Proceedings of the

11th European Lisp Symposium

Centro Cultural Cortijo de Miraflores, Marbella, Spain
April 16 – 17, 2018

In-cooperation with ACM

Dave Cooper (ed.)

acm In-Cooperation

acm SIGPLAN

ISBN-13: 978-2-9557474-2-1

Contents

Preface

Message from the Program Chair

Welcome to the 11thth edition of the European Lisp Symposium!

This year's ELS demonstrates that Lisp continues at the forefront of experimental, academic, and practical "real world" computing. Both the implementations themselves as well as their far-ranging applications remain fresh and exciting in ways that defy other programming languages which rise and fall with the fashion of the day. We have submissions spanning from the ongoing refinement and performance improvement of Lisp implementation internals, to practical and potentially lucrative real-world applications, to the forefront of the brave new world of Quantum Computing.

Virtually all the submissions this year could have been published, and it was a challenge to narrow them down enough to fit the program.

This year's Program also leaves some dedicated time for community-oriented discussions, with the purpose of breathing new life and activity into them. On Day One, the Association of Lisp Users (ALU) seeks new leadership. The ALU is a venerable but sometimes dormant pan-Lisp organization with a mission to foster cross-pollenization among Lisp dialects. On Day Two, the Common Lisp Foundation (CLF) will solicit feedback on its efforts so far and will brainstorm for its future focus.

On a personal note, this year I was fortunate to have a "working retreat" in the week prior to ELS, hosted by Nick & Lauren Levine in their villa nestled in the hills above Marbella. This was a renewing and reflective time which allowed me to recharge and recommit to going back to the "real world" with a strong desire to do great things with Lisp. Thanks Nick and Lauren!

Wishing you all a wonderful time in (what should be a) sunny Marbella. I am honored to be of humble service to this awesome community. Many Thanks,

Dave Cooper, writing from Marbella, April 12 2018.

Message from the Local Chair

When I agreed to help organise the 2018 ELS I expected to be busy, to be stressed, and to end up with a list of people with whom never to speak again. The reality has been different. Yes I've been busy. Yes I've been stressed, but everyone I've had to work with has been unfailingly helpful, from Didier, Dave, Nick and the rest of the ELS team to the local council representatives, from the restaurant owners to the bus driver and tour guides. The result is that this has been a very positive experience for me, for Nick I believe, and I hope that this will be reflected in what should be a very positive experience for us all.

I hope that you all enjoy your time in Marbella.

Andrew Lawson, Marbella, April 13 2018

Organization

Programme Chair

- Dave Cooper, Genworks, USA

Local Chair

- Andrew Lawson — Ravenpack, Marbella, Spain
- Nick Levine — Ravenpack, Marbella, Spain

Programme Committee

- Andy Wingo – Igalia, Spain
- Christophe Rhodes – Goldsmiths University, UK
- Christopher Wellons – Null Program USA
- Ernst van Waning – Infometrics.nl, Netherlands
- Irène Durand – LaBRI, Université de Bordeaux, France
- Julian Padget – University of Bath, UK
- Ludovic Courtès – Inria, France
- Michael Sperber – DeinProgramm, Germany
- Nicolas Neuss – FAU Erlangen-Nürnberg, Germany
- Olin Shivers – Northeastern University USA
- Philipp Marek – Austria
- Robert Strandh – Université de Bordeaux, France
- Sacha Chua – Living an Awesome Life, Canada
- Scott McKay – Future Fuel, USA

Sponsors

We gratefully acknowledge the support given to the 11thth European Lisp Symposium by the following sponsors:

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

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

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

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

RavenPack
Centro de Negocios Oasis Oficina 4
Urb. Villa Parra, Ctra de Cadiz km 176
29602 Marbella Málaga Spain
`www.ravenpack.com`

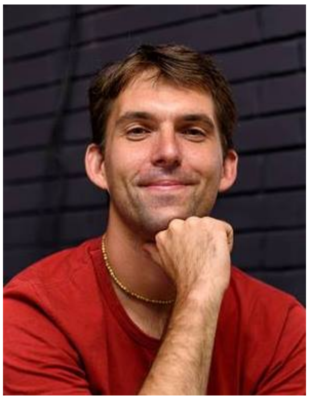
ACM Sigplan
`sigplan.org`

Invited Contributions

Lisp in a Startup: the Good, the Bad, and the Ugly

Vsevolod Dyomkin, m8nware, Ukraine

Over the last 10 years of my software development career, I have mostly programmed in Common Lisp, in two distinct environments: open-source and startup (aka consumer Internet companies). Among the startup projects, in which I managed to introduce Lisp, the most successful is Grammarly where the system we had built continues to playa major role — more than two years after my departure from the company — at the core of its intelligent product used by 10 million people daily.
In this talk, I'd like to share the success stories of developing a number of internet services in Lisp and the merits of the Lisp enviornment that enabled those, as well as the flip sides of the same stories that manifest the problems of the Lisp ecosystem — and propose possible solutions to them. We'll discuss where Lisp fits best and worst among the different professional environments and why.



Vsevolod Dyomkin is currently employed as a Lisp consultant at Franz Inc. working on AllegroGraph. He is a long-time Lisp enthusiast and quit his job 9 years ago to program in Common Lisp: first, his own projects, afterwards, in bigger companies and as a hired consultant. His other area of interest is Natural Language Processing. He has developed a number of open-source Lisp projects, the most notable of which is CL-NLP — a yet incomplete Lisp NLP library. He's also an author of "Lisp Hackers" — a series of interviews with prominent Lisp programmers.

This Old Lisp

R. Matthew Emerson, USA

Lisp was invented 60 years ago. Coral Common Lisp, the ancestor of today's Clozure Common Lisp, was released over 30 years ago.

Over this time, processor architectures and operating systems have come and gone, but Clozure CL (under various names and forms) has survived and is still with us today.

Clozure CL, Common Lisp, and Lisp itself are the product of many intelligent and clever people. Indeed, we find ourselves saying, with Newton, "If I have seen a little farther than others, it is because I have stood on the shoulders of giants."

I will say a few words, looking down from the giant's shoulders, on the subject of Clozure CL, that old Lisp, including where it stands today, and how it might evolve in the future.

R. Matthew Emerson currently leads the development and maintenance of Clozure Common Lisp, a free (Apache 2.0-licensed) Common Lisp implementation. Formerly an employee of the Common Lisp consulting company Clozure Associates, he now works on Clozure CL independently.

Event Detection in Unstructured Text (using Common Lisp)

Jason Cornez, Spain

At RavenPack, we use Common Lisp to extract meaning from unstructured English text. The focus is low-latency processing of real-time news feeds and blogs, to provide actionable intelligence to our clients in the financial industry. This talk discusses our technology for detecting events. We look at what we've done so far, what we are working on now, and some future possibilities.

Jason joined RavenPack in 2003 and is responsible for the design and implementation of the RavenPack software platform. He is a hands-on technology leader, with a consistent record of delivering break-through products.

A Silicon Valley start-up veteran with 20 years of professional experience, Jason combines technical know-how with an understanding of business needs to turn vision into reality. Jason holds a Master's Degree in Computer Science, along with undergraduate degrees in Mathematics and EECS from the Massachusetts Institute of Technology.

Session I: Performance

Petalisp: A Common Lisp Library for Data Parallel Programming

Marco Heisig
FAU Erlangen-Nürnberg
Cauerstraße 11
Erlangen 91058, Germany
marco.heisig@fau.de

ABSTRACT

We describe the design and implementation of Petalisp — a Common Lisp library for data parallel programming. At its core, Petalisp is a lazy, functional array language. All its statements are analyzed, simplified and compiled at runtime. This approach limits expressive power and introduces significant overhead, but also unlocks unprecedented potential for optimization.

We explain Petalisp from a users' perspective, compare its performance with other libraries for data parallel computation and finally discuss important facets of our implementation.

CCS CONCEPTS

•Software and its engineering →Parallel programming languages; Distributed programming languages; *Functional languages; Data flow languages; Just-in-time compilers;*

KEYWORDS

High Performance Computing, Common Lisp, Compilers, SIMD

ACM Reference format:
Marco Heisig. 2018. Petalisp: A Common Lisp Library for Data Parallel Programming. In *Proceedings of the 11th European Lisp Symposium, Marbella, Spain, April 16–17 2018 (ELS'18),* 8 pages.
DOI:

1 INTRODUCTION

For the last 50 years, the performance of our computers has doubled roughly each 20 months — an effect known as Moore's law. However, Gordon Moore's original prediction in 1965 was never about performance, but about the complexity of integrated circuits. In other words, the transistor count and resulting complexity of our computers has grown exponentially for more than 50 years. Translating this transistor count into performance is not for free. It led to the introduction of superscalar execution, speculative execution, vector instructions, caches, out-of-order execution, simultaneous multithreading, multicore CPUs, cache coherent NUMA domains, hybrid hardware and distributed systems. We expect to see even more such technologies, now that our semiconductor manufacturing processes start to hit physical limits.

Each of these technologies places a burden on the performance-aware programmer. The time it takes to develop efficient, parallel software grows at a steady pace. Today, developing a physics simulation to efficiently utilize a parallel machine often takes longer than a PhD thesis. The consequence is that many important scientific problems are not solved due to a lack of software.

To address this important issue, we propose a programming model where the software sacrifices a portion of its resources at runtime to remove the burden of parallel programming entirely from the user. In this model, it must be possible to reliably predict the complexity of future tasks and optimize the schedule, memory layout and code accordingly. The goal is not to develop another general purpose programming language, but to provide a special purpose tool for structured, inherently parallel programs. In its domain, it should rival human expert programmers in skill, but act in a timeframe of microseconds. The result of these considerations is the library Petalisp[1].

2 PREVIOUS WORK

The idea to develop specialized programming models for data parallel programming is not new. There are dedicated array languages, such as APL (Iverson 1962) and its descendants, data parallel extensions for imperative languages, such as High Performance Fortran (Forum 1997), Fortran coarrays (Reid 2010) and CUDA (Nickolls et al. 2008) and fully developed general purpose languages with particular support for parallel computing, such as SISAL (McGraw et al. 1983), ZPL (Snyder 2007), NESL (Blelloch et al. 1994), SAC (Grelck and Scholz 2007), SequenceL (Nemanich et al. 2010), Chapel (Chamberlain et al. 2007), X10 (Charles et al. 2005) and Fortress (Steele 2006). The Lisp language alone spawned plenty of research on parallel computing, such as Qlisp, Multilisp, PaiLisp, Concurrent Scheme and Parcel (Ito 1990), as well the data parallel Lisp dialects *Lisp (Massar 1995) and Connection Machine Lisp (Steele and Hillis 1986).

Our work has been influenced by the design decisions and experiences with most of these tools, but also differs in some significant aspects. The pivotal difference is that Petalisp only ever generates and compiles code at runtime. This increases the compiler's knowledge far beyond what any ahead-of-time compiler can hope to achieve. Our main challenge will be to keep the resulting runtime overhead within reasonable bounds.

3 USING PETALISP

Data structures are a fundamental concept in computer science. In classical Lisps, this role is filled by the `cons` function. While elegant, this approach produces data structures that are far too heterogeneous for any automatic parallelization. Instead Petalisp operates exclusively on strided arrays. Strided arrays are an extension

ELS'18, Marbella, Spain
978-2-9557474-2-1...$15.00
DOI:

[1] www.github.com/marcoheisig/Petalisp

of classical arrays, where the valid indices in each dimension are denoted by three integers: The smallest admissible index, the step size and the highest admissible index. More precisely, we define strided arrays as:

Definition 1 (strided array)**.** A strided array in n dimensions is a function from elements of the cartesian product of n ranges to a set of Common Lisp objects.

Definition 2 (range)**.** A range with the lower bound x_L, the step size s and the upper bound x_U, with x_L, s, $x_U \in \mathbb{Z}$, is the set of integers $\{\, x \in \mathbb{Z} \mid x_L \le x \le x_U \ \land \ (\exists k \in \mathbb{Z})\, [x = x_L + ks] \,\}$.

As a convenience feature, Common Lisp arrays and scalars are automatically converted to Petalisp strided arrays when necessary, rendering the distinction almost invisible to the user.

Working exclusively with strided arrays allows us to perform many domain-specific optimizations that are not possible in the general case. Our philosophy is that a reliable tool for a narrow domain is more useful that a mediocre general-purpose library. Besides, working with arrays is a relatively common case in many disciplines, such as image processing, data analysis or physics simulation. Finally, we want to point out that strided arrays may contain objects of any type, e.g. conses or hash tables.

3.1 Index Spaces

In order to denote strided arrays of a particular size, or to select a subset of the values of a strided array, we introduce index spaces as tangible objects. Index spaces can be created using the notation shown in figure 1.

```
(σ)                               ; the zero-dimensional space
(σ (0 1 8) (0 1 8))               ; index space of a 9 × 9 array
(σ (0 8) (0 8))                   ; ditto
(σ (10 2 98))                     ; all even two-digit numbers
(σ (x0 2 xN))                     ; using variable bounds
(σ (1 2 3) (1 2 3) (1 2 3))       ; corners of a 3 × 3 × 3 cube
```

Figure 1: A notation for strided arrays index spaces.

3.2 Transformations

A crucial difference between Petalisp and many other parallel programming models is that the motion of data is restricted to affine-linear transformations and permutations. This design strikes a balance between the expressiveness on the one hand and the need to perform accurate code analysis on the other hand. To denote such transformations, we introduce transformations themselves as instances of a subclass of `funcallable-standard-object` and choose a notation for them that is deliberately similar to a lambda form. The body forms of a transformation may contain arbitrary Common Lisp code, as long as the net effect is that of an affine-linear operator[2]. Examples of transformations are given in figure 2.

```
(τ () ())            ; mapping the empty space to itself
(τ (i) ((+ i a)))    ; shifting all indices by a
(τ (i) ((* 2 i)))    ; doubling all indices
(τ (i j) (j i))      ; switching 1st and 2nd dimension
(τ (i j) (i j 2))    ; increasing the dimension
(τ (i 1) (i))        ; decreasing the dimension
```

Figure 2: A notation for affine transformations.

Affine-linear transformations are automorphisms with many desirable properties: They can be stored using only three rational numbers per dimension and the composition and inverse of such transformation always exists and is again affine-linear. This knowledge permits our implementation to perform surprisingly smart code transformations later on.

3.3 Moving Data

The most important tool for data creation and movement is the `->` function[3]. It takes any array or scalar as its first argument and produces a new data structure according to the supplied transformations and index spaces. Each index space is either interpreted as a selection of a subset, or as a broadcasting operation — depending on whether it is smaller or larger of the current index space.

```
(-> 0 (σ (0 9) (0 9)))  ; a 10 × 10 array of zeros
(-> #(2 3) (σ (0 0)))   ; the first element only
(-> A (τ (i j) (j i)))  ; transposing A
```

Figure 3: Using the -> function.

Two further functions are provided to combine the values of several arrays into a single one. The `fuse` function creates an array containing the combined values of several supplied arrays. This requires that the supplied arrays are non-overlapping and that the union of their index spaces is again a strided array index space. The `fuse*` function is almost identical, yet permits overlapping arguments, in which case the values of the rightmost arguments are chosen. Usage of these functions is illustrated in figure 4.

```
(defvar B (-> #(2) (τ (i) ((1+ i)))))

(fuse #(1) B)      ; equivalent to (-> #(1 2))
(fuse #(1 3) B)    ; an error!
(fuse* #(1 3) B)   ; equivalent to (-> #(1 2))
(fuse* B #(1 3))   ; equivalent to (-> #(1 3))
```

Figure 4: Using `fuse` and `fuse*`.

3.4 Two Types of Parallelism

Up to now, we have shown how strided arrays can be created, moved and combined, but not how to operate on them. Inspired by CM-Lisp (Steele and Hillis 1986), we provide a function α to apply

[2] There is no magic involved here — we evaluate the forms multiple times to uniquely determine all coefficients of the transformation. With some further evaluations, we can even detect and report most functions that are not affine-linear.

[3] This function name seems hardly ideal. We welcome any constructive discussion regarding the API of Petalisp.

an *n*-ary function to *n* supplied data structures and a function β to reduce the elements of a strided array along its last dimension with a given binary function, as seen in figure 5. These functions are essentially the parallel counterparts of `map` and `reduce`. A small, but crucial difference is that the functions passed to α and β must be referentially transparent.

```
(α #'+ 2 3)      ; adding two numbers
(α #'+ A B C)    ; adding three arrays element-wise
(α #'sin A)      ; the sine of each entry

(β #'+ #(2 3))   ; adding two numbers
(defvar B #2A((1 2 3) (4 5 6)))
(β #'+ B)        ; summing the rows of B
```

Figure 5: The functions α and β.

3.5 Triggering Evaluation

The observant reader might have noticed that, so far, we have not shown any results of a call to a Petalisp function. This is where the lazy, functional nature of the language comes into play. Each result is an instance of a subclass of `strided-array`, whose dimension and element type are known, but whose values have not been computed. This behavior is illustrated in figure 6.

```
   (-> #(1 2 3))
=> #<strided-array-immediate #(1 2 3)>

   (α #'cos #(4 5 6))
=> #<strided-array-application t (σ (0 1 2))>

   (-> 1 (σ (2 2 8)))
=> #<strided-array-reference bit (σ (2 2 8))>
```

Figure 6: Petalisp calls return unevaluated strided arrays.

The reasoning behind this lazy semantics is that the high-level behavior of parallel algorithms is often independent from the contents of the data structures they manipulate. Whenever actual array values are required, their evaluation must be triggered explicitly. To do so, we provide a function `compute` to force evaluation and return a Common Lisp array with the contents of the given strided array. For ease of use, any array strides and starting indices are stripped in the process and zero dimensional arrays are converted to corresponding scalars. Usage examples are shown in figure 7.

```
(compute (-> 0.0 (σ (0 1))))      => #(0.0 0.0)
(defvar A #(1 2 3))
(compute (-> A))                  => #(1 2 3)
(compute (β #'+ A))               => 6
(compute (-> A (τ (i) ((- i)))))  => #(3 2 1)
```

Figure 7: Using the `compute` function.

```
(β #'+
   (α #'*
      (-> A (τ (m n) (m 1 n)))
      (-> B (τ (n k) (1 k n)))))
```

Figure 8: The matrix multiplication of matrices A and B.

As an optimization, Petalisp also features another function — `schedule` — that takes any number of strided arrays as arguments and returns immediately. Its sole purpose is to hint that these values should be computed asynchronously. This way, it is possible to overlap ordinary Lisp computation and Petalisp computation in many cases.

3.6 Example: Matrix Multiplication

The preceding sections exhaustively describe the API of Petalisp. To increase confidence that these few functions and macros are indeed sufficient to denote complex programs, we do now describe an implementation of the product *C* of two matrices *A* and *B*, according to the following definition:

$$C_{ij} = \sum_{p=1}^{n} A_{ip} B_{pj} \tag{1}$$

The equivalent Petalisp program is given in figure 8. It starts by reshaping *A* and *B* to three dimensional arrays of size $m \times 1 \times n$ and $1 \times k \times n$, respectively. Then it relies on the implicit broadcasting of α to obtain a $m \times k \times n$ cube of the products of the elements of *A* and *B*. It then uses the β function to sum the elements of the last dimension of this cube to obtain the $m \times k$ result matrix *C*. While this notation is hardly intuitive, it perfectly captures the data parallel nature of the problem and is almost as short as the original definition.

4 IMPLEMENTATION

4.1 Evaluation of Petalisp Programs

The evaluation of Petalisp programs consists of two phases. In the first phase, ordinary Common Lisp code calls Petalisp API functions to define strided arrays. Each strided arrays is an ordinary CLOS object that tracks how its elements *could* be computed from the values of other strided arrays. In the second phase, the evaluation of certain strided arrays is initiated by a call to `compute` or `schedule`. From this point on, we exploit the fact that each strided array and its inputs can also be viewed as nodes in a data flow graph. Our implementation differentiates between five different kinds of nodes:

- **immediate** Values of these nodes can be accessed in constant time, e.g. because they are internally stored in a Common Lisp array.
- **application** These nodes represent the element-wise application of a function of *n* arguments to *n* strided arrays.
- **reduction** These nodes represent the reduction of the last dimension of a given strided array with some binary function.

- **fusion** A fusion node represents a strided array containing all the values of a given set of non-overlapping strided arrays.
- **reference** These nodes represent a selection of a subset of the values of a strided array, a transformation of the index space of a strided array, or a broadcasting operation.

These five kinds of nodes exhaustively define the intermediate language that is passed to our compiler. We further impose the restriction on the user, that all Common Lisp functions that are used to construct application or reduction nodes must be referentially transparent. The result is a dream come true for any compiler writer — our intermediate language has only five statements, no control flow and is far from Turing-complete. An example of such a data flow graph is given in figure 9.

The price we have to pay for this simplicity is that each desired operation must be constructed, compiled and executed at runtime. Furthermore there is, at least conceptually, no reuse of compiled code. Each program is immediately discarded after the evaluation. And finally, each conditional statement that depends on the value of a strided array introduces significant latency — at least as long as it takes to return the value to Common Lisp, perform the test, assemble a new Petalisp program and compile it.

Given these considerations, we see that the latency and throughput of our compiler completely determines the applicability of our method. In the next subsections, we will discuss how we deal with this challenge.

4.2 Type Inference

We implemented a simple type inference engine to estimate the return type of known functions. Known functions are these from the `CL` package and functions that have been explicitly introduced by the user. Because we only ever deal with data flow graphs, inferring the element type of each node can be done directly during node creation, using the type information of its inputs. Our type system is pragmatic in that it only considers those types that can be represented as a specialized array by the host Common Lisp implementation. Luckily, this usually includes floating-point numbers and complex floating-point numbers, which are particularly relevant to numerical applications.

4.3 Data Flow Graph Optimization

Conceptually, the API functions `α`, `β`, `fuse`, `fuse*` and `->` allocate one or more data flow nodes to express the relation between the values of their inputs and output. To reduce the number of data flow node allocations and also the size of the programs passed to the compiler, we perform many optimizations already at node creation time. In particular, we perform the following optimizations:

- Consecutive reference nodes can be combined into a single reference node, using the functional composition of their transformations and by determining the appropriate transformed subspace. This is unconditionally possible, because all Petalisp transformations are known to be affine-linear.
- Reference nodes that neither transform, nor broadcast, nor select only a subset of their input are replaced by their input.
- Fusion nodes with a single input are meaningless and can be replaced by their input.
- Application nodes whose inputs are immediate nodes with only a single element are eagerly evaluated.

One consequence of these optimizations is that there can never be more than a single consecutive reference node. Or put differently: User code can use any level of indirection, like switching from zero-based arrays to one-based arrays or using different coordinate systems, without any performance penalty.

4.4 Kernel Creation

In our terminology, a kernel is a computation that is identically applied to all elements of a certain iteration space. Each kernel can reference any number of arrays and write to a single target array. The address computation of each array reference in a kernel must be an affine-linear function of the current point in the iteration space. A trivial way to obtain such kernels would be to turn each application, reduction and reference node into a kernel and each fusion node with n inputs into n kernels. However, the performance of this approach would be abysmal — each function call would reside in its own kernel and the result of each operation would have to be written to main memory.

Our challenge is now twofold: We want to determine kernels of maximal size and we want this algorithm to be extremely efficient. The size of the kernel is important, because data within a kernel can be passed via CPU registers instead of main memory. The kernel creation efficiency is a concern because the partitioning of data flow problems into kernels is a non-parallelizable bottleneck of every Petalisp program.

The first step of our algorithm is the detection of *critical nodes*. In our terminology, a critical node is a node that is referenced by multiple other nodes, or appears as the input of a broadcast operation. These nodes are the only ones that must be stored in main memory to avoid redundant computation. We obtain these nodes with a single traversal of the data flow graph, using a hash table to keep track of the users of each node.

Once the set of critical nodes has been found, we know that all remaining nodes have at most one user. As a result, each critical node, together with all its inputs up to the next critical node, form a tree. Figure 9 is an example of such a tree, where the only critical nodes are the immediate nodes and the final fusion node. Each such tree can be further simplified by lifting all reference nodes upwards until they reach the leaf nodes and merging them into a single one. Additionally, all fusion nodes can be eliminated by splitting the space of the target node into suitable fragments. Each of these fragments is now the root of a tree whose only interior nodes are application and reduction nodes and where each leaf is a single reference to a critical node.

In the final step, each fragment space of each critical node is turned into its own kernel. For the example in figure 9, we obtain three kernels, corresponding to the three inputs of the fusion node. The first two kernels simply copy boundary values from some other immediate. The third kernel, that is executed for all interior points of the grid, computes the sum of four references to some other array, multiplies it by 0.25 and stores it at the respective place in the target array.

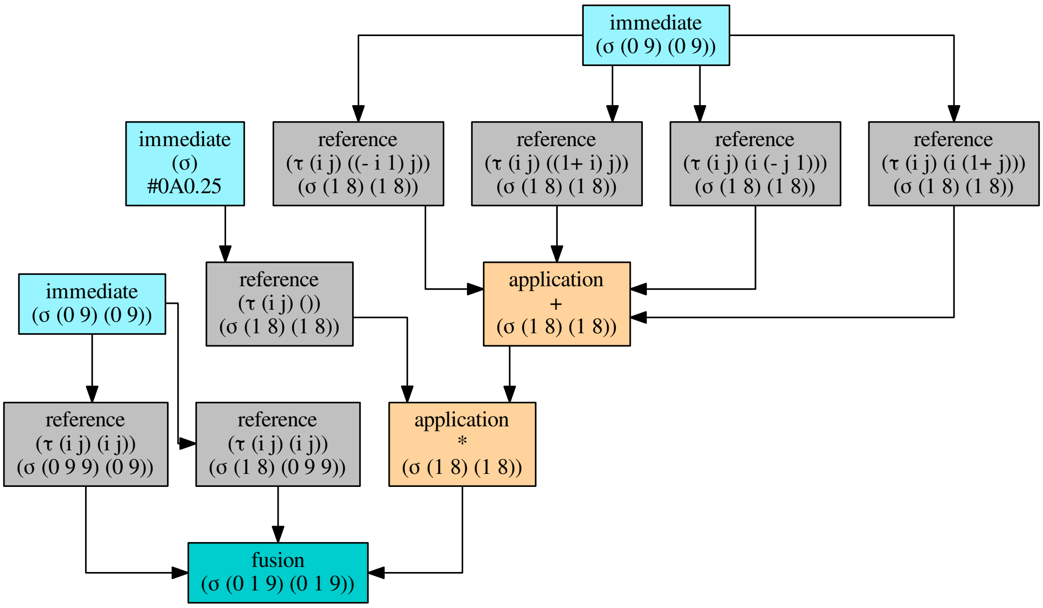


Figure 9: The data flow graph of a single Jacobi iteration on a 10×10 grid.

4.5 Modular Backends

Up to this point, all considerations were independent of the target hardware. But for the next steps — scheduling and code generation — the available resources are significant. Modeling these resources is a challenging problem. Modern hardware is increasingly heterogeneous, with multiple cores, sockets and special purpose accelerators. This problem is amplified in the case of distributed computing.

Our solution is to introduce a flexible, CLOS-based protocol for execution contexts. Every call to `schedule` or `compute` occurs in the context of a particular execution context we call *backend*. A backend is a CLOS object, featuring a single generic function as entry point. This method receives a list of nodes and turns them asynchronously into immediate nodes. It returns a request object to wait for the asynchronous computation to finish.

This approach is essentially a variant of Context-oriented programming (Hirschfeld et al. 2008). Petalisp programs are always executed in the context of a particular backend. One of the benefits of modeling the target platform this way is that functionality like the scheduling algorithm or the caching of compiled code can be shared between all backends by means of inheritance.

Our implementation contains already several specialized backends. In particular, we provide a slow, obviously correct reference backend and a fast backend that generates optimized Lisp code and uses the Lisp compiler of the host system. The latter has been used to conduct the performance measurements in section 5.

4.6 Scheduling

Once a data flow graph has been partitioned into kernels and kernel targets, it is passed to a particular backend for execution. The job of the backend is now to derive a valid order of execution and memory allocation scheme.

At the moment, the scheduling strategy of our implementation is to evaluate each critical node in depth-first order of dependencies. This simple technique is sufficient for many iterative algorithms, where there is only a single, long dependency chain. We intend to use a more sophisticated scheme in the future. For memory management, we use a memory pool, where each allocation first checks whether an array of suitable size and element type is already in the pool and allocates one otherwise. Then we use a reference counter for each array and add it back to the memory pool once this counter reaches zero. This scheme excels for algorithms with many same-sized intermediate arrays, but is wasteful in the general case.

4.7 Kernel Evaluation

The final step in the evaluation of a Petalisp program is the evaluation of individual kernels. At this point, data dependencies and memory management have already been taken care of by the scheduler and the only remaining task is to generate fast code for each kernel, compile it, and execute it for the given iteration space. Given that kernels are the smallest unit of work in our system, it seems prohibitively expensive to generate and compile code on each invocation. Mitigating this problem is a key concern that decides whether our evaluation model is viable or not.

We solve this problem by using aggressive memoization. From each kernel, we extract all information we need to generate fast code, i.e. the approximate size of the iteration space, the names of operators with inline information and the relative offsets and element types of each array reference. We call this structure a *recipe*. Each recipe is stored using a specialized variant of hash consing, such that similar recipes share most of their structure and such that identical recipes are eq to each other. The recipe of a kernel is then used as sole input to the code generator, which caches its compiled results in a hash table. Apart from the first use of a recipe, evaluation consists of a single lookup in an eq hash table to obtain the compiled recipe and the application of that compiled recipe to the correct iteration space, inputs and non-inlined function handles.

4.8 Code Optimization

If the recipe of a kernel is not found in the code generator cache, the backend has to generate, compile and cache such code. Compiling and caching are trivial in Common Lisp, because the language standard provides functions for both tasks (compile and gethash). It remains to generate efficient Common Lisp code for a particular problem. Because kernel recipes are deliberately similar to S-expressions, a straightforward translation can be done using only a few lines of code. Unfortunately this task is complicated by the limitations of the freely available Common Lisp compilers that we use (CCL and SBCL). Neither of these compilers emits efficient address computations for references to multi-dimensional arrays.

The problem with multi-dimensional address computation is that we want the compiler to move as much as possible of the index computation of aref outside of the innermost loop. Our solution is to perform this lifting of loop invariant code ourselves. We directly emit calls to row-major-aref with explicit address computation. We then determine for each subexpression the outermost loop on which it depends, move the expression to this loop, bind its value to a temporary variable and use this temporary variable instead. While doing so, we also perform common subexpression elimination where possible. The result is that we emit Lisp code that is often considerably faster than manually written code.

As a final optimization, we parallelize the outermost loop of each kernel whose iteration space exceeds a certain size. To do so, we use the library lparallel. The decision procedure when and how to parallelize is, in its current state, far from ideal and more intended as a proof of concept. Nevertheless it can significantly accelerate computations on large domains.

5 PERFORMANCE

In the previous sections, we have shown that Petalisp offers a high level of abstraction, but with a potentially high runtime overhead. In this section, we want to quantify this overhead and compare the performance of Petalisp code with other well known technologies.

5.1 Jacobi's Method

The benchmark program that we will use throughout this section is a simple variant of Jacobi's method. It is an iterative algorithm for solving elliptic partial differential equations. We will not discuss its mathematical properties in detail. For the following considerations, it is sufficient to note that it produces a sequence of grids of identical size, where the value of each interior grid point is the average of the values of its four neighboring points in the previous grid. In terms of computation, it consists of an update rule, where for each interior point, four values are loaded, the sum is computed with three additions, then turned into the average by one multiplication and finally stored at the current position.

```
(defun jacobi (u iterations)
 (let ((it ; the interior
        (σ* u ((+ start 1) step (- end 1))
              ((+ start 1) step (- end 1)))))
  (loop repeat iterations do
   (setf u
    (fuse* u
     (α #'* 0.25
       (α #'+
          (-> u (τ (i j) ((1+ i) j)) it)
          (-> u (τ (i j) ((1- i) j)) it)
          (-> u (τ (i j) (i (1+ j))) it)
          (-> u (τ (i j) (i (1- j))) it)))))
   finally (return u))))
```

Figure 10: Jacobi's method in Petalisp.

```
def jacobi(src, dst):
    dst[1:-1, 1:-1] = \
        0.25 * (  src[0:-2,1:-1]
                + src[2:,1:-1]
                + src[1:-1,0:-2]
                + src[1:-1,2:] )
    return dst

for i in range(iterations // 2):
    jacobi(src, dst)
    jacobi(dst, src)
```

Figure 11: Jacobi's method in Python.

5.2 The Benchmark Setup

We implemented the same algorithm using three different techniques. Our first implementation uses Petalisp on SBCL 1.3.21. It is shown in figure 10. The second implementation (figure 11) uses Python 3.5.2 and the numerics library NumPy, version 1.11. The third implementation (figure 12) is written in C++ and compiled with GCC version 5.4 and highest optimization settings. It serves upper bound of we can hope to achieve some day. Our benchmark system is an Intel Xeon E3-1275 CPU, running at 3.6GHz.

To ease comparison, each implementation is run single-threaded. Furthermore, we measure two variants of the C++ code. Both are compiled with highest optimization settings (-O3), but only one of them uses native optimizations(-march=native). We do this, because we aim to reach the performance of the non-native code in the near future, while reaching the other variant will require some changes to SBCL itself, especially adding support for AVX2 operations.

5.3 Benchmark Results

The results of our benchmarks are given in figure 13. We measured the floating point operations per second for different problem sizes,

```

void jacobi(size_t h, size_t w,
            double* src,
            double* dst) {
    for(size_t ih = 1; ih < h-1; ++ih) {
        for(size_t iw = 1; iw < w-1; ++iw) {
            size_t i = ih * w + iw;
            dst[i] = 0.25 * (  src[i+1]
                             + src[i-1]
                             + src[i+w]
                             + src[i-w]);}}}

for(size_t i = 0; i < iterations/2; ++i) {
    jacobi(w, h, src, dst);
    jacobi(w, h, dst, src);}
```

Figure 12: Jacobi's method in C++.

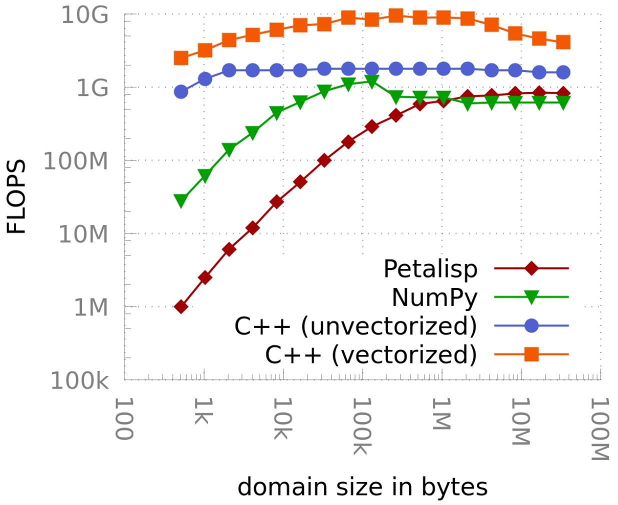


Figure 13: Single-threaded floating point operations per second for Jacobi's method on a square domain.

starting from a grid with 8 × 8 double precision floating point values, increasing the number of grid points by powers of two, until the problem domain started to exceed the cache size of our CPU.

On a first glance, the performance of Petalisp seems disappointing in comparison. However, NumPy and C++ are widely used tools with decades of optimization under the hood, while Petalisp is in an early stage of development. The numbers shown here are some of the first Petalisp benchmarks ever conducted. Our current implementation is naive in many respects and would benefit a lot from improved scheduling, inter-kernel optimization and vectorization. Nevertheless, Petalisp already manages to outperform NumPy (which calls optimized C code internally), and lands within 50% of non-vectorized C++ code. In this light, we are extremely satisfied with these early performance results.

The only case where Petalisp performs much worse is for small domains. This is not surprising, given the constant overhead of about 200 microseconds per Jacobi iteration just to analyze and schedule the code (a number we hope to decrease in the future). But small domains are not the focus of our work. We are after the large problems, where this constant overhead amortizes quickly. And indeed our benchmarks confirm that we reach this point already at problems of about one megabyte in size.

6 CONCLUSIONS AND FUTURE WORK

We have presented a new programming technique, where a purely functional programming language with inherent parallelism and lazy semantics is integrated into the existing general purpose language Common Lisp. We have implemented said approach and studied its applicability to several real-world problems. Thereby, we have made the following pleasant observations:

- Our compilation strategy is feasible. In order to have maximal knowledge about the dynamic state of each computation, we defer compilation of compute-intensive parts until the very last moment. Yet by using efficient algorithms, asynchronous compilation and different kinds of memoization, we manage to compile and execute more than 10^5 parallel Petalisp instructions per second, with a latency of less than 100 microseconds.
- By moving all optimization and analysis to the runtime, our compiler has total knowledge. It can predict the exact number of loads and stores, which instructions are used and what bottlenecks may arise. Furthermore, it has unprecedented freedom in the choice of memory layouts and execution strategy. This is an enormous strategic advantage over other optimizing compilers.
- With about 5000 lines of code, our implementation is — with respect to the complexity of the task — simple and maintainable. We attribute this to the decision to integrate Petalisp tightly into Common Lisp and building on existing libraries and infrastructure where possible.
- Even in the current, early stage, our system is able to outperform the existing numerics framework NumPy.
- Our programming model leads to a clean separation between description and execution. Petalisp reliably normalizes every description of a particular algorithm to the same intermediate representation, before determining a reasonable execution strategy for it. The programmer can focus entirely on correctness and clarity.

The last of these points — the separation of concerns — is the most dear to us. In many fields of computer science, the need for high performance encourages programmers to mix optimization and description. Many of these optimizations, e.g. special purpose memory layouts, are pervasive. They affect many functions and greatly increase the cognitive burden on the developer. Our long term goal is to make Petalisp a viable alternative in these cases, by achieving competitive performance to hand-tuned applications.

A lot of work remains until the full potential of this programming model is unlocked. In particular we intend to add the following features over the next few years:

- **Performance Modeling** Before doing further improvements to the code generator and scheduler, we need a way to determine the performance characteristics of a particular computation. In particular we care whether some code is memory bound or compute bound. We intend to use the Roofline (Williams et al. 2009) or ECM (Stengel et al. 2015) model.
- **Sophisticated Scheduling** Currently, we evaluate each array in depth-first fashion according to the data dependency graph. The new scheduler should utilize the results of the performance model to determine an order of execution and to change memory layouts to optimize locality. In particular, we want to be able to apply temporal blocking on arbitrary memory-bound computations.
- **Improved Shared-Memory Parallelization** Instead of naively parallelizing the outermost loop of a computation, we want to base the division of labor on the memory access patterns of a computation and on the result from the performance analysis.
- **Distributed Memory Parallelization** The potential for automatic parallelization in Petalisp is not limited to a single node. Indeed, the whole system has been carefully designed to permit concurrent execution on multiple machines. We intend to use the message passing standard MPI for distributed communication.
- **Vectorization** Once performance analysis detects that a particular step is compute bound, our code generator should use vectorized instructions when possible.
- **GPU Offloading** We are not aware of any existing technique to run arbitrary Common Lisp functions on GPUs. Nevertheless we could determine the subset of Petalisp kernels that use only primitive hardware operations and offload their evaluation to the GPU. Again, the capability of Petalisp to exactly predict the cost of its operations can help to make qualified decisions when such offloading is profitable.

The name "Petalisp" emphasizes our commitment to provide an enjoyable programming model for petascale systems, i.e. systems able to execute around 10^{15} operations per second. With our current results ranging between 10^8 and 10^{10} operations per second, we still have a long way to go. Yet the capabilities of our system so far make us confident that we may eventually succeed.

7 ACKNOWLEDGMENTS

We are grateful for the support and feedback from Nicolas Neuss, Matthias Bauer, Pascal Costanza, Sebastian Kuckuk, Jonas Schmitt and Harald Köstler. Furthermore we want to thank all maintainers of Common Lisp packages and developers of Common Lisp implementations. This work would not have been possible if not for the excellent quality of the Common Lisp ecosystem.

REFERENCES

Guy E. Blelloch, Siddhartha Chatterjee, Jonathan C. Hardwick, Jay Sipelstein, and Marco Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, April 1994.

B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007. doi: 10.1177/1094342007078442. URL http://dx.doi.org/10.1177/1094342007078442.

Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40 (10):519–538, October 2005. ISSN 0362-1340. doi: 10.1145/1103845.1094852. URL http://doi.acm.org/10.1145/1103845.1094852.

High Performance Fortran Forum. High performance fortran language specification, 1997.

Clemens Grelck and Sven-Bodo Scholz. Sac: Off-the-shelf support for data-parallelism on multicores. In *Proceedings of the 2007 Workshop on Declarative Aspects of Multicore Programming*, DAMP '07, pages 25–33, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5. doi: 10.1145/1248648.1248654. URL http://doi.acm.org/10.1145/1248648.1248654.

Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, March 2008. ISSN 1660-1769. doi: 10.5381/jot.2008.7.3.a4. URL http://www.jot.fm/contents/issue_2008_03/article4.html.

Robert H. Jr. Ito, Takayasu und Halstead. *Parallel Lisp: Languages and Systems: US/Japan Workshop on Parallel Lisp, Sendai, Japan, June 5-8, 1989, Proceedings (Lecture Notes in Computer Science)*. Springer, 1990. ISBN 3540527826.

Kenneth E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA, 1962. ISBN 0-471430-14-5.

J.P. Massar. Starsim: Thinking machines' *lisp simulator. http://www.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/lisp/impl/starlisp/0.html, 1995. Accessed: 2018-02-08.

J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, I. Dobes, and P. Hohensee. *SISAL: streams and iteration in a single-assignment language. Language reference manual, Version 1. 1.* Jul 1983.

Brad Nemanich, Daniel Cooke, and J. Nelson Rushtom. Sequencel: Transparency and multi-core parallelisms. In *Proceedings of the 5th ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming*, DAMP '10, pages 45–52, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-859-9. doi: 10.1145/1708046.1708057. URL http://doi.acm.org/10.1145/1708046.1708057.

John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with cuda. *Queue*, 6(2):40–53, March 2008. ISSN 1542-7730. doi: 10.1145/1365490.1365500. URL http://doi.acm.org/10.1145/1365490.1365500.

