2	8	9

Table 1: Questions from our user study of RacketViz. The last 5 questions are answered on a scale of agreement from 0 (indicating a negative user experience) to 10 (indicating a positive user experience).

allows the logged textual changes to be used by other plugins. Fluorite only logs textual events, merging changes whenever possible. It does not capture AST changes.

Omori et al. [7] developed OperationRecorder, a change-logging plugin for Eclipse. The goal of this plugin is to more deeply understand code evolution. Traditionally, code evolution is studied by using snapshots in repositories, but the authors found that this traditional way of studying code evolution is incomplete since intermediate changes in the editor are lost. OperationRecorder uses its own inferencing algorithm, that also relies on textual changes.

Negara et al. [6] have developed CodingTracker, a change-logging plugin for Eclipse. Its main goal is to study software evolution in a more complete and precise way. The tool has been used to answer five research questions. One of the questions served as motivation for RacketLogger: “How much code evolution data is not stored using Version Control (VC)?”. By performing a study with 15 participants, across 2000 commits and 23002 committed files, they found that on average 37 percent of changes never reach version control. This finding motivates change loggers since they give a more complete picture of code evolution than version control repositories. Negara et al. [6] have also implemented a state-of-the-art AST node operations inferencing algorithm, which we have adapted to Scheme syntax.

Hattori and Lanza have developed Syde, a change-logging plugin for Eclipse. Syde is developed as a tool for collaborative software development. For example, changes are broadcast to all team members of a project, and real-time visualisations of the evolution of the system.

Beller et al. [1] have developed WatchDog, a change-logging plugin for Eclipse, IntelliJ, and Visual Studio Code. By using WatchDog, Beller et al. performed a large-scale study of the habits of developers whilst testing. Their results have shed light on how several activities of developers relate to each other. For example, they found that developers often overestimate the time they spend on testing software.

The most notable difference between existing work and RacketLogger is the compatible IDE. To the best of our knowledge, RacketLogger is the first change-logging plugin for DrRacket. RacketLogger also has common DNA with other change loggers: it logs textual changes [6, 7, 14], merges changes [6, 14], and uses CodingTracker’s state-of-the-art AST node operations inferencing algorithm [6].

7 CONCLUSION

In this paper, we have presented RacketLogger, the first change-logging plugin for the DrRacket IDE. We explained that RacketLogger uses a hierarchy of changes, and merges changes, similar to many state-of-the-art change-logging plugins [6], and we discussed AST changes. RacketLogger required adaptation of an existing state inferencing algorithm in order to support Scheme-like languages. We have explained how the nested structure of Scheme code implies node update operations at matching ancestors of the common covering nodes. We have also introduced the challenges that had to be overcome in determining whether an s-expression is affected by textual changes. Finally, we have shown how RacketLogger may be used to build other plugins that require detailed information about changes. To that end, we implemented a change visualiser, RacketViz, which shows the AST of the program as it evolves in a DrRacket frame. We have conducted a preliminary evaluation of RacketViz through a user study, indicating that it was well received by the participants.

REFERENCES

- [1] Moritz Beller, Georgios Gousios, Annibale Panichella, Sebastian Proksch, Sven Amann, and Andy Zaidman. Developer Testing in the IDE: Patterns, Beliefs, and Behavior. *IEEE Trans. Software Eng.*, 45(3):261–284, 2019. doi: 10.1109/TSE.2017.2776152. URL <https://doi.org/10.1109/TSE.2017.2776152>.
- [2] John Ellson, Emden R. Gansner, Eleftherios Koutsofos, Stephen C. North, and Gordon Woodhull. Graphviz - open source graph drawing tools. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing, 9th International Symposium, GD 2001*, volume 2265 of *Lecture Notes in Computer Science*, pages 483–484. Springer, 2001.
- [3] Lile Hattori and Michele Lanza. Syde: a tool for collaborative software development. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2, ICSE 2010*, pages 235–238. ACM, 2010. doi: 10.1145/1810295.1810339. URL <https://doi.org/10.1145/1810295.1810339>.
- [4] Zarina Kurbatova, Yaroslav Golubev, Vladimir Kovalenko, and Timofey Bryksin. The IntelliJ platform: a framework for building plugins and mining software data. *arXiv preprint arXiv:2110.00141*, 2021.
- [5] Tim Molderez, Reinout Stevens, and Coen De Roover. Mining change histories for unknown systematic edits. In *Proceedings of the 14th International Conference on Mining Software Repositories (MSR17)*, 2017.
- [6] Stas Negara, Mohsen Vakilian, Nicholas Chen, Ralph E. Johnson, and Danny Dig. Is It Dangerous to Use Version Control Histories to Study Source Code Evolution? In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference*, volume 7313 of *Lecture Notes in Computer Science*, pages 79–103. Springer, 2012. doi: 10.1007/978-3-642-31057-7_5. URL https://doi.org/10.1007/978-3-642-31057-7_5.
- [7] Takayuki Omori and Katsuhisa Maruyama. A change-aware development environment by recording editing operations of source code. In Ahmed E. Hassan, Michele Lanza, and Michael W. Godfrey, editors, *Proceedings of the 2008 International Working Conference on Mining Software Repositories, MSR*

- 2008, pages 31–34. ACM, 2008. doi: 10.1145/1370750.1370758. URL <https://doi.org/10.1145/1370750.1370758>.
- [8] Turgut Reis Kursun. RacketLogger: Logging changes from the dracket code editor. Bachelor's thesis, Vrije Universiteit Brussel, Brussels, Belgium, 2021.
- [9] Reinout Stevens and Coen De Roover. Querying the history of software projects using QwalkEko. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution, Tool Demo Track, (ICSM14)*, 2014.
- [10] Reinout Stevens, Tim Molderez, and Coen De Roover. Querying distilled code changes to extract executable transformations. *Empirical Software Engineering*, 24(1):491–535, 2019.
- [11] Tamás Szabó. *Incrementalizing Static Analyses in Datalog*. PhD thesis, Universitätsbibliothek der Johannes Gutenberg-Universität Mainz, 2021.
- [12] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. Inca: a DSL for the definition of incremental program analyses. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*, pages 320–331. ACM, 2016. doi: 10.1145/2970276.2970298. URL <https://doi.org/10.1145/2970276.2970298>.
- [13] Jens Van der Plas, Quentin Stiévenart, Noah Van Es, and Coen De Roover. Incremental Flow Analysis through Computational Dependency Reification. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020*, pages 25–36. IEEE, 2020. doi: 10.1109/SCAM51674.2020.00008. URL <https://doi.org/10.1109/SCAM51674.2020.00008>.
- [14] YoungSeok Yoon and Brad A. Myers. Capturing and analyzing low-level events from the code editor. In Craig Anslow, Shane Markstrum, and Emerson R. Murphy-Hill, editors, *Proceedings of the 3rd ACM SIGPLAN workshop on Evaluation and usability of programming languages and tools, PLATEAU 2011*, pages 25–30. ACM, 2011. doi: 10.1145/2089155.2089163. URL <https://doi.org/10.1145/2089155.2089163>.