John Reid. Coarrays in the next fortran standard. *SIGPLAN Fortran Forum*, 29(2):10–27, July 2010. ISSN 1061-7264. doi: 10.1145/1837137.1837138. URL http://doi.acm.org/10.1145/1837137.1837138.

Lawrence Snyder. The design and development of zpl. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 8–1–8–37, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. doi: 10.1145/1238844.1238852. URL http://doi.acm.org/10.1145/1238844.1238852.

Guy L. Steele, Jr. Parallel programming and code selection in fortress. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 1–1, New York, NY, USA, 2006. ACM. ISBN 1-59593-189-9. doi: 10.1145/1122971.1122972. URL http://doi.acm.org/10.1145/1122971.1122972.

Guy L. Steele, Jr. and W. Daniel Hillis. Connection machine lisp: Fine-grained parallel symbolic processing. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, LFP '86, pages 279–297, New York, NY, USA, 1986. ACM. ISBN 0-89791-200-4. doi: 10.1145/319838.319870. URL http://doi.acm.org/10.1145/319838.319870.

Holger Stengel, Jan Treibig, Georg Hager, and Gerhard Wellein. Quantifying performance bottlenecks of stencil computations using the execution-cache-memory model. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 207–216, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3559-1. doi: 10.1145/2751205.2751240. URL http://doi.acm.org/10.1145/2751205.2751240.

Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, April 2009. ISSN 0001-0782. doi: 10.1145/1498765.1498785. URL http://doi.acm.org/10.1145/1498765.1498785.

Dynamic optimizations for SBCL garbage collection

Ethan H. Schwartz
Google
ethanhschwartz@gmail.com

ABSTRACT

Using a garbage collected language poses a challenge for latency-sensitive applications. The garbage collector is an integral part of a lisp implementation; optimizing or replacing the GC may be infeasible without a substantial rewrite of the lisp compiler/runtime. By taking advantage of the container which many modern processes run inside we can tune the garbage collector only to keep the lisp process's heap within the bounds provided by the container. We make 3 simple changes to SBCL's generational garbage collector which result in improved application performance. We find that the application's runtime throughput and latency are improved, with less time spent in the GC, and that behavior in a multi-core environment is improved.

CCS CONCEPTS

• **Software and its engineering → Garbage collection**; *Memory management*; *Runtime environments*; Virtual machines;

ACM Reference Format:
Ethan H. Schwartz. 2018. Dynamic optimizations for SBCL garbage collection. In *Proceedings of the 11th European Lisp Symposium (ELS'18).* ACM, New York, NY, USA, 3 pages. https://doi.org/

1 INTRODUCTION

During early startup of a lisp process, the Steel Bank Common Lisp (SBCL) GENerational Conservative Garbage Collector[1] (gencgc) uses `mmap()` to allocate the entire dynamic space that will be accessible for use by the lisp heap for the life of the process which accounts for the process's its virtual size (VSZ). This space is divided into equal size *cards* or *pages*, which represent the smallest unit of heap which can be allocated or garbage collected[2]. Each lisp object on the heap is assigned to one or more of these GC cards.

Once a lisp process has allocated (consed) a specified number of bytes the SBCL runtime halts all threads and triggers garbage collection. The GC transitively searches objects which are live, or presumed live (in the case of objects residing in an older GC *generation*), looking for references to other objects, which are thus enlivened. Surviving objects are copied from their existing GC card to a new card; objects which are not copied to a new card are garbage. An object which survives garbage collection may be promoted to an older GC generation, according to GC policy, meaning that it may be presumed live during a subsequent GC. Generations are numbered 0 to 5, with generation 0 holding the newest objects. When SBCL finds it needs to collect garbage from a sufficiently old generation (a *large* GC, typically: older than gen 1) it releases all unused GC cards to the kernel via `madvise()` (i.e. so they are no longer are part of the process's resident space (RSS)).

2 PROBLEM

SBCL's gencgc assumes several optimizations which make it very useful for a general purpose computing: objects typically become garbage in a roughly LIFO order; the existence of collectable garbage is typically correlated with having allocated new objects; the flexibility of having multiple generations allows the runtime to scale to an application which may have unpredictable memory use over time; and the GC can usually prevent the application from prematurely exhausting its preconfigured dynamic space size (e.g. due to fragmentation), which would cause the process to fail. However the constraints of the garbage collector make it suboptimal for a latency-sensitive application – especially one running on a modern platform with multiple processing cores and finite memory. One root of these problems is that application code cannot run concurrently with garbage collection. Compounding this problem, the runtime may trigger GC even if there is no garbage to collect or if system memory reserved for the application is not yet scarce, and while GC is running, the process utilizes only a single CPU core, even if the application is otherwise multithreaded and capable of utilizing many cores. System memory utilization is further compromised by the GC's behavior, which may not return memory to the kernel when memory reserved for the process is running out, and when the GC does `madvise()` a GC card back to the system, a subsequent write to the returned GC card requires the kernel to remap the page, which takes time.

3 IMPROVEMENTS

We can significantly improve the performance characteristics of the SBCL garbage collector with only very minor code changes to the gencgc implementation. We propose three optimizations. The first two optimizations cause gencgc to be more responsive to the host platform in which the lisp process runs. These optimizations directly benefit from use of a container API, used to manage the memory and processing resources of the application by providing virtual environment as opposed to "bare metal". The third optimization allows gencgc to take advantage of information from the application logic for some types of programs. The optimizations are best suited to applications that process logically independent work items (e.g. an RPC server or part of a data processing pipeline).

[1] SBCL also supports a precise or Cheney GC implementation on some platforms. This paper does not address that GC implementation.
[2] 32kB on x86-64

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1.

3.1 Optimization #1: Trigger GC based on host memory constraints

The first optimization is to use memory utilization metrics to indicate to the lisp runtime when collecting garbage is actually required to avoid an out of memory condition. This entails configuring up to two memory thresholds. For an application running within a container, these thresholds are most logically expressed as a fraction of the container's total memory size. When the container's memory in use crosses a specified threshold, the lisp process is notified of the impending need to collect garbage. These triggers may replace the standard behavior of beginning garbage collection after allocating a fixed number of bytes on the lisp heap.

At the first threshold, free memory is low and garbage should be collected soon. The application is signaled that it should stop processing additional work items. The application waits for outstanding work items to complete, performs garbage collection, and then resumes processing new work items. Deferring the GC until the process is idle means the GC pause does not adversely affect latency-sensitive processing.

At the higher threshold free memory is critical. The process runs GC immediately, even if it has work items in progress. The GC pause still imposes latency on concurrent work items, but the work items have an opportunity to complete before the process completely runs out of memory.

3.2 Optimization #2: Control when memory is given back to the system.

SBCL gencgc's conventional behavior is to release memory back to the system only after a *large* GC. After such a GC it releases the memory associated with all disused GC cards. This optimization changes gencgc's behavior to always release memory, but to only release a number of GC cards sufficient to bring resident memory below a specified threshold. As with optimization #1, if the application runs inside of a container, the GC aims to shrink resident memory below a given fraction of the container's total memory size. When gencgc has released enough GC cards to bring the resident memory size below the specified threshold, the container provides notification back to the lisp process that no further GC cards need to be released.

SBCL gencgc[3] normally loops through free pages sequentially from 0 up to the largest page index used, returning each contiguous block of pages to the operating system. This loop is reversed, such that the first card to be reused for subsequent heap allocations would be the last card released back to the system, and the loop is halted once enough pages have been released to bring the processes resident size down below the specified threshold.

The target number of total GC cards held by the runtime after garbage collection is allowed to float, as some of the process's resident space is consumed by memory other than the lisp heap. This approach allows the garbage collector to keep the application's resident size within acceptable limits, and avoids the penalty incurred by the kernel remapping the GC card memory, unless and until the lisp heap subsequently grows to exceed the previous threshold.

This optimization may be used in combination with optimization #1, which allows the garbage collector to run when and only when memory needs to be returned back to the system. Used together, and with appropriate thresholds configured, the two optimizations form a positive feedback loop: the GC runs only when too much memory is in use, and every GC reduces the amount of memory in use back to an acceptable level.

3.3 Optimization #3: A generational garbage collector aware of application request processing

Arena allocation or region-based memory management is an approach allowing a program to efficiently deallocate objects with a known lifetime. Lisp doesn't natively support arena allocation, but an application which processes logically independent work items can make use of SBCL's garbage collector's generation 0 to achieve some of the benefit of allocation against a single arena. When the GC runs while one or more work items are in progress, only generation 0 is scavenged, and no surviving objects are promoted to an older generation[4]. Heap objects which survive a gen 0 GC are presumed to be logically associated with in-progress work items and are expected to be garbage when those work items complete. When the GC runs while no work items are in progress, generation 1 is scavenged, and all surviving objects are promoted to gen 1. Since no work items are in-progress (e.g. the server is quiescent), surviving heap objects are assumed to be persistent or long-lived structures. For best results, this optimization may require changes to application code to not keep live references to obsolete state after the end of a request.

4 PUTTING IT ALL TOGETHER

Used in combination, these 3 optimizations yield a system that GCs only when it needs to release memory to the system, always releases memory to the system when resident space is too high, and never prematurely pessimizes the performance of future GC card use. At the high memory threshold the system waits for the application to become idle before issuing a slow (gen 1) GC. At the critical memory threshold the system performs an immediate fast (gen 0) GC. Objects on the heap are not promoted to an older generation when short-lived work items are known to be in progress.

5 RESULTS

The performance impact of each of these 3 optimizations depends on the particular application. Each application will have its own particular memory usage pattern under conventional GC, and its performance will suffer proportionally to how frequently it needs to collect garbage and how many threads are blocked from doing useful work because the garbage collector needs to run. The more total memory allowed for an application, the more optimization #1 will reduce unnecessary collections and therefore benefit the application's performance. Optimization #2 can benefit any application, but the magnitude of the benefit is bounded by the frequency with which the application needs to perform *large* GCs and the amount

[3] Specifically, `remap_free_pages()`.

[4] Controlled via `(sb-ext:generation-number-of-gcs-before-promotion 0)`.

of memory the application subsequently needs to reclaim afterwards. An application which never reaches a quiescent state will not benefit from optimization #3; this optimization is most beneficial to an application which is given enough memory to completely eschew garbage collections during work item processing.

5.1 Latency sensitive RPC server

A production low latency RPC server using a single service thread is able to process most queries without needing to collect garbage during any single query. The infrequent garbage collections therefore impact only the tail latency of the service. However increasing the parallelism of the process aggravates the latency imposed by garbage collection. The same server running with 9 service threads offers 9x the throughput, but also allocates into its heap 9x as fast, triggering garbage collection 9x as often, and with every concurrent RPC suffering the full latency penalty imposed by GC.

By enabling all 3 of the optimizations described here, we find that in the service running with 9 threads, tail latency at the 99.9th percentile is improved by by 65%, and 21% at the 95th percentile, with no adverse impact to overall throughput caused by the slower, less frequent GCs. The optimized server declines to process new inbound RPCs once it detects it must GC soon, and since it has enough memory to process a query on every service thread without GCing, it is able to defer effectively all GCs to time where the server is idle – GC does not impact tail latency. Each of the less frequent GCs takes longer, but there is no adverse effect on overall throughput.

5.2 SBCL Compilation

Whereas the benefit of optimizations #1 and #3 is very much particular to a given application, optimization #2 lends itself to a much more straightforward and concrete measurement. We use the familiar benchmark of the SBCL compiler repeatedly compiling itself. We compare the baseline performance of the SBCL self-build to two alternate configurations in order to demonstrate the efficacy of optimization #2. In both variations, the garbage collector is triggered using SBCL's conventional settings, and the peak observed size of the lisp heap is around 450MB.

The baseline uses gencgc's default behavior, which only returns GC cards when collecting generations older than gen 1.

In the first variation, gencgc is made to never release memory back to the system after GC. That is, the GC runs just as often as it does in the baseline, but the process's resident size never recedes from its peak. This configuration is 2.42% faster than the baseline. Faster GCs make up a tiny portion of the time savings, with the vast majority being attributable to obviating the work the kernel needs to do remap pages back to the application that the garbage collector had previously returned. This figure serves as an upper bound on the amount of savings we can expect from optimization #2.

The second variation employs the optimization in a configuration such that gencgc (still triggered by the conventional mechanisms) always releases memory to the system after every GC, but retains the lowest 250MB of lisp heap, regardless of the actual heap size. This configuration is 1.71% faster than the baseline. That is, optimization #2 preserves nearly three quarters of the performance benefit of not returning memory to the system, while keeping the process's resident size significantly below its peak.

Session II: Implementation

Incremental Parsing of Common Lisp Code

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

ABSTRACT

In a text editor for writing Common Lisp [1] source code, it is desirable to have an accurate analysis of the buffer contents, so that the role of the elements of the code can be indicated to the programmer. Furthermore, the buffer contents should preferably be analyzed after each keystroke so that the programmer has up-to-date information resulting from the analysis.

We describe an incremental parser that can be used as a key component of such an analyzer. The parser, itself written in Common Lisp, uses a special-purpose implementation of the Common Lisp `read` function in combination with a *cache* that stores existing results of calling the reader.

Since the parser uses the standard Common Lisp reader, the resulting analysis is very accurate. Furthermore, the cache makes the parser very fast in most common cases; re-parsing a buffer in which a single character has been altered takes only a few milliseconds.

CCS CONCEPTS

• **Applied computing → Text editing**; • **Software and its engineering → Syntax**; **Parsers**; **Development frameworks and environments**; **Integrated and visual development environments**; *Functional languages*; *Multiparadigm languages*;

KEYWORDS

Common Lisp

ACM Reference Format:
Irène Durand and Robert Strandh. 2018. Incremental Parsing of Common Lisp Code. In *Proceedings of the 11th European Lisp Symposium (ELS'18).* ACM, New York, NY, USA, 7 pages.

1 INTRODUCTION

Whether autonomous or part of an integrated development environment, an editor that caters to Common Lisp programmers must analyze the buffer contents in order to help the programmer understand how this contents would be analyzed when submitted to the Common Lisp compiler or interpreter. Furthermore, the editor analysis must be *fast* so that it is up to date shortly after each keystroke generated by the programmer. Miller [5] indicates that an upper bound on the delay between a keystroke and the updated result is around 0.1 seconds.

In order to obtain such speed for the analysis, it must be *incremental.* A complete analysis of the entire buffer for each keystroke is generally not feasible, especially for buffers with a significant amount of code.

Furthermore, the analysis is necessarily *approximate.* The reader macro `#.` (hash dot) and the macro `eval-when` allow for arbitrary computations at read time and at compile time, and these computations may influence the environment in arbitrary ways that may invalidate subsequent, or even preceding analyses, making an analysis that is both precise and incremental impossible in general.

The question, then, is how approximate the analysis has to be, and how much of it we can allow ourselves to recompute, given the performance of modern hardware and modern Common Lisp implementations.

In this paper, we describe an analysis technique that represents an improvement compared with the ones used by the most widespread editors for Common Lisp code used today. The technique is more precise than existing ones, because it uses the Common Lisp `read` function, which is a better approximation than the regular-expression techniques most frequently used. We show that our analysis is sufficiently fast because it is done incrementally, and it requires very little incremental work for most simple editing operations.

The work in this paper is specific to the Common Lisp language. This language has a number of specific features in terms of its syntax, some of which make it harder to write a parser for it, and some of which make it easier:

- The reader algorithm is defined in terms of a non-tokenizing recursive descent parser. This fact makes our task easier, because the result of calling `read` at any location in the source code is well defined and yields a unique result. For other languages, the meaning of some sequence of characters may depend on what comes much later.
- The Common Lisp reader is *programmable* in that the application programmer can define *reader macros* that invoke arbitrary code. This feature makes our task harder, because it makes it impossible to establish fixed rules for the meaning of a sequence of characters in the buffer. The technique described in this paper can handle such arbitrary syntax extensions.
- As previously mentioned, arbitrary Common Lisp code may be invoked as part of a call to `read`, and that code may modify the readtable and/or the global environment. This possibility

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1.

makes our task harder, and we are only able to address some of the problems it creates.

2 PREVIOUS WORK

In this section, we present a selection of existing editors, and in particular, we discuss the technique that each selected editor uses in order to analyze a text buffer containing Common Lisp code.

We do not cover languages other than Common Lisp, simply because our technique crucially depends on ability to analyse the buffer contents with a non-tokenizing (i.e, based on reading a character at a time) recursive descent parser. The Common Lisp `read` function is defined this way, but most other languages require more sophisticated parsing techniques for a correct analysis.

2.1 Emacs

GNU Emacs [2, 3] is a general-purpose text editor written partly in C but mostly in a special-purpose dialect of Lisp [4].

In the editing mode used for writing Common Lisp source code, highlighting is based on string matching, and no attempt is made to determine the symbols that are present in the current package. Even when the current package does not use the `common-lisp` package, strings that match Common Lisp symbols are highlighted nevertheless.

In addition, no attempt is made to distinguish between the role of different occurrences of a symbol. In Common Lisp where a symbol can simultaneously be used to name a function, a variable, etc., it would be desirable to present occurrences with different roles in a different way.

Indentation is also based on string matching, resulting in the text being indented as Common Lisp code even when it is not. Furthermore, indentation does not take into account the role of a symbol in the code. So, for example, if a lexical variable named (say) `prog1` is introduced in a `let` binding and it is followed by a newline, the following line is indented as if the symbol `prog1` were the name of a Common Lisp function as opposed to the name of a lexical variable.

2.2 Climacs

Climacs[1] is an Emacs-like editor written entirely in Common Lisp. It uses McCLIM [9] for its user interface, and specifically, an additional library called ESA [10].

The framework for syntax analysis in Climacs [6] is very general. The parser for Common Lisp syntax is based on table-driven parsing techniques such as LALR parsing, except that the parsing table was derived manually. Like Emacs, Climacs does not take the current package into account. The parser is incremental in that the state of the parser is saved for certain points in the buffer, so that parsing can be restarted from such a point without requiring the entire buffer to be parsed from the beginning.

Unlike Emacs, the Climacs parser is more accurate when it comes to the role of symbols in the program code. In many cases, it is able to distinguish between a symbol used as the name of a function and the same symbol used as a lexical variable.

2.3 Lem

Lem[2] is a relatively recent addition to the family of Emacs clones. It is written in Common Lisp and it uses `curses` to display buffer contents.

Like Emacs (See Section 2.1.), it uses regular expressions for analyzing Common Lisp code, with the same disadvantages in terms of precision of the analysis.

3 OUR TECHNIQUE

3.1 Buffer update protocol

Our incremental parser parses the contents of a buffer, specified in the form of a CLOS protocol[8]. We include a brief summary of that protocol here.

The protocol contains two sub-protocols:

(1) The *edit protocol* is used whenever items are inserted or deleted from the buffer. An edit operation is typically invoked as a result of a keystroke, but an arbitrary number of edit operations can happen as a result of a keystroke, for example when a region is inserted or deleted, or when a keyboard macro is executed.
(2) The *update protocol* is used when the result of one or more edit operations must be displayed to the user. This protocol is typically invoked for each keystroke, but it can be invoked less frequently if some view of the buffer is temporarily hidden. Only when the view becomes visible is the update protocol invoked.

This organization solves several problems with the design of similar protocols:

- The edit protocol does not trigger any updates of the views. The edit operations simply modify the buffer contents, and marks modified lines with a *time stamp*. Therefore the operations of the edit protocol are fast. As a result, complex operations such as inserting or deleting a region, or executing a complicated keyboard macro, can be implemented as the repeated invocation of simpler operations in the edit protocol. No special treatment is required for such complex operations, which simplifies their overall design.
- There is no need for the equivalent of *observers* as many object-oriented design methods require. Visible views are automatically updated after every keystroke. Each view contains a time stamp corresponding to the previous time it was updated, and this time stamp is transmitted to the buffer when the update protocol is invoked.
- Views that are invisible are not updated as a result of a keystroke. Such views are updated if and when they again become visible.

When a view invokes the update protocol, in addition to transmitting its time stamp to the buffer, it also transmits four *callback functions*. Conceptually, the view contains some mirror representation of the lines of the buffer. Before the update protocol is invoked, the view sets an index into that representation to zero, meaning the first line. As a result of invoking the update protocol, the buffer informs the view of changes that happened after the time indicated by the time stamp by calling these callback functions as follows:

[1] See: https://common-lisp.net/project/climacs/

[2] See https://github.com/cxxxr/lem.

- The callback function *skip* indicates to the view that the index should be incremented by a number given as argument to the function.
- The callback function *modify* indicates a line that has been modified since the last update. The line is passed as an argument. The view must delete lines at the current index until the correct line is the one at the index. It must then take appropriate action to reflect the modification, typically by copying the new line contents into its own mirror data structure.
- The callback function *insert* indicates that a line has been inserted at the current index since the last update. Again, the line is passed as an argument. The view updates its mirror data structure to reflect the new line.
- The callback function *sync* is called with a line passed as an argument. The view must delete lines at the current index until the correct line is the one at the index.

Notice that there is no *delete* callback function. The buffer does not hold on to lines that have been deleted, so it is incapable of supplying this information. Instead, the *modify* and *sync* operations provide this information implicitly by supplying the next line to be operated on. Any lines preceding it in the mirror data structure are no longer present in the buffer and should be deleted by the view.

The buffer protocol is *line-oriented* in two different ways:

(1) The editing operations specified by the protocol define a *line* abstraction, in contrast to a buffer of GNU Emacs [2] which exposes a single sequence containing newline characters to indicate line separation.
(2) The update protocol works on the granularity of a line. An entire line can be reported as being modified or inserted.

In the implementation of the buffer protocol, a line being edited is represented as a gap buffer. Therefore, editing operations are very fast, even for very long lines. However, the update protocol works on the granularity of an entire line. This granularity is acceptable for Common Lisp code, because lines are typically short. For other languages it might be necessary to use a different buffer library.

For the purpose of this paper, we are only interested in the update protocol, because we re-parse the buffer as a result of the update protocol having been invoked. We can think of such an invocation as resulting in a succession of operations, sorted by lines in increasing order. There can be three different update operations:

- Modify. The line has been modified.
- Insert. A new line has been inserted.
- Delete. An existing line has been deleted.

Although the presence of a *delete* operation may seem to contradict the fact that no such operation is possible, it is fairly trivial to derive this operation from the ones that are actually supported by the update protocol. Furthermore, this derived set of operations simplifies the presentation of our technique in the rest of the paper.

In order to parse the buffer contents, we use a custom `read` function. This version of the `read` function differs from the standard one in the following ways:

- Instead of returning S-expressions, it returns a nested structure of instances of a standard class named `parse-result`. These instances contain the corresponding S-expression and the start and end position (line, column) in the buffer of the parse result.
- The parse results returned by the reader also include entities that would normally not be returned by `read` such as comments and, more generally, results of applying reader macros that return no values.
- Instead of attempting to call `intern` in order to turn a token into a symbol, the custom reader returns an instance of a standard class named `token`.

The reader from the SICL project[3] was slightly modified to allow this kind of customization, thereby avoiding the necessity of maintaining the code for a completely separate reader.

No changes to the mechanism for handling reader macros is necessary. Therefore, we handle custom reader macros as well. Whenever a reader macro calls `read` recursively, a nested parse result is created in the same way as with the standard reader macros. More information about the required modifications to the reader are provided in Appendix B.

For a visible view, the buffer update protocol is invoked after each keystroke generated by the end user, and the number of modifications to the buffer since the previous invocation is typically very modest, in that usually a single line has been modified. It would be wasteful, and too slow for large buffers, to re-parse the entire buffer character by character, each time the update protocol is invoked. For that reason, we keep a *cache* of parse results returned by the customized reader.

3.2 Cache organization

The cache is organized as a sequence[4] of top-level parse results. Each top-level parse result contains the parse results returned by nested calls to the reader. Here, we are not concerned with the details of the representation of the cache. Such details are crucial in order to obtain acceptable performance, but they are unimportant for understanding the general technique of incremental parsing. Refer to appendix A for an in-depth description of these details.

When the buffer is updated, we try to maintain as many parse results as possible in the cache. Updating the cache according to a particular succession of update operations consists of two distinct phases:

(1) Invalidation of parse results that span a line that has been modified, inserted, or deleted.
(2) Rehabilitation of the cache according to the updated buffer contents.

3.3 Invalidation phase

As mentioned in Section 3.1, the invocation of the buffer-update protocol results in a sequence of operations that describe how the buffer has changed from the previous invocation.

As far as the invalidation phase is concerned, there are only minor variations in how the different types of operations are handled. In all cases (line modification, line insertion, line deletion), the existing parse results that straddle a place that has been altered must be

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

[4] Here, we use the word *sequence* in the meaning of a set of items organized consecutively, and not in the more restrictive meaning defined by the Common Lisp standard.

invalidated. Notice that when a top-level parse result straddles such a modification, that parse result is invalidated, but it is very likely that several of its children do not straddle the point of modification. Therefore such children are not invalidated, and are kept in the cache in case they are needed during the rehabilitation phase.

In addition to the parse results being invalidated as described in the previous paragraph, when the operation represents the insertion or the deletion of a line, remaining valid parse results following the point of the operation must be modified to reflect the fact that they now have a new start-line position.

As described in Appendix A, we designed the data structure carefully so that both invalidating parse results as a result of these operations, and modifying the start-line position of remaining valid parse results can be done at very little cost.

3.4 Rehabilitation phase

Conceptually, the rehabilitation phase consists of parsing the entire buffer from the beginning by calling `read` until the end of the buffer is reached. However, three crucial design elements avoid the necessity of a full re-analysis:

- Top level parse results that precede the first modification to the buffer do not have to be re-analyzed, because they must return the same result as before any modification.
- When `read` is called at a buffer position corresponding to a parse result that is in the cache, we can simply return the cache entry rather than re-analyzing the buffer contents at that point.
- If a top-level call to `read` is made beyond the last modification to the buffer, and there is a top-level parse result in the cache at that point, then every remaining top-level parse result in the cache can be re-used without any further analysis required.

4 PERFORMANCE OF OUR TECHNIQUE

The performance of our technique can not be stated as a single figure, nor even as a function of the size of the buffer, simply because it depends on several factors such as the exact structure of the buffer contents and the way the user interacts with that contents.

Despite these difficulties, we can give some indications for certain important special cases. We ran these tests on a 4-core Intel Core processor clocked at 3.3GHz, running SBCL version 1.3.11.

4.1 Parsing with an empty cache

When a buffer is first created, the cache is empty. The buffer contents must then be read, character by character, and the cache must be created from the contents.

We timed this situation with a buffer containing 10000 lines of representative Common Lisp code. The total time to parse was around 1.5 seconds. This result deserves some clarifications:

- It is very unusual to have a file of Common Lisp code with this many lines. Most files contain less than 2000 lines, which is only 1/5 of the one in our test case.
- This result was obtained from a very preliminary version of our parser. In particular, to read a character, several generic functions where called, including the `stream-read-char` function of the Gray streams library, and then several others in order to access the character in the buffer. Further optimizations are likely to decrease the time spent to read a single character.
- This situation will happen only when a buffer is initially read into the editor. Even very significant subsequent changes to the contents will still preserve large portions of the cache, so that the number of characters actually read will only be a tiny fraction of the total number of characters in the buffer.
- Parsing an entire buffer does not exercise the incremental aspect of our parser. Instead, the execution time is a complex function of the exact structure of the code, the performance of the reader in various situations, the algorithm for generic-function dispatch of the implementation, the cost of allocating standard objects, etc. For all these reasons, a more thorough analysis of this case is outside the scope of this paper, and the timing is given only to give the reader a rough idea of the performance in this initial situation.
- This particular case can be handled by having the parser process the original stream from which the buffer contents was created, rather than giving it the buffer protocol wrapped in a stream protocol after the buffer has been filled. That way, the entire overhead of the Gray-stream protocol is avoided altogether.

4.2 Parsing after small modifications

We measured the time to update the cache of a buffer with 1200 lines of Common Lisp code. We used several variations on the number of top-level forms and the size of each top-level form. Three types of representative modifications were used, namely inserting/deleting a constituent character, inserting/deleting left parenthesis, and inserting/deleting a double quote. All modifications were made at the very beginning of the file, which is the worst-case scenario for our technique.

For inserting and deleting a constituent character, we obtained the results shown in Table 1. For this benchmark, the performance is independent of the distribution of forms and sub-forms, and also of the number of characters in a line. The execution time is roughly proportional to the number of lines in the buffer. For that reason, the form size is given only in number of lines. The table shows that the parser is indeed very fast for this kind of incremental modification to the buffer.

nb forms	form size	time
120	10	0.14ms
80	15	0.14ms
60	20	0.14ms
24	100	0.23ms
36	100	0.32ms

Table 1: Inserting and deleting a constituent character.

For inserting and deleting a left parenthesis, we obtained the results shown in Table 2. For this benchmark, the performance is independent of the size of the sub-forms of the top-level forms. For that reason, the form size is given only in number of lines. As shown in the table, the performance is worse for many small top-level forms, and then the execution time is roughly proportional to the number of forms. When the number of top-level forms is small, the execution time decreases asymptotically to around 0.5ms. However, even the slowest case is very fast and has no impact on the perceived overall performance of the editor.

nb forms	form size	time
120	10	1.3ms
80	15	1.0ms
60	20	0.5ms
40	30	0.7ms
30	40	0.6ms
24	50	0.5ms
12	100	0.5ms

Table 2: Inserting and deleting a left parenthesis.

Finally, for inserting and deleting a double quote, we obtained the results shown in Table 3. For this benchmark, the performance is roughly proportional to the number of characters in the buffer when the double quote is inserted, and completely dominated by the execution time of the reader when the double quote is deleted. The execution time thus depends not only on the number of characters in the buffer, but also on how those characters determine what the reader does. As shown by the table, these execution times are borderline acceptable. In the next section, we discuss possible ways of improving the performance for this case.

nb forms	form size	characters per line	time
120	10	1	18ms
80	15	1	15ms
60	20	1	17ms
24	100	1	33ms
36	100	1	50ms
120	10	30	70ms

Table 3: Inserting and deleting a double quote.

5 CONCLUSIONS AND FUTURE WORK

Currently, parse results that are not part of the final structure of the buffer are discarded. When the user is not using an editor mode that automatically balances characters such as parentheses and double quotes, inserting such a character often results in a large number of parse results being discarded, only to have to be created again soon afterward, when the user inserts the balancing character of the pair. We can avoid this situation by keeping parse results that are not part of the final structure in the cache, in the hopes that they will again be required later. We then also need a strategy for removing such parse results from the cache after some time, so as to avoid that the cache grows without limits.

Parsing Common Lisp source code is only the first step in the analysis of its structure. In order to determine the role of each symbol and other information such as indentation, further analysis is required. Such analysis requires a *code walker*, because the role of a symbol may depend on the definitions of macros to which it is an argument. Similarly, computing standard *indentation*, also requires further analysis. To implement this code walker, we consider using the first phase of the Cleavir compiler framework.[5]

We plan to investigate the use of a new implementation of *first-class global environments* [7]. This new implementation of the existing CLOS protocol would use *incremental differences* to the *startup environment*[6] so as to define a *compilation environment*[7] that is different for each top-level form in the editor buffer. This technique would allow us to restart the compiler in an appropriate environment without having to process the entire buffer from the beginning.

The combination of the use of the first pass of the Cleavir compiler framework and the use of incremental first-class global environments will allow us to handle compile-time evaluation of certain top-level forms in a way that corresponds to the semantics of the file compiler. In particular, imperative environment operations such as changing the current package or modifying the readtable in the middle of a buffer will have the expected consequences, but only to subsequent forms in the buffer.

A more precise analysis of Common Lisp code opens the possibility for additional functionality that requires knowledge about the role of each expression. In particular, such an analysis could be the basis for sophisticated code transformations such as variable renaming and code refactoring.

6 ACKNOWLEDGMENTS

We would like to thank Philipp Marek and Cyrus Harmon for providing valuable feedback on early versions of this paper.

REFERENCES

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

[2] Craig A. Finseth. Theory and practice of text editors, or, A cookbook for an Emacs. Thesis (B.S.), M.I.T., Department of Electrical Engineering and Computer Science, Cambridge, MA, USA, 1980. Supervised by David P. Reed.

[3] Craig A. Finseth. *The Craft of Text Editing – Emacs for the Modern World.* Springer-Verlag, 1991. ISBN 0-387-97616-7 (New York), 3-540-97616-7 (Berlin).

[4] Bill Lewis, Dan LaLiberte, and Richard Stallman. *GNU Emacs Lisp Reference Manual.* Free Software Foundation, Boston, MA, USA, 2014. ISBN 1-882114-74-4.

[5] Robert B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*, AFIPS '68 (Fall, part I), pages 267–277, New York, NY, USA, 1968. ACM. doi: 10.1145/1476589.1476628. URL http://doi.acm.org/10.1145/1476589.1476628.

[6] Christophe Rhodes, Robert Strandh, and Brian Mastenbrook. Syntax Analysis in the Climacs Text Editor. In *Proceedings of the International Lisp Conference*, ILC 2005, June 2005.

[7] Robert Strandh. First-class Global Environments in Common Lisp. In *Proceedings of the 8th European Lisp Symposium*, ELS 2015, pages 79 – 86, April 2015. URL http://www.european-lisp-symposium.org/editions/2015/ELS2015.pdf.

[8] Robert Strandh. A CLOS Protocol for Editor Buffers. In *Proceedings of the 9th European Lisp Symposium*, ELS 2016, pages 3:21–3:28. European Lisp Scientific Activities Association, 2016. ISBN 978-2-9557474-0-7. URL http://dl.acm.org/citation.cfm?id=3005729.3005732.

[9] Robert Strandh and Timothy Moore. A Free Implementation of CLIM. In *Proceedings of the International Lisp Conference*, ILC 2002, October 2002.

[10] Robert Strandh, David Murray, Troels Henriksen, and Christophe Rhodes. ESA: A CLIM Library for Writing Emacs-Style Applications. In *Proceedings of the 2007 International Lisp Conference*, ILC '07, pages 24:1–24:10, New York, NY,

[5] Cleavir is currently part of SICL. See the directory named `Code/Cleavir` in the SICL repository.

[6] Recall that the startup environment is the environment in which the compiler was invoked.

[7] Recall that the compilation environment is the environment used by the compiler for definitions and side effects of the compilation process.

USA, 2009. ACM. ISBN 978-1-59593-618-9. doi: 10.1145/1622123.1622150. URL http://doi.acm.org/10.1145/1622123.1622150.

A CACHE REPRESENTATION

Figure 1 illustrates the representation of the cache for parse results. The buffer contents that corresponds to that cache contents might for instance be:

```
(a
 (b c))
(d)
(e
 f)
(g
 (h))
```

The sequence of top-level parse results is split into a *prefix* and a *suffix*, typically reflecting the current position in the buffer being edited by the end user. The suffix contains parse results in the order they appear in the buffer, whereas the prefix contains parse results in reverse order, making it easy to move parse results between the prefix and the suffix.

Depending on the location of the parse result in the cache data structure, its position may be *absolute* or *relative*. The prefix contains parse results that precede updates to the buffer. For that reason, these parse results have absolute positions. Parse results in the suffix, on the other hand, follow updates to the buffer. In particular, if a line is inserted or deleted, the parse results in the suffix will have their positions changed. For that reason, only the first parse result of the suffix has an absolute position. Each of the others has a position relative to its predecessor. When a line is inserted or deleted, only the first parse result of the suffix has to have its position updated. When a parse result is moved from the prefix to the suffix, or from the suffix to the prefix, the positions concerned are updated to maintain this invariant.

To avoid having to traverse all the descendants of a parse result when its position changes, we make the position of the first child of some parse result P relative to that of P, and the children, other than the first, of some parse result P, have positions relative to the previous child in the list.

As a result of executing the invalidation phase, a third sequence of parse results is created. This sequence is called the *residue*, and it contains valid parse results that were previously children of some top-level parse result that is no longer valid. So, for example, if the line containing the symbol `f` in the buffer corresponding to the cache in Figure 1 were to be modified, the result of the invalidation phase would be the cache shown in Figure 2.

As Figure 2 shows, the top-level parse result corresponding to the expression `(e f)` has been invalidated, in addition the child parse result corresponding to the expression `f`. However, the child parse result corresponding to the expression `e` is still valid, so it is now in the residue sequence. Furthermore, the suffix sequence now contains only the parse result corresponding to the expression `(g (h))`.

For the rehabilitation phase, we can imagine that a single character was inserted after the `f`, so that the line now reads as `fi)`.

At the start of the rehabilitation phase, the position for reading is set to the end of the last valid top-level parse result in the prefix, in this case at the end of the line containing the expression `(d)`. When the reader is called, it skips whitespace characters until it is positioned on the left parenthesis of the line containing `(e`. There is no cache entry, neither in the residue nor in the suffix, corresponding to this position, so normal reader operation is executed. Thus, the reader macro associated with the left parenthesis is invoked, and the reader is called recursively on the elements of the list. When the reader is called with the position corresponding to the expression `e`, we find that there is an entry for that position in the residue, so instead of this expression being read by normal reader operation, the contents of the cache is used instead. As a result, the position

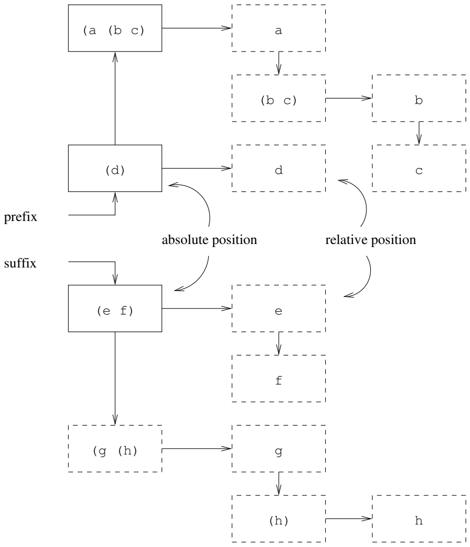


Figure 1: Representation of the cache.

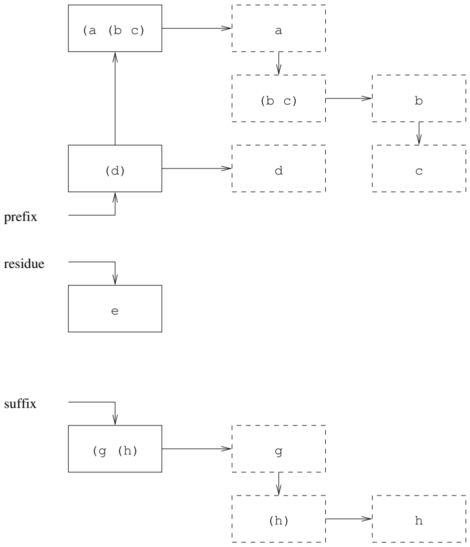


Figure 2: Cache contents after invalidation.

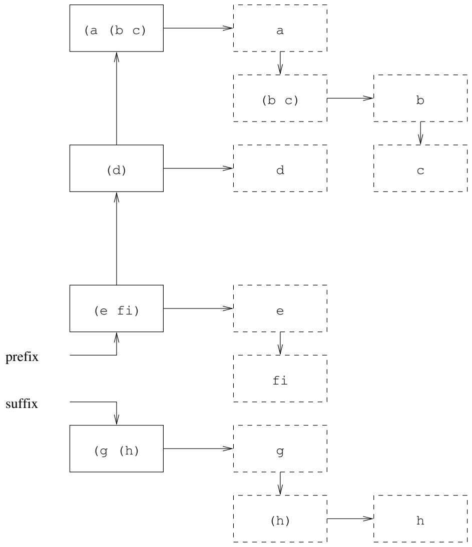


Figure 3: Cache contents after read.

in the buffer is set to the end of the cached parse result, i.e. at the end of the expression `e`. The remaining top-level expression is read using then normal reader operation resulting in the expression `(e fi)`. This parse result is added to the prefix resulting in the cache contents shown in figure 3.

The reader is then invoked again in order to read another top-level expression. In this invocation, whitespace characters are first skipped until the reader is positioned immediately before the expression `(g (h))`. Not only is there a parse result in the cache corresponding to this position, but that parse result is the first in the suffix sequence. We therefore know that all parse results on in the suffix are still valid, so the we can terminate the rehabilitation phase.

B READER CUSTOMIZATION

In order for it to be possible for the Common Lisp `read` function to serve as a basis for the incremental parser described in this paper, it must be adapted in the ways described below.

B.1 Returning parse results

In addition to the nested expressions returned by an unmodified `read` function, it must also return a nested structure of *parse results*, i.e. expressions wrapped in standard instances that also contain information about the location in the source code of the wrapped expressions.

To accomplish this additional functionality, it is not possible to create a custom `read` function that returns parse results *instead of* expressions, simply because the function must handle custom reader macros, and those reader macros return expressions, and not parse results. Also, it would create unnecessary maintenance work if all standard reader macros had to be modified in order to return parse results instead of expressions.

It is also not possible to modify the `read` function to return the parse result as a second value, in addition to the normal expression. One reason is that we would like for the modified `read` function to be compatible with the standard version, and it is not permitted by the Common Lisp standard to return additional values.

Instead, the modified `read` function maintains an explicit stack of parse results in parallel with the expressions that are normally returned. This explicit stack is kept as the value of a special variable that our parser accesses after a call to `read`.

B.2 Returning parse results for comments

The modified `read` function must return parse results that correspond to source code that the standard `read` function does not return, such as comments and expressions that are not selected by a read-time conditional. We solve this problem by checking when a reader macro returns no values, and in that case, a corresponding parse result is pushed onto the explicit stack mentioned in the previous section.

B.3 Intercepting symbol creation

The modified `read` function must not call `intern` in all situations that the ordinary `read` function would, and it must not signal an error when a symbol with an explicit package prefix does not exist. For that reason, the modified reader calls a generic function with the characters of a potential token instead. The unmodified `read` function just calls `intern`, whereas the custom `read` function creates a particular parse result that represents the token, and that can be exploited by the editor.

Strategies for typecase optimization

Jim E. Newton
Didier Verna
jnewton@lrde.epita.fr
didier@lrde.epita.fr
EPITA/LRDE
Le Kremlin-Bicêtre, France

ABSTRACT

We contrast two approaches to optimizing the Common Lisp `typecase` macro expansion. The first approach is based on heuristics intended to estimate run time performance of certain type checks involving Common Lisp type specifiers. The technique may, depending on code size, exhaustively search the space of permutations of the type checks, intent on finding the optimal order. With the second technique, we represent a `typecase` form as a type specifier, encapsulating the side-effecting non-Boolean parts so as to appear compatible with the Common Lisp type algebra operators. The encapsulated expressions are specially handled so that the Common Lisp type algebra functions preserve them, and we can unwrap them after a process of Boolean reduction into efficient Common Lisp code, maintaining the appropriate side effects but eliminating unnecessary type checks. Both approaches allow us to identify unreachable code, test for exhaustiveness of the clauses and eliminate type checks which are calculated to be redundant.

CCS CONCEPTS

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

ACM Reference Format:
Jim E. Newton and Didier Verna. 2018. Strategies for typecase optimization. In *Proceedings of the 11th European Lisp Symposium (ELS'18).* ACM, New York, NY, USA, 9 pages. https://doi.org/

1 INTRODUCTION

The `typecase` macro is specified in Common Lisp [4] as is a run-time mechanism for selectively branching as a function of the type of a given expression. Figure 1 summarizes the usage. The type specifiers used may be simple type names such as `fixnum`, `string`, or `my-class`, but may also specify more expressive types such as range checks `(float -3.0 3.5)`, membership checks such as `(member 1 3 5)`, arbitrary Boolean predicate checks such as `(satisfies oddp)`, or logical combinations of other valid type specifiers such as `(or string (and fixnum (not (eql 0))) (cons bignum))`.

```
(typecase keyform
  (Type.1 body-forms-1...)
  (Type.2 body-forms-2...)
  (Type.3 body-forms-3...)
  ...
  (Type.n body-forms-n...))
```

Figure 1: Synopsis of `typecase` syntax.

In this article we consider several issues concerning the compilation of such a `typecase` usage.

- *Redundant checks*[1] — The set of type specifiers used in a particular invocation of `typecase` may have subtype or intersection relations among them. Consequently, it is possible (perhaps likely in the case of auto-generated code) that the same type checks be performed multiple times, when evaluating the `typecase` at run-time.
- *Unreachable code* — The specification *suggests* but does not require that the compiler issue a warning if a clause is not reachable, being completely shadowed by earlier clauses. We consider such compiler warnings desirable, especially in manually written code.
- *Exhaustiveness* — The user is allowed to specify a set of clauses which is non-exhaustive. If it can be determined at compile time that the clauses are indeed exhaustive, even in the absence of a `t/otherwise` clause, then in such a case, the final type check may be safely replaced with `otherwise`, thus eliminating the need for that final type check at run-time.

The `etypecase` macro (exhaustive type case) promises to signal a run-time error if the object is not an element of any of the specified types. The question of whether the clauses are exhaustive is a different question, namely whether it can be determined at compile time that all possible values are covered by at least one of the clauses.

Assuming we are allowed to change the `typecase` evaluation order, we wish to exploit evaluation orders which are more likely to be faster at run-time. We assume that most type checks are fast, but some are slower than others. In particular, a `satisfies` check may be arbitrarily slow. Under certain conditions, as will be seen, there are techniques to *protect* certain type checks to allow reordering without effecting semantics. Such reordering may consequently enable particular optimizations such as elimination of redundant

[1] Don't confuse *redundant check* with *redundancy check*. In this article we address the former, not the latter. A type check is viewed as *redundant*, and can be eliminated, if its Boolean result can determined by static code analysis.

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1…$15.00

checks or the exhaustiveness optimization explained above. Elimination of redundant type checks has an additional advantage apart from potentially speeding up certain code paths, it also allows the discovery of unreachable code.

In this article we consider different techniques for evaluating the type checks in different orders than that which is specified in the code, so as to maintain the semantics but to eliminate redundant checks.

In the article we examine two very different approaches for performing certain optimizations of `typecase`. First, we use a *natural* approach using s-expression based type specifiers (Section 2), operating on them as symbolic expressions. In the second approach (Section 3) we employ Reduced Ordered Binary Decision Diagrams (ROBDDs). We finish the article with an overview of related work (Section 4) and a summary of future work (Section 5).

2 TYPE SPECIFIER APPROACH

We would like to automatically remove redundant checks such as `(eql 42)`, `(member 40 41 42)`, and `fixnum` in Example 1.

Example 1 (typecase with redundant type checks)**.**

```
(typecase OBJ
  ((eql 42)
   body-forms-1...)
  ((and (member 40 41 42) (not (eql 42)))
   body-forms-2...)
  ((and fixnum (not (member 40 41 42)))
   body-forms-3...)
  ((and number (not fixnum))
   body-forms-4...))
```

The code in Example 2 is semantically identical to that in Example 1, because a type check is only reached if all preceding type checks have failed.

Example 2 (typecase after removing redundant checks)**.**

```
(typecase OBJ
  ((eql 42) body-forms-1...)
  ((member 40 41 42) body-forms-2...)
  (fixnum body-forms-3...)
  (number body-forms-4...))
```

In the following sections, we initially show that certain duplicate checks may be removed through a technique called forward-substitution and reduction (Section 2.2). A weakness of this technique is that it sometimes fails to remove particular redundant type checks. Because of this weakness, a more elaborate technique is applied, in which we augment the type tests to make them mutually disjoint (Section 2.4). With these more complex type specifiers in place, the `typecase` has the property that its clauses are reorderable, which allows the forward-substitution and reduction algorithm to search for an ordering permitting more thorough reduction (Section 2.6). This process allows us to identify unreachable code paths and to identify exhaustive case analyses, but there are still situations in which redundant checks cannot be eliminated.

2.1 Semantics of type specifiers

There is some disagreement among experts of how to interpret certain semantics of type specifiers in Common Lisp. To avoid confusion, we state explicitly our interpretation.

There is a statement in the `typecase` specification that each *normal-clause* be *considered* in turn. We interpret this requirement not to mean that the type checks must be evaluated in order, but rather than each type test must assume that type tests appearing earlier in the `typecase` are not satisfied. Moreover, we interpret this specified requirement so as not to impose a run-time evaluation order, and that as long as evaluation semantics are preserved, then the type checks may be done in any order at run-time, and in particular, that any type check which is redundant or unnecessary need not be preformed.

The situation that the user may specify a type such as `(and fixnum (satisfies evenp))` is particularly problematic, because the Common Lisp specification contains a dubious, non-conforming example in the specification of `satisfies`. The problematic example in the specification says that `(and integer (satisfies evenp))` is a type specifier and denotes the set of all even integers. This claim contradicts the specification of the `AND` type specifier which claims that `(and A B)` is the intersection of types `A` and `B` and is thus the same as `(and B A)`. This presents a problem, because `(typep 1.0 '(and fixnum (satisfies evenp)))` evaluates to `nil` while `(typep 1.0 '(and (satisfies evenp) fixnum))` raises an error. We implicitly assume, for optimization purposes, that `(and A B)` is the same as `(and B A)`.

Specifically, if the `AND` and `OR` types are commutative with respect to their operands, and if type checks have side effects (errors, conditions, changing of global state, IO, interaction with the debugger), then the side effects cannot be guaranteed when evaluating the optimized code. Therefore, in our treatment of types we consider that type checking with `typep` is side-effect free, and in particular that it never raises an error. This assumption allows us to reorder the checks as long as we do not change the semantics of the Boolean algebra of the `AND`, `OR`, and `NOT` specifiers.

Admittedly, that `typep` never raise an error is an assumption we make knowing that it may limit the usefulness of our results, especially since some non-conforming Common Lisp programs may happen to perform correctly absent our optimizations. That is to say, our optimizations may result in errors in some non-conforming Common Lisp programs. The specification clearly states that certain run-time calls to `typep` even with well-formed type specifiers must raise an error, such as if the type specifier is a list whose first element is `values` or `function`. Also, as mentioned above, an evaluation of `(typep obj '(satisfies F))` will raise an error if `(F obj)` raises an error. One might be tempted to interpret `(typep obj '(satisfies F))` as `(ignore-errors (if (F obj) t nil))`, but that would be a violation of the specification which is explicit that the form `(typep x '(satisfies p))` is equivalent to `(if (p x) t nil)`.

There is some wiggle room, however. The specification of `satisfies` states that its operand be the name of a *predicate*, which is elsewhere defined as a function which returns. Thus one might be safe to conclude that `(satisfies evenp)` is not a valid type specifier, because `evenp` is specified to signal an error if its argument is not an `integer`.

We assume, for this article, that no such problematic type specifier is used in the context of `typecase`.

2.2 Reduction of type specifiers

There are legitimate cases in which the programmer has specifically ordered the clauses to optimize performance. A production worthy `typecase` optimization system should take that into account. However, for the sake of simplicity, the remainder of this article ignores this concern.

We introduce a macro, `reduced-typecase`, which expands to a call to `typecase` but with cases reduced where possible. Latter cases assuming previous type checks fail. This transformation preserves clause order, but may simplify the executable logic of some clauses. In the expansion, in Example 3 the second `float` check is eliminated, and consequently, the associated `AND` and `NOT`.

Example 3 (Simple invocation and expansion of `reduced-typecase`).

```
(reduced-typecase obj
   (float body-forms-1...)
   ((and number (not float)) body-forms-2...))

(typecase obj
   (float body-forms-1...)
   (number body-forms-2...))
```

How does this reduction work? To illustrate we provide a sightly more elaborate example. In Example 4 the first type check is `(not (and number (not float)))`. In order that the second clause be reached at run-time the first type check must have already failed. This means that the second type check, `(or float string (not number))`, may assume that `obj` is not of type `(not (and number (not float)))`.

Example 4 (Invocation and expansion `reduced-typecase` with unreachable code path).

```
(reduced-typecase obj
   ((not (and number (not float))) body-forms-1...)
   ((or float string (not number)) body-forms-2...)
   (string body-forms-3...))

(typecase obj
   ((not (and number (not float))) body-forms-1...)
   (string body-forms-2...)
   (nil body-forms-3...))
```

The `reduced-typecase` macro rewrites the second type test `(or float string (not number))` by a technique called forward-substitution. At each step, it substitutes implied values into the next type specifier, and performs Boolean logic reduction. Abelson *et al.* [1] discuss lisp[2] algorithms for performing algebraic reduction; however, in addition to the Abelson algorithm reducing Boolean expressions representing Common Lisp types involves additional reductions representing the subtype relations of terms in question. For example `(and number fixnum ...)` reduces to `(and fixnum ...)` because `fixnum` is a subtype of `number`. Similarly, `(or number fixnum ...)` reduces to `(or number ...)`. Newton *et al.* [21] discuss techniques of Common Lisp type reduction in the presence of subtypes.

$$\begin{aligned}
\texttt{(not (and number (not float)))} &= \texttt{nil}\\
\Longrightarrow \texttt{(and number (not float))} &= \texttt{t}\\
\Longrightarrow \texttt{number} &= \texttt{t}\\
\text{and } \texttt{(not float)} &= \texttt{t}\\
\Longrightarrow \texttt{float} &= \texttt{nil}\\
\texttt{(or float string}\quad &\\
\texttt{(not number))} &= \texttt{(or nil string (not t))}\\
&= \texttt{(or nil string nil)}\\
&= \texttt{string}
\end{aligned}$$

With this forward substitution, `reduced-typecase` is able to rewrite the second clause `((or float string (not number)) body-forms-2...)` simply as `(string body-forms-2...)`. Thereafter, a similar forward substitution is made to transform the third clause from `(string body-forms-3...)` to `(nil body-forms-3...)`.

Example 4 illustrates a situation in which a type specifier in one of the clauses reduces completely to `nil`. In such a case we would like the compiler to issue warnings about finding unreachable code, and in fact it does (at least when tested with SBCL[3]) because the compiler finds `nil` as the type specifier. The clauses in Example 5 are identical to those in Example 4, and consequently the expressions `body-forms-3...` in the third clause cannot be reached. Yet contrary to Example 4, SBCL, AllegroCL[4], and CLISP[5] issue no warning at all that `body-forms-3...` is unreachable code.

Example 5 (Invocation of `typecase` with unreachable code).

```
(typecase obj
   ((not (and number (not float))) body-forms-1...)
   ((or float string (not number)) body-forms-2...)
   (string body-forms-3...))
```

2.3 Order dependency

We now reconsider Examples 1 and 2. While the semantics are the same, there is an important distinction in practice. The first `typecase` contains mutually exclusive clauses, whereas the second one does not. *E.g.*, if the `(member 40 41 42)` check is moved before the `(eql 42)` check, then `(eql 42)` will never match, and the consequent code, `body-forms-2...` will be unreachable.

For the order of the type specifiers given Example 1, the types can be simplified, having no redundant type checks, as shown in Example 2. This phenomenon is both a consequence of the particular types in question and also the order in which they occur. As a contrasting example, consider the situation in Example 6 where the first two clauses of the `typecase` are reversed with respect to Example 1. In this case knowing that `OBJ` is not of type `(and (member 40 41 42) (not (eql 42)))` tells us nothing about whether `OBJ` is of type `(eql 42)` so no reduction can be inferred.

Example 6 (Re-ordering clauses sometimes enable reduction).

```
(typecase OBJ
  ((and (member 40 41 42) (not (eql 42)))
   body-forms-2...)
```

[2] In this article we use *lisp* (in lower case) to denote the family of languages or the concept rather than a particular language implementation, and we use *Common Lisp* to denote the language.

[3] We tested with SBCL 1.3.14. SBCL is an implementation of ANSI Common Lisp. `http://www.sbcl.org/`

[4] We tested with the International Allegro CL Free Express Edition, version 10.1 [32-bit Mac OS X (Intel)] (Sep 18, 2017 13:53). `http://franz.com`

[5] We tested with GNU CLISP 2.49, (2010-07-07). `http://clisp.cons.org/`

```
  ((eql 42)
   body-forms-1...)
  ((and fixnum (not (member 40 41 42)))
   body-forms-3...)
  ((and number (not fixnum))
   body-forms-4...))
```

Programmatic reductions in the typecase are dependent on the order of the specified types. There are many possible approaches to reducing types despite the order in which they are specified. We consider two such approaches. Section 2.6 discusses automatic reordering of disjoint clauses, and Section 3 uses decision diagrams.

As already suggested, a situation as shown in Example 6 can be solved to avoid the redundant type check, (eql 42), by reordering the disjoint clauses as in Example 1. However, there are situations for which no reordering alleviates the problem. Consider the code shown in Example 7. We see that some sets of types are reorderable, allowing reduction, but for some sets of types such ordering is impossible. We consider in Section 3 typecase optimization where reordering is futile. For now we concentrate on efficient reordering where possible.

Example 7 (Re-ordering cannot always enable reduction)**.**

```
(typecase OBJ
  ((and unsigned-byte (not bignum))
   body-forms-1...)
  ((and bignum (not unsigned-byte))
   body-forms-2...))
```

2.4 Mutually disjoint clauses

As suggested in Section 2.3, to arbitrarily reorder the clauses, the types must be disjoint. It is straightforward to transform any typecase into another which preserves the semantics but for which the clauses are reorderable. Consider a typecase in a general form.

Example 8 shows a set of type checks equivalent to those in Figure 1 but with redundant checks, making the clauses mutually exclusive, and thus reorderable.

Example 8 (typecase with mutually exclusive type checks)**.**

```
(typecase OBJ
  (Type.1
   body-forms-1...)
  ((and Type.2
        (not Type.1))
   body-forms-2...)
  ((and Type.3
        (not (or Type.1 Type.2)))
   body-forms-3...))
  ...
  ((and Type.n
        (not (or Type.1 Type.2 ... Type.n-1)))
   body-forms-n...))
```

In order to make the clauses reorderable, we make them more complex which might seem to defeat the purpose of optimization. However, as we see in Section 2.6, the complexity can sometimes be removed after reordering, thus resulting in a set of type checks which is *better* than the original. We discuss what we mean by *better* in Section 2.5.

We proceed by first describing a way to judge which of two orders is better, and with that comparison function, we can visit every permutation and choose the best.

One might also wonder why we suffer the pain of establishing heuristics and visiting all permutations of the mutually disjoint types in order to find the best order. One might ask, why not just put the clauses in the best order to begin with. The reason is because in the general case, it is not possible to predict what the best order is. As is discussed in Section 4, ordering the Boolean variables to produce the smallest binary decision diagram is an NP-hard problem. The only solution in general is to visit every permutation. The problem of ordering a set of type tests for optimal reduction must also be NP-hard because if we had a better solution, we would be able to solve the BDD NP-hard problem as a consequence.

2.5 Comparing heuristically

Given a set of disjoint and thus reorderable clauses, we can now consider finding a good order. We can examine a type specifier, typically after having been reduced, and heuristically assign a *cost*. A high cost is assigned to a satisfies type, a medium cost to AND, OR, and NOT types which takes into account the cost of the types specified therein, and a low cost to atomic names.

To estimate the relative *goodness* of two given semantically identical typecase invocations, we can heuristically estimate the complexity of each by using a weighted sum of the costs of the individual clauses. The weight of the first clause is higher because the type specified therein will be always checked. Each type specifier thereafter will only be checked if all the preceding checks fail. Thus the heuristic weights assigned to subsequent checks is chosen successively smaller as each subsequent check has a smaller probability of being reached at run-time.

2.6 Reduction with automatic reordering

Now that we have a way to heuristically measure the complexity of a given invocation of typecase we can therewith compare two semantically equivalent invocations and choose the better one. If the number of clauses is small enough, we can visit all possible permutations. If the number of clauses is large, we can sample the space randomly for some specified amount of time or specified number of samples, and choose the best ordering we find.

We introduce the macro, auto-permute-typecase. It accepts the same arguments as typecase and expands to a typecase form. It does so by transforming the specified types into mutually disjoint types as explained in Section 2.4, then iterating through all permutations of the clauses. For each permutation of the clauses, it reduces the types, eliminating redundant checks where possible using forward-substitution as explained in Section 2.2, and assigns a cost heuristic to each permutation as explained in Section 2.5. The auto-permute-typecase macro then expands to the typecase form with the clauses in the order which minimizes the heuristic cost.

Example 9 shows an invocation and expansion of auto-permute-typecase. In this example auto-permute-typecase does a good job of eliminating redundant type checks.

Example 9 (Invocation and expansion of auto-permute-typecase)**.**

```
(auto-permute-typecase obj
  ((and unsigned-byte (not (eql 42)))
   body-forms-1...)
  ((eql 42)
   body-forms-2...)
  ((and number (not (eql 42)) (not fixnum))
   body-forms-3...)
  (fixnum
    body-forms-4...))

(typecase obj
  ((eql 42) body-forms-2...)
  (unsigned-byte body-forms-1...)
  (fixnum body-forms-4...)
  (number body-forms-3...))
```

As mentioned in Section 1, a particular optimization can be made in the situation where the type checks in the `typecase` are exhaustive; in particular the final type check may be replaced with `t/otherwise`. Example 10 illustrates such an expansion in the case that the types are exhaustive. Notice that the final type test in the expansion is `t`.

Example 10 (Invocation and expansion of `auto-permute-typecase` with exhaustive type checks)**.**

```
(auto-permute-typecase obj
  ((or bignum unsigned-byte) body-forms-1...)
  (string body-forms-2...)
  (fixnum body-forms-3...)
  ((or (not string) (not number)) body-forms-4...))

(typecase obj
  (string body-forms-2...)
  ((or bignum unsigned-byte) body-forms-1...)
  (fixnum body-forms-3...)
  (t body-forms-4...))
```

3 DECISION DIAGRAM APPROACH

In Section 2.6 we looked at a technique for reducing `typecase` based solely on programmatic manipulation of type specifiers. Now we explore a different technique based on a data structure known as Reduced Ordered Binary Decision Diagram (ROBDD).

Example 7 illustrates that redundant type checks cannot always be reduced via reordering. Example 11 is, however, semantically equivalent to Example 7. Successfully mapping the code from of a `typecase` to an ROBDD will guarantee that redundant type checks are eliminated. In the following sections we automate this code transformation.

Example 11 (Suggested expansion of Example 7)**.**

```
(if (typep OBJ 'unsigned-byte)
    (if (typep obj 'bignum)
        nil
        (progn body-forms-1...))
    (if (typep obj 'bignum)
        (progn body-forms-2...)
        nil))
```

The code in Example 11 also illustrates a concern of code size explosion. With the two type checks `(typep OBJ 'unsigned-byte)` and `(typep obj 'bignum)`, the code expands to 7 lines of code. If this code transform be done naïvely, the risk is that each `if/then/else` effectively doubles the code size. In such an undesirable case, a `typecase` having N unique type tests among its clauses, would expand to $2^{N+1} - 1$ lines of code, even if such code has many congruent code paths. The use of ROBDD related techniques allows us to limit the code size to something much more manageable. Some discussion of this is presented in Section 4.

ROBDDs (Section 3.1) represent the semantics of Boolean equations but do not maintain the original evaluation order encoded in the actual code. In this sense the reordering of the type checks, which is explicit and of combinatorical complexity in the previous approach, is automatic in this approach. A complication is that normally ROBDDs express Boolean functions, so the mapping from `typecase` to ROBDD is not immediate, as a `typecase` may contain arbitrary side-effecting expressions which are not restricted to Boolean expressions. We employ an encapsulation technique which allows the ROBDDs to operate opaquely on these problematic expressions (Section 3.1). Finally, we are able to serialize an arbitrary `typecase` invocation into an efficient `if/then/else` tree (Section 3.3).

ROBDDs inherently eliminate duplicate checks. However, ROBDDs cannot easily guarantee removing all unnecessary checks as that would involve visiting every possible ordering of the leaf level types involved.

3.1 An ROBDD compatible type specifier

An ROBDD is a data structure used for performing many types of operations related to Boolean algebra. When we use the term ROBDD we mean, as the name implies, a decision diagram (directed cyclic graph, DAG) whose vertices represent Boolean tests and whose branches represent the consequence and alternative actions. An ROBDD has its variables **O**rdered, meaning that there is some ordering of the variables $\{v_1, v_2, ..., v_N\}$ such that whenever there is an arrow from v_i to v_j then $i < j$. An ROBDD is deterministically **R**educed so that all common sub-graphs are shared rather than duplicated. The reader is advised to read the lecture nodes of Andersen [3] for a detailed understanding of the reduction rules. It is worth noting that there is variation in the terminology used by different authors. For example, Knuth [18] uses the unadorned term BDD for what we are calling an ROBDD.

A unique ROBDD is associated with a canonical form representing a Boolean function, or otherwise stated, with an equivalence class of expressions within the Boolean algebra. In particular, intersection, union, and complement operations as well as subset and equivalence calculations on elements from the underlying space of sets or types can be computed by straightforward algorithms. We omit detailed explanations of those algorithms here, but instead we refer the reader to work by Andersen [3] and Castagna [13].

We employ ROBDDs to convert a `typecase` into an `if/then/else` diagram as shown in Figure 2. In the figure, we see a decision diagram which is similar to an ROBDD, at least in all the internal nodes of the diagram. Green arrows lead to the consequent if a specified type check succeeds. Red arrows lead to the alternative. However, the leaf nodes are not Boolean values as we expect for an ROBDD.

We want to transform the clauses of a `typecase` as shown in Figure 1 into a binary decision diagram. To do so, we associate a distinct `satisfies` type with each clause of the `typecase`. Each such `satisfies` type has a unique function associated with it, such as `P1`, `P2`, etc, allowing us to represent the diagram shown in Figure 2 as an actual ROBDD as shown in Figure 3.

In order for certain Common Lisp functions to behave properly (such as `subtypep`) the functions `P1`, `P2`, etc. must be real functions, as opposed to place-holder functions types as Baker[7] suggests, so that `(satisfies P1)` etc, have type specifier semantics. `P1`, `P2`, etc, must be defined in a way which preserves the semantics of the `typecase`.

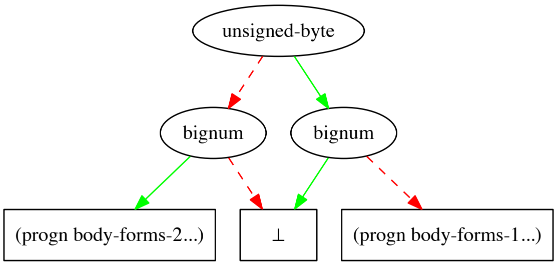


Figure 2: Decision Diagram representing irreducible `typecase`. This is similar to an ROBDD, but does not fulfill the definition thereof, because the leaf nodes are not simple Boolean values.

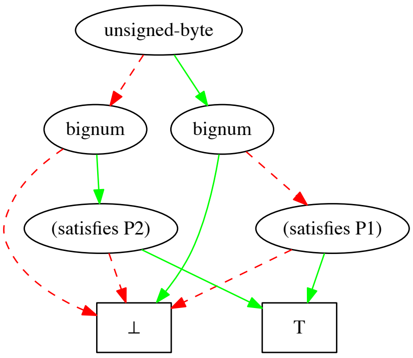


Figure 3: ROBDD with temporary valid `satisfies` types

Ideally we would like to create type specifiers such as the following:

```
(satisfies (lambda (obj)
             (typep obj '(and (not unsigned-byte)
                              bignum))))
```

Unfortunately, the specification of `satisfies` explicitly forbids this, and requires that the operand of `satisfies` be a symbol representing a globally callable function, even if the type specifier is only used in a particular dynamic extent. Because of this limitation in Common Lisp, we create the type specifiers as follows. Given a type specifier, we create such a functions at run-time using the technique shown in the function `define-type-predicate` defined in Implementation 1, which programmatically defines function with semantics similar to those shown in Example 12.

Implementation 1 (define-type-predicate).

```
(defun define-type-predicate (type-specifier)
  (let ((function-name (gensym "P")))
    (setf (symbol-function function-name)
          #'(lambda (obj)
              (typep obj type-specifier)))
    function-name))
```

Example 12 (Semantics of `satisfies` predicates).

```
(defun P1 (obj)
  (typep obj '(and (not unsigned-byte) bignum)))
(defun P2 (obj)
  (typep obj '(and (not bignum) unsigned-byte)))
```

The `define-type-predicate` function returns the name of a named closure which the calling function can use to construct a type specifier. The name and function binding are generated in a way which has dynamic extent and is thus friendly with the garbage collector.

To generate the ROBDD shown in Figure 3 we must construct a type specifier equivalent to the entire invocation of `typecase`. From the code in Figure 1 we have to assemble a type specifier such as in Example 13. This example is provided simply to illustrate the pattern of such a type specifier.

Example 13 (Type specifier equivalent to Figure 1).

```
(let ((P1 (define-type-predicate 'Type.1))
      (P2 (define-type-predicate
           '(and Type.2 (not Type.1))))
      (P3 (define-type-predicate
           '(and Type.3 (not (or Type.1 Type.2)))))
      ...
      (Pn (define-type-predicate
           '(and Type.n (not (or Type.1 Type.2
                                 ... Type.n-1))))))
   `(or (and Type.1
             (satisfies ,P1))
        (and Type.2
             (not Type.1)
             (satisfies ,P2))
        (and Type.3
             (not (or Type.1 Type.2))
             (satisfies ,P3))
        ...
        (and Type.n
             (not (or Type.1 Type.2
                      ... Type.n-1))
             (satisfies ,Pn))))
```

3.2 BDD construction from type specifier

Functions which construct an ROBDD need to understand a complete, deterministic ordering of the set of type specifiers via a *compare* function. To maintain semantic correctness the corresponding *compare* function must be deterministic. It would be ideal if the function were able to give high priority to type specifiers which are likely to be seen at run time. We might consider, for example, taking clues from the order specified in the `typecase` clauses. We do not attempt to implement such decision making. Rather we choose to give high priority to type specifiers which are *easy* to check at run-time, even if they are less likely to occur.

We use a heuristic similar to that mentioned in Section 2.5 except that type specifiers involving `AND`, `OR`, and `NOT` never occur, rather such types correspond to algebraic operations among the ROBDDs themselves such that only non-algebraic types remain. More precisely, the heuristic we use is that atomic types such as `number` are considered fast to check, and `satisfies` types are considered slow. We recognize the limitation that the user might have used `deftype` to define a type whose name is an atom, but which is slow to type check. Ideally, we should fully *expand* user defined types

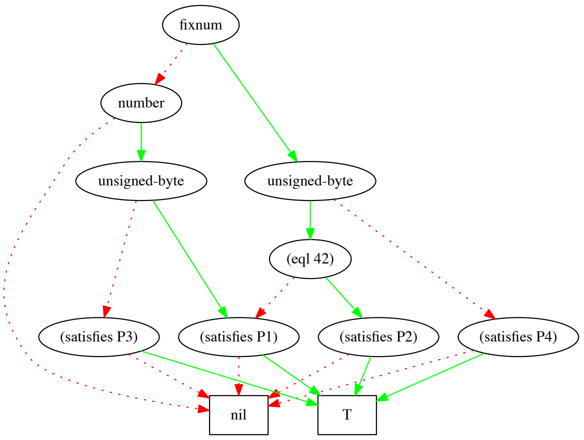


Figure 4: ROBDD generated from `typecase` clauses in Example 14

into Common Lisp types. Unfortunately this is not possible in a portable way, and we make no attempts to implement such expansion in implementation specific ways. It is not even clear whether the various Common Lisp implementations have public APIs for the operations necessary.

A crucial exception in our heuristic estimation algorithm is that to maintain the correctness of our technique, we must assure that the `satisfies` predicates emanating from `define-type-predicate` have the lowest possible priority. *I.e.*, as is shown in Figure 3, we must avoid that any type check appear below such a `satisfies` type in the ROBDD.

There are well known techniques for converting an ROBDD which represents a pure Boolean expression into an `if/then/else` expression which evaluates to `true` or `false`. However, in our case we are interested in more than simply the Boolean value. In particular, we require that the resulting expression evaluate to the same value as corresponding `typecase`. In Figure 1, these are the values returned from `body-forms-1...`, `body-forms-2...`, ... `body-forms-n...`. In addition we want to assure that any side effects of those expressions are realized as well when appropriate, and never realized more than once.

We introduce the macro `bdd-typecase` which expands to a `typecase` form using the ROBDD technique. When the macro invocation in Example 14 is expanded, the list of `typecase` clauses is converted to a type specifier similar to what is illustrated in Example 13. That type specifier is used to create an ROBDD as illustrated in Figure 4. As shown in the figure, temporary `satisfies` type predicates are created corresponding to the potentially side-effecting expressions `body-forms-1`, `body-forms-2`, `body-forms-3`, and `body-forms-4`. In reality these temporary predicates are named by machine generated symbols; however, in Figure 4 they are denoted `P1`, `P2`, `P3`, and `P4`.

Example 14 (Invocation of `bdd-typecase` with intersecting types).

```
(bdd-typecase obj
  ((and unsigned-byte (not (eql 42)))
   body-forms-1...)
  ((eql 42)
   body-forms-2...)
  ((and number (not (eql 42)) (not fixnum))
   body-forms-3...)
  (fixnum
   body-forms-4...))
```

3.3 Serializing the BDD into code

The macro `bdd-typecase` emits code as in Example 15, but just as easily may output code as in Example 16 based on `tagbody/go`. In both example expansions we have substituted more readable labels such as `L1` and `block-1` rather than the more cryptic machine generated uninterned symbols `#:l1070` and `#:|block1066|`.

Example 15 (Macro expansion of Example 14 using `labels`).

```
((lambda (obj-1)
   (labels ((L1 () (if (typep obj-1 'fixnum)
                       (L2)
                       (L7)))
            (L2 () (if (typep obj-1 'unsigned-byte)
                       (L3)
                       (L6)))
            (L3 () (if (typep obj-1 '(eql 42))
                       (L4)
                       (L5)))
            (L4 () body-forms-2...)
            (L5 () body-forms-1...)
            (L6 () body-forms-4...)
            (L7 () (if (typep obj-1 'number)
                       (L8)
                       nil))
            (L8 () (if (typep obj-1 'unsigned-byte)
                       (L5)
                       (L9)))
            (L9 () body-forms-3...))
     (L1)))
 obj)
```

The `bdd-typecase` macro walks the ROBDD, such as the one illustrated in Figure 4, visiting each non-leaf node therein. Each node corresponding to a named closure type predicate is serialized as a tail call to the clauses from the `typecase`. Each node corresponding to a normal type test is serialized as left and right branches, either as a label and two calls to `go` as in Example 16, or a local function definition with two tail calls to other local functions as in Example 15.

Example 16 (Alternate expansion of Example 14 using `tagbody/go`).

```
((lambda (obj-1)
   (block block-1
     (tagbody
      L1 (if (typep obj-1 'fixnum)
             (go L2)
             (go L7))
      L2 (if (typep obj-1 'unsigned-byte)
             (go L3)
             (go L6))
      L3 (if (typep obj-1 '(eql 42))
             (go L4)
             (go L5))
      L4 (return-from block-1
           (progn body-forms-2...))
      L5 (return-from block-1
           (progn body-forms-1...))
      L6 (return-from block-1
           (progn body-forms-4...))
      L7 (if (typep obj-1 'number)
             (go L8)
             (return-from block-1 nil))
     L8 (if (typep obj-1 'unsigned-byte)
             (go L5)
             (go L9))
     L9 (return-from block-1
           (progn body-forms-3...)))))
obj)
```

3.4 Emitting compiler warnings

The ROBDD, as shown in Figure 4, can be used to generate the Common Lisp code semantically equivalent to the corresponding typecase as already explained in Section 3.3, but we can do even better. There are two situations where we might wish to emit warnings: (1) if certain code is unreachable, and (2) if the clauses are not exhaustive. Unfortunately, there is no standard way to incorporate these warnings into the standard compiler output. One might tempted to simply emit a warning of type style-warning as is suggested by the typecase specification. However, this would be undesirable since there is no guarantee that the corresponding code was human-generated—ideally we would only like to see such style warnings corresponding to human generated code.

The list of unreachable clauses can be easily calculated as a function of which of the P1, P2 ... predicates are missing from the serialized output. As seen in Figure 4, each of body-forms-1, body-forms-2, body-forms-3, and body-forms-4 is represented as P1, P2, P3, and P4, so no such code is unreachable in this case.

We also see in Figure 4 that there is a path from the root node to the nil leaf node which does not pass through P1, P2, P3, or P4. This means that the original typecase is not exhaustive. The type of any such value can be calculated as the particular path leading to nil. In the case of Figure 4, (and (not fixnum) (not number)), which corresponds simply to (not number), is such a type. *I.e.*, the original bdd-typecase, shown in Example 14, does not have a clause for non numbers.

4 RELATED WORK

This article references the functions make-bdd and bdd-cmp whose implementation is not shown herein. The code is available via GitLab from the EPITA/LRDE public web page:
https://gitlab.lrde.epita.fr/jnewton/regular-type-expression.git. That repository contains several things. Most interesting for the context of BDDs is the Common Lisp package, LISP-TYPES.

As there are many individual styles of programming, and each programmer of Common Lisp adopts his own style, it is unknown how widespread the use of typecase is in practice, and consequently whether optimizing it is effort well spent. A casual look at the code in the current public Quicklisp[6] repository reveals a rule of thumb. 1 out of 100 files, and 1 out of 1000 lines of code use or make reference to typecase. When looking at the Common Lisp code of SBCL itself, we found about 1.6 uses of typecase per 1000 lines of code. We have made no attempt to determine which of the occurrences are comments, trivial uses, or test cases, and which ones are used in critical execution paths; however, we do loosely interpret these results to suggest that an optimized typecase either built into the cl:typecase or as an auxiliary macro may be of little use to most currently maintained projects. On the contrary, we suggest that having such an optimized typecase implementation, may serve as motivation to some programmers to make use of it, at least in machine generated code such as Newton *et al.* [20] explain. Since generic function dispatch conceptually bases branching choices on Boolean combinations of type checks, one naturally wonders whether our optimizations might be of useful within the implementation of CLOS[17].

Newton *et al.* [20] present a mechanism to characterize the type of an arbitrary sequence in Common Lisp in terms of a rational language of the types of the sequence elements. The article explains how to build a finite state machine and from that construct Common Lisp code for recognizing such a sequence. The code associates the set of transitions existing from each state as a typecase. The article notes that such a machine generated typecase could greatly benefit from an optimizing typecase.

The map-permutations function (Section 2.6) works well for small lists, but requires a large amount of stack space to visit all the permutations of large lists. Knuth[18] explores several iterative (not recursive) algorithms using various techniques, in particular by plain changes[18, Algorithm P, page 42], by cyclic shifts[18, Algorithm C, page 56], and by Erlich swaps[18, Algorithm E, page 57]. A survey of these three algorithms can also be found in the Cadence SKILL Blog[7] which discussions an implementation in SKILL[8], another lisp dialect.

There is a large amount of literature about Binary Decision Diagrams of many varieties [2, 3, 9, 10, 14]. In particular Knuth [18, Section 7.1.4] discusses worst-case and average sizes, which we alluded to in Section 3. Newton *et al.* [21] discuss how the Reduced Ordered Binary Decision Diagram (ROBDD) can be used to manipulate type specifiers, especially in the presence of subtypes. Castagna [13] discusses the use of ROBDDs (he calls them BDDs in that article) to perform type algebra in type systems which treat types as sets [4, 12, 16].

BDDs have been used in electronic circuit generation[15], verification, symbolic model checking[11], and type system models such as in XDuce [16]. None of these sources discusses how to extend the BDD representation to support subtypes.

Common Lisp does not provide explicit pattern matching [5] capabilities, although several systems have been proposed such as Optima[8] and Trivia[9]. Pierce [23, p. 341] explains that the addition of a typecase-like facility (which he calls typecase) to a typed λ-calculus permits arbitrary run-time pattern matching.

Decision tree techniques are useful in the efficient compilation of pattern matching constructs in functional languages[19]. An important concern in pattern matching compilation is finding the best ordering of the variables which is known to be NP-hard. However, when using BDDs to represent type specifiers, we obtain representation (pointer) equality, simply by using a consistent ordering; finding the *best* ordering is not necessary for our application.

In Section 2.2 we mentioned the problem of symbolic algebraic manipulation and simplification. Ableson *et al.* [1, Section 2.4.3] discuss this with an implementation of rational polynomials. Norvig

[6] https://www.quicklisp.org/

[7] https://community.cadence.com/tags/Team-SKILL, SKILL for the Skilled, *Visiting all Permutations*

[8] https://github.com/m2ym/optima

[9] https://github.com/guicho271828/trivia

[22, Chapter 8] discusses this in a use case of a symbolic mathematics simplification program. Both the Ableson and Norvig studies explicitly target a lisp-literate audience.

5 CONCLUSION AND FUTURE WORK

As illustrated in Example 9, the exhaustive search approach used in the auto-permute-typecase (Section 2.6) can often do a good job removing redundant type checks occurring in a typecase invocation. Unfortunately, as shown in Example 7, sometimes such optimization is algebraically impossible because the particular type interdependencies. In addition, an exhaustive search becomes unreasonable when the number of clauses is large. In particular there are $N!$ ways to order N clauses. This means there are $7! = 5040$ orderings of 7 clauses and $10! = 3,628,800$ orderings of 10 clauses.

On the other hand, the bdd-typecase macro, using the ROBDD approach (Section 3.2), is always able to remove duplicate checks, guaranteeing that no type check is performed twice. Nevertheless, it may fail to eliminate some unnecessary checks which need not be performed at all.

It is known that the size and *shape* of a reduced BDD depends on the ordering chosen for the variables [9]. Furthermore, it is known that finding the *best* ordering is NP-hard, and in this article we do not address questions of choosing or improving variable orderings. It would be feasible, at least in some cases, to apply the exhaustive search approach with ROBDDs. *I.e.*, we could visit all orders of the type checks to find which gives the smallest ROBDD. In situations where the number of different type tests is large, the development described in Section 3.1 might very well be improved employing some known techniques for improving BDD size though variable ordering choices[6]. In particular, we might attempt to use the order specified in the typecase as input to the sorting function, attempting in at least the simple cases to respect the user given order as much as possible.

In Section 2.4, we presented an approach to approximating the cost a set of type tests and commented that the heuristics are simplistic. We leave it as a matter for future research as to how to construct good heuristics, which take into account how compute intensive certain type specifiers are to manipulate.

We believe this research may be useful for two target audiences: application programmers and compiler developers. Even though the currently observed use frequency of typecase seems low in the majority of currently supported applications, programmers may find the macros explained in this article (auto-permute-typecase and bdd-typecase) to be useful in rare optimization cases, but more often for their ability to detect certain dubious code paths. There are, however, limitations to the portable implementation, namely the lack of a portable expander for user defined types, and an ability to distinguish between machine generated and human generated code. These shortcomings may not be significant limitations to the compiler implementer, in which case the compiler may be able to better optimize user types, implement better heuristics regarding costs of certain type checks, and emit useful warnings about unreachable code.

REFERENCES

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

[2] S. B. Akers. Binary decision diagrams. *IEEE Trans. Comput.*, 27(6):509–516, June 1978. ISSN 0018-9340. doi: 10.1109/TC.1978.1675141. URL http://dx.doi.org/10.1109/TC.1978.1675141.

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

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

[5] Lennart Augustsson. Compiling pattern matching. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 368–381, New York, NY, USA, 1985. Springer-Verlag New York, Inc. ISBN 3-387-15975-4. URL http://dl.acm.org/citation.cfm?id=5280.5303.

[6] Adnan Aziz, Serdar Taşiran, and Robert K. Brayton. Bdd variable ordering for interacting finite state machines. In *Proceedings of the 31st Annual Design Automation Conference*, DAC '94, pages 283–288, New York, NY, USA, 1994. ACM. ISBN 0-89791-653-0. doi: 10.1145/196244.196379. URL http://doi.acm.org/10.1145/196244.196379.

[7] Henry G. Baker. A Decision Procedure for Common Lisp's SUBTYPEP Predicate. *Lisp and Symbolic Computation*, 5(3):157–190, 1992. URL http://dblp.uni-trier.de/db/journals/lisp/lisp5.html#Baker92a.

[8] T.J. Barnes. SKILL: a CAD system extension language. In *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pages 266–271, Jun 1990. doi: 10.1109/DAC.1990.114865.

[9] Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, 35:677–691, August 1986.

[10] Randal E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Comput. Surv.*, 24(3):293–318, September 1992. ISSN 0360-0300. doi: 10.1145/136035.136043. URL http://doi.acm.org/10.1145/136035.136043.

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 1020 states and beyond. *Inf. Comput.*, 98(2):142–170, June 1992. ISSN 0890-5401. doi: 10.1016/0890-5401(92)90017-A. URL http://dx.doi.org/10.1016/0890-5401(92)90017-A.

[12] G. Castagna and V. Lanvin. Gradual typing with union and intersection types. *Proc. ACM Program. Lang.*, (1, ICFP '17, Article 41), sep 2017.

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

[14] Maximilien Colange. *Symmetry Reduction and Symbolic Data Structures for Model Checking of Distributed Systems.* Thèse de doctorat, Laboratoire de l'Informatique de Paris VI, Université Pierre-et-Marie-Curie, France, December 2013. URL http://www.lsv.ens-cachan.fr/Publis/PAPERS/PDF/colange-phd13.pdf.

[15] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 365–373, London, UK, UK, 1990. Springer-Verlag. ISBN 3-540-52148-8. URL http://dl.acm.org/citation.cfm?id=646691.703286.

[16] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Trans. Program. Lang. Syst.*, 27(1):46–90, January 2005. ISSN 0164-0925. doi: 10.1145/1053468.1053470. URL http://doi.acm.org/10.1145/1053468.1053470.

[17] Gregor J. Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[18] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams.* Addison-Wesley Professional, 12th edition, 2009. ISBN 0321580508, 9780321580504.

[19] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN Workshop on ML*, ML '08, pages 35–46, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-062-3. doi: 10.1145/1411304.1411311. URL http://doi.acm.org/10.1145/1411304.1411311.

[20] Jim Newton, Akim Demaille, and Didier Verna. Type-Checking of Heterogeneous Sequences in Common Lisp. In *European Lisp Symposium*, Kraków, Poland, May 2016.

[21] Jim Newton, Didier Verna, and Maximilien Colange. Programmatic manipulation of Common Lisp type specifiers. In *European Lisp Symposium*, Brussels, Belgium, April 2017.

[22] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp.* Morgan Kaufmann, 1992.

[23] Benjamin C. Pierce. *Types and Programming Languages.* The MIT Press, 1st edition, 2002. ISBN 0262162091, 9780262162098.

Method Combinators

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

ABSTRACT

In traditional object-oriented languages, the dynamic dispatch algorithm is hardwired: for every polymorphic call, only the most specific method is used. Clos, the Common Lisp Object System, goes beyond the traditional approach by providing an abstraction known as *method combinations*: when several methods are applicable, it is possible to select several of them, decide in which order they will be called, and how to combine their results, essentially making the dynamic dispatch algorithm user-programmable.

Although a powerful abstraction, method combinations are under-specified in the Common Lisp standard, and the Mop, the Meta-Object Protocol underlying many implementations of Clos, worsens the situation by either contradicting it or providing unclear protocols. As a consequence, too much freedom is granted to conforming implementations. The exact or intended behavior of method combinations is unclear and not necessarily coherent with the rest of Clos.

In this paper, we provide a detailed analysis of the problems posed by method combinations, the consequences of their lack of proper specification in one particular implementation, and a Mop-based extension called *method combinators*, aiming at correcting these problems and possibly offer new functionality.

CCS CONCEPTS

• **Software and its engineering → Object oriented languages**; **Extensible languages**; **Polymorphism**; *Inheritance*; *Classes and objects*; *Object oriented architectures*; *Abstraction, modeling and modularity*;

KEYWORDS

Object-Oriented Programming, Common Lisp Object System, Meta-Object Protocol, Generic Functions, Dynamic Dispatch, Polymorphism, Multi-Methods, Method Combinations, Orthogonality

ACM Reference Format:
Didier Verna. 2018. Method Combinators. In *Proceedings of the 11th European Lisp Symposium (ELS'18)*. ACM, New York, NY, USA, 10 pages. https://doi.org/

ELS'18, April 16–17 2018, Marbella, Spain
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-2-9557474-2-1.

1 INTRODUCTION

Common Lisp was the first programming language equipped with an object-oriented (OO) layer to be standardized [16]. Although in the lineage of traditional class-based OO languages such as Smalltalk and later C++ and Java, Clos, the Common Lisp Object System [2, 5, 7, 9], departs from those in several important ways.

First of all, Clos offers native support for multiple dispatch [3, 4]. Multiple dispatch is a generalization of single dispatch *a.k.a.* inclusion polymorphism [12]. In the classic message-passing style of single dispatch, the appropriate method is selected according to the type of the receiver (the object through which the method is called). In multiple dispatch however, the method selection algorithm may use as many arguments as requested in the generic call. Because this kind of polymorphism doesn't grant any object argument a particular status (message receiver), methods (herein called *multi-methods*) are naturally decoupled from classes and generic function calls look like ordinary function calls. The existence of multi-methods thus pushes dynamic dispatch one step further in the direction of separation of concerns: polymorphism and inheritance are clearly separated.

Next, Clos itself is written on top of a meta-object protocol, the Clos Mop [10, 13]. Although not part of the ANSI specification, the Clos Mop is a *de facto* standard well supported by many implementations. In supporting implementations, the Mop layer not only allows for Clos to be implemented in itself (classes being instances of their meta-classes *etc.*), but also lets the programmer extend or modify its very semantics, hence providing a form of homogeneous behavioral reflection [11, 14, 15].

Yet another improvement over the classical OO approach lies in the concept of *method combination*. In the traditional approach, the dynamic dispatch algorithm is hardwired: every polymorphic call ends up executing the most specific method available (applicable) and using other, less specific ones requires explicit calls to them. In Clos however, a generic function can be programmed to implicitly call several applicable methods (possibly all of them), not necessarily by order of specificity, and combine their results (not necessarily all of them) in a particular way. Along with multiple dispatch, method combinations constitute one more step towards *orthogonality* [8, chapter 8]: a generic function can now be seen as a 2D concept: 1. a set of methods and 2. a specific way of combining them. As usual with this language, method combinations are also fully programmable, essentially turning the dynamic dispatch algorithm into a user-level facility.

Richard P. Gabriel reports[1] that at the time Common Lisp was standardized, the standardization committee didn't believe that

[1] in a private conversation

```
(defgeneric details (human)
  (:method-combination append :most-specific-last))
(defmethod details append ((human human)) ...)
(defmethod details append ((employee employee)) ...))
```

Figure 1: Short Method Combination Usage Example

method combinations were mature enough to make people implement them in one particular way (the only industrial-strength implementation available back then was in Flavors on Lisp Machines). Consequently, they intentionally under-specified them in order to leave room for experimentation. At the time, the Mop was not ready either, and only added later, sometimes with unclear or contradictory protocols. The purpose of this paper is to provide a detailed analysis of the current status of method combinations, and also to offer possible improvements over them.

Section 2 provides a detailed analysis of the specification for method combinations (both Clos and the Mop) and points out its caveats. Section 3 describes how Sbcl[2] implements method combinations, and exhibits some of their inconsistent or unfortunate (although conformant) behavior. Sections 4 and 5 provide an extension to method combinations, called *method combinators*, aimed at fixing the problems previously described. Finally, Section 6 demonstrates an additional feature made possible with method combinators, which increases yet again the orthogonality of generic functions in Common Lisp.

2 METHOD COMBINATIONS ISSUES

In this section, we provide an analysis of how method combinations are specified and point out a set of important caveats. This analysis is not only based on what the Common Lisp standard claims, but also on the additional requirements imposed by the Clos Mop. In the remainder of this paper, some basic knowledge on method combinations is expected, notably on how to define them in both short and long forms. The reader unfamiliar with `define-method-combination` is invited to look at the examples provided in the Common Lisp standard first[3].

2.1 Lack of Orthogonality

As already mentioned, method combinations help increase the separation of concerns in Common Lisp's view on generic functions. The orthogonality of the concept goes only so far however, and even seems to be hindered by the standard itself occasionally. This is particularly true in the case of method combinations defined in short form (or built-in ones, which obey the same semantics).

Figure 1 demonstrates the use of the append built-in combination, concatenating the results of all applicable methods. In this particular example, and given that employees are humans, calling `details` on an employee would collect the results of both methods. Short combinations require methods to have exactly *one* qualifier: either the combination's name for primary methods (append in our example), or the `:around` tag[4]. This means that one cannot change a generic function's (short) method combination in a practical way, as it would basically render every primary method unusable (the standard also mandates that an error be signaled if methods without a qualifier, or a different one are found). Hence, method combinations are not completely orthogonal to generic functions. On the other hand, `:around` methods remain valid after a combination change, a behavior inconsistent with that of primary methods.

Perhaps the original intent was to improve readability or safety: when adding a new method to a generic function using a short method combination, it may be nice to be reminded of the combination's name, or make sure that the programmer remembers that it's a non-standard one. If such is the case, it also fails to do so in a consistent way. Indeed, short method combinations support an option affecting the order in which the methods are called, and passed to the `:method-combination` option of a `defgeneric` call (`:most-specific-first/last`, also illustrated in Figure 1). Thus, if one is bound to restate the combination's name anyway, why not restate the potential option as well? Finally, one may also wonder why short method combinations didn't get support for `:before` and `:after` methods as well as `:around` ones.

Because short method combinations were added to enshrine common, simple cases in a shorter definition form, orthogonality was not really a concern. Fortunately, short method combinations can easily be implemented as long ones, without the limitations exhibited in this section (see Appendix A).

2.2 Lack of Structure

The Common Lisp standard provides a number of concepts related to object-orientation, such as objects, classes, generic functions, and methods. Such concepts are usually gracefully integrated into the type system through a set of classes called *system classes*. Generic functions, classes, and methods are equipped with two classes: a class named *C* serving as the root for the whole concept hierarchy, and a class named `standard-`*C* serving as the default class for objects created programmatically. In every case, the standard explicitly names the APIs leading to the creation of objects of such standard classes. For example, `standard-method` is a subclass of `method` and is "the default class of methods defined by the `defmethod` and `defgeneric` forms"[5].

Method combinations, on the other hand, only get one standardized class, the `method-combination` class. The Mop further states that this class should be abstract (not meant to be instantiated), and also explicitly states that it "does not specify the structure of method combination metaobjects"[10, p. 140]. Yet, because the standard also requires that method combination objects be "indirect instances" of the `method-combination` class[6], it is mandatory that subclasses are provided by conforming implementations (although no provisions are made for a `standard-method-combination` class for instance). Although this design may seem inconsistent with the rest of Clos, the idea, again, was to leave room for experimentation. For example, knowing that method combinations come in two forms, *short* and *long*, and that short combinations may be implemented as long ones, implementations can choose whether to represent short and long combinations in a single or as separate hierarchies. The unfortunate consequence, however, is that it is

[2] http://www.sbcl.org
[3] http://www.lispworks.com/documentation/lw70/CLHS/Body/m_defi_4.htm
[4] http://www.lispworks.com/documentation/lw70/CLHS/Body/07_ffd.htm
[5] http://www.lispworks.com/documentation/lw70/CLHS/Body/t_std_me.htm
[6] http://www.lispworks.com/documentation/lw70/CLHS/Body/t_meth_1.htm

impossible to specialize method combinations in a portable way, because implementation-dependent knowledge of the exact method combination classes is needed in order to subclass them.

Yet another unfortunate consequence of this under-specification lies in whether method combinations should be objects or classes to be instantiated, although the original intent was to consider them as some kind of macros involved in method definition. The Common Lisp standard consistently talks of "method combination types", and in particular, this is what is supposed to be created by `define-method-combination`[7]. This seems to suggest the creation of classes. On the other hand, method combinations can be parametrized when they are used. The long form allows a full ordinary lambda-list to be used when generic functions are created. The short form supports one option called `:identity-with-one-argument`, influencing the combination's behavior at creation-time (originally out of a concern for efficiency), and another one, the optional `order` argument, to be used by generic functions themselves. The long form also has several creation-time options for method groups such as `:order` and `:required`, but it turns out that these options can also be set at use-time, through the lambda-list.

2.3 Unclear Protocols

The third and final issue we see with method combinations is that the Mop, instead of clarifying the situation, worsens it by providing unclear or inconsistent protocols.

2.3.1 `find-method-combination`. In Common Lisp, most global objects can be retrieved by name one way or another. For example, `symbol-function` and `symbol-value` give you access to the Lisp-2 namespaces [6], and other operators perform a similar task for other categories of objects (`compiler-macro-function` being an example). The Common Lisp standard defines a number of `find-*` operators for retrieving objects. Amongst those are `find-method` and `find-class` which belong to the Clos part of the standard, but there is no equivalent for method combinations.

The Mop, on the other hand, provides a generic function called `find-method-combination` [10, p. 191]. However, this protocol only adds to the confusion. First of all, the arguments to this function are a generic function, a method combination type name, and some method combination options. From this prototype, we can deduce that contrary to `find-class` for example, it is not meant to retrieve a globally defined method combination by name. Indeed, the description of this function says that it is "called to determine the method combination object used by a generic function". Exactly *who* calls it and *when* is unspecified however, and if the purpose is to retrieve the method combination used by a generic function, then one can wonder what the second and third arguments (method combination type and options) are for, and what happens if the requested type is *not* the type actually used by the generic function. In fact, the Mop already provides a more straightforward way of inquiring a generic function about its method combination. `generic-function-method-combination` is an accessor doing just that.

2.3.2 `compute-effective-method`. Another oddity of method combinations lies in the design of the generic function invocation protocol. This protocol is more or less a two steps process. The first step consists in determining the set of applicable methods for a particular call, based on the arguments (or their classes). The Common Lisp standard specifies a function (which the Mop later refines), `compute-applicable-methods`, which unsurprisingly accepts two arguments: a generic function and its arguments for this specific call. The second step consists in computing (and then calling) the *effective method*, that is, the combination of applicable methods, precisely combined in a manner specified by the generic function's method combination. While the Common Lisp standard doesn't specify how this is done, the Mop does, via a function called `compute-effective-method`. Unsurprisingly again, this function accepts two arguments: a generic function and a set of (applicable) methods that should be combined together. More surprisingly however, it takes a method combination as a third (middle) argument. One can't help but wonder why such an argument exists, as the generic function's method combination can be retrieved through its accessor which, as we saw earlier, is standardized. Here again, we may be facing a aborted attempt at more orthogonality. Indeed, this protocol makes it possible to compute an effective method for *any* method combination, not just the one currently in use by the generic function (note also that the Mop explicitly mentions that `compute-effective-method` may be called by the user [10, p. 176]). However, the rest of Clos or the Mop doesn't support using `compute-effective-method` in this extended way. It is, however, an incentive for more functionality (see Section 6).

2.3.3 Memoization. One final remark in the area of protocols is about the care they take for performance. The Mop describes precisely how and when a discriminating function is allowed to cache lists of applicable methods [10, p. 175]. Note that nothing is said about the location of such a cache however (within the discriminating function, in a lexical closure over it, globally for every generic function *etc.*), but it doesn't really matter. On the other hand, the Mop says nothing about caching of effective methods. This means that conforming implementations are free to do what they want (provided that the semantics of Clos is preserved). In particular, if caching of effective methods is done, whether such a cache is maintained once for every generic function, or once for every generic function/method combination pair is unspecified. This is rather unfortunate, both for separation of concerns, and also for the extension that we propose in Section 6.

3 THE CASE OF SBCL

In this section, we analyse Sbcl's implementation of Clos, and specifically the consequences of the issues described in the previous section. Note that with one exception, the analysis below also stands for Cmucl[8] from which Sbcl is derived, and which in turn derives its implementation of Clos from Pcl [1].

3.1 Classes

The Sbcl method combination classes hierarchy is depicted in Figure 2. It provides the `standard-method-combination` class that

[7] http://www.lispworks.com/documentation/lw70/CLHS/Body/m_defi_4.htm

[8] https://www.cons.org/cmucl/

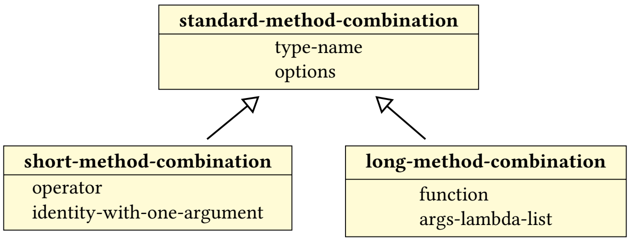


Figure 2: Sbcl Method Combination Classes Hierarchy

was missing from the standard (see Section 2.2), although this class doesn't serve as the default implementation for method combinations, as two subclasses are provided for that, one for each combination form. The `options` slot in the base class stores the use-time options (the ones passed to the `:method-combination` option to a `defgeneric` call). New method combination definitions are not represented by new classes; only by instances of either the `short` or `long-method-combination` ones. As a result, method combinations retrieved later on are objects containing a mix of definition-time and use-time options.

3.2 Short Method Combinations

Investigating how short method combinations are created in Sbcl uncovers a very peculiar behavior. `define-method-combination` expands to a call to `load-short-defcombin`, which in turn creates a method on `find-method-combination`, `eql`-specialized on the combination's name and ignoring its first argument (the generic function). This method is encapsulated in a closure containing the method combination's parameters, and recreates and returns a *new* method combination object on the fly *every time it is called.*

This has at least three important consequences.

(1) Short method combinations never actually globally exist *per se* (they don't have a namespace proper). Indeed, what is defined is not a method combination object (not even a class), but a means to create one on-demand. In particular, every generic function gets its own brand new object representing its method combination.

(2) `find-method-combination` neither does what its name suggests, nor what the Mop seems to imply. Indeed, because the generic function argument is ignored, it doesn't "determine the method combination object used by a generic function", but just creates whatever method combination instance you wish, of whichever known type and use-time option you like.

(3) It also turns out that redefining a short method combination (for example by calling `define-method-combination` again) doesn't affect the existing generic functions using it (each have a local object representing it). This is in contradiction with how every other Clos component behaves (class changes are propagated to live instances, method redefinitions update their respective generic functions *etc.*).

3.3 Long Combinations

The case of long method combinations is very similar, although with one additional oddity. Originally in Pcl (and still the case in Cmucl), long method combinations are compiled into so-called *combination functions*, which are in turn called in order to compute effective methods. In both Pcl and Cmucl, these functions are stored in the `function` slot of the long method combination objects (see Figure 2). In Sbcl however, this slot is not used anymore. Instead, Sbcl stores those functions in a *global* hash table named `*long-method-combination-functions*` (the hash keys being the combination names). The result is that long method combinations are represented half-locally (local objects in generic functions), half-globally with this hash table.

Now suppose that one particular long method combination is redefined while some generic functions are using it. As for the short ones, this redefinition will not (immediately) affect the generic functions in question, because each one has its own local object representing it. However, the combination function in the global hash table *will* be updated. As a result, if any concerned generic function ever needs to recompute its effective method(s) (for instance, if some methods are added or removed, if the set of applicable methods changes from one call to another, or simply if the generic function needs to be reinitialized), then the updated hash table entry will be used and the generic function's behavior will indeed change according to the updated method combination. With effective methods caching (as is the case in Sbcl) and a little (bad) luck, one may even end up with a generic function using different method combinations for different calls at the same time (see Appendix B).

4 METHOD COMBINATORS

In this section, we propose an extension to method combinations called *method combinators*, aiming at correcting the problems described in Sections 2 and 3. Most importantly, method combinators have a global namespace and generic functions using them are sensitive to their modification. Method combinators come with a set of new protocols inspired from what already exists in Clos, thus making them more consistent with it. As an *extension* to method combinations, they are designed to work on top of them, in a non-intrusive way (regular method combinations continue to work as before). Finally, their implementation tries to be as portable as possible (although, as we have already seen, some vendor-specific bits are unavoidable).

4.1 Method Combinator Classes

Figure 3 depicts the implementation of method combinators in Sbcl. We provide two classes, `short/long-method-combinator`, themselves subclasses of their corresponding, implementation-dependent, method combination classes. A `method-combinator-mixin` is also added as a superclass for both, maintaining additional information (the `clients` slot will be explained in Section 5.3) and serving as a means to specialize on both kinds of method combinators at the same time.

4.2 Method Combinators Namespace

Method combinators are stored globally in a hash table and accessed by name. This hash table is manipulated through an accessor called

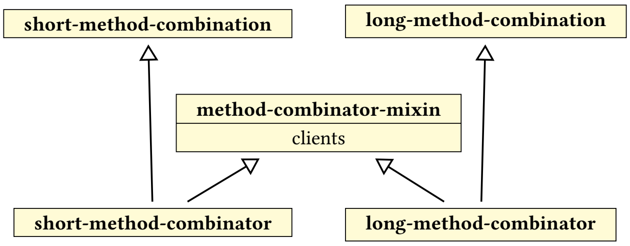


Figure 3: Method Combinator Classes Hierarchy

find-method-combinator (and its accompanying setf function). This accessor can be seen as the equivalent of find-class for method combinators, and has the same prototype. It thus considerably departs from the original Mop protocol, but is much more consistent with Clos itself.

4.3 Method Combinators Management

4.3.1 Protocols. The short method combinator protocol is designed in the same layered fashion as the rest of the Mop. First, we provide a macro called define-short-method-combinator behaving as the short form of define-method-combination, and mostly taking care of quoting. This macro expands to a call to ensure-short-method-combinator. In turn, this (regular) function calls the ensure-short-method-combinator-using-class generic function. Unsurprisingly, this generic function takes a method combinator as its first argument, either null when a new combinator is created, or an existing one in case of a redefinition. Note that the Mop is not always clear or consistent with its ensure-* family of functions, and their relation to the macro layer. In method combinators, we adopt a simple policy: while the functional layer may default some optional or keyword arguments, the macro layer only passes down those arguments which have been explicitly given in the macro call.

The same protocol is put in place for long method combinators. Note that it is currently undecided whether we want to keep distinct interfaces and protocols for short and long forms. The current choice of separation simply comes from the fact that Pcl implements them separately. Another yet undecided feature is how to handle definition-time *vs.* use-time options. Currently, in order to simplify the matter as a proof of concept, the (normally) use-time option :most-specific-first/last is handled when a short combinator is defined rather than when it is used, and the lambda-list for long forms is deactivated. In other words, use-time options are not supported. Note that this is of little consequence in practice: instead of using the same combination with different use-time arguments, one would just need to define different (yet similar) combinations with those arguments hard-wired in the code.

4.3.2 Creation. A new method combinator is created in 3 steps.

(1) define-method-combination is bypassed. Because regular method combinations do not have any other protocol specified, we use Sbcl's internal functions directly. Recall that the effect of these functions is to add a new method to find-method-combination.
(2) We subsequently call this new method in order to retrieve an actual combination object, and upgrade it to a combinator by calling change-class.
(3) Finally, this upgraded object is stored in the global combinators hash table by calling (setf find-method-combinator).

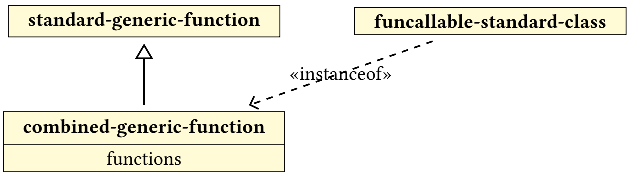

standard-generic-function
funcallable-standard-class
«instanceof»
combined-generic-function
functions

Figure 4: Combined Generic Functions

All of this is done in layer 3 of the protocols, except that in the case of long combinators, the combination function is computed at the macro level (this is how Sbcl does it). Additionally, as Cmucl still does, but contrary to Sbcl, we update the function slot in the long combinator objects.

The advantage of this process is that defining a combinator also inserts a regular method combination in the system. Regular generic functions may thus use the new combination without any of the combinator extensions.

4.3.3 Modification. An existing method combinator may be updated by the user via the first two protocol layers (the define-* macro layer or the ensure-* functional one). The updating process is quite simple: it merely involves a call to reinitialize-instance or to change-class if we are switching combinator forms. The definition change is also propagated to the regular combination layer, and in the case of the long form, care is taken to update not only the function slot of the combinator object, but Sbcl's *long-method-combination-functions* hash table as well.

4.3.4 Built-in Combinators. Finally, we provide new versions of the standard and built-in method combinations as combinators. These combinators are named with keywords, so as to both co-exist gracefully with the original Common Lisp ones, and still be easily accessible by name. On top of that, the built-in method combinators are defined in long forms, so as to provide support for :before and :after methods, and also avoid requiring the combinator's name as a qualifier to primary methods. In fact, a user-level macro called define-long-short-method-combinator is provided for defining such "pseudo-short" combinators easily.

5 COMBINED GENERIC FUNCTIONS

At that point, generic functions can seamlessly use method combinators as regular combinations, although with not much benefit (apart from the extended versions of the built-in ones). Our next goal is to ensure that the global method combinator namespace is functioning properly.

5.1 Generic Functions Subclassing

As usual, in order to remain unobtrusive with standard Clos, we specialize the behavior of generic functions with a subclass handling method combinators in the desired way. This class, called combined-generic-function, is depicted in Figure 4 (an explanation for the functions slot will be provided in Section 6.2.2). For convenience, a macro called defcombined is provided as a wrapper around defgeneric. This macro takes care of setting the generic function class to combined-generic-function (unless otherwise specified). Also for convenience, a new :method-combinator option is provided to replace the regular :method-combination one, but ultimately transformed back into

Finally, the (not portable) method-combination slot of generic functions is extended to recognize a :method-combinator initarg, and a method-combinator accessor.

5.2 Method Combinator Management

5.2.1 Initialization. In the absence of an explicit method combinator option, new combined generic functions should use the :standard one. This is easily done by providing a default initarg for :method-combinator to the combined-generic-function class, with a value of (find-method-combinator :standard).

The case of a provided method combinator name is more interesting. Normally, we would wrap ensure-generic-function/-using-class with specialized versions to look up a combinator instead of a combination. However, at the expense of portability (a necessity anyway), we can do a little simpler. As it turns out, Sbcl initializes a generic function's method combination by calling find-method-combination on the generic function's class prototype. Consequently, we can simply specialize this function with an eql specializer on the combined-generic-function class prototype, and look up for the appropriate global method combinator object there. Note that in order to specialize on a class prototype, the class needs to have been finalized already. Because of that, we need to call finalize-inheritance explicitly and very early on the class combined-generic-function.

5.2.2 Sanitation. This is also a good opportunity for us to sanitize the find-method-combination protocol for combined generic functions. A new method specialized on such functions is provided. Contrary to the default behavior, this method ensures that the requested method combinator is indeed the one in use by the function, and then returns it (recall that this is a global object). Otherwise, an error is signaled.

5.2.3 Updating. In order to change a combined generic function's method combinator, we provide a convenience function called change-method-combinator. This function accepts a combined generic function (to be modified) and a method combinator *designator* (either a name, or directly an object) which it canonicalizes. In the ideal case, this function should be able to only invalidate the generic function's effective method cache. Unfortunately, this cannot be done in a portable way. Thus, the only thing we can do portably is to call reinitialize-instance with the new method combinator.

5.3 Client Maintenance

The last thing we need to do is make sure that method combinator updates are correctly propagated to relevant combined generic functions. A combined generic function using a method combinator is called its *client*. Every method combinator maintains a list of clients, thanks to the the clients slot of the mixin (see Figure 3).

5.3.1 Registration. Registering a combined generic function as a method combinator client is implemented via two methods. One, on initialize-instance, adds a new combined generic function to its method combinator's clients slot. The other one, on reinitialize-instance, checks whether an existing combined generic function's combinator has changed, and performs the updating accordingly (recall that reinitializing the instance is the only portable way to change a generic function's method combination).

Note that while the Common Lisp standard allows a generic function's class to change, provided that both classes are "compatible" (a term which remains undefined)[9], the Mop seems to imply that meta-classes are only compatible with themselves (it is forbidden to change a generic function's meta-class [10, p. 187]). This restriction makes the client registration process simpler, as a regular generic function cannot become a combined one, or *vice versa.*

5.3.2 Updating. When a method combinator is redefined, it can either remain in the same form, or switch from short to long and *vice versa.* These two situations can be easily detected by specializing reinitialize-instance and u-i-f-d-c[10] (we could also use shared-initialize). Two such :after methods are provided, which trigger updating of all the method combinator's clients.

Client updating is implemented thanks to a new protocol inspired from the instance updating one: we provide a generic function called make-clients-obsolete, which starts the updating process. During updating, the generic function u-c-g-f-f-r-m-c[11] is called on every client. As mentioned previously, there is no portable way to invalidate an effective methods cache in the Clos Mop, so the only thing we can do safely is to completely reinitialize the generic function.

The problem we have here is that while the method combinator has been redefined, the object identity is preserved. Still, we need to trick the implementation into believing that the generic function's method combinator object has changed. In order to do that, we first set the combined generic function's method-combination slot to nil manually (and directly; bypassing all official protocols), and then call reinitialize-instance with a :method-combinator option pointing to the same combinator as before. The implementation then mistakenly thinks that the combinator has changed, and effectively reinitializes the instance, invalidating previously cached effective methods.

6 ALTERNATIVE COMBINATORS

In Section 4.3.4, we provided new versions of the built-in method combinations allowing primary methods to remain unqualified. In Section 5.2.3 we offered a convenience function to change the method combinator of a combined generic function more easily (hence the use for unqualified methods). In the spirit of increasing the separation of concerns yet again, the question of *alternative combinators* follows naturally: what about calling a generic function

[9] http://www.lispworks.com/documentation/lw70/CLHS/Body/f_ensure.htm

[10] update-instance-for-different-class

[11] update-combined-generic-function-for-redefined-method-combinator

with a different, temporary method combinator, or even maintaining several combinators at once in the same generic function?

In the current state of things, we can already change the method combinator temporarily, call the generic function, and then switch the combinator back to its original value. Of course, the cost of doing it this way is prohibitive, as the generic function would need to be reinitialized as many times as one changes its combinator. There is however, a way to do it more efficiently. While highly experimental, it has been tested and seems to work properly in Sbcl.

6.1 Protocols

At the lowest level lies a function called `call-with-combinator`. This function takes a combinator object, a combined generic function object and a `&rest` of arguments. Its purpose is to call the generic function on the provided arguments, only with the temporary combinator instead of the original one. On top of this function, we provide a macro called `call/cb` (pun intended) accepting designators (*e.g.* names) for the combinator and generic function arguments, instead of actual objects. Finally, it is not difficult to extend the Lisp syntax with a reader macro to denote alternative generic calls in a concise way. For demonstration purposes, a `#!` dispatching macro character may be installed and used like this:

```
#!combinator(func arg1 arg2 ...)
```

This syntax is transformed into the following macro call:

```
(call/cb combinator func arg1 arg2 ...)
```

In turn, this is finally expanded into:

```
(call-with-combinator
    (find-method-combinator 'combinator)
  #'func arg1 arg2 ...)
```

6.2 Implementation

Method combinations normally only affect the computation of effective methods. Unfortunately, we have already seen that the Clos Mop doesn't specify how or when effective methods may be cached. Consequently, the only portable way of changing them is to reinitialize a generic function with a different combination. Although effective methods cannot be portably accessed, the generic function's discriminating function can, at least in a half-portable fashion. This gives us an incentive towards a possible implementation.

6.2.1 Discriminating Functions / Funcallable Instances. A generic function is an instance of a *funcallable* class (see Figure 4), which means that generic function objects may be used where functional values are expected. When a generic function (object) is "called", its discriminating function is actually called. The Mop specifies that discriminating functions are installed by the (regular) function `set-funcallable-instance-function`. This strongly suggests that the discriminating function is stored somewhere in the generic function object. Unfortunately, the Mop doesn't specify a reader for that potential slot, although every implementation will need one (this is why we said earlier that discriminating functions could be accessed in a half-portable way). In Sbcl, it is called `funcallable-instance-fun`.

6.2.2 Discriminating Function Caches. The idea underlying our implementation of alternative combinators is thus the following. Every combined generic function maintains a cache of discriminating functions, one per alternative combinator used (this is the `functions` slot seen in Figure 4). When an alternative combinator is used for the first time (via a call to `call-with-combinator`), the generic function is reinitialized with this temporary combinator, called, and the new discriminating function is memoized. The function is then reinitialized back to its original combinator, and the values from the call are returned. It is important to actually execute the call *before* retrieving the new discriminating function, because it may not have been calculated before that.

If the alternative combinator was already used before with this generic function, then the appropriate discriminating function is retrieved from the cache and called directly. Of course, care is also taken to call the generic function directly if the alternative combinator is in fact the generic function's default one.

6.2.3 Client Maintenance. Alternative combinators complicate client maintenance (see Section 5.3), but the complication is not insurmountable. When an alternative combinator is used for the first time, the corresponding generic function is registered as one of its clients. The client updating protocol (see Section 5.3.2) is extended so that if the modified combinator is not the generic function's original one, then the generic function is *not* reinitialized. Instead, only the memoized discriminating function corresponding to this combinator is invalidated.

6.2.4 Disclaimer. Generic functions were never meant to work with multiple combinations in parallel, so there is no guarantee on how or where applicable and effective method caches, if any, are maintained. Our implementation of alternative combinators can only work if each discriminating function gets its own set of caches, for example by closing over them. According to both the result of experimentation and some bits of documentation[12], it appears to be the case in Sbcl. If, on the other hand, an implementation maintains a cache of effective methods *outside* the discriminating functions (for instance, directly in the generic function object), then, this implementation is guaranteed to *never* work.

7 PERFORMANCE

Because method combinators are implemented in terms of regular combinations, the cost of a (combined) generic call shouldn't be impacted. In Sbcl, only the `standard` combination is special-cased for bootstrapping and performance reasons, so some loss could be noticeable with the `:standard` combinator. Method combinator updates or changes do have a cost, as clients need to be reinitialized, but this is not different from updating a regular generic function for a new method combination. Again, the only portable way to do so is also to completely reinitialize the generic function.

Alternative combinators, on the other hand, do come at a cost, and it is up to the programmer to decide whether the additional expressiveness is worth it. Using an alternative combinator for the first time is very costly, as the generic function will be reinitialized twice

[12] http://www.sbcl.org/sbcl-internals/Discriminating-Functions.html#Discriminating-Functions

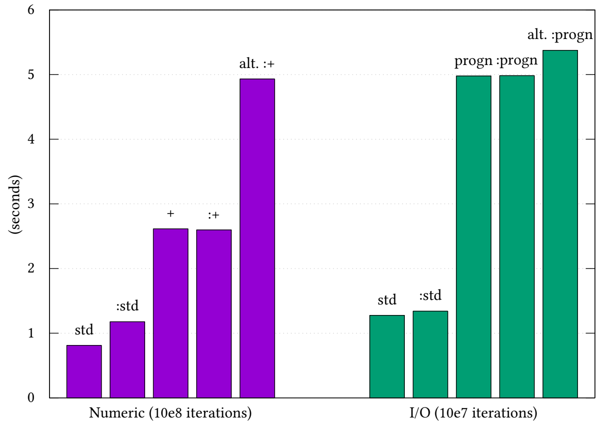


Figure 5: Benchmarks

(hence affecting the next regular call to it as well) and client maintenance will be triggered. Once an alternative discriminating function has been memoized, an "alternative call" will essentially require looking it up in a hash table (twice if `find-method-combinator` is involved in the call) before calling it.

In order to both confirm and illustrate those points, some rough performance measurements have been conducted and are reported in Figure 5. The first batch of timings involve a generic function with 4 methods simply returning their (numerical) argument. The second one involves a generic function with 4 methods printing their argument on a stream with `format`. The idea is that the methods in the numerical case are extremely short, while the ones performing I/O take much longer to execute. The timings are presented in seconds, for 10^8 and 10^7 consecutive iterations respectively.

The first two bars show the timings for a regular generic function with the standard method combination, and an equivalent combined generic function with the `:standard` combinator. In the numerical case, we observe a 45% performance loss, while in the I/O case, the difference is of 5%. This is due to SBCL optimizing the `standard` method combination but not the `:standard` combinator.

The next two bars show the timings for a built-in method combination compared to its equivalent combinator (`+` for the numerical case, `progn` for the I/O one). Recall that short combinators are in fact implemented as long ones, so the comparison is not necessarily fair. Nevertheless, the difference in either case is not measurable. Again, this is due to the fact that method combinators are implemented in terms of regular combinations.

Finally, the last bars show the timings involved in calling a generic function with an alternative combinator. Recall that this simply means calling a memoized discriminating function (hence taking the time displayed by the 4th bars) after having looked it up in a hash table. The large number of iterations measured ensures that the overhead of first-time memoization is cushioned). In the first (numerical) case, the overhead of using the `:+` combinator as an alternative instead of as the original one is of 90%. The methods being very short, the impact of an additional has table lookup is important. In the (longer) I/O case and for the `:progn` combinator however, this impact is amortized and falls down to 8%.

8 RELATED WORK

Greg Pfeil has put up a set of useful method combination utilities on Github[13]. These utilities include functions and macros frequently used in the development of new combinations or helping in the debugging of their expansion, and also some pre-made ones.

His library addresses some of the orthogonality concerns raised in Section 2.1. In particular, the `append/nconc` combination allows one to switch between the `append` and `nconc` operators without the need for requalifying all the primary methods (they still need to be qualified as `append/nconc` though, so are short forms defined with the `basic` combination).

Greg Pfeil's library does not attempt to address the primary concern of this paper, namely the overall consistence of the design of method combinations, and more specifically their namespace behavior. In one particular case, it even takes the opposite direction. The `basic` combination implements an interesting idea: it serves as a unique short form, making the operator a use-time value. In this way, it is not necessary anymore to define short combinations globally before using them. Every short combination essentially becomes local to one generic function.

Note that even though we attempted to do the exact opposite with method combinators, it is also possible to use them locally. Indeed, one can always break the link from a name to a combinator by calling `(setf (find-method-combinator name) nil)`. After this, the combinator will only be shared by combined generic functions already using it. Again, this behavior is similar to that of `find-class`[14].

Finally, the `basic` combination also addresses some of the concerns raised in Section 2.2. On top of allowing `:before` and `:after` methods in short forms, the distinction between definition-time and use-time options is removed. Indeed, since the operator has become a use-time option itself, the same holds for the option `:identity-with-one-argument`. What we have done, on the contrary, is to turn the `order` option into a definition-time one (see Section 4.3.1).

9 CONCLUSION

Method combinations are one of the very powerful features of CLOS, perhaps not used as much as they deserve, due to their apparent complexity and the terse documentation that the standard provides. The additional expressiveness and orthogonality they aim at providing is also hindered by several weaknesses in their design.

In this paper, we have provided a detailed analysis of these problems, and the consequences on their implementation in SBCL. Basically, the under-specification or inconsistency of the associated protocols can lead to non-portable, obscure, or even surprising, yet conforming behavior.

We have also proposed an extension called *method combinators* designed to correct the situation. Method combinators work on top of regular combinations in a non-intrusive way and behave in a more consistent fashion, thanks to a set of additional protocols following some usual patterns in the CLOS MOP. The full code is available on Github[15]. It has been successfully tested on SBCL.

[13] https://github.com/sellout/method-combination-utilities

[14] http://www.lispworks.com/documentation/lw70/CLHS/Issues/iss304_w.htm

[15] https://github.com/didierverna/ELS2018-method-combinators

10 PERSPECTIVES

Method combinators are currently provided as a proof of concept. They still require some work and also raise a number of new issues. First of all, it is our intention to properly package them and provide them as an actual Asdf system library. Next, we plan on investigating their implementation for vendors other than Sbcl, and in particular figuring out whether alternative combinators are possible or not. As of this writing, the code is in fact already ported to Cmucl, but surprisingly enough, it doesn't work as it is. Most of the tests fail or even trigger crashes of the Lisp engine. It seems that Cmucl suffers from many bugs in its implementation of Pcl, and it is our hope that fixing those bugs would suffice to get combinators working.

One still undecided issue is whether to keep long and short forms implemented separately (as in Pcl), or unify everything under the long form. We prefer to defer that decision until more information on how other vendors implement combinations is acquired. The second issue is on the status of the long form's lambda-list (currently deactivated) and consequently whether new combinators should be represented by new classes or only instances of the general one (see Section 4.3.1).

As we have seen, the lack of specification makes it impossible to implement method combinators in a completely portable way, and having to resort to `reinitialize-instance` is overkill in many situations, at least in theory. Getting insight on how exactly the different vendors handle applicable and effective methods caches could give us hints on how to implement method combinators more efficiently, alternative combinators in particular.

Apart from the additional functionality, several aspects of method combinators and their protocols only fill gaps left open in the Mop. Ultimately, these protocols (generic function updating notably) should belong in the Mop itself, although a revised version of it is quite unlikely to see the day. It is our hope, however, that this paper would be an incentive for vendors to refine their implementations of method combinations with our propositions in mind.

Finally, one more step towards full orthogonality in the generic function design can still be taken. The Common Lisp standard forbids methods to belong to several generic functions simultaneously. By relaxing this constraint, we could reach full 3D separation of concerns. Method combinators exist as global objects, so would "floating" methods, and generic functions simply become mutable sets of shareable methods, independent from the way(s) their methods are combined.

ACKNOWLEDGMENTS

Richard P. Gabriel provided some important feedback on the history of method combinations, Christophe Rhodes some documentation on Sbcl's internals, and Pascal Costanza and Martin Simmons some valuable insight or opinions on several aspects of the Mop.

REFERENCES

[1] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. Commonloops: Merging lisp and object-oriented programming. *SIGPLAN Notices*, 21(11):17–29, June 1986. ISSN 0362-1340. doi: 10.1145/960112.28700. URL http://doi.acm.org/10.1145/960112.28700.

[2] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common lisp object system specification. *ACM SIGPLAN Notices*, 23(SI):1–142, 1988. ISSN 0362-1340.

[3] Giuseppe Castagna. *Object-Oriented Programming, A Unified Foundation*. Progress in Theoretical Computer Science. Birkhäuser Boston, 2012. ISBN 9781461241386.

[4] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *SIGPLAN Lisp Pointers*, 5(1):182–192, January 1992. ISSN 1045-3563. doi: 10.1145/141478.141537. URL http://doi.acm.org/10.1145/141478.141537.

[5] Linda G. DeMichiel and Richard P. Gabriel. The common lisp object system: An overview. In *European Conference on Object Oriented Programming*, pages 151–170, 1987.

[6] Richard P. Gabriel and Kent M. Pitman. Technical issues of separation in function cells and value cells. *Lisp and Symbolic Computation*, 1(1):81–101, June 1988. ISSN 1573-0557. doi: 10.1007/BF01806178. URL https://doi.org/10.1007/BF01806178.

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

[8] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-61622-X.

[9] Sonja E. Keene. *Object-Oriented Programming in Common Lisp: a Programmer's Guide to Clos*. Addison-Wesley, 1989. ISBN 0-20117-589-4.

[10] Gregor J. Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[11] Patty Maes. Concepts and experiments in computational reflection. In *OOPSLA*. ACM, December 1987.

[12] Kristen Nygaard and Ole-Johan Dahl. The development of the SIMULA languages. *SIGPLAN Notices*, 13(8):245–272, August 1978. ISSN 0362-1340. doi: 10.1145/960118.808391. URL http://doi.acm.org/10.1145/960118.808391.

[13] Andreas Paepcke. User-level language crafting – introducing the Clos metaobject protocol. In Andreas Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, chapter 3, pages 65–99. MIT Press, 1993. Downloadable version at http://infolab.stanford.edu/~paepcke/shared-documents/mopintro.ps.

[14] Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, volume 2196 of *Lecture Notes in Computer Science*, pages 2–44. Springer Berlin / Heidelberg, 2001. ISBN 978-3-540-42558-8.

[15] Brian C. Smith. Reflection and semantics in Lisp. In *Symposium on Principles of Programming Languages*, pages 23–35. ACM, 1984.

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

A LONG SHORT METHOD COMBINATIONS

The Common Lisp standard provides several examples of built-in method combinations, and their equivalent definition in long form[16]. In a similar vein, the macro proposed in Figure 6 defines method combinations similar to those created with the short form, only with the following differences:

(1) the primary methods must not be qualified,
(2) `:before` and `:after` methods are available.

As in the original short form of `define-method-combination`, `identity-with-one-argument` is available as an optimization avoiding the call to the operator when a single method is invoked. The long form's lambda-list is used to define the `order` optional argument, directly passed along as the value of the `:order` keyword to the `primary` method group.

B LONG METHOD COMBINATION WOES

This section demonstrates an inconsistent behavior of generic functions using long method combinations in Sbcl, when the combination is redefined. First, we define a `progn`-like long method combination, ordering the methods in the default, most specific first way.

```
(define-method-combination my-progn ()
  ((primary () :order :most-specific-first :required t))
  `(progn ,@(mapcar (lambda (method)
```

[16] http://www.lispworks.com/documentation/lw70/CLHS/Body/m_defi_4.htm

```
(defmacro define-long-short-method-combination
    (name &key documentation identity-with-one-argument (operator name))
  "Define NAME as a long-short method combination.
OPERATOR will be used to define a combination resembling a short method
combination, with the following differences:
- the primary methods must not be qualified,
- :before and :after methods are available."
  (let ((documentation (when documentation (list documentation)))
        (single-method-call (if identity-with-one-argument
                              '`(call-method ,(first primary))
                              ``(,',operator (call-method ,(first primary))))))
    `(define-method-combination ,name (&optional (order :most-specific-first))
       ((around (:around))
        (before (:before)) ;; :before methods provided
        (primary (#| combination name removed |#) :order order :required t)
        (after (:after)))  ;; :after methods provided
       ,@documentation
       (flet ((call-methods (methods)
                (mapcar (lambda (method) `(call-method ,method)) methods)))
         (let* ((primary-form (if (rest primary)
                                `(,',operator ,@(call-methods primary))
                                ,single-method-call))
                (form (if (or before after)
                        `(multiple-value-prog1
                             (progn ,@(call-methods before) ,primary-form)
                           ,@(call-methods (reverse after)))
                        primary-form)))
           (if around
             `(call-method
               ,(first around) (,@(rest around) (make-method ,form)))
             form))))))
```

Figure 6: Long Short Method Combinations

```
                      `(call-method ,method))
                    primary)))
```

Next, we define a generic function using it with two methods.

```
(defgeneric test (i) (:method-combination my-progn)
  (:method ((i number)) (print 'number))
  (:method ((i fixnum)) (print 'fixnum)))
```

Calling it on a `fixnum` will execute the two methods from most to least specific.

```
CL-USER> (test 1)
FIXNUM
NUMBER
```

Next, we redefine the combination to reverse the ordering of the methods.

```
(define-method-combination my-progn ()
  ((primary () :order :most-specific-last :required t))
  `(progn ,@(mapcar (lambda (method)
                      `(call-method ,method))
                    primary)))
```

This does not (yet) affect the generic function.

```
CL-USER> (test 1)
FIXNUM
NUMBER
```

We now add a new method on `float`, which normally reinitializes the generic function.

```
(defmethod test ((i float)) (print 'float))
```

However, a `fixnum` call is not affected, indicating that some caching of the previous behavior is still going on.

```
CL-USER> (test 1)
FIXNUM
NUMBER
```

A first `float` call, however, will notice the new combination function.

```
CL-USER> (test 1.5)
NUMBER
FLOAT
```

Meanwhile, `fixnum` calls continue to use the old one.

```
CL-USER> (test 1)
FIXNUM
NUMBER
```

Session III: Cutting Edge

The Computational Structure of the Clifford Groups

Robert S. Smith
Rigetti Quantum Computing
Berkeley, CA, USA
robert@rigetti.com

ABSTRACT

We describe computational considerations of the Pauli and Clifford groups—including efficient data structures—for the purpose of randomized benchmarking of superconducting quantum computers. We then show particular implementation details in Common Lisp, including embedded domain-specific languages for specifying complex mathematical objects as Lisp macros.

ACM Reference format:
Robert S. Smith. 2018. The Computational Structure of the Clifford Groups. In *Proceedings of the 11th European Lisp Symposium, Marbella, Spain, April 16–17 2018 (ELS'18),* 10 pages.
https://doi.org/

1 INTRODUCTION

There are many ways to characterize the quality of a quantum computer. Consider a given collection of qubits whose composite state resides in a finite dimensional Hilbert space $\mathscr{H}$, and whose state is initially $v \in \mathscr{H}$. One pattern of characterization is to perform some unitary operation $f : \mathscr{H} \to \mathscr{H}$ on v to produce $f(v)$, and then perform the inverse $f^{-1}(f(v))$ to again produce v. On real devices however, the process of computing $f^{-1} \circ f$ won't exactly have the same effect as an identity operator, due to noise and limited execution fidelity. We distinguish between the ideal mathematical model of $f^{-1} \circ f$, and the physical process of computing the composition. In notation, we consider the transformation of v to v' identified by

$$v \xrightarrow{\text{the physical process of } f^{-1}\circ f} v'.$$

We can characterize our qubits with respect to an operator f or a family of such operators. Let us look at one simple example.

For a one-qubit system, we might consider rotations on the Bloch sphere about the x-axis. These are simple operations that are fundamental to quantum computation, and evaluating our ability to perform them would be helpful in characterizing our quantum computing system. The rotations are a continuous family of operators, which we can write down as the set

$$R_x := \left\{ \exp(-\tfrac{1}{2} i\theta X) : 0 \le \theta < 2\pi \right\},$$

where X is the Pauli X operator. Starting in the ground state $v := |0\rangle$, we select an element $f_\theta \in R_x$ which amounts to selecting a θ, then we perform the rotation, and then perform the inverse rotation. Pictorially, we have

$$|0\rangle \xrightarrow{\text{the physical process of } f_{-\theta}\circ f_\theta} v' := v'_0 |0\rangle + v'_1 |1\rangle .$$

What can we diagnose from v'? If v' always lies on the yz-plane, that is, if v' assumes the form

$$\begin{pmatrix} v'_0 \\ v'_1 \end{pmatrix} = \begin{pmatrix} \cos\frac{\alpha}{2} \\ -i\sin\frac{\alpha}{2} \end{pmatrix}$$

for some $\alpha \ne 0$, then we might conclude that our x-rotations are improperly tuned since we are overshooting by α.

The family of operations R_x forms a group under composition because of the following facts:

- Any composition of elements of R_x will also be in R_x.
- Composition of rotations is associative.
- Every rotation has an inverse rotation.
- There exists an identity rotation (i.e., $\theta = 0$).

Since R_x forms a group, it constrains the space being characterized. The constraints are able to help one diagnose issues when the constraints are violated. This is a pattern we want to maintain in other families of operations.

While R_x has nice algebraic properties, and also has a nice corresponding physical interpretation, it also comes with difficulties.

First, the group is continuous and hence infinite, which doesn't allow for any exhaustive testing. We might consider resolving this by choosing some discrete subgroup, like those generated integer multiples of some angle that partition the circle. But which subgroup? What does the fineness of angular resolution have to do with the quality of our qubits? These questions don't have a clean answer.

Second, even the continuous group doesn't explore much of the state space of the qubit, which in its entirety is the Bloch sphere. By attempting to extend R_x with other rotations, we quickly "saturate" the Bloch sphere. This also has the issue of being difficult to discretize.

Third, the group R_x (or any naive extension) doesn't generalize to multi-qubit systems well. Properties such as entanglement aren't explored by taking direct products of R_x.

One such collection of groups that satisfy all of our desired properties are the Clifford groups, denoted C_n. They have many benefits:

- They are discrete subgroups of $U(2^n)$, yet they broadly sample all dimensions of $U(2^n)$.
- They contain many operators of interest in quantum computing, such as controlled gates, phase gates, and even the Hadamard gate.

ELS'18, April 16–17 2018, Marbella, Spain
© 2018 Association for Computing Machinery.
ACM ISBN 978-2-9557474-2-1…$15.00
https://doi.org/

- The elements of the Clifford groups of any order are efficiently classically simulatable operators [6]. This means a classical computer can be used, with perfect accuracy, to verify computations a quantum computer makes.
- They use the familiar Pauli spin operators as a foundational construction.
- They are generated by comparatively few operators.
- They are, in some sense, the largest interesting groups that aren't computationally universal[1].

In this paper, we explore the Clifford groups, and talk about how we can compute with them[2]. We then prototype a way to do a specific kind of characterization of the above flavor called *randomized benchmarking*, using our development of the Clifford groups. We present in traditional mathematical style, interleaved with how the mathematics presents itself in Common Lisp.

We do not expound on exactly how quantum computer characterization works or how to do an analysis of data generated by a randomized benchmarking procedure. We also don't delve into important and relevant mathematical topics, such as the symplectic[3] representation of Clifford groups.

2 THE PAULI GROUPS

Before attempting to wrangle the Clifford groups, it's necessary to have a thorough understanding of the so-called Pauli group P_n, which forms the foundation of the definition of the Clifford groups. As we shall see, the Pauli groups are easy to define based off of the **Pauli spin matrices**

$$I := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \qquad X := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix},$$
$$Y := \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \text{ and } \qquad Z := \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

However, we wish to have an efficient means of manipulation on a computer, so we will transform the Pauli group into a series of more convenient representations. In particular, we will establish the following homomorphisms:

$$P_n \xrightarrow{\text{epimorphism}} \mathbb{F}_2^{2n} \xleftrightarrow{\text{isomorphism}} (\mathbb{Z}/4\mathbb{Z})^n.$$

The first homomorphism (an epimorphism in particular) will reveal a vector space structure on the group, and the second homomorphism (an isomorphism) will provide an efficient means of storage and manipulation of group elements.

The transformations between groups are not without complication, however. We will need to take care to maintain information about accumulated phases, which are complex numbers of unit modulus. A crescendo of algebra will finally climax at a computational representation of P_n—in fact, a family of isomorphic groups—called the computational Pauli groups $\mathscr{P}_n$. The computational Pauli groups will be used to efficiently construct the Clifford groups.

We start by defining the phases we will care about, and give an efficient representation of them. This will be used to define the Pauli groups.

Definition 1. The **phase group** $\Gamma := \langle i \rangle$ is the group under multiplication generated by $i = \sqrt{-1}$. This is exactly $\Gamma = \{1, i, -1, -i\}$.

Lemma 2. *The phase group is isomorphic to the additive group of $\mathbb{Z}/4\mathbb{Z}$, via the isomorphism $i^k \mapsto k$.*

Proof. Every element of Γ can be written as an integer power of i. Consider $g = i^m$ and $h = i^n$. Then $gh = i^m i^n = i^{m+n}$. The rest is straightforward to verify. □

Definition 3. The **Pauli group** of order n is the group under multiplication defined by

$$P_n := \left\{ \gamma \sigma_n \otimes \cdots \otimes \sigma_1 : \sigma_k \in \{I, X, Y, Z\} \right\},$$

where $\gamma \in \Gamma$ is called the **phase** of an element.

Lemma 4. *The size of the Pauli group $|P_n| = 4^{n+1}$.*

Proof. The Pauli group is equivalent to all length-n strings of the four Pauli spin matrices with any of the $|\Gamma| = 4$ phases. This is $4 \cdot 4^n = 4^{n+1}$. □

Fact 5. *The Pauli group P_n is indeed a group, and is a subgroup of the unitary group $U(2^n)$.*

Proof. By direct computation, it can be shown that P_1 is closed under multiplication and is a subgroup of $U(2)$. This structure lifts trivially to an n-fold tensor product of P_1, which is homomorphic to P_n. □

Often we will want to embed lower-dimensional Pauli group elements in a higher-dimensional Pauli group. We can do this with a rather trivial homomorphism.

Fact 6. *The group P_{n-1} is homomorphic to P_n via f defined by*

$$\begin{aligned} f &: P_{n-1} \longrightarrow P_n \\ &:= \sigma \mapsto I \otimes \sigma. \end{aligned}$$

As is, the elements of the Pauli groups would have a cumbersome representation on a computer, either as matrices or as formal symbols. We wish to create a data structure for the Pauli group elements. Because $Y = iXZ$, in some sense we might consider Y to be redundant, so long as we can keep track of the phases properly. Redundancy is best expressed by way of linear dependence, so we wish to identify a vector space to which we can map the Pauli groups. To go about this, we break the Pauli group into its phase portion Γ and a phaseless portion P_1/Γ, and attempt to construct something like looks like $\Gamma \times P_1/\Gamma$.

First, we will consider P_1 up to phases and then generalize to P_n.

[1] For $n \geq 1$, the addition of *any* operator in $U(2^n) \setminus C_n$ to the Clifford group of two or more qubits will make it suitable for universal computation [9].

[2] While the fundamental representations presented here are not new, the author feels they've hitherto lacked clear exposition, especially in the context of computer implementation.

[3] For the interested reader, the symplectic groups are in some sense the best and most compact way to understand the structure of the Clifford groups, particularly in regards to preserving commutation relations in the basis maps, as we will see in Lemma 23.

Lemma 7. *Consider the map $\varphi_1 : P_1/\Gamma \to \mathbb{F}_2^2$ defined exhaustively by*

$$\varphi_1(\sigma) = \begin{cases} (0,0)^\intercal & \text{for } \sigma = I, \\ (1,0)^\intercal & \text{for } \sigma = X, \\ (0,1)^\intercal & \text{for } \sigma = Z, \text{ and} \\ (1,1)^\intercal & \text{for } \sigma = Y. \end{cases}$$

The map φ_1 is an isomorphism between $(P_1/\Gamma, \cdot)$ and $(\mathbb{F}_2^2, +)$.

Proof. It is straightforward to verify that $\mathbb{F}_2^2$ forms a group. To show that the group structure is maintained up to phase, we note that the identity elements of each group—I and $(0,0)^\intercal$—are in correspondence by definition, and that $Y \sim XZ$ corresponds to $(1,1)^\intercal = (1,0)^\intercal + (0,1)^\intercal$, where $\sim$ is equality up to Γ-factors. □

The space $\mathbb{F}_2^2$ forms a vector space over $\mathbb{F}_2$ whose natural basis is $\{\varphi_1(X), \varphi_1(Z)\}$. We can create many copies of φ_1 to generalize to $\mathbb{F}_2^{2n}$.

Lemma 8. *Consider the map $\varphi_n : P_n/\Gamma \to \mathbb{F}_2^{2n}$ defined by*

$$\varphi_n(\sigma_n \otimes \cdots \otimes \sigma_1) := \bigtimes_{k=1}^{n} \varphi_1(\sigma_k).$$

The map φ_n is an isomorphism between $(P_n/\Gamma, \cdot)$ and $(\mathbb{F}_2^{2n}, +)$.

Proof. This is a direct consequence of the fact that

$$P_n/\Gamma = \underbrace{P_1/\Gamma \times \cdots \times P_1/\Gamma}_{n\text{ copies}}. \qquad □$$

The space $\mathbb{F}_2^{2n}$ also forms a vector space, and has a natural basis.

Definition 9. Define the set

$$B_n := \bigcup_{k=1}^{n} \{X_k, Z_k\}$$

called the **Pauli basis** of order n, where the subscript k indicates a tensor product of $n-1$ identity factors with the operator in the k^{th} position from the right.

The notational conventions X_k and Z_k will be used for the remainder of the paper.

Lemma 10. *The set $\varphi_n(B_n)$ forms a basis for $\mathbb{F}_2^{2n}$.*

Proof. The Pauli matrices X and Z are in direct correspondence with the standard basis of $\mathbb{F}_2^2$ from Lemma 7. The natural basis of a direct product of vector spaces is just the union of the original basis elements of each vector space with the zero element of each vector in all other positions of the product. □

The vector space structure of $\mathbb{F}_2^{2n}$, and hence the existence of a basis, is all that's important to us. We will reap its benefits, but we want to transform into an isomorphic space useful for building efficient data structures. The natural representation of $\mathbb{F}_2^{2n}$ would be $2n$-tuples of binary digits. While these can be stored compactly on a computer as bit arrays, their access and manipulation as Pauli operators are inconvenient. These can be transformed into length-n arrays of integers mod 4, which have many nice properties on a computer. As with our previous constructions, we will start with defining a base case, and then generalize.

Lemma 11. *The group $(\mathbb{F}_2^2, +)$ is isomorphic to $(\mathbb{Z}/4\mathbb{Z}, \oplus)$, where the symbol $\oplus$ denotes a binary exclusive-or on the transversal $\{0,1,2,3\}$ of $\mathbb{Z}/4\mathbb{Z}$.*

Proof. Binary exclusive-or is a bitwise operation equivalent to addition mod 2. The element $(b_0, b_1) \in \mathbb{F}_2^2$ corresponds to the binary number $b_1 b_0 \in \mathbb{Z}/4\mathbb{Z}$ written in positional notation. □

This representation is convenient on a computer since computing exclusive-or is very efficient, and storing integers as opposed to pairs of binary numbers is more convenient. Note that this lemma implies a particular choice of coset representative.

This construction generalizes to direct products easily.

Lemma 12. *The group $(\mathbb{F}_2^{2n}, +)$ is isomorphic to $\big((\mathbb{Z}/4\mathbb{Z})^n, \oplus\big)$, where $\oplus$ is understood to operate componentwise.*

Proof. This is a direct consequence of the construction of direct products of groups and Lemma 11. □

While $(\mathbb{Z}/4\mathbb{Z})^n \cong \mathbb{Z}/2^{2n}\mathbb{Z}$, which means that we can store elements of $\mathbb{F}_2^{2n}$ as arbitrary precision integers of $2n$ bits, it will be useful to query the terms of the direct product easily within a computer program for the purpose of phase tracking, which is the next order of business.

Since we have accomplished an efficient computer representation of the essential objects of the Pauli group—namely $\Gamma \cong \mathbb{Z}/4\mathbb{Z}$ and $P_n/\Gamma \cong (\mathbb{Z}/4\mathbb{Z})^n$—all that is left to fix up is the way phases accumulate under our group operation. As such, we make the group operation richer over the direct product of our isomorphic spaces.

Definition 13. The **computational Pauli group** of first order $\mathscr{P}_1$ is defined by elements of the form $\mathbb{Z}/4\mathbb{Z} \times \mathbb{Z}/4\mathbb{Z}$ with the operation $\odot$ defined by

$$(\alpha, u) \odot (\beta, v) := (\alpha + \beta + \varepsilon_{u,v}, u \oplus v),$$

where

$$\varepsilon_{\bullet,\bullet} : \underbrace{\mathbb{Z}/4\mathbb{Z} \times \mathbb{Z}/4\mathbb{Z}}_{\cong P_1/\Gamma \times P_1/\Gamma} \to \underbrace{\mathbb{Z}/4\mathbb{Z}}_{\cong \Gamma}$$

is a non-standard Levi–Civita symbol defined by

$$\varepsilon_{u,v} := \begin{cases} 1 & \text{if } (u,v) \in \{(1,3),(3,2),(2,1)\}, \\ 3 & \text{if } (v,u) \in \{(1,3),(3,2),(2,1)\}, \text{ and} \\ 0 & \text{otherwise.} \end{cases}$$

This symbol comes directly from the multiplication rules of the Pauli spin matrices.

The group $\mathscr{P}_1$ is very nearly isomorphic to the group $\Gamma \times P_1/\Gamma$—the phase group together with the phaseless Pauli elements—except for the additional phase correction determined by the elements being considered in P_1/Γ. Again, we can generalize to direct products.

Definition 14. The **computational Pauli group** of order n, denoted $\mathscr{P}_n$, is defined by elements of the form $\mathbb{Z}/4\mathbb{Z} \times (\mathbb{Z}/4\mathbb{Z})^n$ with the operation $\odot$ defined by

$$(\alpha, \vec{u}) \odot (\beta, \vec{v}) = (\alpha + \beta + \varepsilon_{\vec{u},\vec{v}}, \vec{u} \oplus \vec{v}), \qquad (2.1)$$

where

$$\varepsilon_{\bullet,\bullet} : \underbrace{(\mathbb{Z}/4\mathbb{Z})^n \times (\mathbb{Z}/4\mathbb{Z})^n}_{\cong P_n/\Gamma \times P_n/\Gamma} \to \underbrace{\mathbb{Z}/4\mathbb{Z}}_{\cong \Gamma}$$

is a generalized non-standard Levi–Civita symbol defined by

$$\varepsilon_{\vec{u},\vec{v}} := \sum_{k=1}^{n} \varepsilon_{u_k, v_k}.$$

Now we present the most important result of this section, which gives us a conveniently computable representation of P_n.

Theorem 15. *The group* $(P_n, \cdot)$ *is isomorphic to* $(\mathscr{P}_n, \odot)$.

Proof. We have shown that $\Gamma \cong \mathbb{Z}/4\mathbb{Z}$ and that $P_n/\Gamma \cong (\mathbb{Z}/4\mathbb{Z})^n$, so we can conclude the set under consideration is the right size and contains the requisite information to construct a bijection between the *sets* P_n and $\mathscr{P}_n$. We have also shown isomorphism up to phase by way of Lemma 7 and Lemma 12. What is left to show is that we recover our group structure via the Levi–Civita term in the phase.

Label the Pauli matrices (X, Y, Z) as $(\sigma_1, \sigma_2, \sigma_3)$. Then the multiplication rule for σ_a and σ_b is

$$\sigma_a \sigma_b = \delta_{ab} I + \sum_{c=1}^{3} i\epsilon_{abc}\sigma_c. \qquad (2.2)$$

Here, ϵ_{abc} is the standard three-dimensional Levi–Civita symbol, equal to 1 when abc is an even permutation, -1 when it is odd, and 0 otherwise. (Note that this means the sum will always have a single non-vanishing term.) This relates to our Levi–Civita symbol in the following way. Let $c \neq a, b$. Then the accumulated phase is

$$i\epsilon_{abc} = i^{\varepsilon_{\pi a, \pi b}},$$

where $\pi = (1, 2, 3) \mapsto (1, 3, 2)$ is used to translate between indexing conventions. Since (2.2) is an equation relating objects in P_n, we invoke the isomorphism of Lemma 2 to recover the phase found in (2.1). This is the only additional phase that gets accumulated. □

2.1 Implementation in Lisp

In the presentation of Lisp code throughout this paper, we omit unnecessary definitions and details, and provide the code to give a sense of its efficient implementation in Lisp.

The main data structures involved in the representation of Pauli operators are as follows. Using Steel Bank Common Lisp (SBCL), the `simple-array` of `Z/4Z` will be a specialized array.

```
(deftype Z/4Z ()
  `(mod 4))

(deftype pauli-components ()
  '(simple-array Z/4Z (*)))

(defstruct pauli
  (components nil :type pauli-components))

(defmethod num-qubits ((p pauli))
  (1- (length (pauli-components p))))

(defun make-components (num-qubits)
  (make-array (1+ num-qubits) :element-type 'Z/4Z
                              :initial-element 0))

(defun phase-factor (p)
  (aref (pauli-components p) 0))

(defun pauli-tensor-product (p)
  (coerce (subseq (pauli-components p) 1) 'list))
```

The principle operation of interest is the implementation of the group operation.

```
(defmacro pair-membership (u v &rest cases)
  `(or ,@(loop :for (ui vi) :in cases
               :collect `(and (= ,u ,ui)
                              (= ,v ,vi)))))

(defun levi-civita (u v)
  (cond
    ((pair-membership u v (1 3) (3 2) (2 1))
     1)
    ((pair-membership v u (1 3) (3 2) (2 1))
     3)
    (t
     0)))

(defun multiply-components (a b)
  (let* ((n (length a))
         (c (make-components (1- n))))
    ;; Get the initial phase.
    (setf (aref c 0) (%phase-mul (aref a 0) (aref b 0)))
    ;; Get the tensor components, modifying the
    ;; phase along the way.
    (loop :for i :from 1 :below n
          :for ai :of-type Z/4Z := (aref a i)
          :for bi :of-type Z/4Z := (aref b i)
          :do (setf (aref c i) (logxor ai bi))
              (setf (aref c 0) (%phase-mul (aref c 0)
                                           (levi-civita ai bi))))
    c))
```

The function `%phase-mul` is as described earlier.

Finally, we share a function to convert Pauli Y operators into XZ products.

```
(defun pauli-basis-decompose (pauli)
  (let ((p (phase-factor pauli))
        (n (num-qubits pauli)))
    (loop :for idx :below (num-qubits pauli)
          :for base4-rep := (aref (pauli-components pauli)
                                  (1+ idx))
          :append (ecase base4-rep
                    (0 nil)
                    (1 (list (embed +X+ n (list idx))))
                    (2 (list (embed +Z+ n (list idx))))
                    (3 ; side-effect: update the phase *= i
                     (setq p (%phase-mul p 1))
                     (list (embed +X+ n (list idx))
                           (embed +Z+ n (list idx)))))
            :into b
          :finally (return (values b p)))))
```

3 THE CLIFFORD GROUPS

Now we turn our attention to the Clifford groups. As outlined, the Clifford groups are a particularly interesting subgroup of the unitary group that remain computationally efficient to explore and manipulate. The Clifford groups, roughly speaking, consist of the unitary matrices which conjugate Paulis to Paulis. These "conjugation invariant" groups are called "normalizers."

Definition 16. Consider a group G and a subgroup $S \subseteq G$. The **normalizer** of S with respect to G is defined by

$$N_G(S) := \{g \in G : \forall s \in S,\, gsg^{-1} \in S\}.$$

We consider normalizers with respect to the unitary group, where all non-projective quantum operators live. Without consideration of a particular subgroup S, the unitary group presents a slight problem: the normalizers are infinite. This can be seen easily. Consider an element $e^{i\theta}I \in U(2^n)$. Then this element is in $N_{U(2^n)}(S)$ because[4]

$$(e^{i\theta}I)S(e^{i\theta}I)^{-1} = (e^{i\theta}I)S(e^{-i\theta}I) = S.$$

Various remedies to this issue have been presented in the literature. Calderbank et al. [3] decide to limit the ring over which matrix elements of the normalizer can live—in particular, over the ring[5] $\mathbb{Q}[\eta]$ for $\eta := e^{i\pi/4} = (1+i)/\sqrt{2}$. This is convenient algebraically and computationally, because all elements can be represented precisely as formal polynomials of a variable η, but this representation over-counts quantum-mechanically equivalent operators by a factor of 8, because η is a primitive eighth root of unity.

Another remedy is to brute force the issue by modding out the normalizer by all complex phases, i.e., $U(1)$. This is the approach we take with our definition of the Clifford group, but will require care in the representation of elements.

Definition 17. The **Clifford group** of order n is

$$C_n := N_{U(2^n)}(P_n)/U(1)$$

under matrix multiplication.

This definition is concise, and is equivalent to the group of equivalence classes of

$$\{U \in U(2^n) : \forall \sigma \in P_n,\, U\sigma U^\dagger \in P_n\}$$

under the equivalence relation $U \sim e^{i\theta}U$.

We wish to show that the Clifford groups actually form groups. The following simple algebra fact will assist us.

Lemma 18. *For an invertible matrix U, the map $\kappa_U(x) := UxU^{-1}$ is a group homomorphism.*

Proof. To show that κ_U is a homomorphism, we show how it transforms products. Consider the product of square matrices AB. We show that $\kappa_U(AB) = \kappa_U(A)\kappa_U(B)$. This is straightforward algebra:

$$\begin{aligned}\kappa_U(AB) &= U(AB)U^{-1}\\ &= UA(U^{-1}U)BU^{-1}\\ &= (UAU^{-1})(UBU^{-1})\\ &= \kappa_U(A)\kappa_U(B).\end{aligned}$$

One can additionally easily see that $\kappa_U(I) = I$, concluding the proof. □

This is important because it means that in order to verify a product $\sigma\tau \in P_n$ gets conjugated back into P_n, we must only verify that each factor σ and τ get conjugated into P_n. This homomorphic property is critical to proving some later results (e.g., Lemma 22).

Theorem 19. *The Clifford groups are groups under matrix multiplication.*

Proof. We enumerate each of the group axioms.

- **Identity** The identity matrix is the identity element of the Clifford group. It is straightforward to verify this.
- **Closure** If $A, B \in C_n$, then by Lemma 18, AB is also in C_n.
- **Associativity** Matrix multiplication is associative.
- **Inverses** First, it's clear that all elements of C_n are invertible because they are unitary. Second, we show the inverses are also in C_n. Let $A \in C_n$ and let $A' = A^{-1}$. By definition, for all $\sigma \in P_n$, there exists a $\tau \in P_n$ such that $A\sigma A^{-1} = \tau$. Rewriting this as $\sigma = A'\tau(A')^{-1}$ shows that $A' \in C_n$. □

So far, the Clifford groups have an implicit definition; they're defined by their relationship with the Pauli groups. As we did with the Pauli groups, we would like to find an isomorphism between C_n and some computable representation of C_n. To do this, we start by probing the structure of the group to get to a computable representation of the group. We first show that no element, except identity, can be conjugated to identity.

Lemma 20. *For all $g \in C_n$, the only $\sigma \in P_n$ that gets conjugated by g to identity is identity.*

Proof. Consider an arbitrary $g \in C_n$ and a $\sigma \in P_n$ such that $g\sigma g^{-1} = I$. Then $g\sigma = g$ and therefore $\sigma = I$. □

We can use Lemma 18 and the vector space structure of P_n/Γ to derive an important structural theorem about the Clifford groups.

Definition 21. For $g \in C_n$, define $\mathscr{M}_g : B_n \longrightarrow P_n$ as

$$\mathscr{M}_g(b) := gbg^{-1}.$$

Call this the **basis map** of g.

While basis maps are defined with maps on P_n, one would actually use $\mathscr{P}_n$ in a computer implementation. Practically, if we order[6] the basis map, this amounts to storing a length-$2n$ array of length-$(n+1)$ integer arrays.

Lemma 22. *The basis map of g determines g uniquely.*

Proof. Because of Lemma 10, every element $\sigma \in P_n$ can be written uniquely as a possibly empty product $\gamma b_1 \cdots b_l$ for $b_i \in B_n$ and $\gamma \in \Gamma$. If $\sigma' := g\sigma g^{-1}$, which is also in P_n by definition, then σ' can be written as

$$\gamma \prod_{k=1}^{l} gb_kg^{-1}$$

by Lemma 18. As such, for a particular σ, it suffices to only consider how g conjugates each b_k. Considering every element of

[4] These elements are said to be at the **center** of the normalizer.

[5] This turns out to be the "smallest" ring you can choose to represent all matrices of the Clifford group.

[6] For instance, $X_i \mapsto 2i$ and $Z_i \mapsto 2i+1$.

P_n as opposed to just one, we conclude that all elements of B_n shall be considered. The collection of such maps forms a total function on B_n. Since conjugation by unitaries is invertible, no two independent b_k will conjugate equally. Therefore, the map is unique. □

This theorem doesn't tell us how to recover g from $\mathscr{M}_g$ in any particular representation (e.g., matrix), and doesn't tell us exactly which basis maps in general (i.e., any such map from B_n to P_n) are representative of Clifford group elements. One approach to answering the latter question is to consider how the operators should commute.

Lemma 23. *All pairs of elements in B_n either commute or anti-commute.*

Proof. All pairs of elements of B_n trivially commute, except for X_i and Z_i for $1 \le i \le n$. □

Lemma 24. *If Pauli basis elements b and b' commute (resp. anti-commute), then $\mathscr{M}_g(b)$ and $\mathscr{M}_g(b')$ commute (resp. anti-commute).*

Proof. We show both of these implications through direct computation. First we consider commuting b and b' :

$$\begin{aligned}[\mathscr{M}_g(b), \mathscr{M}_g(b')] &= \mathscr{M}_g(b)\mathscr{M}_g(b') - \mathscr{M}_g(b')\mathscr{M}_g(b)\\ &= (gbg^{-1})(gb'g^{-1}) - (gb'g^{-1})(gbg^{-1})\\ &= gbb'g^{-1} - gb'bg^{-1}\\ &= gbb'g^{-1} - gbb'g^{-1}\\ &= 0.\end{aligned}$$

Next we consider the anti-commuting b and b':

$$\begin{aligned}\{\mathscr{M}_g(b), \mathscr{M}_g(b')\} &= \mathscr{M}_g(b)\mathscr{M}_g(b') + \mathscr{M}_g(b')\mathscr{M}_g(b)\\ &= (gbg^{-1})(gb'g^{-1}) + (gb'g^{-1})(gbg^{-1})\\ &= gbb'g^{-1} + gb'bg^{-1}\\ &= gbb'g^{-1} - gbb'g^{-1}\\ &= 0.\end{aligned}$$ □

Another way to probe the structure of the basis maps is to look at how unitary operators conjugate Hermitian operators. This leads to a result about the properties of the image of $\mathscr{M}_g$.

Lemma 25. *Let U be a unitary matrix and A be Hermitian. Then UAU^{-1} is Hermitian.*

Proof. Recall that $UU^\dagger = I$ so $U^{-1} = U^\dagger$. We wish to show that $UAU^{-1} = (UAU^{-1})^\dagger$ by expanding the right-hand side:

$$\begin{aligned}(UAU^{-1})^\dagger &= (UAU^\dagger)^\dagger\\ &= (U^\dagger)^\dagger A^\dagger U^\dagger\\ &= UA^\dagger U^\dagger\\ &= UAU^\dagger \qquad (A = A^\dagger \text{ because } A \text{ is Hermitian})\\ &= UAU^{-1}.\end{aligned}$$ □

Lemma 26. *The image of a basis map consists only of Hermitian operators, and as such, Pauli operators with eigenvalues ±1.*

Proof. The basis map is a collection of maps of the form $\sigma \mapsto g\sigma g^{-1}$ for $\sigma \in B$ and a unitary operator g. Since σ is Hermitian, then so is $g\sigma g^{-1}$. Since unitary transformations are isometries, the eigenvalues of σ remain ±1. □

Now we can construct a computationally convenient version of the Clifford groups.

Definition 27. The **computational Clifford group** of order n, denoted $\mathscr{C}_n$, is defined as the group of basis maps under composition. Specifically, it is the set

$$\mathscr{C}_n := \{\mathscr{M}_g : g \in C_n\}.$$

Lemma 28. *The computational Clifford group is a group and is isomorphic to C_n.*

Proof. The essential fact to verify is that $\mathscr{M}_{fg} = \mathscr{M}_f \circ \mathscr{M}_g$, i.e., that $u \mapsto \mathscr{M}_u$ is an isomorphism. To this end, we have shown the existence of a bijection between C_n and $\mathscr{C}_n$ in Lemma 22. Now, we do algebra:

$$\begin{aligned}\mathscr{M}_{fg} &= b \mapsto (fg)b(fg)^{-1}\\ &= b \mapsto (fg)b(g^{-1}f^{-1})\\ &= b \mapsto f(gbg^{-1})f^{-1}\\ &= (b \mapsto fbf^{-1}) \circ (b \mapsto gbg^{-1})\\ &= \mathscr{M}_f \circ \mathscr{M}_g.\end{aligned}$$ □

The proof of this lemma actually has a very computationally convenient tool: the composition of elements of the computable Clifford group. In fact, we can calculate its computational complexity.

Fact 29. *The time complexity of composition in $\mathscr{C}_n$ is $O(n^2)$.*

Proof. Consider two elements $f, g \in \mathscr{C}_n$. Elements of $\mathscr{C}_n$ are maps whose domain is B_n, which has a cardinality of $2n$. So $f \circ g$ amounts to iterating through the map of g, a linear-time operation, which are essentially $(b, \gamma p)$ pairs, and computing $(b, \gamma f(p))$. Since p may contain Pauli Y operators, it must be re-expressed as basis elements (linear in the length of p, which is n). In the worst case, for each pair in g, we may have to do a linear-time operation to multiply by the elements of f, giving $O(n^2)$. □

We are finally equipped with enough structure on the basis maps to prove an important result about the Clifford groups: their size.

Theorem 30. *The sizes of the Clifford groups can be calculated by the recurrence $|C_1| = 24$ and $|C_n| = 2(4^n - 1)4^n|C_{n-1}|$.*

Proof. The size of the Pauli basis is $|B_n| = 2n$. We proceed to construct the base case $|C_1|$ and then calculate a recurrence for $|C_n|$ based on Lagrange's theorem.

Let P_n' be the Hermitian subset of $P_n \backslash \{\pm I, \pm iI\}$, whose size is $\frac{1}{2}(4^{n+1} - 4) = 2(4^n - 1)$, which we consider because of Lemma 20 and Lemma 26. The size of C_1 consists of all possible ways to map X into P_1', which gives $|P_1'| = 2(4^1 - 1) = 6$ ways, and all possible ways of mapping Z into elements of P_1' which anti-commute with the image of X, of which there will be two fewer choices, giving $|C_1| = 6 \cdot 4 = 24$.

n	$\lvert C_n \rvert$
1	24
2	11,520
3	92,897,280
4	12,128,668,876,800
5	25,410,822,678,459,187,200

Table 1: Sizes of the first few Clifford groups.

Computing $|C_n|$ amounts to selecting a map for X_n and Z_n, then computing $|C_{n-1}|$. This is because fixing $\{X_n, Z_n\}$ amounts to constructing a group isomorphic to C_{n-1}, and by Lagrange's theorem, $|C_n| = |C_n/C_{n-1}| \cdot |C_{n-1}|$.

Let us fix the image of X_n; call it $\mathscr{M}(X_n)$. We can choose any element of P'_n, giving us $|P'_n|$ choices.

After fixing $\mathscr{M}(X_n)$, we must fix an image of Z_n, call it $\mathscr{M}(Z_n)$, which anticommutes with $\mathscr{M}(X_n)$. To do this, we select an arbitrary non-identity factor of the tensor product $\mathscr{M}(X_n)$, and freely select any Hermitian operators in P_n for the other $n-1$ factors. There are 4^{n-1} choices here. For the last factor, we choose one of $\{I, X, Y, Z\}$ to make sure the operator anticommutes. (This will be based on the previous $n-1$ choices, of course.) There will be 2 such choices. Lastly, we can choose our eigenvalue, ± 1, giving us 2 more choices. In total, there are $2 \cdot 2 \cdot 4^{n-1} = 4^n$ choices for $\mathscr{M}(Z_n)$.

In total, there are $|P'_n| \cdot 4^n = 2(4^n - 1)4^n$ choices for $\mathscr{M}(\{X_n, Z_n\})$, which are the number of cosets $|C_n/C_{n-1}|$. This gives us

$$|C_n| = 2(4^n - 1)4^n|C_{n-1}|,$$

our desired result. □

The recurrence can be written in closed form by multiplying it out:

$$|C_n| = 2^{n^2+2n}\prod_{k=1}^{n}(4^k - 1).$$

The first few values of this can be seen in Table **??**.

3.1 Implementation in Lisp

The following code shows the fundamentals of how to implement Clifford group elements in Common Lisp. We show one optimization where we can store the basis map as a vector, where the indexes $(2j, 2j+1)$ correspond to the Pauli operators (X_j, Z_j) respectively.

```
(defun map-pauli-basis (n f)
  (dotimes (idx n)
    (funcall f (* 2 idx)      (embed +X+ n (list idx)))
    (funcall f (1+ (* 2 idx)) (embed +Z+ n (list idx)))))

(defstruct clifford
  (basis-map nil :type simple-vector))

;; clifford * pauli
(defun apply-clifford (c p)
  (multiple-value-bind (b ph) (pauli-basis-decompose p)
    (reduce #'group-mul b
            :initial-value (pauli-identity (num-qubits p) ph)
            :key (lambda (e)
                   (aref (basis-map c)
                         (pauli-to-index e))))))

(defun clifford-identity (n)
  (let ((bm (make-array (* 2 n))))
    (map-pauli-basis n (lambda (j p) (setf (aref bm j) p)))
    (make-clifford :basis-map bm)))

;; clifford * clifford
(defmethod group-mul ((c1 clifford) (c2 clifford))
  (let ((c1c2 (clifford-identity (num-qubits c1))))
    (map-into (basis-map c1c2)
              (lambda (c2p) (apply-clifford c1 c2p))
              (basis-map c2))
    c1c2))
```

3.2 A Lisp Macro for Specifying Clifford Elements

Specifying Clifford group elements is cumbersome if we use `make-pauli` and `make-clifford` manually. Writing a Clifford element in the notation of mathematics, however, is straightforward. We adopt the mathematical notation as a domain specific language, allowing us to write Clifford elements like so:

```
(defvar *hadamard*
  (clifford-element
    X -> Z
    Z -> X))

(defvar *phase*
  (clifford-element
    X -> Y))

(defvar *cnot*
  (clifford-element
    XI -> XX
    IZ -> ZZ))
```

These are the Hadamard, phase, and controlled-not gates, which we introduce in the next section. The macro enabling this can be found in Figure 6.1.

4 SUBROUTINES FOR RANDOMIZED BENCHMARKING

The reason we have studied the Clifford groups is to provide a flexible yet efficient way to characterize the quality of operations on a quantum computer. A specific technique called **randomized benchmarking** amounts to applying a Clifford gate followed by its inverse, and computing fidelity statistics.

The fundamental procedure used in randomized benchmarking is the generation of a sequence n uniformly random Clifford operators $g_1, \ldots, g_n$ and the generation of its inverse $g^{-1} := (g_n \cdots g_1)^{-1}$. As described in §1, the process of computing $g^{-1}g$ may not have the same effect as computing identity.

Fundamentally, the following subroutines must be implemented:

Composition The composition of a sequence of elements of $\mathscr{C}_n$.

Random Selection The uniformly random selection of an element of $\mathscr{C}_n$.

Inversion The inversion of an element of $\mathscr{C}_n$.

Generator Decomposition The computation of a sequence of elements of $S \subset \mathscr{C}_n$—called the *generators*—that reconstruct any element of $\mathscr{C}_n = \langle S \rangle$.

We are interested in Generator Decomposition because quantum computers usually have a limited selection of native gates; it is usually not possible to apply a Clifford gate directly, but rather indirectly through a sequence of native gates.

In the next few subsections, we will describe how to do these operations. First, however, we will discuss a brute-force approach to studying $\mathscr{C}_n$ for small n.

4.1 Exhaustive Generation of the Clifford Group

The number of elements of the Clifford groups for $1 \leq n \leq 3$ permits us to store the entire group without much difficulty[7]. Generating and storing the entire group allows us to perform random selection, inversion, and generator decomposition very efficiently.

Generating necessitates a set of generators. We will define this well-known term formally.

Definition 31. Consider a finite group G along with a set of elements $S \subseteq G$. The set S is said to **generate** G if every element of G can be represented as a combination of elements of S. The elements of S are called **generators** of G. One writes $\langle S \rangle$ to denote the group generated by S.

The usual set of generators for C_n are

$$\mathsf{H} := \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \ \mathsf{P} := \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \text{ and } \mathsf{CNOT} := \begin{pmatrix} 1&0&0&0 \\ 0&1&0&0 \\ 0&0&0&1 \\ 0&0&1&0 \end{pmatrix}.$$

It is understood that the generators for C_n are actually tensor products of these with identity. We will not prove that these generate the Clifford groups.

We must transform the generators of C_n to those in $\mathscr{C}_n$. This is done directly from Definition 21. Once we have these, we can generate the group exhaustively. We do this by way of the construction of a table called a "God's table."

Definition 32. Consider a finite group $G = \langle S \rangle$ and $S := \{s_1, \ldots, s_k\}$. A[8] **God's table** of G is a total function $T : G \to S \cup \{I\}$ such that the fixed point iteration of Δ_T always converges to I for all $g \in G$, where Δ_T is defined to be the **decomposition map** $\Delta_T(g) := T(g)^{-1}g$.

Fact 33. *The only $g \in G$ such that $T(g) = I$ is $g = I$.*

Proof. Suppose there was a $g \neq I$ such that $T(g) = I$. One step of the fixed point iteration on g would give $\Delta_T(g) = T(g)^{-1}g = g$. But $\Delta_T(g) = g$ is a non-identity fixed point, violating the hypothesis that T is a God's table. □

A God's table is, simply put, a table mapping elements of G to a generator which takes the element one step closer to identity. The decomposition map is important because it solves the "Generator Decomposition" and "Inversion" problems; repeated application of Δ_T, recording each $T(g)$ along the way, gives us a sequence to reconstruct g, and hence, its inverse g^{-1}. The number of elements in the generator decomposition implied by T is a useful metric.

Definition 34. Given a group G and a God's table T, the **length** of an element $g \in G$ with respect to T is the minimum non-negative integer k such that $\Delta_T^k(g) = I$. We write $\ell_T(g)$ or sometimes just $\ell(g)$ if the context is clear.

Definition 35. A God's table T is said to be **optimal** iff there does not exist another God's table T' such that for any $g \in G$, $\ell_{T'}(g) < \ell_T(g)$.

In some literature, a God's table is in fact assumed to be optimal. We do not make that assumption here.

The construction of an optimal God's table is simple. One does a breadth-first traversal of the group based off of the generators, recording the generators that produced the new elements in the table. See algorithm 1.

Algorithm 1 Optimal God's table generation.

```
INPUT: Generators S
OUTPUT: God's table T

T[I] = I
for s in S:
    T[s] = s

q = empty queue

for s in S:
    queue s in q

while q has elements:
    next = pop q
    for s in S:
        g = s * next
        if g does not exist in T:
            queue g in q
            T[g] = s
```

Theorem 36. *Algorithm 1 produces an optimal God's table.*

Proof. This is immediate from the fact that breadth-first search on a graph will visit each node in the fewest number of steps. □

A God's table can be created by any generating set, even redundant ones. (A redundant generating set $\{s_1, \ldots, s_k\}$ is one such that there exist s_i expressible as combinations of $s_{\neq i}$.) This can be used to prioritize certain generators.

As a final note, if a God's table uses a sufficient data structure, then random selection amounts to a randomly selected non-negative integer below $|C_n|$.

4.2 Composition

Composition was described as a polynomial process in Lemma 28.

4.3 Random Selection

Random selection amounts to randomly selecting the image of a basis map. This can be done easily, though a bit laboriously, if commutation relations are kept according to Lemma 24. Refer

[7] As a back-of-the-envelope calculation, an element of $\mathscr{P}_n$ consumes about $1 + \lceil 2n/(4 \cdot 8) \rceil$ quad-words of memory. Since each element of $\mathscr{C}_n$ is a collection of $2n$ maps to elements of $\mathscr{P}_n$, an element of $\mathscr{C}_n$ will consume around $n^2/8 + 2n \in O(n^2)$ quad-words of memory. For $n = 2$, storing $\mathscr{C}_n$ amounts to absolutely no more than a megabyte.

[8] A God's table isn't necessarily unique.

d	C_2 Count	C_3 Count
0	1	1
1	6	12
2	26	99
3	96	668
4	292	3930
5	734	20,626
6	1494	97,273
7	2448	409,153
8	3035	1,506,547
9	2424	4,706,972
10	912	11,870,008
11	52	22,653,233
12	0	29,319,401
13	0	19,641,316
14	0	2,663,872
15	0	4169
$\sum$	11,520	92,897,280

Table 2: Number of elements at depth d.

to Theorem 30 for the considerations in counting the number of elements of the Clifford group constructively, which may be amended to create a procedure to generate random elements of the Clifford group.

4.4 Inversion

One very simple method to invert elements of $\mathscr{C}_n$ is to compute their order.

Definition 37. Given a finite group G, the **order** of an element $g \in G$ is the smallest positive integer k such that $g^k = I$.

This definition implies that the order of I is 1.

Lemma 38. *If the order of an element $g \in G$ is k, then $g^{-1} = g^{k-1}$.*

Proof. Since $g^k = I$, then $g^{-1}g^k = g^{-1} \implies g^{k-1} = g^{-1}$. □

In the worst case, the order will need a number of multiplications equal to the diameter of the group. (Upper bounds of the diameter of permutation groups can be found in [2].)

4.5 Generator Decomposition

Generator decomposition is most difficult to do without a full description of the group. In the general case, there are other approaches we would suggest, though we have not tried them.

The first approach is to *search* for the decompositions by viewing the group as a Cayley graph and traversing it. One can explore the graph using techniques such as iterative deepening depth-first search [8]. This would require, however, pruning strategies to be made feasible.

The second approach would be to use the tools of computational group theory to explore the Clifford group. In particular, we can study $\mathrm{Sp}(2n, \mathbb{F}_2) \cong C_n/P_n$ which admits a permutation representation of bounded degree [4]. In a permutation representation, using a computational group theory library such as cl-permutation [10], there are many optimal and sub-optimal techniques for computing generator factorizations.

5 CONCLUSION

We have presented a data structure for elements of the Clifford groups, and both practical and efficient ways to manipulate them. Using this, we have presented a sufficient number of subroutines to implement randomized benchmarking of a quantum computer for a small number of qubits. We have also shown how Common Lisp can assist the development of efficient Clifford group computations.

6 ACKNOWLEDGEMENTS

We acknowledge Peter Karalekas, Chris Osborn, and Anthony Polloreno of Rigetti Quantum Computing for their helpful comments and corrections. We would also like to thank Toby Thain for his valuable improvements.

REFERENCES

[1] Aaronson, S., and Gottesman, D. Improved simulation of stabilizer circuits. *Phys. Rev. A 70* (Nov 2004), 052328.

[2] Babai, L., and Seress, Á. On the diameter of permutation groups. *European Journal of Combinatorics 13*, 4 (1992), 231 – 243.

[3] Calderbank, A. R., Rains, E. M., Shor, P. W., and Sloane, N. J. A. Quantum error correction via codes over GF(4), 1996.

[4] Cooperstein, B. N. Minimal degree for a permutation representation of a classical group. *Israel Journal of Mathematics 30*, 3 (Sep 1978), 213–235.

[5] Dehaene, J., and Moor, B. D. The Clifford group, stabilizer states, and linear and quadratic operations over GF(2). Phys. Rev. A 68, 042318 (2003), 2003.

[6] Gottesman, D. The Heisenberg representation of quantum computers. Proceedings of the XXII International Colloquium on Group Theoretical Methods in Physics, eds. S. P. Corney, R. Delbourgo, and P. D. Jarvis, pp. 32–43 (Cambridge, MA, International Press, 1999), 1998.

[7] Haah, J. Algebraic methods for quantum codes on lattices, 2016.

[8] Korf, R. E. Depth-first iterative-deepening: An optimal admissible tree search. *Artif. Intell. 27*, 1 (Sept. 1985), 97–109.

[9] Nielsen, M. A., and Chuang, I. L. *Quantum Computation and Quantum Information: 10th Anniversary Edition*, 10th ed. Cambridge University Press, New York, NY, USA, 2011.

[10] Smith, R., and Pawlowski, B. Efficient finite permutation groups and homomesy computation in common lisp. In *Proceedings of ILC 2014 on 8th International Lisp Conference* (New York, NY, USA, 2014), ILC '14, ACM, pp. 60:60–60:67.

```lisp
(defmacro clifford-element (&body body)
  (let ((clifford (gensym "CLIFFORD-"))
        (table (gensym "TABLE-"))
        (maps nil)
        (num-qubits 0))
    (labels ((pauli-string-p (string)
               ;; Is STRING a valid Pauli string?
               (let ((n (length string)))
                 (and (or (= 1 (count #\X string))
                          (= 1 (count #\Z string)))
                      (= (1- n) (count #\I string)))))
             (dimension (string)
               ;; What is the dimension of the Pauli STRING?
               (if (char= #\- (char string 0))
                   (1- (length string))
                   (length string)))
             (embed-pauli-string (string)
               ;; Embed the Pauli string into a NUM-QUBITS space.
               (let* ((p (pauli-from-string string))
                      (p-num-qubits (num-qubits p)))
                 (with-output-to-string (s)
                   (print-pauli
                    (embed p
                           num-qubits
                           (loop :for i :below p-num-qubits :collect i))
                    s)))))
      ;; Parse out all of the map data, including the number of qubits
      ;; of the entire Clifford element.
      (loop :for (from arrow to) :on body :by #'cdddr
            :for from-name := (string from)
            :for to-name := (string to)
            :do (progn
                  (assert (string= "->" arrow))
                  (assert (pauli-string-p from-name)
                          ()
                          "The symbol ~S is not a Pauli basis element"
                          from)
                  ;; Get the max dimension
                  (alexandria:maxf num-qubits
                                   (dimension from-name)
                                   (dimension to-name))
                  ;; Record the map.
                  (push (cons from-name to-name)
                        maps)))

      ;; Embed into proper dimension.
      (mapc (lambda (m)
              (rplaca m (embed-pauli-string (car m)))
              (rplacd m (embed-pauli-string (cdr m))))
            maps)

      ;; Set the mappings.
      `(let* ((,clifford (clifford-identity ,num-qubits))
              (,table (basis-map ,clifford)))
         ,@(loop :for (from . to) :in maps
                 :collect
                 `(setf (aref ,table
                              (pauli-to-index
                               (pauli-from-string ,from)))
                        (pauli-from-string ,to)))
         ;; Return the table.
         ,clifford))))
```

Figure 6.1: The `clifford-element` macro.

Simulating Quantum Processor Errors by Extending the Quantum Abstract Machine

Nikolas A. Tezak
Rigetti Quantum Computing
Berkeley, CA, USA
nikolas@rigetti.com

Robert S. Smith
Rigetti Quantum Computing
Berkeley, CA, USA
robert@rigetti.com

ABSTRACT

We describe how the Common Lisp Object System can be used to define a protocol for a machine called the *quantum abstract machine*, a mechanism used to simulate the operational semantics of a quantum instruction language called Quil[7]. Furthermore, we describe implementations of ideal and noisy simulators for hybrid classical/quantum program execution.

CCS CONCEPTS

• **Theory of computation** → *Quantum information theory*; *Abstract machines*; • **Applied computing** → *Physics*;

ACM Reference format:
Nikolas A. Tezak and Robert S. Smith. 2018. Simulating Quantum Processor Errors by Extending the Quantum Abstract Machine. In *Proceedings of the 11th European Lisp Symposium, Marbella, Spain, April 16–17 2018 (ELS'18),* 5 pages.
https://doi.org/

1 INTRODUCTION

The semantics of programming languages, especially instruction languages, are often specified in terms of state transitions of a mathematical device called an *abstract machine*. Often, these machines admit a straightforward representation on a computer. In the context of the language being described, it is often desirable to amend or augment the semantics of the machine to study different behavior for a variety of reasons. In this paper, we principally focus on how we realize the quantum abstract machine (QAM) and amend the semantics it encodes to support quantum error models. We demonstrate how using the Common Lisp Object System (CLOS) has eased the extensibility of the quantum simulation software at Rigetti Quantum Computing, and in particular, we illustrate how it has made the implementation of realistic simulations of *noisy* quantum hardware very straightforward.

As Moore's law is increasingly challenged by the physics of semiconductor technology, alternative paradigms for computation are required. Advances in engineering and controlling devices governed by the laws of quantum mechanics have ushered in a new era of noisy intermediate-scale quantum (NISQ) technology [3] with the goal of harnessing quantum physics for more powerful information processing. A long-term goal of this field is the creation of a fault-tolerant quantum computer [5]. As a computational framework, quantum computing differs substantially from classical computing and carries the ultimate promise of super-polynomial speed-ups on hard problems such as factoring [4]. Recently Smith et al. have defined the computational framework based on Quil [7], a low-level quantum instruction language whose operational semantics are defined by way of a mathematical device called a QAM. Using this framework, Rigetti Quantum Computing has demonstrated the basic feasibility of unsupervised machine learning on their current 19-qubit quantum processing unit (QPU) [2].

Prototyping and benchmarking of such near-term applications are enabled by our classically simulated QAM, the quantum virtual machine (QVM), which is publicly accessible through the open-source pyQuil[1] library and API. As discussed at length in [3], current quantum devices—such as our 19-qubit QPU—are not yet error-corrected and therefore their operation deviates from the ideal quantum programming model in a way that can be characterized and, luckily, simulated as well.

In the following we briefly introduce the necessary mathematical abstractions underlying quantum computation as well as parts of the basic protocol encapsulated by the QAM. We continue this with a description of how we can model the effect of noisy quantum gates and noisy readout in a way that is compatible with the QVM and how the introduction of semantics-amending `PRAGMA` directives can provide a simple configuration mechanism that transparently modifies the interpretation of a given Quil program, provided the underlying QAM understands it. Finally, we present a few examples of how the features of CLOS make it remarkably easy to add support for noise models to the QVM.

2 QUANTUM COMPUTING ILLUSTRATED VIA THE QAM AND QVM

Our computational framework is explicitly hybrid in that it encapsulates both quantum registers (qubits) and classical registers as well as operations on them. The two are connected in three ways:

(1) Via quantum measurement, which in general stochastically and irreversibly projects the quantum state into a particular measurement outcome.
(2) Through classical control flow in which quantum gates are applied conditionally on the value of classical registers.
(3) Through parametrization of quantum gates with numerical quantities encoded in classical memory.

ELS'18, April 16–17 2018, Marbella, Spain
© 2017-2018 Association for Computing Machinery.
ACM ISBN 978-2-9557474-2-1...$15.00
https://doi.org/

[1] `https://github.com/rigetticomputing/pyquil`

We refer the interested reader to [7] to learn more about the underlying model of hybrid classical/quantum computation.

2.1 Quantum State Transitions: Measurement and Gates

A pure quantum state of n qubits can always be represented as a complex vector $\psi = (\psi_{0\ldots 00}, \psi_{0\ldots 01}, \ldots, \psi_{1\ldots 11})^{\intercal} \in \mathbb{C}^d$ of unit length in the canonical bitstring basis (also called the *Z-basis* or *computational basis*), with dimension $d = 2^n$. Measurement generally causes a non-deterministic change to a quantum state. The probability of each measurement outcome depends on the pre-measurement state. If all qubits in a register are measured, the outcome probability of a given bitstring $s = s_{n-1}s_{n-2}\cdots s_0$ is given by $|\psi_s|^2$. If only a particular qubit q is measured, the probability for the bit-value outcome $s_q = 1$ is given by

$$\mathbb{P}(s_q = 1; \psi) = \sum_{s \text{ with } s_q=1} |\psi_s|^2,$$

and the post-measurement state coefficients are given by

$$\psi'_s = \begin{cases} \frac{1}{\sqrt{\mathbb{P}(s_q=1;\psi)}}\psi_s & \text{if } s_q = 1 \\ 0 & \text{otherwise.} \end{cases}$$

In Quil, a measurement of qubit 2 into the 0th classical register is expressed as:

```
MEASURE 2 [0]
```

A quantum state may also be modified by a quantum gate which, when represented in the same basis, is given by a unitary matrix $U \in \mathbb{C}^{d\times d}$ that maps an initial state ψ to a final state ψ' via

$$\psi' = U\psi \Leftrightarrow \psi'_j = \sum_{k=0}^{d-1} U_{jk}\psi_k,$$

where we have switched from bit string representation of the state label to an enumeration $s \longrightarrow j \in \{0, 1, \ldots, d-1\}$ such that s gives the binary representation of j. Our convention is to take the lowest index qubit $q = 0$ to also be the least significant bit of j as motivated in [6]. We will continue using both the bitstring (base-2) and the full enumeration as convenient.

Oftentimes, quantum gates are decomposed into a device specific gate set and in such a way that they act locally on a subset of all qubits. A gate acting only on two qubits q, r would result in

$$\psi'_{s_{n-1}\ldots s'_q\ldots s'_r\ldots s_0} = \sum_{\substack{s_q\in\{0,1\} \\ s_r\in\{0,1\}}} U_{s'_q s'_r\, s_q s_r}\psi_{s_{n-1}\ldots s_q\ldots s_r\ldots s_0}.$$

In Quil, a gate U applied to qubits 3 and 2 is written as:

```
U 3 2
```

Note that in general this is different from

```
U 2 3
```

as the enumeration of U's matrix elements depends on the order of qubits.

2.2 The QAM and QVM

The formal definition of a QAM is operational; it implies a particular way of working, and hence, it implies what a Quil interpreter looks like.

Rigetti Quantum Computing's classical implementation of the QAM is called the "Rigetti Quantum Virtual Machine". In this report, we simply refer to it as *the* QVM.

The QVM as well as the compiler software are implemented in Common Lisp[1]. While much of the source code is portable to any ANSI-conforming implementation, we use Steel Bank Common Lisp (SBCL). It includes a machine-code compiler whose numerical performance is capable of being competitive with state-of-the-art C compilers.

The simulator makes heavy use of object-oriented idioms of Common Lisp, including the notion of a protocol. The QAM protocol is defined by a single abstract class (the superclass of all QAMs) as well as two generic functions for executing code on a QAM.

```
;;; Superclass of all QAM classes.
(defclass quantum-abstract-machine ()
  ()
  (:metaclass abstract-class))

;;; Run the QAM until completion.
(defgeneric run (qam))

;;; Execute an instruction INSTR on a QAM.
(defgeneric transition (qam instr))
```

While protocol-based programming in Common Lisp usually does not require the presence of an explicit superclass, its presence as the root class of all QAMs makes for better documentation and introspection.

As one might imagine, execution dispatches not only on the QAM class, but also the instruction class. The simplest example is Quil's `NOP` instruction. Here, `pure-state-qvm` is a class of machine that represents an ideal QAM that is always propagating a pure quantum state with ideal operations.

```
(defmethod transition ((qvm pure-state-qvm) (instr no-operation))
  (declare (ignore instr))
  (values qvm (1+ (pc qvm))))
```

The execution of a quantum program—both in terms of the classical and quantum values stored, as well as the control flow—is non-deterministic, because quantum measurements are probabilistic. This usually implies that one must execute a quantum program many times and infer the solution to the underlying computational problem from the joint outcome statistics.

3 NOISY OPERATIONS OF A QUANTUM COMPUTER

Above, we discussed the two types of state transitions most relevant to this exposition: quantum gate applications and qubit measurement. In this section we will briefly explain how those operations can be corrupted on a real device and how to model such errors on our QVM.

3.1 Noisy Quantum Gate Operations

In general, the details of gate imperfections depend on the qubits they act on. This is a direct consequence of the fact that qubits are separate physical entities—microwave circuits in the case of superconducting qubits—which have different physical properties such as their transition frequency, and which inadvertently couple to their environment. To accommodate this, the QVM supports either augmenting or outright replacing ideal gate operations by non-unitary maps, specified as Kraus maps. Kraus maps provide a convenient way to write down physical quantum maps or "channels". A full introduction to quantum channels is beyond the scope of this work, but at the level of pure state simulations, a general channel can be realized as a non-deterministic transition that replaces the actually intended unitary gate. In particular, the noisy realization of a quantum gate transition is defined via a set of Kraus operators

$$\mathcal{K} = \left\{ K_m \in \mathbb{C}^{d\times d}, m = 0, 1, \ldots, r-1 \mid \sum_{m=0}^{r-1} K_m^\dagger K_m = I \right\}.$$

Here $X^\dagger$ denotes the Hermitian conjugate matrix to X, obtained by transposing it and taking the element-wise complex conjugate. In each intended gate transition one of the Kraus operators is randomly selected to act on the state such that

$$\psi \mapsto \psi' = \frac{1}{\sqrt{p_m}} K_m \psi \text{ with probability } p_m = \|K_m \psi\|_2^2 \qquad (1)$$

where $\|v\|_2^2 = \sum_{j=0}^{d-1} |v_j|^2$ is the squared 2-norm of a vector $v \in \mathbb{C}^d$. As in the case of unitary gate matrices, we usually also consider Kraus operators that only affect a small subset of the available qubits, in which case the Kraus matrices are elements of $\mathbb{C}^{2^{n'}\times 2^{n'}}$ where $n' \leq n$ is the number of affected qubits. Quantum computation is already inherently probabilistic, but gate noise can greatly reduce the success probability of a quantum algorithm. It is an exciting and active area of research to develop applications that exhibit some robustness to such errors.

3.2 Readout Noise

For superconducting qubits as developed by Rigetti Computing, the dominant error mechanism associated with qubit readout is thermal noise that perturbs the readout signal and causes it to be mislabeled. Consequently, an appropriate model for this error is to assume initially a correct quantum measurement yielding a result s_q which is then subsequently corrupted by a noisy channel $s_q \mapsto s_q'$ with conditional output probability

$$P_{s_q'|s_q} = \begin{pmatrix} p(0 \mid 0) & p(0 \mid 1) \\ p(1 \mid 0) & p(1 \mid 1) \end{pmatrix}.$$

This matrix is usually called *assignment probability* matrix. The average of its diagonal elements is called *assignment fidelity* $F = \frac{1}{2}\left[p(0 \mid 0) + p(1 \mid 1)\right]$ and it provides a simple figure of merit for the quality of a qubit's readout.

3.3 Pauli noise

There exists a special class of noise that is quite popular in the Quantum Computing community due to its mathematical properties: The Pauli noise model corresponds to appending to each unitary gate U a non-deterministic Pauli channel realized by the Kraus operators

$$\mathcal{K}_{\text{pauli}} = \{\sqrt{1 - p_x - p_y - p_z} I, \sqrt{p_x} X, \sqrt{p_y} Y, \sqrt{p_z} Z\}$$

where $0 \leq p_x + p_y + p_z \leq 1$ and the Pauli operators are given by

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

Furthermore, each readout is preceded by a Pauli channel with $p_y = p_z = 0$, i.e., a pure bit-flip error. Note that this is a different readout noise model than considered above as this actively changes the quantum state, rather than just the classical measurement bit.

4 NOISE INJECTION THROUGH THE COMMON LISP OBJECT SYSTEM

In this section we outline how class inheritance and generic dispatch allow selectively replacing ideal operations with faulty ones.

4.1 Pauli Noise simulation with the `depolarizing-qvm`

The Rigetti QVM preprocesses any incoming Quil program and extracts noise model definitions from `PRAGMA` directives. In the next section, we will describe this functionality more precisely. If a program contains noise models, then a special QVM instance is constructed, of class `noisy-qvm`. The `noisy-qvm` is a subclass of the briefly aforementioned `pure-state-qvm`. The methods for applying gate transitions or measurements are augmented or replaced to implement the noisy versions by using CLOS generic function dispatch.

For the purpose of simulating just Pauli noise, we subclass the pure-state QVM and store the Pauli error probabilities [2] for gates and measurements in additional slots. The convenient availability of Common Lisp's `:before` and `:after` method specializations allows us to simply extend the behavior of the non-noisy pure-state QVM and append or prepend the appropriate errors:

```
;;; Noise gets added to only the qubits being changed.
(defmethod transition :after ((qvm depolarizing-qvm)
                                (instr gate-application))
  (dolist (q (application-arguments instr))
    (add-depolarizing-noise qvm q
                             (probability-gate-x qvm)
                             (probability-gate-y qvm)
                             (probability-gate-z qvm))))
;;; Noise gets added to only the qubit being measured, before
;;; measurement occurs.
(defmethod transition :before ((qvm depolarizing-qvm)
                                 (instr measurement))
  (let ((q (measurement-qubit instr)))
    (add-depolarizing-noise qvm q
                             (probability-measure-x qvm)
                             0.0d0         ; Y and Z noise are not
                             0.0d0)))      ; observable.
```

4.2 General Kraus models and Noisy Readout with the `noisy-qvm`

We can also realize the more general gate noise model of arbitrary Kraus maps and the measured-bit-flip error model. The following subclass adds slots to store the noisy gate and noisy readout definitions. The first slot corresponds to the collection of Kraus maps for each noisy operation for each qubit specified, and the second slot corresponds to the measurement noise model.

[2] Here, the error probabilities are the same for all qubits.

```
(defclass noisy-qvm (pure-state-qvm)
  ((noisy-gate-definitions
    :initarg :noisy-gate-definitions
    :accessor noisy-gate-definitions
    :initform (make-hash-table :test 'equalp))
   (readout-povms
    :initarg :readout-povms
    :accessor readout-povms
    :initform (make-hash-table)))
  (:documentation "A quantum virtual machine
with noisy gates and readout."))
```

4.3 Extending Quil with PRAGMA directives

The QAM's semantics are relatively fixed, and do not contain any notion of noise. However, Quil has provisions for extensions to its semantics with a PRAGMA directive.

The syntax of PRAGMA is as follows:

```
PRAGMA <identifier> (<identifier> | <integer>)* <freeform string>?
```

While PRAGMA directives may introduce any arbitrary functionality, a useful rule-of-thumb to constrain their effects is to not introduce anything which fundamentally alters the ideal behavior of the program; one should be able to strip out all PRAGMA directives and retain a valid program.

Rigetti Quantum Computing's software stack is able to recognize a certain set of PRAGMA directives, which are used to modify simulation behavior or to suggest optimizations[3].

In general, PRAGMA directives are ignored, as shown with this transition rule.

```
(defmethod transition ((qvm pure-state-qvm) (instr pragma))
  (warn "Ignoring PRAGMA: ~A" instr)
  (values qvm (1+ (pc qvm))))
```

When we parse program text, PRAGMA directives are parsed in the most general way possible, into a pragma class instance representing the abstract syntax. So how shall these general entities, so far ignored by the abstract machine, affect how the machine operates?

Consider a PRAGMA directive for tracing execution:

```
H 0
PRAGMA PRINT stdout "Superposition!"
CNOT 0 1
```

In this case, supposing that we have not done any form of program reordering[4], our interpreter could use this PRAGMA to print some useful information, such as the fact that we have created an equiprobable superposition. To make use of CLOS's multiple dispatch, we will want a dedicated class for this PRAGMA directive.

```
(defclass pragma-print (pragma)
  ((out-stream :initarg :out-stream
               :reader pragma-print-stream))
  (:default-initargs :words '("PRINT"))
  (:documentation "PRAGMA to trace execution."))
```

How might we actually get this instance? In usual operation, PRAGMA directives are parsed as instances of the pragma class. We could change this behavior and add more specialized knowledge of PRAGMA directives to the parser. However, this would make the parser more complicated, and additionally complicate what is insofar a clean instruction code definition.

Instead, a compiler pass called *pragma refinement* is performed. Using pragma refinement, we scan through the program, find this recognized PRAGMA directive by way of its leading identifier, and create an instance of a class defined. Creation is *not* done through make-instance, but rather through change-class, which gives us hooks for interpreting the PRAGMA directive further. Specifically, we specialize update-instance-for-different-class for the directive.

```
(defmethod update-instance-for-different-class
    :before ((from pragma)
             (to special-pragma)
             &key)
  ...)
```

The above described usage mode of PRAGMA directives has been through direct interpretation. However, the far more important way to use these directives is to command the operation of the compiler or to *prepare* a machine for execution. In these use cases, it is still viable to use PRAGMA directives without further parsing (i.e., one may look at the contents of the root pragma class directly), but we have found that practice difficult to debug and maintain.

4.4 Noise Annotation via Quil PRAGMA directives

4.4.1 Gate Noise Example: Amplitude Damping. In this section we present a very simple noise model especially relevant to superconducting qubits: Often the two energy levels used to encode a qubit differ in energy. If such a qubit is in the higher energy state and not perfectly isolated from the environment, then energy can leak out leading to a relaxation to the low energy state. We can parametrize such a single qubit noise process by annotating a program with PRAGMA's.

```
PRAGMA ADD-KRAUS I 0 "(1.0 0.0 0.0 0.979795897)"
PRAGMA ADD-KRAUS I 0 "(0.0 0.2 0.0 0.0)"
I 0
I 0
# ...
I 0
MEASURE 0 [0]
```

The PRAGMA directives in effect replace the ideal identity gate I on qubit 0 with an amplitude damping channel with two Kraus operators that are given in flattened row-major form in the string PRAGMA arguments. The first of these Kraus operators gives the outcome in which the qubit state is mostly preserved except for a re-weighting of occupation probabilities. The second of these Kraus operators is proportional to a single qubit lowering operator that induces relaxation transitions from $1 \longrightarrow 0$. If the qubit is excited, the damping operator has a single step probability of $p_{\text{damp}} = 4\%$. Consequently, when starting in 1, the probability of not having decayed after M steps is given by $(1 - p_{\text{damp}})^M$. It is straightforward to generalize the above to even more accurate models for gates under amplitude damping by exponentiating the summed generators of the unitary gate evolution and the noise. Alternatively, one may empirically estimate the actual physical channel that a gate corresponds to through tomography and subsequently use this for realistic simulations. In Figure 1 we present the result of simulating a sequence of Quil programs with an amplitude damping noise model.

4.4.2 Readout Noise Example: RX(θ). Here we show how single qubit readout noise generally reduces the contrast in qubit measurements. A good example for this is given by a sequence of so

[3] Some of these can be found in the pyQuil documentation.

[4] Quantum programs are very much unlike their classical analogs because quantum programs operate—at least on real hardware—on resources in parallel. It is generally unwise to think of a quantum instruction code as a linear instruction code.

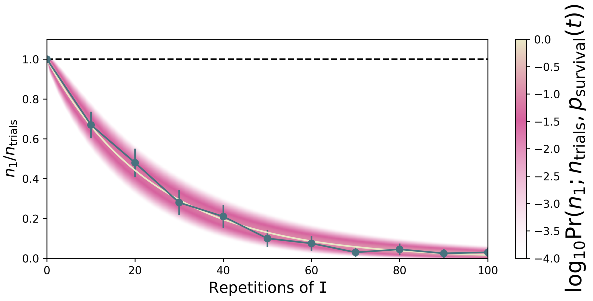


Figure 1: Single Qubit Amplitude Damping. The dashed black line gives the expected 1-state population for an *ideal* qubit, the color map gives the theoretical $\log$**-probability of the 1-state population for our noise model and the dark teal points give the stochastically simulated estimates from running $N = 200$ executions of a program with repeated noisy** I **gates on the QVM.**

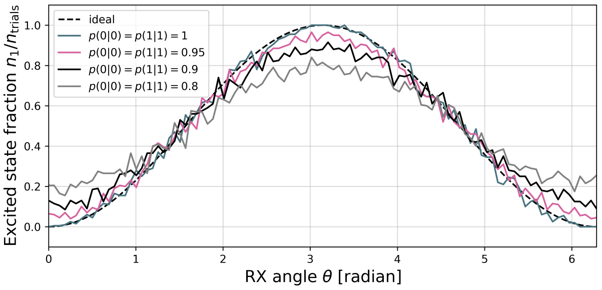


Figure 2: A Rabi-program with readout corrupted with different noise levels (see legend). Each data point represents the average of $N = 200$ samples generated by the QVM.

called *Rabi-programs* that prepare qubit states rotated at increasing angles in the $Y - Z$ plane using the rotation operator RX(θ). An example program of such a sequence is given by

```
RX(1.1*pi) 0
PRAGMA READOUT-POVM 0 "(0.8 0.2 0.2 0.8)"
MEASURE 0 [0]
```

Here the `PRAGMA READOUT-POVM 0 "(...)"` statement configures the assignment probability matrix for qubit 0, here given in flattened row-major form.

In Figure 2 we present the result of simulating a sequence of Quil programs with a readout noise model.

4.5 Noise Simulation through Generic Dispatch

We use generic dispatch on both the QVM type and the Quil instruction type to overload the ideal implementation of a gate or measurement transition with a noisy one. Listing 1 contains the actual Lisp code for the case of noisy gates.

```
(defmethod transition ((qvm noisy-qvm) (instr gate-application))
  (let ((kraus-ops (lookup-kraus-ops instr (noisy-gate-definitions qvm))))
    (cond
      (kraus-ops
       ;; Found noisy realization of gate.
       (let ((wf (make-wavefunction (number-of-qubits qvm)))
             (qubits (application-arguments instr))
             (r (random 1.0d0))
             (s 0.0d0))
         (loop :for K :in kraus-ops
               :do (replace wf (wavefunction qvm))
                   (apply-operator K wf qubits)
                   (incf s (sum #'probability wf))
               :until (>= s r))
         (replace (wavefunction qvm) (normalize-wavefunction amps))
         (values
          qvm
          (1+ (pc qvm)))))
      (t
       ;; no noise model, proceed simulation as usual
       (call-next-method qvm instr)))))
```

Listing 1: Overloaded gate application transition method for the noisy QVM.

5 CONCLUSIONS

In this paper, we have shown how to make non-trivial modifications to the operational semantics of Quil—using the QAM as an implementation formalism—by way of facilities provided by Common Lisp. We have shown how CLOS has allowed us to rapidly and economically equip an ideal QVM with functionality to simulate real-world noise present in near term devices. The flexibility of the semantics afforded by the ease of implementation is but one important aspect of using Common Lisp; maintaining the high speed required for computing with exponentially large state vectors is another. This is only a small sample of the ways in which our simulator—a key component to quantum software engineering—benefits from the utilization of Common Lisp.

REFERENCES

[1] American National Standards Institute and Information Technology Industry Council. American National Standard for Information Technology: programming language—Common LISP, 1994. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA. 1996.
Approved December 8, 1994.

[2] J. S. Otterbach, R. Manenti, N. Alidoust, A. Bestwick, M. Block, B. Bloom, S. Caldwell, N. Didier, E. Schuyler Fried, S. Hong, P. Karalekas, C. B. Osborn, A. Papageorge, E. C. Peterson, G. Prawiroatmodjo, N. Rubin, Colm A. Ryan, D. Scarabelli, M. Scheer, E. A. Sete, P. Sivarajah, Robert S. Smith, A. Staley, N. Tezak, W. J. Zeng, A. Hudson, Blake R. Johnson, M. Reagor, M. P. da Silva, and C. Rigetti. Unsupervised Machine Learning on a Hybrid Quantum Computer. *ArXiv peprints arxiv:1712.05771*, 12 2017. URL http://arxiv.org/abs/1712.05771.

[3] John Preskill. Quantum Computing in the NISQ era and beyond. *ArXiv e-prints*, pages 1–22, 1 2018. URL http://arxiv.org/abs/1801.00862.

[4] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994. ISSN 0272-5428. doi: 10.1109/SFCS.1994.365700.

[5] P.W. Shor. Fault-tolerant quantum computation. In *Proceedings of 37th Conference on Foundations of Computer Science*, pages 56–65. IEEE Comput. Soc. Press, 1996. ISBN 0-8186-7594-2. doi: 10.1109/SFCS.1996.548464. URL http://arxiv.org/abs/quant-ph/9605011http://ieeexplore.ieee.org/document/548464/.

[6] Robert S. Smith. Someone shouts, "|01000>!" Who is Excited? *ArXiv e-prints*, pages 1–4, 11 2017. URL http://arxiv.org/abs/1711.02086.

[7] Robert S. Smith, Michael J. Curtis, and William J. Zeng. A practical quantum instruction set architecture. *arXiv:1608.03355v2*, 2016. URL http://arxiv.org/abs/1608.03355.

Clasp Common Lisp Implementation and Optimization

Alex Wood
Temple University
1901 N. 13th Street
Philadelphia, Pennsylvania 19122
alex.m.wood@outlook.com

Christian Schafmeister
Temple University
1901 N. 13th Street
Philadelphia, Pennsylvania 19122
meister@temple.edu

ABSTRACT

We describe implementation strategies and updates made in the last two years to CLASP,[1, 2] a new Common Lisp implementation that interoperates with C++, uses the CLEAVIR compiler, and uses the LLVM backend[4]. Improvements in CLASP have been made in many areas. The most important changes are: (1) Tagged pointers and immediate values have been incorporated. (2) A fast generic function dispatch approach has been implemented that allows CLASP to carry out generic function dispatch as fast as SBCL, a highly optimized free implementation of Common Lisp. The generic function dispatch memoization approach was developed by Robert Strandh[8] and demonstrates a 20x improvement in performance of generic function dispatch. (3) The new LLVM feature “Thin Link Time Optimization” has been added, which speeds up generated code by removing call overhead throughout the system. (4) Type inference has been added to the CLEAVIR compiler, which is part of CLASP. Type inference removes redundant run-time type checks and dead code paths. Type inference currently provides about a 30% speedup in microbenchmarks.[9] (5) Constants and literals have been moved close to the code and “instruction pointer addressing” has been incorporated to speed access to literals and constants. (6) Pre-emptive multithreading has been added to CLASP, based on pthreads, supporting the Bordeaux threads library. (7) The overall build time for CLASP has been reduced from five to eight hours over two years ago to approximately one hour at present.

CCS CONCEPTS

•**Software and its engineering → Compilers;** *Dynamic compilers; Runtime environments;*

ACM Reference format:
Alex Wood and Christian Schafmeister. 2016. Clasp Common Lisp Implementation and Optimization. In *Proceedings of the 11th European Lisp Symposium, Marbella, Spain, April 16–17 2018 (ELS'18),* 6 pages.
DOI:

ELS'18, Marbella, Spain
© 2016 ACM. 978-2-9557474-2-1…$15.00
DOI:

1 INTRODUCTION

We describe implementation strategies and updates made in the last two years to CLASP[2] - a new Common Lisp implementation that interoperates with C++ and uses the LLVM backend[4] and CLEAVIR[1] to generate efficient native code for an increasing variety of processors. CLASP currently targets macOS and Linux. CLASP is being developed as a scientific programming language and specifically to support CANDO.[2] CANDO is being designed as a Common Lisp implementation to support the development of new software tools for molecular modeling and molecular design and it extends CLASP by adding hundreds of thousands of lines of C++ and Common Lisp code that implement solutions to computational chemistry problems. CANDO is designed to enable interactive programming to develop tools for designing molecules while at the same time generating fast native code for demanding chemistry and molecular modeling applications.

2 TAGGED POINTERS AND MEMORY MANAGEMENT

CLASP makes use of pointer tagging to indicate the type of a pointer, and also encodes some values directly as immediate values. CLASP is currently only targeting 64-bit architectures, and so the following currently applies to the 64-bit implementation. The type of a pointer is indicated using a three bit tag in the least significant bits of a 64-bit word. The meaning of the tags and their current values in binary are fixnum (#B000 and #B100), general pointer (#B001), character (#B010), cons (#B011), vaslist (#B101), and single-float (#B110). #B111 is currently unused.

“Vaslist” is a special CLASP type used to operate on variable-length lists of arguments without allocating an actual Lisp list structure. These can be constructed in Lisp using the &CORE:VA-REST lambda list keyword, similar to e.g. &MORE in SBCL. The vaslist itself is a pointer to a structure incorporating a C/C++ va_list structure, together with a count of arguments remaining. One way we can work with vaslists in CLASP is using the BIND-VA-LIST special operator, which is analogous to DESTRUCTURING-BIND.

The immediate types are fixnum, character, and single-float; they are stored in the high bits of the word. Fixnum values are 62 bits wide and they are indicated by the value #B00 in the two least significant bits. This allows fixnum addition and comparison to be carried out without bit shifting. Character values are 32 bits wide, enough to encode Unicode.[10] Single-float values are 32 bit wide, IEEE754 format, corresponding to the C++ 'float' type.

Cons pointers are indicated by the tag value #B011. The cons itself is two sequential 8-byte words. The consp test is extremely efficient because the tag is sufficient to indicate a cons. Traversal

[1] https://github.com/robert-strandh/SICL/tree/master/Code/Cleavir

of lists thus involves only one tag test per element, excluding the ultimate element.

All other general object pointers share the pointer tag #B001. General objects consist of one header word followed by data. The header word is used by the garbage collector to identify the layout of the data, and to determine C++ inheritance relationships to avoid the use of the slow C++ template function "dynamic_cast".

There are four kinds of general objects:

(1) Instances of C++ classes. These must inherit from Clasp's General_O, and have their layouts known to the garbage collector.
(2) Instances of Common Lisp classes, i.e. conditions, structure-objects, standard-objects, and funcallable-standard-objects. These are implemented as the C++ class Instance_O (Table 1). These consist of a "Signature" (a description of the object's slots used for obsolete instance updating), a pointer to their class, and a *rack* of slots. Funcallable instances include more data (implemented as FuncallableInstance_O, Table 2), but keep the class and rack at the same positions as non-funcallable instances, to facilitate uniform access.
(3) Instances of the C++ Wrapper⟨T⟩ template class, which wraps C++ pointers and keeps track of their types. These can be used as pointers to C++ objects outside of managed memory, so that such objects can be operated on from Lisp.
(4) Instances of the C++ Derivable⟨T⟩ template class, which is used to create Common Lisp classes that inherit from C++ classes.

Lisp's other built in classes, such as symbols, complex numbers, and arrays, are implemented as C++ classes, i.e. the first kind.

Because the header does not include information about Lisp classes, among other things, it is not totally sufficient for Lisp type checking. It can however be used for rapid discrimination of instances of built in classes.

General objects and conses are stored on the heap and managed by the memory manager. CLASP can be built to use one of two automatic memory management systems: the Boehm garbage collector[3], or the Memory Pool System (MPS)[7].

Boehm is a conservative collector originally designed for C programs, meaning that it treats objects as opaque blobs of memory, identifies memory words as pointers without using type information, and does not move objects. This allows it to allocate and collect quickly, but as it does not compact memory it can leave the heap fragmented, impacting long-running CLASP programs and ones that allocate large objects.

The Boehm build of CLASP is required to run a special C++ static analyzer, written in CLASP, that determines memory layouts for all C++ classes in the CLASP C++ source.

The Memory Pool System is a precise collector more suitable for Lisp. It requires information per-type about where pointers are located; this is derived from the static analyzer, indexed using the general object headers. MPS can move and compact memory, using the least significant two bits of the header word to indicate forwarding pointers and padding. The MPS will run in a fixed memory footprint, making it suitable as the default memory manager for CLASP for scientific programming.

Table 1: Memory layout of Instance_O.

Offset	C++ type	Field
0	T_sp	Signature
8	Class_sp	Class
16	SimpleVector_sp	Rack

Table 2: Memory layout of FuncallableInstance_O.

Offset	C++ type	Field
0	std::atomic⟨claspFunction⟩	entry
8	Class_sp	Class
16	SimpleVector_sp	Rack
24	std::atomic⟨T_sp⟩	CallHistory
32	std::atomic⟨T_sp⟩	SpecializerProfile
40	std::atomic⟨T_sp⟩	DispatchFunction

Table 3: Memory layout of simple vectors, the SimpleVector_O class.

Offset	C++ type	Field
0	size_t	Length
8	TYPE	Data[length]

3 ARRAYS

Common Lisp vectors and arrays are implemented using the structures shown (Table 3 and Table 4). The type of a simple vector can be specialized within the CLASP C++ code to be any C++ type or class. The types that are currently supported are T_sp (the general Common Lisp T type), fixnum, double, float, and signed and unsigned integer types of length 8, 16, 32 and 64 bits. Simple bit vectors are implemented by manipulating bits in unsigned 32 bit words. The offset of the Length field in simple vectors and the FillPointerOrLengthOrDummy field of complex arrays is the same so that the length or fill-pointer can be accessed quickly for both simple and complex vectors. For multi-dimensional arrays FillPointerOrLengthOrDummy is ignored. The Flags field stores whether the array has ARRAY-FILL-POINTER-P and whether the array is displaced.

Array operations can be complex, and they are inlined for both simple vector and complex array operations. CLASP does this by undisplacing the array to the simple vector that ultimately stores the array contents, and then indexing into that simple vector. Inlining is performed for SVREF, ROW-MAJOR-AREF, SCHAR, CHAR, ELT and AREF.

4 FAST GENERIC FUNCTION DISPATCH

CLASP implements the fast generic function dispatch approach developed by Robert Strandh.[8] Fast generic function dispatch is important in CLASP because it uses the CLEAVIR compiler (also written by Robert Strandh), which makes heavy use of generic functions in its operation.

To enable the dispatch technique, all objects have a 64-bit value called the *stamp*, unique to the class it was defined with. For general

Table 4: Memory layout of complex arrays, the MDArray_O class.

Offset	C++ type	Field
0	size_t	FillPointerOrLengthOrDummy
8	size_t	ArrayTotalSize
16	Array_sp	Data
24	size_t	DisplacedIndexOffset
32	Flags	Flags
40	size_t	Rank
48	size_t	Dimensions[Rank]

objects, the stamp is within the header word for instances of C++ classes, but it is stored within the object otherwise.

The fast generic function dispatch approach works by compiling discriminating functions that dispatch to precompiled effective methods based on the stamps of their arguments (Figure 1). This reduces discrimination to a series of integer comparisons, making it very efficient. The "slow path" of dealing with actual classes and calling the MOP-specified generic functions only comes into play if the integer comparisons fail to branch to a known effective method. The "slow path" compiles a new "fast path" through the normal MOP-specified protocols, and so the method works independently of MOP customizations, novel method combinations, etc., except for the EQL specializer optimization described below.

If a Lisp class is redefined, its stamp is changed and existing discriminators have their dispatching for the old stamp removed. Calls to discriminators involving obsolescent instances can therefore update instances only in the slow path. This in particular, is a major improvement over ECL, which checks object obsolescence for all calls to accessors.

CLASP incorporates a small extension to the dispatch technique. Calls involving EQL specializers cannot in general be memoized, because they imply the necessity of tests more specific than stamp tests. However, CLASP does memoize some calls involving EQL specializers, for functions without customizable generic function invocation behavior. The core requirement to memoize such calls to such functions is that any argument in an EQL-specialized position does match an EQL-specialized method, and not only class-specialized methods. For example, if a generic function of one parameter has one method specialized on the symbol class, and one EQL-specialized on a particular symbol, calls involving the latter will be memoized while the former will not be. This speeds the most common uses of EQL specializers while preserving correctness.

In a micro-benchmark where a regular function of two arguments, a generic function that accepts two arguments, and a CLOS accessor were called 200,000,000 times - the timing values in Table 5 were obtained.

The performance is remarkable given that CLASP started out using the ECL CLOS source code and reimplemented the ECL generic function dispatch cache. With the Strandh fast generic function dispatch, CLASP is comparable to the performance of SBCL, a highly optimized implementation of Common Lisp.

There is a "warm-up" time associated with this fast generic function dispatch method as it is currently implemented within CLASP. Discriminating functions are compiled lazily when a generic function is called. Invoking components that use many generic functions for the first time forces a cascade of lazy compilation. The compiler itself, for example, must compile about 1,200 functions on its first run after startup. This takes tens of seconds of real time. This only happens once at startup, and thereafter discriminating functions are invalidated and recompiled only when methods are added or removed, or when classes are redefined.

Table 5: A micro-benchmark of regular calls, generic function calls and CLOS accessor calls (200M calls).

Implementation	call (s)	gf call (s)	accessor (s)
Clasp(553e35a)	0.40	1.44	1.23
SBCL(1.4.4)	0.45	1.25	0.72
ECL(16.1.3)	0.81	26.96	7.79
ccl(1.11-store-r16714)	0.38	3.82	3.73

5 COMMON FOREIGN FUNCTION INTERFACE

CLASP has incorporated an implementation of the Common Foreign Function Interface (CFFI) written by Frank Goenninger. This gives CLASP access to C libraries that have been exposed to Common Lisp using CFFI. This is in addition to CLASP's built in C++ interoperation facility *clbind*.[1]

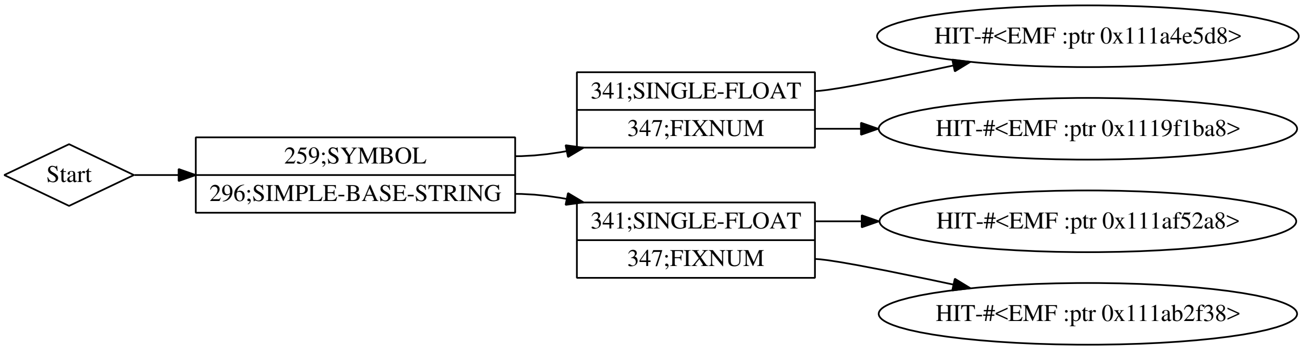
6 LINK TIME OPTIMIZATION

A new LLVM feature called "Link Time Optimization" (LTO) has been incorporated into CLASP. With LTO, all code that is compiled with COMPILE-FILE and all of the CLASP C++ code is compiled to the LLVM intermediate representation (LLVM-IR) and written to files as LLVM "bitcode". The link stage of building then does a final round of optimization, wherein LLVM-IR from Common Lisp code and LLVM-IR from C++ code for internal functions with symbols that are not exported, are inlined within each other. Those that are not inlined are converted to the "fastcc" calling convention, which passes arguments as much as possible in registers. The overall effect is to reduce the overhead of calls within CLASP.

7 C CALLING CONVENTION

Currently, CLASP only supports the x86-64 (System V) Application Binary Interface (ABI) because it makes non-portable references to calling convention details to improve performance. The calling convention passes six integer word arguments in registers and returns two integer word arguments in registers, and this is used by CLASP to utilize registers as much as possible when making function calls. Calls that pass four or fewer arguments are very fast in CLASP when compared to other Lisps (Table 5). CLASP passes arguments with the following C calling signature: (void* closure, size_t number-of-arguments, void* arg0, void* arg1, void* arg2, void* arg3, ...). So, up to four general objects are passed in registers and when the lambda list for a function uses &rest or &key arguments then CLASP uses the ABI details of the C calling convention "var-args" facility to spill the register arguments into a register save area on the stack. A C va_list is then "rewound" to point to the *arg0*

Figure 1: A generic function specialized on two arguments. The left box represents stamp values that are matched to the first argument stamp and the right boxes represent stamp values matched to the second argument depending on the first argument.


HIT-#<EMF :ptr 0x111a4e5d8>
341;SINGLE-FLOAT
347;FIXNUM
HIT-#<EMF :ptr 0x1119f1ba8>
Start
259;SYMBOL
296;SIMPLE-BASE-STRING
341;SINGLE-FLOAT
HIT-#<EMF :ptr 0x111af52a8>
347;FIXNUM
HIT-#<EMF :ptr 0x111ab2f38>

argument so that all arguments can be accessed one at a time using the clasp "vaslist" facility. clasp returns the first return value in the first return register and the number of return values in the second register. The remaining return values are written into a thread-local vector (one per thread).

8 INSTRUCTION POINTER ADDRESSING

clasp uses the LLVM library, which defines C++ classes for Modules, Functions, BasicBlocks, Instructions and GlobalVariables. Code generated by LLVM cannot currently be managed by the memory manager and must live outside of the managed memory space, at fixed locations. Functions therefore accumulate in memory as clasp runs. Memory depletion has not been a problem because modern computer memories are large, but it does make referencing Common Lisp objects from LLVM generated code problematic. To deal with this, each LLVM module has a block of garbage collector roots. These roots point to all non-immediate constants referenced by the functions in the module. These constants can then exist in memory managed space without being collected inappropriately.

9 STACK UNWINDING

Stack unwinding is achieved in clasp using C++ exception handling to allow clasp to inter-operate with C++ code. C++ stack unwinding on x86-64 uses Itanium ABI Zero-cost Exception Handling.[2] It is "zero-cost" in that there is no runtime cost when it is not used; however if any actual unwinding of the stack does occur, it does take quite some time. Timing of a simple CATCH/THROW pair demonstrates that unwinding the stack in clasp is about 90x slower than it is in SBCL, and thus stack unwinding must be done judiciously.

10 INLINING OF ARITHMETIC FUNCTIONS

Arithmetic can now be done without function calls in common cases. When the operands to a binary operation such as addition are of the same kind - fixnum, single float, or double float - the operation is carried out without a function call. This also occurs if one operand can be easily coerced to a value of the other type, e.g., a fixnum plus single float operation becomes a single float plus single float operation once a single float corresponding to the fixnum value is produced.

Facilities are now in place to deal with unboxed values. These are special register values, outside of the normal managed memory regime, representing number values. For example, even though 64-bit floats cannot be immediates due to lacking space for a tag, they can be dealt with as unboxed values. Inlined arithmetic essentially consists of "unboxing" values (e.g. extracting double floats from memory), performing machine arithmetic operations, and then boxing the results. A future compiler stage will remove unboxing operations followed immediately by boxing operations, as are done for arithmetic operations using the results of other operations as operands.

Arithmetic involving nontrivial allocations, such as on bignums or complex numbers, still goes through function calls.

11 PROFILING

Since all clasp Common Lisp and C++ code compiles to LLVM-IR, all functions look like standard C functions to standard profiling tools. This allows standard tracing tools like "dtrace"[3] to be used to profile clasp code (Figure 2). This puts all Common Lisp, C++, C, and Fortran code on a level playing field - making it possible to compare timing of functions written in different languages, linked together in the clasp mixed language environment.

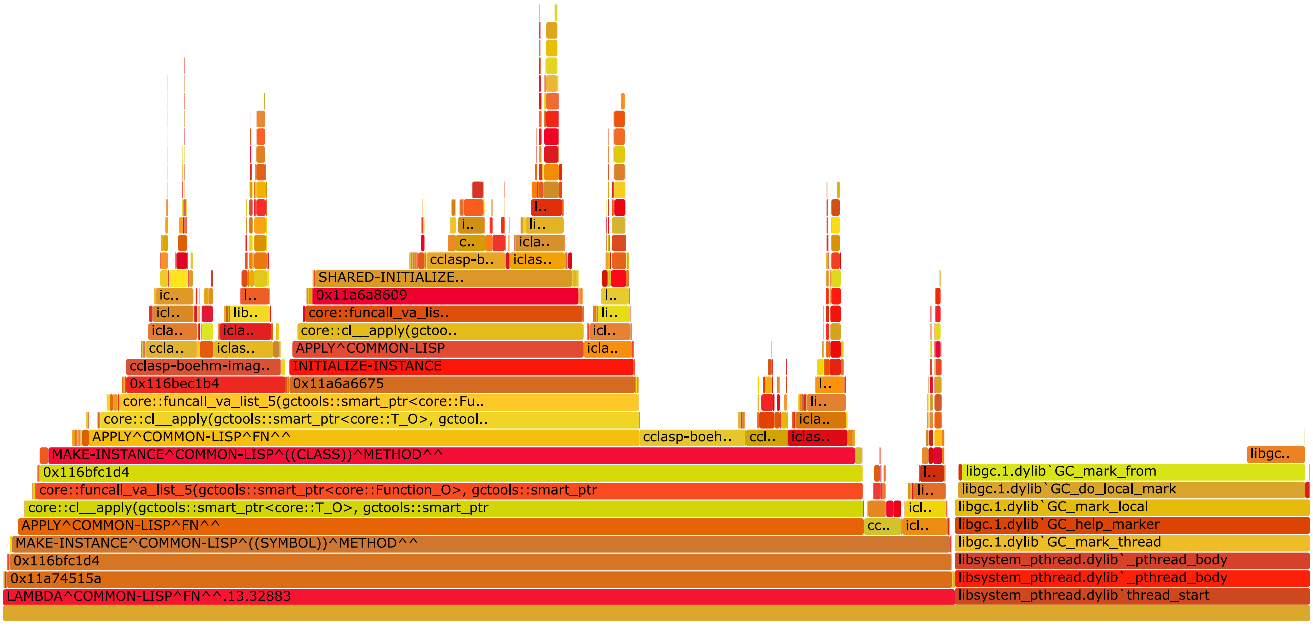
12 SOURCE TRACKING

clasp has incorporated source tracking using facilities from the SICL project. clasp makes extensive use of the nascent SICL project, including the cleavir compiler, and the SICL reader. The cleavir compiler has recently been upgraded to generate "Abstract Syntax Trees" (AST) from "Concrete Syntax Trees" (CST). CSTs are a representation of Common Lisp source code that has source location information attached to every atomic token and every list. One of the purposes of this is to carry source code location information into the AST and then all the way down to the instruction level in the final generated machine code to facilitate debugging. Other applications for CST's include tools that carry out source-to-source translation, and programming tools like syntax highlighters.

[2] http://itanium-cxx-abi.github.io/cxx-abi/abi-eh.html

[3] http://dtrace.org/blogs/about/

Figure 2: A flame graph generated by profiling CLASP repeatedly calling MAKE-INSTANCE. 19.7% of the time is spent in SHARED-INITIALIZE and 27.2% of the time is consumed by the memory manager.



13 DEBUGGING USING DWARF

CLASP generates DWARF debugging information using the DIBuilder API of the LLVM C++ library. This allows CLASP compiled programs to be inspected and debugged using the industry standard debuggers "gdb" and "lldb". DWARF debugging information is used by these debuggers to provide information about source line information, and with future work will provide the locations of variables in stack frames. The DWARF source line information is generated with the aid of source tracking information provided by Concrete Syntax Trees. The uniform use of DWARF debugging information for Common Lisp and C++ code allows the debugging of CLASP programs that make use of C++, C and Fortran libraries. The Lisp debugger does not yet use the DWARF debugging information fully due to the lack of a Common Lisp accessible DWARF interpreter. The Lisp debugger does provide backtraces with function names and arguments, through the use of an interim "shadow stack" mechanism.

14 MULTITHREADING

Pre-emptive multithreading based on the "pthreads" library[5] has been added to CLASP, fully supporting the "Bordeaux threads" library.[4] Special variables are bound thread locally and hash-tables can be declared "thread-safe". Several C++ classes like Function_O and FuncallableInstance_O contain C++ std::atomic⟨...⟩ fields that are updated atomically. Multithreading allows CLASP to run SLIME in the multithreaded :spawn mode. Multithreading also allows CLASP to run a JUPYTER[6] notebook server and SLIME server simultaneously - this allows interactive, web based, JUPYTER notebook applications to be developed with the full SLIME interactive development environment. Internal components such as CLOS, the Cleavir compiler and LLVM are all implemented in a thread-safe way - enabling us in the future to parallelize compilation.

15 CONCLUSIONS AND FUTURE WORK

CLASP is a new implementation of Common Lisp that interoperates with C++ and uses the LLVM library as a backend. It supports novel interoperation features with C++ libraries and industry standard profiling and debugging tools. CLASP incorporates the CLEAVIR compiler that is a platform for exploring new ideas in compiling dynamic languages. For future work we plan to eliminate the "warm-up" time of the fast generic function dispatch by preassigning stamps for system classes and building discrminating functions into the image at compile time. We plan to incorporate more inlining, type-inference, dead-code elimination and optimizations. We also plan to implement a DWARF interpreter to allow CLASP to access DWARF debugging information and provide it to the SLIME debugger.

16 ACKNOWLEDGMENTS

We want to thank professor Robert Strandh for his constant patience and guidance which made this work possible. We would also like to thank Stas Boukarev for help with some nasty bugs. We thank Frank Goenninger for his implementation of the low level CFFI interface.

REFERENCES

[1] *Schafmeister, Christian E. (2015).* Clasp - A Common Lisp that Interoperates with C++ and Uses the LLVM Backend. Proceedings of the 8th European Lisp Symposium: 90-91. http://github.com/drmeister/clasp

[4] https://trac.common-lisp.net/bordeaux-threads/wiki/ApiDocumentation

[2] *Schafmeister, Christian. E. (2016).* CANDO - a Compiled Programming Language for Computer-Aided Nanomaterial Design and Optimization Based on Clasp Common Lisp. Proceedings of the 9th European Lisp Symposium: 75-82.

[3] *Boehm, H. (1996).* Simple Garbage-Collector-Safety. Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation: 89-98.

[4] *Lattner, C. (2005).* Masters Thesis: LLVM: An Infrastructure for Multi-Stage Optimization. Computer Science Dept., University of Illinois at Urbana-Champaign, http://llvm.cs.uiuc.edu

[5] *Butenhof, David R. (1997).* Programming with POSIX Threads. Addison-Wesley. ISBN 0-201-63392-2.

[6] *Kluyver, T., Ragan-Kelley, B., Prez, F., Granger, B. E., Bussonnier, M., Frederic, J., ... & Ivanov, P. (2016, May).* Jupyter Notebooks-a publishing format for reproducible computational workflows. In ELPUB (pp. 87-90). doi:10.3233/978-1-61499-649-1-87

[7] *Barnes, N., & Brooksby, R. (2002)* "The Memory Pool System: Thirty person-years of memory management development goes Open Source." ISMM02.

[8] *Strandh, R. (2014).* Fast generic dispatch for Common Lisp. Proceedings of ILC 2014 on 8th International Lisp Conference - ILC 14. doi:10.1145/2635648.2635654

[9] *Wood, Alex. (2017).* Type Inference in Cleavir. Proceedings of the 10th European Lisp Symposium: 36-39.

[10] *The Unicode Consortium.* The Unicode Standard. http://www.unicode.org/versions/latest/

Session IV: Teaching & Learning

pLisp: A Friendly Lisp IDE for Beginners

Rajesh Jayaprakash
TCS Research, India
rajesh.jayaprakash@tcs.com

ABSTRACT

This abstract describes the design and implementation of pLisp, a Lisp dialect and integrated development environment modeled on Smalltalk that targets beginners.

CCS CONCEPTS

• **Software and its engineering** → **Integrated and visual development environments**;

KEYWORDS

lisp, integrated development environment

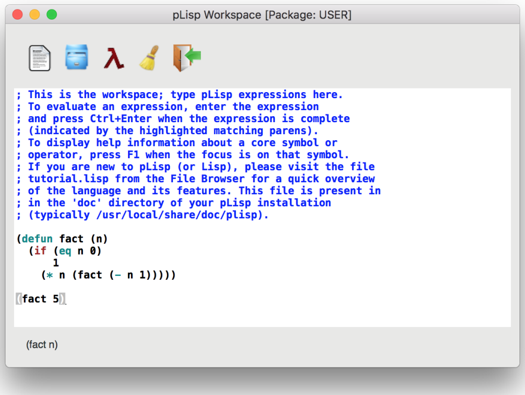
1 INTRODUCTION

pLisp is an integrated development environment (IDE) and an underlying Lisp dialect (based on Common Lisp) that is targeted towards beginners. It is an attempt at developing a Lisp IDE that matches (or at least approaches) the simplicity and elegance of typical Smalltalk environments and thereby hopefully providing a friendlier environment for beginners to learn Lisp.

Smalltalk environments are characterized by three interface components: the workspace, the transcript, and the system browser. The workspace and the transcript windows together serve the purpose of the canonical Read-Eval-Print Loop (REPL) used to interact with programming systems in the command-line mode, while the system browser is used to view the universe of objects available to the user and to define new objects. pLisp adopts the same idioms to model this interaction. Figures 1 and 2 illustrate sample screenshots where the user has entered an expression and has issued the command for evaluating the expression.

pLisp supports the following features:

- Graphical IDE with context-sensitive help, syntax coloring, autocomplete, and auto-indentation
- Native compiler
- Continuations
- Exception handling
- Foreign function interface
- Serialization at both system- and object level
- Package/Namespace system

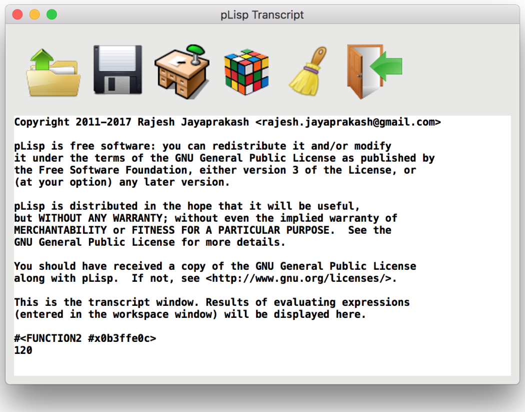


```
; This is the workspace; type pLisp expressions here.
; To evaluate an expression, enter the expression
; and press Ctrl+Enter when the expression is complete
; (indicated by the highlighted matching parens).
; To display help information about a core symbol or
; operator, press F1 when the focus is on that symbol.
; If you are new to pLisp (or Lisp), please visit the file
; tutorial.lisp from the File Browser for a quick overview
; of the language and its features. This file is present in
; in the 'doc' directory of your pLisp installation
; (typically /usr/local/share/doc/plisp).

(defun fact (n)
  (if (eq n 0)
      1
    (* n (fact (- n 1)))))

(fact 5)
```

Figure 1: The pLisp Workspace window



```
Copyright 2011-2017 Rajesh Jayaprakash <rajesh.jayaprakash@gmail.com>

pLisp is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

pLisp is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with pLisp.  If not, see <http://www.gnu.org/licenses/>.

This is the transcript window. Results of evaluating expressions
(entered in the workspace window) will be displayed here.

#<FUNCTION2 #x0b3ffe0c>
120
```

Figure 2: The pLisp Transcript window

The productivity-enhancing features like expression evaluation, autocompletion and auto-indentation of code, and context-sensitive help are available in all code-editing contexts (Workspace, code panels in the System Browser and Callers window, and the File Browser). Another useful Smalltalk-inspired feature implemented in pLisp is the ability to store the entire programming session—including the GUI state—in the serialized image; this enables the user to carry over the programming experience seamlessly across sessions, even in the middle of a debugging exercise.

pLisp has been released under the GPL 3.0 license and is freely available for download [1]. At present, pLisp is available for Linux (both 32-bit and 64-bit), Windows (32-bit), and Mac OS X platforms. pLisp is written in C, and relies on open-source components (GTK+,

ELS'18, April 16–17 2018, Marbella, Spain
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-2-9557474-2-1.

GtkSourceView, Tiny C Compiler, the Boehm Garbage Collector, and libffi).

2 IMPLEMENTATION

pLisp is a Lisp-1 dialect, i.e., functions share the same namespace as the other objects in the system. The syntax of pLisp closely mirrors that of Common Lisp (e.g., `defun`, `defmacro`, `progn`, and macro-related constructs like backquote, comma, and comma-at), however, notations from Scheme are also used (`call/cc`). The design philosophy of pLisp is to be more-or-less source-code compatible with Common Lisp so that users can easily transition to Common Lisp and carry over their knowledge and code.

2.1 Syntax

The pLisp s-expression grammar is shown in Figure 3. Except for the language constructs and primitive operators, the core of pLisp is written in itself. The support for continuations and the `call/cc` construct, coupled with the use of macros, enables this and the implementation of sophisticated programming constructs like loops and exception handling at the library level.

E ::= $L \mid I$
| (define I_{name} E_{defn})
| (set I_{name} E_{defn})
| (lambda (I^{*}_{formal}) E^{*}_{body})
| (macro (I^{*}_{formal}) E^{*}_{body})
| (error E)
| (if E_{test} E_{then} E_{else})
| (E_{rator} E^{*}_{rand})
| (let ((I_{name} E_{defn})*) E^{*}_{body})
| (letrec ((I_{name} E_{defn})*) E^{*}_{body})
| (call/cc E)

Figure 3: pLisp informal s-expression grammar

2.2 Object Model

pLisp supports the following object types:

- Integers
- Floating point numbers
- Characters
- Strings
- Symbols
- Arrays
- CONS cells
- Closures
- Macros

All objects are internally represented by `OBJECT_PTR`, a `typedef` for `uintptr_t`, the C language data type used for storing pointer values of the implementation platform. The four least significant bits of the value are used to tag the object type (e.g., `0001` for symbol objects, `0010` for string literals, and so on), while the remaining *(n-4)* bits (where *n* is the total number of bits) of the value take on different meanings depending on the object type, i.e., whether the object is a boxed object or an immediate object. If the object is a boxed object, the remaining bits store the referenced memory location. The loss of the four least significant bits is obviated by making use of the `GC_posix_memalign()` call for the memory allocation and thus ensuring that the four least significant bits of the returned address are zeros.

2.3 Compiler

The pLisp compiler transforms the code to continuation-passing style (CPS) [2] and emits C code, which is then passed to the Tiny C Compiler (TCC) to produce native code. The compiler does the transformation in the following passes [3]:

- Desugaring/Macro expansion
- Assignment conversion
- Translation
- Renaming
- CPS conversion
- Closure conversion
- Lift transformation
- Conversion to C

These passes produce progressively simpler pLisp dialects, culminating in a version with semantics close enough to C. Since TCC is utilized for the native code generation, the transformation pipeline does not include passes like register allocation/spilling.

2.4 Debugger

Since pLisp uses the continuation-passing style, all the functions invoked in the course of evaluating the expression are extant at any point in time, and are displayed in the debug call stack. At present, only the break/resume functionality (and inspection of function arguments) is supported in pLisp.

The compilation process introduces a large number of internal continuation functions as part of the CPS conversion pass; the debugging infrastructure needs to filter out these continuations so that the user is presented with only those functions they need to be aware of (i.e., those that have external source representations). This is accomplished by logic in the C conversion phase, which generates code to store a closure in the debug stack only if that closure maps to a top-level definition.

3 CONCLUSION

This abstract describes the design and implementation of pLisp, a Lisp dialect and integrated development environment modeled on Smalltalk that targets Lisp beginners. While pLisp is oriented towards beginners, its feature-set is complete enough (and its performance robust enough) to serve the needs of a typical medium-sized Lisp development project. Introduction of multithreading capabilities and enhancements to the debugger to enable continuing or restarting a computation with user-supplied values are part of the future work being considered.

REFERENCES

[1] R. Jayaprakash. pLisp IDE. https://github.com/shikantaza/pLisp, 2018.

[2] Guy L Steele Jr. Rabbit: A compiler for scheme. Technical report, Technical Report AI-TR-474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, 1978.

[3] Franklyn Turbak, David Gifford, and Mark A Sheldon. *Design concepts in programming languages.* MIT press, 2008.

Using Lisp-based pseudocode to probe student understanding

Christophe Rhodes
Goldsmiths, University of London
London, United Kingdom

ABSTRACT

We describe our use of Lisp to generate teaching aids for an Algorithms and Data Structures course taught as part of the undergraduate Computer Science curriculum. Specifically, we have made use of the ease of construction of domain-specific languages in Lisp to build an restricted language with programs capable of being pretty-printed as pseudocode, interpreted as abstract instructions, and treated as data in order to produce modified distractor versions. We examine student performance, report on student and educator reflection, and discuss practical aspects of delivering using this teaching tool.

CCS CONCEPTS

• **Applied computing** → **Computer-assisted instruction**; • **Social and professional topics** → *Computational thinking*; • **Theory of computation** → *Program constructs*; *Program reasoning*; • **Software and its engineering** → *Multiparadigm languages*;

KEYWORDS

multiple choice questions, Lisp

ACM Reference Format:
Christophe Rhodes. 2018. Using Lisp-based pseudocode to probe student understanding. In *Proceedings of the 11th European Lisp Symposium (ELS'18).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

In this paper, we discuss the development and use of a large question bank of multiple-choice, short-answer and numerical-answer questions in teaching a course in Algorithms & Data Structures, as a component of degree programmes in Computer Science and in Games Programming in the United Kingdom. We report specifically on the use of automation, using Lisp among other tools, to develop questions with specific distractors and specific feedback corresponding to likely or common student mistakes.

Gamification of learning has been experimented with and studied in detail in recent years, with the increasing availability of platforms and integrations allowing for more and varied gamification techniques to be applied at all stages of a student's education; the benefits of gamification include higher student engagement, with the curriculum material (as the tasks are intended to probe or reinforce the course content) and with the rest of the cohort (through the social elements of game-playing). The approach we describe here can be viewed as an application of gamification techniques; in the categorization of a recent systematic mapping [3], we describe an element of gamification in a blended-learning course delivered in conjunction with a learning management system, with rapid feedback and countdown clocks, specifically in the context of Computer Science education but applicable more widely.

In the remainder of this introductory section, we provide some relevant context for the Algorithms & Data Structures course in which we have implemented this pedagogy: the conventions of Higher Education in the UK, of Computing education in UK Higher Education, and of the particular programmes of study at Goldsmiths. Section 2 covers the development of multiple-choice question banks suitable for our purposes, including the use of Lisp to help to generate the questions, specific feedback, and assure their correctness. Section 3 describes the in-course delivery of quizzes including these questions, presenting quantitative results and qualitative reflections from students and educators, and we conclude in section 4.

1.1 UK Higher Education

In the UK, Tony Blair in 1999 famously gave as an aim that half of all young people should go to University. Since giving that aim in a party conference speech, the UK Higher Education landscape has shifted substantially, with the introduction and raising of tuition fees (from £1k per year, to £3k and then £9k per year), and the removal of quotas and caps in student recruitment, and the situation is indeed that half of people under 30 in the UK have started a programme of Higher Education, compared with approximately one quarter two decades ago.

This rapid growth in student numbers has inevitably led to pressure on resources: campus space, lecture halls, and staff time. Additionally, placing more of the costs of Higher Education on the student, even if through a notional student loan (which operates more like a tax), has led to more consumerist and arguably transactional approaches to education from the students: it is more common to hear from students now that they are paying for content that they will consume than it would have been twenty years ago. In addition, students nowadays are digital natives; they are accustomed to online delivery of content, though perhaps not so much of material requiring substantial engagement; they are used to the affordances provided by online platforms, and indeed are somewhat intolerant of the perceived backwardness of some Learning Management Systems.

One might expect, given that the students are at least notionally responsible for the cost of their own higher education, that students would have made an informed choice about their programme of study and have clarity about their reasons for entering Higher Education. However, this is not always the case [5, 6], and even when it is, those reasons may not align with the educator's reason

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1…$15.00

for teaching in Higher Education; a majority of students enter into Higher Education seeing it as a means to an end, of getting a job that would otherwise be inaccessible, or having a better chance at a particular career – whereas few teachers in Higher Education have the career development of their students as their primary motivation.

Teachers in Higher Education do operate under constraints, sometimes quite severe ones. One such constraint is that the system works in a way that expects it to be unlikely for students to fail courses. Even minimal engagement with the material is expected to yield a passing grade; degrees are further *classified*, with classifications of a "first-class" or an "upper second" being considered of high enough quality to act as an entry qualification for typical graduate trainee schemes or study for advanced degrees, while "lower second" or "third-class" classifications, while indicating that the degree was passed, are seen as being of lesser quality[1]. Conventionally throughout the sector, a mark of 40% is a pass, and a mark of 60% is the boundary between lower- and upper-second degree classifications.

1.2 Computing education

When offering a degree programme in Computer Science or a related discipline, we must be conscious of the fact that we will have at least three constituencies in our student cohort. We may have some students who will go on to further academic study of the discipline itself; however, we would expect those students to be small in number compared with the students who are studying Computer Science as a means to an end (such as a career in Informatics) or who do not have a particular reason for studying Computer Science at all.

In designing our curricula and our teaching methods, we must therefore accommodate multiple different styles of learning and a wide range of current and prior engagement. We will have to teach students who are already accomplished programmers and wish to deepen their theoretical understanding, and students who believe that a University course can teach them to programme so that they can go out and get a job. We must therefore be careful to nurture development of applicable and transferrable ways of thinking, helping the students to develop computational thinking [11] or build mental models or "notional machines" [9] of the systems that they interact with.

Teaching students to programme, and to reason about programmes, is difficult – and assessing whether students have mastered individual elements of the skill [1, 8] potentially has a high cost. We do not claim to have found a panacea, but one aspect which we believe is particularly demotivating is the somewhat binary nature of assessment: it is common to see bimodal distributions of outcomes, or at least high failure rates [7], typically corresponding to a failure by the student to get anything working at all – an experience seen in microcosm by anyone faced with inscrutable compiler or linker error messages. As educators, we should aim to find ways to allow students to receive partial credit for partial solutions, so as to recognize forward progress even if it has not yet led to a fully-functional implementation.

1.3 Algorithms & Data Structures at Goldsmiths

We report in this paper on a course in Algorithms & Data Structures at Goldsmiths. There is a particular issue in the delivery of this course: it is taken as a compulsory part of the programme by students on the BSc in Computer Science (CS) programme and those on the BSc in Games Programming (GP). The CS students are taught programming in Java, while the GP students are taught programming in C++. This creates a particular challenge, in that examples need to be in both or neither programming language in order not to give the perception of unfair or second-class treatment to either cohort. In this course, students are given practical programming work in the form of small automatically-marked lab assignments as well as more open tasks, but theory is presented in a language-neutral pseudocode format.

2 QUESTIONS

One of the components of our delivery of this material is a series of multiple choice quizzes, delivered through the Moodle[2] Learning Management System (LMS). These quizzes are intended to be part measurement instrument – the mark achieved contributes to the final grade in the course – but chiefly a pedagogical tool, to help the students recognize whether they have understood the material sufficiently to identify or generate solutions to problems.

The function of our questions is similar to the *root question* concept described in the Gradiance documentation [10]: we aim to produce questions, or question templates, with the following characteristics:

- a student who has understood the material should find answering the question to be straightforward;
- a student who has not begun mastering the material should have a low probability of being able to guess the correct answer;
- individual or groups of students should find it easier to master the material than to acquire and search through a set of questions with corresponding answers;
- for multiple choice questions, distractor answers corresponding to common misunderstandings or misconceptions should be present.

The reason for the last characteristic, that distractor answers should be present, is to be able to identify individual students, or measure the fraction of students, with a particular misunderstanding, and to give them targetted feedback aimed to improve their understanding. There is no need for distractors in numerical- or short-answer questions, but the questions we produce must still be done with that understanding, in order to be able to give targetted feedback for particular wrong answers. The subsections below give examples of targetted feedback in both short-answer and multiple-choice questions.

[1] This is a highly simplified description, as there are also distinctions between the perceived quality of degrees awarded by different institutions, being a combination of reputational teaching quality and expected student attainment at intake.

[2] https://moodle.org/

What is the return value of this block of code? You may assume that the value of all variables before the start of this block is 0.

```
x ← 4
for -5 ≥ i > -15 do
    x ← x + 1
end for
return x
```

Figure 1: example simple loop question, question 6 of the Pseudocode quiz

What is the return value of this block of code? You may assume that the value of all variables before the start of this block is 0.

```
x ← 8
for 4 ≤ i < 16 do
    x ← x + 1
    break
    x ← x + 1
end for
return x
```

Figure 2: example loop question, question 8 of the Pseudocode quiz

2.1 Pseudocode

As described in section 1.3, the class taking this course consists of two separate cohorts. To establish a common language, therefore, an early lecture established the pseudocode conventions to be used throughout the course (essentially a subset of the algpseudocode notation provided by the LaTeX algorithmicx package).

One of the questions (see figure 1) asked participants to compute the final value of a variable after it was incremented within a loop: the intent of the question was to make sure that the students could identify the number of times the loop body was executed. As well as the generic feedback given to a student after an attempt, specific feedback was included to be shown to the student when they had made an off-by-one error, reminding them to check the boundaries of the iteration carefully.

A subsequent question in the same quiz used the same question format, but introduced the keywords **break** and **continue**. Again, students were given the generic indication for correct or incorrect answers, but also specific feedback for particular wrong answers given if the student had computed the return value for the wrong keyword, or for no keyword present at all (see figure 2).

The Moodle LMS provides for automatic generation of variants of questions through its Calculated question type, where a template is filled in with randomly-chosen values, and a symbolic expression (supporting standard mathematical operators) is interpreted with each variable from the template bound to the corresponding value. This facility is sufficient for questions based on simple calculations, but has disadvantages for our purposes: the interface for writing calculated questions requires a connection to the Moodle server, and cannot be done off-line; it requires hand-editing each question, which is error-prone; and generating non-numerical variants automatically (*e.g.* choosing between **break** and **continue**) is not possible.

We therefore took a different approach. We defined a sexp-based mini-language to represent the constructs supported in our pseudocode, and implemented a pretty-printer and an interpreter in Emacs Lisp. The definition and implementation were extended as necessary from an initial set of six operators (the basic mathematical operators, variable setting, and **return**) to encompass conditionals, loops, function definition, and various elementary data structures and operations on them (such as lists and vectors).

We could then generate valid forms in our mini-language, somewhat reminiscent of generation of random forms for compiler testing [4]; see listing 1, which is the code to generate random examples of the block presented in figure 2. These sexp-based forms are then pretty-printed to Moodle's GIFT input format[3], and surrounded with question markup to produce questions such as the ones presented in figures 1 and 2.

```
(defun make-break-continue-for-form ()
  (let* ((ascend (flip))
         (comps (if ascend '(< ≤) '(> ≥)))
         (lc (elt comps (if (flip) 0 1)))
         (uc (elt comps (if (flip) 0 1)))
         (start (* (maybe-sign) (random 10)))
         (diff (+ (random 10) (random 10) 1))
         (end (if ascend (+ start diff) (- start diff))))
    `(progn
       ,(make-form 'setq 'x)
       (for (,start ,lc i ,uc ,end)
            (progn
              (incf x 1)
              ,(if (flip) `(break) `(continue))
              (incf x 1)))
       (return x))))
```

Listing 1: Emacs Lisp code to generate a loop in our mini-language containing a break or continue within a for loop, with reasonable start- and end-points.

Not only this, but if we could express a likely mistake that a student might make in code (such as the off-by-one errors or the confusion between **break** and **continue**), we could generate the corresponding form, interpret it, and write specific feedback based on that specific mistake, while checking that it did not accidentally replicate the correct answer. Code to pretty-print, add the question, answer and feedback is demonstrated in listing 2.

This approach also allowed for more fine-grained mistake detection in questions such as in figure 1, where instead of generic feedback related to off-by-one errors (or -two, one at each end of the loop), the feedback was generated based on the specific confusions in each randomly-generated question between < and ≤ and between > and ≥.

2.2 Recursive functions

Another aspect that students often struggle with is grasping recursion, though that is a component of computational thinking (and, arguably, a shibboleth to be probed in job interviews). Students

[3] https://docs.moodle.org/en/GIFT_format

```
(defun return-break-continue-for (n)
  (dotimes (i n)
    (let* ((form (make-break-continue-for-form))
           (answer (interpret-form form))
           (osub '((break . continue) (continue . break)))
           (oform (sublis osub form)
           (other (interpret-form oform))
           (nsub '((break . progn) (continue . progn)))
           (nform (sublis nsub form)
           (neither (interpret-form nform)))
      (insert (format "::R.%s::" (make-name form)))
      (insert (format "What is the return value from the
following block of pseudocode?\nYou may assume that the
value for all variables before the start of this block
is 0.<br/>\n"))
      (dolist (x (format-form form))
        (insert
         (format
          "&nbsp;&nbsp;%s<br/>\n"
          (replace-regexp-in-string " " "&nbsp;" x))))
      (insert (format "{#
  =%s\n
  =%%0%%%s#have you mixed up break and continue?\n
  =%%0%%%s#are both increments executed?\n}\n\n"
                      answer other neither)))))
```

Listing 2: Emacs Lisp code using the form generator from listing 1, modifying and interpreting the form in order to generate answers which might be given by students with a mistaken mental model. This function outputs n questions of this form in Moodle's GIFT format, ready to be imported.

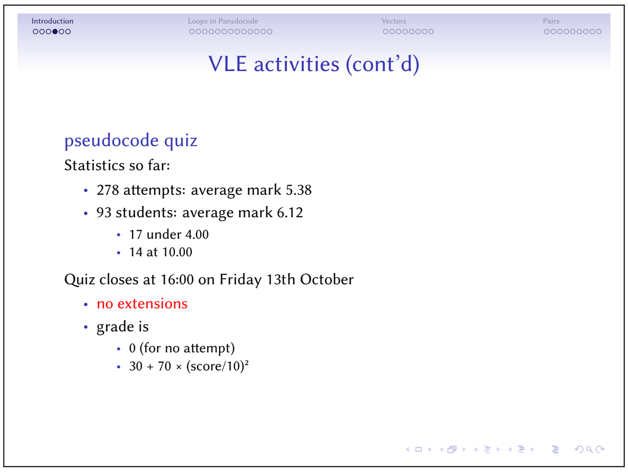
What code fragment should replace Z for function A to return the difference between a and b? You may assume that the initial arguments to the function A are positive integers and that b ≤ a.

```
  function A(a,b)
     if a = b then
        return 0
     else
        return Z + 1
     end if
  end function
```

○ A(a, b + 1)
○ A(a, b - 1)
○ A(a - 1, b - 1)
○ A(a - 1, b + 1)
○ A(a + 1, b - 1)
○ A(a + 1, b + 1)
○ A(a - b, b)
○ none of the other answers

Figure 3: example code building question, question 6 of the Recursive Algorithms quiz

are encouraged to think about base cases, and to consider transforming one or more solutions to a subproblem into the solution to the whole problem, but the details are important and it is easy for students to be lulled into a false sense of security by doing a small number of exercises – or, alternatively, to not experience the desired moment of enlightenment, and feel that forward progress is not possible.



VLE activities (cont'd)

pseudocode quiz

Statistics so far:

- 278 attempts: average mark 5.38
- 93 students: average mark 6.12
  - 17 under 4.00
  - 14 at 10.00

Quiz closes at 16:00 on Friday 13th October

- no extensions
- grade is
  - 0 (for no attempt)
  - $30 + 70 \times (\text{score}/10)^2$

Figure 4: A slide from the lecture given after the quiz on pseudocode had been open for a week.

In order to help our students measure their understanding of recursion, we generated in our mini-language multiple recursive implementations of basic mathematical operations (addition, subtraction, multiplication, division, exponentiation), and used our pretty-printer to generate questions where the conditional, base case, or recursive call had been elided. We collected from the variants the corresponding possibilities for each of these code locations, and generated multiple-choice questions with a subset of these possibilities as choices (see figure 3 for an example). We were able to ensure that we did not mistakenly include an accidentally-correct answer from the possibilities as a distractor, by substituting in each possible response into the corresponding functional form, interpreting it for randomly-chosen arguments, and checking that it did *not* return the mathematically correct answer.

3 DELIVERY AND RESULTS

Throughout the course, a new quiz on an individual topics (such as those described in sections 2.1 and 2.2) was made available to the students each week, with each quiz open for a 12-day period, from Monday at 09:00 until 16:00 on the Friday of the following week. The students were informed that each such quiz would be worth 1% of their final grade, and at the mid-point of the open period were shown a summary of the current cohort performance in that quiz (see figure 4). The non-linear transformation in that slide, from quiz score (out of 10) to awarded mark (out of 100) is to encourage participation (a mark of 30% is a fail, but not a bad one) and to avoid rewarding guesswork (a quiz score of 2 or 3 out of 10, as might occur through chance, still leads to a failing mark).

The cohort of 120 students took the pseudocode quiz 588 times, achieving scores plotted in figure 5. We tracked the improvement in quiz scores relative to each student's first attempt in that quiz, to attempt to measure the effect of practice and feedback (figure 6), which displayed a general improvement with diminishing returns and levelling off at around eight attempts; and students' best scores

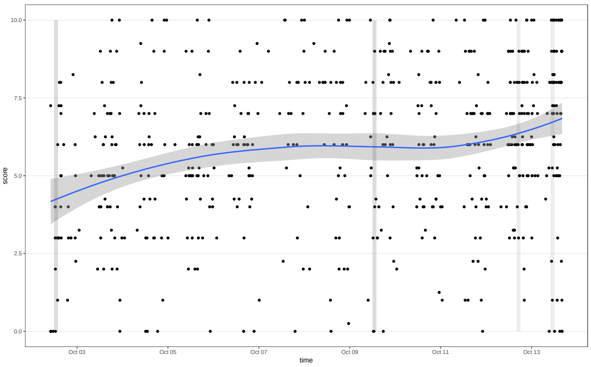


Figure 5: Individual scores (out of 10) in the pseudocode quiz over the period of its availability. The vertical shaded areas represent contact times (lectures and lab sessions).

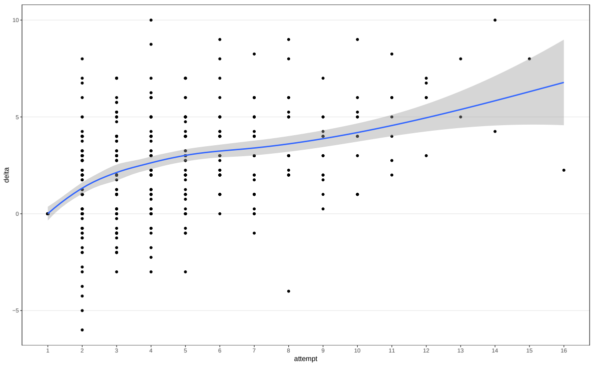


Figure 6: Improvement in student scores in the pseudocode quiz compared with the score attained in their first attempt.

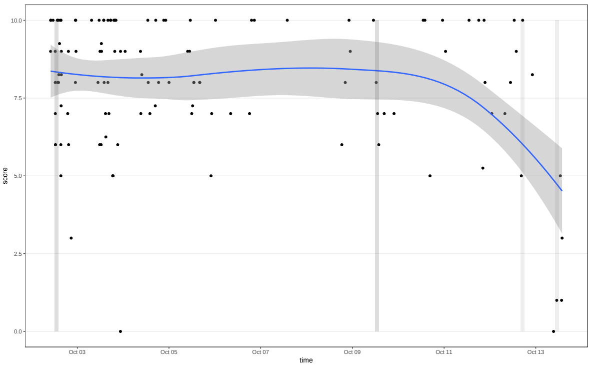


Figure 7: Students' best scores in the pseudocode quiz, plotted against the time when they took their first attempt at the quiz.

	correct	off-by-one	incorrect	unanswered
best	83	11	21	6
other	193	90	135	49

Table 1: classified results for question 6 of the Pseudocode quiz: the "incorrect" column refers to answers given but neither correct nor off-by-one.

	correct	incorrect	unanswered
best	76	35	10
other	123	218	124

Table 2: classified results for question 8 of the Pseudocode quiz. Unfortunately the different categories of incorrect answers (confusion between break and continue, failure to consider how it interacts with the for loop) are not easy to differentiate from the Moodle reports.

	correct	incorrect	unanswered
best	68	46	1
other	69	165	14

Table 3: classified results for question 6 of the Recursive algorithms quiz.

in the quiz plotted against the time of their first attempt (figure 7), where we found that there was no difference in the final outcome provided the student started the quiz activity before two or three days before the deadline.

The cohort's performance in question six (the simple loop question) is given in table 1, while their performance in question eight (the loop with **break**/**continue**) is shown in table 2.

As can be seen from table 1, student performance in this question is considerably better in the aggregate of the best performance of each student than in the other questions. This is expected; what might be unexpected is the degree to which the specific issue of off-by-one errors has been reduced. In the questions representing the best attempts by each student, the error rate corresponding to off-by-one errors is 11 in 121, or 9.1%, this is a reduction from 19.3% in the population of non-best attempts, or roughly a halving of this error. By contrast, other incorrect answers decreased from 28.9% in general attempts to 17.4% in the best attempts; a decrease of generic errors of roughly 40%. The decrease in the proportion of unanswered question reflects the observed pattern that for many students early attempts at the quiz under time pressure means that they run out of time before answering the harder questions in the quiz.

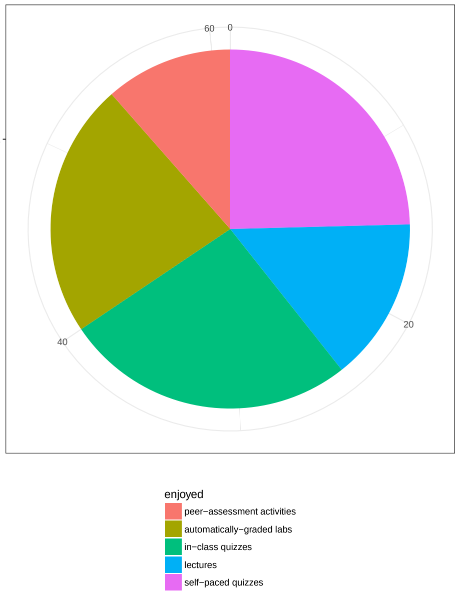


Figure 8: Student responses to the question “Which activity in this course so far have you most enjoyed?”

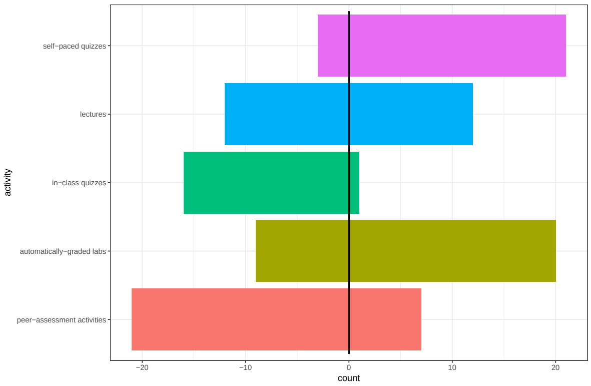


Figure 9: Student responses to the question “Which activity in this course so far have you learnt most [positive counts] / least [negative counts] from?”

3.1 Student perspectives

Near the half-way point of the course, the students were asked to provide feedback, first in a non-anonymous custom questionnaire delivered using the Learning Management System, and second anonymously using the standard course evaluation questionnaire provided by the University. Neither method of soliciting feedback reached complete coverage of the students; indeed, only approximately half the cohort (n=61 students) completed the non-anonymous questionnaire, and even fewer (n=40 students) the standard course evaluation.

As well as the self-paced multiple-choice quizzes described in this paper, the students were given:

- automatically-graded lab exercises, typically to implement particular algorithms or data structures, with their implementation assessed for correctness and targetted feedback generated using JUnit[4] and cppunit[5], managed by the INGInious platform [2];
- peer-assessed extended exercises, with more open briefs than the lab exercises, and an assessment rubric set up for them to assess each others' submissions;
- in-class multiple-choice quizzes, typically given at the half-way point of a lecture, reinforcing or revising particular points, delivered using kahoot![6];
- standard weekly lectures of two hours' duration.

Figure 8 shows the student answers to the question of which of the various activities they most enjoyed in the non-anonymous questionnaire. The 61 respondents divide fairly evenly between the five classes of activity, with a slight preference for self-paced quizzes, in-class quizzes and lab exercises compared with lectures and peer-assessment. The responses to the question of which activities the students considered most or least instructional, however, are starkly different; figure 9 illustrates the answers to those two questions, with positive counts representing answers to the “most” variant and negative counts representing “least”. From these responses, we see that the students value highly the automatically-graded lab exercises and particularly the multiple-choice quizzes; very few students considered the quizzes the least instructional activity, compared with over one-third who considered them the most; students are clearly distinguishing between enjoyment and pedagogy, in that the in-class quizzes, which were considered to be most enjoyable by many students, were rated as being most instructional by very few.

Students were also encouraged to leave free-text comments in both questionnaires. Some expressed frustration about particular aspects of multiple-choice quiz delivery, requesting that the time limit for quiz attempts be raised or the enforced gap between attempts be lowered; however, several commented on the level of challenge posed by the quizzes, and there have been in-person requests to make the quizzes available after the deadline for that quiz to help students guide their further learning and revision.

Student engagement in the quiz activities has remained high; in the 16 completed quizzes in this academic year, the students have submitted 6792 quiz attempts, each with 10 questions (so each student has, on average, received automated feedback on 566 individual questions).

3.2 Educator perspectives

Using multiple-choice questions with the approach given in this paper has several benefits from the perspective of an educator. Firstly, and most importantly, it provides for instruments which the students can take, multiple times, in order to judge their own state of understanding of the foundational components of the material and receive feedback regarding where and how that understanding might be lacking. Importantly, it allows that feedback to be delivered and received at a time of the student's own convenienence; once

[4] https://junit.org/junit4/
[5] https://freedesktop.org/wiki/Software/cppunit/
[6] https://kahoot.com/

the questions are generated and the automation set up, there is no additional cost, freeing up educator time to devise more useful activities or provide extra material.

In addition, this approach can scale to the required size; this entire course was delivered to a cohort of 120 students using one instructor and one teaching assistant; this course *does* have some non-automated components of delivery, such as moderation of peer-assessment, marking of an individual written assignment, and marking a final exam with longer-form questions. Scaling to larger student numbers, as in a fully-online or MOOC setting, might require some additional instructor time to monitor student questions on online forums – though our experience in running this course this year is that the students themselves are well equipped to assist each other on public forums, and indeed it is acknowledged that helping each other in this way helps to consolidate learning and build mastery; the teaching staff participation on the forums is largely limited to administrative announcements and to provision of material beyond the formal syllabus.

Further, it is important for us to know whether our students (as a whole) have a good understanding of the material, a mixed understanding, or maybe that a substantial part of the cohort has misunderstood some topic. One benefit of having the quizzes open for 12 days was that, at the half-way point, we could examine the results so far and identify whether any specific part of the quiz showed substantially worse (or worse than expected) performance – and if so, that specific item could be addressed in a plenary session such as a lecture.

A concern sometimes raised about using self-paced, remotely-administered tests as a component of a final grade is that students might be incentivised to cheat, for example by asking other people to take the test on their behalf. One mitigation is that, since each of these tests is worth 1% of their final grade, and students can get a mark of 30% simply by submitting a blank entry, there is limited upside to cheating; meanwhile, we performed spot-checks on individual elements of suspicious behaviour, by requiring some students to take quizzes under controlled conditions after identifying anomalies in the logs (such as two students taking the same quiz from the same IP address in quick succession – the students replicated their scores of 10, and revealed that they had been racing each other!). If the concern is strong, there is nothing in the approach described here which would prevent quizzes being used primarily formatively, and possibly assessed for part of a course mark under controlled conditions.

We would not expect to be able to be able to build a community of users of our specific mini-language and toolset; it grew to meet immediate needs, and it fulfils those needs minimally. The general approach – identifying potential pitfalls or barriers to understanding, designing assessments that probe those barriers, and providing specific feedback in the case of students demonstrating that they are struggling with those barriers – is, we believe, sound, and we have demonstrated that at least in some subject areas this can be done in a scalable way. It is perhaps surprising not to see this approach taken up more widely; we speculate that this is because the technical sophistication level required to operate the toolchain is fairly high; the up-front cost of development is steep; and that the pedagogical approach taken implies more empathy with the student and more responsibility for the learning journey, which are not necessarily aspects selected for in hiring teaching staff at University.

4 CONCLUSIONS AND FUTURE WORK

We believe that providing automated tools where students can probe in detail their own understanding of the fundamentals of the curriculum that they are studying is valuable. This provision will also become more necessary as Higher Education Institutions become more resource-constrained, as students expect more for their tuition fees, and as competitors such as OpenCourseWare and MOOCs establish the principle in students’ minds that pedagogical materials are available for anyone to access for free.

In the specific case of a Computer Science curriculum, and Algorithms & Data Structures specifically, we identified tangible benefits to the practical pedagogy from the use of a Lisp with strengths both in treating code-as-data and data-as-code, and for text manipulation. As well as the Lisp-based mini-language for interpreting and formatting pseudocode described in this paper, we implemented an Emacs major mode for editing GIFT-format files, to make hand-edits to generated questions fast and practical; we implemented simplified versions of many elementary and more complex data structures, in order to be able to generate questions on the behaviour of hashtables or the properties of graphs; and all this under time pressure and on a budget.

For this specific course, one potential improvement would be to implement a parser for the surface syntax of pseudocode, converting it back to our sexp-based language. This, in combination with a plugin for the LMS, would allow us to set free-text rather than multiple-choice questions for topics such as recursive algorithms, where even with the large number of distractor questions there is some chance that some students will select right answers by pattern-matching, without a full understanding of the material.

There are improvements that can be made in delivering multiple-choice quizzes compared with this year. While giving specific feedback to the students about particular mistakes is helpful, it is also useful for instructors to know that this has happened. The Moodle LMS does not make it easy to see the frequency that particular wrong answers are given from the reports that it produces; however, we could allocate distinct fractional marks, rather than zero, to specific wrong answers; this would not substantially affect the quiz score (a wrong answer scoring 0.01 points instead of 0.00 will not have a material effect on a student outcome) but would make it much more straightforward to analyse data extracted from the LMS.

REFERENCES

[1] Jaap Boender, E. Currie, M. Loomes, Giuseppe Primiero, and Franco Raimondi. Teaching functional patterns through robotic applications. In *Proceedings of the 4th and 5th International Workshop on Trends in Functional Programming in Education, TFPIE 2016, Sophia-Antipolis, France and University of Maryland College Park, USA, 2nd June 2015 and 7th June 2016.*, pages 17–29, 2016. doi: 10.4204/EPTCS.230.2. URL https://doi.org/10.4204/EPTCS.230.2.

[2] Guillaume Derval, Anthony Gego, Pierre Reinbold, Benjamin Frantzen, and Peter Van Roy. Automatic grading of programming exercises in a MOOC using the INGInious platform. In *Proceedings of the European MOOC Stakeholder Summit*, pages 86–91, Mons, May 2015.

[3] Darina Dicheva, Christo Dichev, Gennady Agre, and Galia Angelova. Gamification in Education: A Systematic Mapping Study. *Educational Technology & Society*, 18(3):75–88, 2015.

[4] Paul F. Dietz. The GNU ANSI Common Lisp Test Suite. Presented at International Lisp Conference, Stanford, July 2005, 2005. http://cvs.savannah.gnu.org/viewvc/*checkout*/gcl/gcl/ansi-tests/doc/ilc2005-slides.pdf.

[5] Susan Harter. A new self-report scale of intrinsic versus extrinsic motivation in the classroom: motivational and informational components. *Developmental Psychology*, 17:302–312, 1981.

[6] Stephen E. Newstead, Arlene Franklyn-Stokes, and Penny Armstead. Individual differences in student cheating. *Journal of Educational Psychology*, 88(2):229–241, 1996.

[7] Elizabeth Patitsas, Jesse Berlin, Michelle Craig, and Steve Easterbrook. Evidence that computer science grades are not bimodal. In *Proceedings of the 2016 ACM Conference on International Computing Education Research*, ICER '16, pages 113–121, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4449-4. doi: 10.1145/2960310.2960312. URL http://doi.acm.org/10.1145/2960310.2960312.

[8] Franco Raimondi, Giuseppe Primiero, Kelly Androutsopoulos, Nikos Gorogiannis, Martin Loomes, Michael Margolis, Puja Varsani, Nick Weldin, and Alex Zivanovic. A racket-based robot to teach first-year computer science. In *European Lisp Symposium*, IRCAM, Paris, May 2014.

[9] Juha Sorva. Notional machines and introductory programming education. *Trans. Comput. Educ.*, 13(2):8:1–8:31, July 2013. ISSN 1946-6226. doi: 10.1145/2483710.2483713. URL http://doi.acm.org/10.1145/2483710.2483713.

[10] Jeffrey D. Ullman. Gradiance on-line accelerated learning. In *Proceedings of the Twenty-eighth Australasian Conference on Computer Science - Volume 38*, ACSC '05, pages 3–6, Darlinghurst, Australia, Australia, 2005. Australian Computer Society, Inc. ISBN 1-920-68220-1. URL http://dl.acm.org/citation.cfm?id=1082161.1082162.

[11] Jeannette M. Wing. Computational thinking. *Commun. ACM*, 49(3):33–35, March 2006. ISSN 0001-0782. doi: 10.1145/1118178.1118215. URL http://doi.acm.org/10.1145/1118178.1118215.

ACKNOWLEDGMENTS

Thanks to Alexandre Rademaker for pointing out the similarity of this approach to the *root question* concept of Gradiance. Jonas Bernoulli and Alvarez Gonzales Sotillo made contributions to the Emacs Lisp `gift-mode` library[7] for editing GIFT-format quizzes, and Steve Purcell accepted a pull request for its inclusion in MELPA. Thanks are due also to colleagues and (particularly) students at Goldsmiths for making helpful suggestions, pointing out errors, and participating in the activities.

[7] https://github.com/csrhodes/gift-mode

Session V: Graphics & Modeling

Interactive flow simulation with Common Lisp

Nicolas Neuss
Friedrich-Alexander Universität (FAU) Erlangen-Nürnberg
Erlangen, Erlangen
neuss@math.fau.de

ACM Reference Format:
Nicolas Neuss. 2018. Interactive flow simulation with Common Lisp. In *Proceedings of the 11th European Lisp Symposium (ELS'18).* ACM, New York, NY, USA, 2 pages.
https://doi.org/

1 NAVIER-STOKES EQUATION

The Navier-Stokes equation (NSE) is a special partial differential equation (PDE) which is used for modeling fluid flows occurring in nature. There is a large variety of different flows, and correspondingly, there are a lot of variations of the NSE: compressible or incompressible flow, Newtonian or non-Newtonian flow, etc. For incompressible, Newtonian flows these equations have the form

$$\begin{aligned} \rho\frac{\partial\vec{v}}{\partial t} + \rho(\vec{v}\cdot\nabla)\vec{v} - \mu\Delta\vec{v} + \nabla p &= \vec{f} && (1)\\ \nabla\cdot\vec{v} &= 0\,. && (2)\end{aligned}$$

The parameter μ is the so-called dynamic viscosity of the fluid.

Although this equation is successfully used for modeling flows in many situations, the mathematical theory is only complete for two space dimensions and ensures that —given suitable initial conditions— a unique solution exists for all times. In contrast, the same result for three space dimensions (i.e. in relevant cases) is known only for small Reynolds numbers

$$\mathrm{Re} = \frac{U\cdot D}{\mu}$$

where U is a characteristic flow velocity and D a characteristic length of the specific situation, and μ denotes the viscosity of the fluid from above. From the formula, we can see that problems with small Reynolds number, say $\mathrm{Re} \lesssim 100$, characterize situations with slow flows, small geometries, and/or large viscosities (honey instead of water or air).

Unfortunately, for most interesting flow problems we have $\mathrm{Re} = 10^5 \ldots 10^8$, and the theoretical result ensuring existence of a unique solution does not apply. Indeed, flows with large Reynolds numbers feature regions of vortices of many scales (so-called *turbulence*) which cannot and/or should not be simulated in detail. The remedy is to introduce additional models of turbulence, and the construction of suitable models is still under current research.

2 NUMERICAL APPROXIMATION

The first step of solving a partial differential equation like the NSE is *discretization*, which results in a *discrete* problem whose solution is an approximation to the PDE solution. The Finite Element Method (FEM) is a particularly successful method for PDE discretization, because of its flexibility and its relatively sound mathematical foundation. It is based on

1. choosing a mesh consisting of so-called *cells* (triangles and/or quadrangles in 2D, tetrahedra, cubes etc in 3D),
2. constructing finite-dimensional function spaces on these meshes, and
3. formulating a discrete, finite-dimensional problem for replacing the infinite-dimensional original one.

Applied to a stationary (i.e. time-independent) flow problem, the result is a nonlinear equation of the form

$$\text{Find } u \in \mathbb{R}^N \text{ such that } F(u) = 0\,. \tag{3}$$

Usually, this equation is then solved by some variant of Newton's method, and leads to some vector $u \in \mathbb{R}^N$ which can be interpreted as a FE function (hopefully) approximating the solution of the continuous problem.

One should keep in mind, that the whole process is non-trivial and that it inherits the difficulties of complicated flow situations. Especially,

1. The number N, which is necessary for having a good approximation of u to the solution $\vec{v}$ of the continuous problem, may be very large. (Note that $N = O(h^{-d})$ where h is the mesh size and d the space dimension, which means that a resolution of 100 mesh points in each dimension already leads to $N = O(10^6)$ in 3D.)
2. Newton's method for (3) may not converge.
3. A basic step inside Newton's method for systems of N nonlinear equations is solving a system of N linear equations. Simple linear solvers require something in between $O(N^2)$ and $O(N^3)$ operations for solving such systems, which rapidly becomes unacceptable, and although more sophisticated solvers can perform much better, they are also more sensitive to complexities inherited from complicated flows.

3 FLOW AROUND AN AIRFOIL

A well-known model CFD problem is the one of computing the flow around an obstacle, for example around the wing of an aircraft. Here, interesting results can be obtained already using a two-dimensional calculation which essentially simulates the flow in the cross-section of an infinitely long wing.

Since in this contribution we were not interested in finding the optimal airfoil model, we have used the stationary NSE with a rather large viscosity $\tilde{\mu}$ for the flow region which ensures laminar

ELS'18, April 16–17 2018, Marbella, Spain
© 2018 Association for Computing Machinery.
ACM ISBN 978-2-9557474-2-1…$15.00

flow, but with a small friction along the boundary corresponding to the real size of viscosity μ. That is, we use

$$\begin{aligned} -\tilde{\mu}\Delta\vec{v} + \rho(\vec{v}\cdot\nabla)\vec{v} + \nabla p &= \vec{f} && (4) \\ \nabla\cdot\vec{v} &= 0\,. && (5) \end{aligned}$$

These equations are posed in a domain Ω of the form $\Omega = Q \setminus A$ where Q is a rectangular channel and $A \subset Q$ is the airfoil region. On the exterior boundary ∂Q, a constant velocity is prescribed and on the interior boundary ∂A, we require that the normal component of the flow as well as the normal component of the *stress tensor* T vanishes, i.e.

$$\vec{v}\cdot\vec{n} = 0\,, \quad T_{nn} = 0 \text{ where } T := \mu(\nabla\vec{v} + (\nabla\vec{v})^T) + p\mathbb{1}\,. \qquad (6)$$

The force which the flow exerts on the airfoil can be calculated as the curve integral

$$F = \int_{\partial A} T\cdot\vec{n}\,ds\,. \qquad (7)$$

4 INTERACTIVE SIMULATION

In 2017, our university organized an event for the general public called the "Long Night of the Sciences", where we took part with a web application that allowed drawing an airfoil which was then evaluated and ranked according to the largest ratio of lift/drag. This application was written in Common Lisp (CL) and based on the CL libraries Femlisp[2], cl-who and Hunchentoot. In the following we describe the setup in more detail:

- Accessing `index.html`, the web server provides the client with a modified and extended version of the Javascript application Paper.js [1] which allows the drawing of an airfoil as a curve.
- When this curve has been drawn, the Javascript application sends a request `calculate-drag-lift` back to the web server, which contains a name (who has drawn the airfoil) together with the curve and initiates the following calculation on the server:
  - The curve is scaled and fitted into a rectangular channel as an additional boundary. The resulting region is triangulated and discretized with finite elements. Finally, the discretized problem is solved with the help of Newton's method.
  - The force on the airfoil is calculated using (7). A score for the airfoil is calculated as the quotient vertical component (lift) divided by horizontal component (drag).
  - These numbers as well as pictures of the mesh and the flow (see Fig. 1 for an example) are written into a data directory. An entry for the calculation is written in a hash-table mapping the name to the score of the calculation.
- The server also provides a status page `show-scores` where all requests together with their score are shown in latest-first order, together with a top-10 list where the best ranked calculations are shown. The details of each calculation (see Fig. 1) can also be retrieved by clicking at some entry on this page.

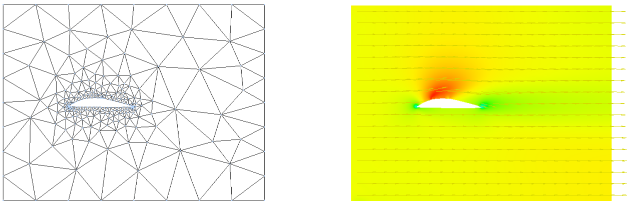


Figure 1: Mesh and velocity/pressure.

5 FUTURE WORK

Although simple, this application proved to be quite successful and a lot of people played around with it. Therefore, we probably will repeat this performance at another "Long Night of the Sciences". Nevertheless, several things could (and should) be improved in the next future. First, the most important improvement is ensuring that the underlying model equation really gives adequate results. Next, each calculation of our simulation took about 40 seconds last time. Although this was acceptable, because several of those calculations could run in parallel, there are quite a few ways of accelerating them, and even an instantaneous feedback might be achievable which would lead to a really satisfactory experience for the user.

6 ACKNOWLEDGMENTS

I would like to thank especially my colleague Florian Sonner from the chair of Applied Mathematics 3 (FAU) for providing the user interface by adapting [1] to our needs.

REFERENCES

[1] J. Lehni and J. Puckey. `http://paperjs.org`.
[2] N. Neuss. `http://www.femlisp.org`.

Object Oriented Shader Composition Using CLOS

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

ABSTRACT

We present a new approach to applying object oriented principles to GPU programming via GLSL shaders. Using the CLOS Meta Object Protocol, shader code can be attached to classes within standard Lisp code and is automatically combined by the system according to inheritance rules. This approach results in a native coupling of both CPU and GPU logic, while preserving the advantages of object oriented inheritance and behaviour. The system allows the use of arbitrary shaders written directly in GLSL, without the need for language extensions.

CCS CONCEPTS

• **Software and its engineering → Object oriented architectures**; *Abstraction, modeling and modularity*; *Object oriented frameworks*; Parsers; • **Computing methodologies →** *Computer graphics*;

KEYWORDS

Common Lisp, GLSL, OpenGL, GPU, CLOS, MOP, Object Orientation

ACM Reference Format:
Nicolas Hafner. 2018. Object Oriented Shader Composition Using CLOS. In *Proceedings of the 11th European Lisp Symposium (ELS'18).* ACM, New York, NY, USA, 4 pages.

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1.

1 INTRODUCTION

In modern real-time computer graphics applications, advanced effects and GPU computations require the use of a programmable graphics pipeline. This is usually done using domain-specific languages. The programmer passes the programs as textual source code to the graphics driver, which then compiles it down to GPU instructions.

For OpenGL, this language is called GLSL[1] and follows a mostly C-like syntax. A program written in GLSL is called a shader and is used to fill one of several stages in the graphics pipeline. Whenever a primitive is rendered, these shaders are executed on the GPU, each providing a particular stage of processing until finally an image is produced.

However, only a single shader can live in a particular stage at a time. This limitation presents an issue for modularity, as effects represented by shaders cannot be easily combined. Instead, it is usually the task of a programmer to craft a single shader for each stage that produces the desired results.

In this paper we present a new solution to this problem that is accomplished in two steps. The first step is the parsing and manipulation of native GLSL code. The second step is the integration of shader code into the Common Lisp Object System to allow for automatic combination through inheritance.

2 RELATED WORK

Several different approaches to shader combination exist today.

Trapp et al[2] use additional constructs introduced to GLSL and a preprocessor to combine effects. Their approach differs from ours in that we do not extend GLSL syntax in any way and do not present a standard interface to use between shader fragments.

McCool et al.[3] present an extension of the C++ language to allow writing GLSL-like code in C++ source. Shader fragments are parsed into an abstract syntax that can be used to perform static analysis and combination of fragments. Combination is however not automatic and needs to be explicitly requested by the programmer.

Kuck[4] presents an evolution of McCool's approach by allowing the use of C++ method declarations for shader function definitions. In this approach method dispatch is mirrored in the emitted GLSL code using unique identifiers for each class. This system is thus intended as a complete carry-over into shader code, rather than a simple combination of behaviour.

In VTK[5] GLSL's `ifdef` preprocessor macros are used to conditionally exclude or include certain shader functionality. This use of preprocessor macros means that a full shader program of all possible combinations is meticulously hand-crafted, and features are then added or removed as needed by the program by prepending appropriate preprocessor definitions to the shader source.

Khronos proposed a GLSL extension[6] for an `include` preprocessor directive that would splice other shader source files in at the requested position, similar to C's include facility. This extension would merely allow controlling concatenation of source text within GLSL itself, though.

3 GLSL PARSING AND MANIPULATION

Typically a GLSL program will have form similar to this example:

```
uniform mat4 transform_matrix;
layout (location = 0) in vec3 position;
out vec4 vertex;

void main(){
  vertex = transform_matrix * vec4(position, 1.0);
}
```

Listing 1: A small example of a vertex shader computing the vertex position based on a transform matrix.

More specifically, it consists of a set of variable declarations that are either `uniform` (exchangeable with the CPU), `in` (coming from the previous stage), or `out` (going out to the next stage), and a set of function definitions. The `main` function presents the entry point for the shader and is required.

One goal of our approach was to allow the user to keep on using existing GLSL shaders, ideally without having to change anything about them. In order to accomplish merging of shader fragments with these constraints, the system needs to be able to automatically rewrite shader code to resolve name conflicts and to combine effects.

In order to do this, we implemented a full parser for the GLSL language in Lisp that turns the textual representation into an AST. Based on this AST, code walking and semantic analysis can be performed to detect definitions and to rewrite behaviour. By analysing two code segments, matching variable declarations can be fused together and their references in the code rewritten as appropriate. The main functions can be rewritten and called sequentially in a newly emitted main function.

For instance, combining the shaders of listing 1 and listing 2 results in listing 3.

```
layout (location = 0) in vec3 vertex_data;
out vec4 vertex;

void main(){
  vertex.x += sin(vertex_data.y);
}
```

Listing 2: A fragment shader that warps the vertex position.

```
uniform mat4 transform_matrix;
layout (location = 0) in vec3 position;
out vec4 vertex;

void _GLSLTK_main_1(){
  vertex = transform_matrix * vec4(position, 1.0);
}

void _GLSLTK_main_2(){
  vertex.x += sin(position.y);
}

void main(){
  _GLSLTK_main_1();
  _GLSLTK_main_2();
}
```

Listing 3: A combination of the shaders from listing 1 and listing 2.

The system automatically recognises that the `vertex` variable is the same in both shaders and omits the second instance. It also recognises that the `position` and `vertex_data` variables denote the same input, omits the second declaration, and renames the variable references to match the first declaration. An example rendering using listing 1 and listing 3 can be seen in figure 1.

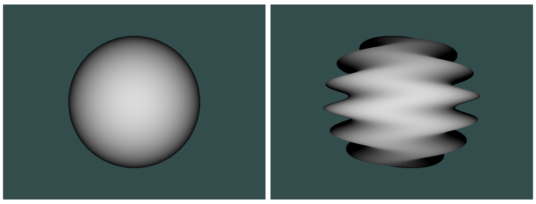


Figure 1: Left: rendering of a sphere using listing 1. Right: the same with listing 2 added.

Using this technique, a wide variety of shaders can be combined, with a minimal amount of awareness of other shaders being necessary. Shortcomings of this technique are elaborated in section 6.

4 SHADER INTEGRATION WITH CLOS

Usually the shaders do not act on their own. They need corresponding CPU-side code in order to provide the inputs and parameters for the pipeline to execute properly. Thus our system couples the definition of relevant shader code and CPU code together in a single class. Combination of features is then automatically provided through the inheritance of classes.

We implement a new metaclass that includes a slot for a set of shader fragments, each grouped according to their pipeline stage. Shader fragments can then be attached to an instance of this metaclass to form the direct shader fragments. When the class hierarchy is finalised, shader fragments from all transitive superclasses are gathered in order of class precedence to form the list of effective shader fragments. This list is then combined into a full shader for each stage using the technique from section 3.

Accompanied by this metaclass is a set of generic functions that allow specifying the loading and drawing logic of a class. These functions encompass the CPU-side code required to run the shaders properly. Using the standard method combination, the behaviour can be passed down and combined alongside the shader code. An example of such a behaviour can be seen in listing 4.

```
(defclass colored-entity ()
  ((color :initform (vec 0 0 1 1) :reader color))
  (:metaclass shader-class))

(defmethod paint :before ((obj colored-entity))
  (let ((shader (shader-program obj)))
    (setf (uniform shader "objectcolor") (color obj))))

(define-class-shader (colored-entity :fragment-shader)
  "uniform vec4 objectcolor;
out vec4 color;

void main(){
  color *= objectcolor;
}")
```

Listing 4: A `colored-entity` class that encompasses object colouring functionality.

This `colored-entity` class can now be used as a superclass in order to inherit the full behaviour. In effect this approach allows us to encapsulate a variety of small behaviours in the form of mixins, which can then be combined to form a full implementation of an object to be drawn. For instance, we could imagine an entity to emit geometry for a sphere, an entity to apply a wave deformation, an entity to texture the geometry, an entity to apply shading, and so forth.

5 APPLICATIONS

We currently use this technique for two primary applications: the encapsulation of common display functionality into mixins, and the implementation of effects passes. The former allows us to separate various concerns such as the read-out of texture data, the transformation of vertex data, and the fragment rendering behaviour into small, self-contained classes, that can then be combined by the user to produce the result they desire.

The latter allows us to write arbitrary rendering effects as self-contained render pass objects. To elaborate on this application, imagine a visual effect that requires you to manipulate some, but not all of the aspects of how an object is rendered. A simple example for such an effect is the “god rays” that you can see in figure 2.

In order to achieve this effect, all opaque objects in the scene are rendered once normally, and once in black. Then the black pass is blurred and added to the normal render. Neither the normal render pass nor the combination step require any insight from the pass into how the objects are drawn. However, for the black rendering pass, the way the objects are drawn has to be modified in such a way that all fragments are output as black, but everything else, such as vertex transformations, must remain the same.

With shader combination we can achieve this quite easily: we write a fragment shader that simply outputs the colour black, and then combine this shader with whatever the objects' shaders are. In this way we are virtually inserting a superclass for each object in the scene depending on which render pass is currently active. Render passes themselves are implemented with the same class shader combination as regular objects, so they too can profit from encapsulating and sharing shader functionality.

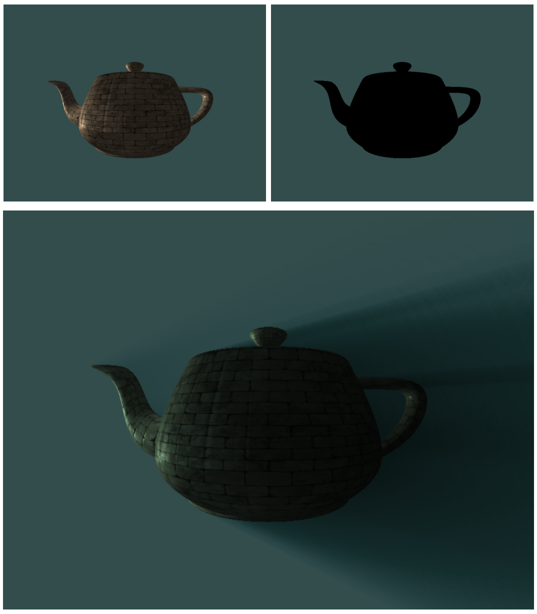


Figure 2: Left: phong rendered teapot. Right: black rendered teapot. Bottom: combined god-rays effect.

6 CONCLUSION

Source code analysis allows us to easily re-use shader code written independently of our Lisp program while retaining the ability to merge the effects of multiple shaders together. Integrating this analysis into the object system combines the GPU and CPU logic in a single class, allowing the inheritance and re-use of both.

While the merging works automatically for a large class of shaders, certain ambiguities cannot be automatically rectified and require user input. For instance, unless the names of in or out declarations match, or their location is explicitly specified, the static analysis cannot determine whether they denote the same thing. The shader effects that write to the same out variable must also be adapted to re-use the existing value in order for the effects to combine. For instance, a colour should be combined by multiplying with the previous value, rather than directly setting the new one. Typically multiplying the colour will yield the desired combination effect, whereas setting it would simply discard the colour computed by other mixins. Finally, if the declarations differ in qualifiers such as type or allocation, the merger cannot automatically determine how to resolve this conflict, even if the authors of the shaders meant it to denote the same conceptual variable.

7 FURTHER WORK

The current merging strategy employed is rather simplistic. For instance, no attempt at recognising identical function definitions is made. The system could also be extended with a variety of static analysis algorithms to optimise the code ahead of time, or to provide errors and warnings about potential user mistakes.

While allowing the use of native GLSL code is handy, providing the user with a system to write shaders in Lisp syntax would improve the system quite a bit. To facilitate this change, the Varjo[7] compiler could be integrated. A further development from there could be integration with an IDE to provide the user with automated help information about not just Lisp, but also shader functions.

8 ACKNOWLEDGEMENTS

I would like to thank Robert Strandh and Philipp Marek for feedback on the paper.

9 IMPLEMENTATION

An implementation of the proposed system can be found at https://github.com/Shirakumo/trial/blob/e37e0dc73085c401e43da58d5098a9cf02167f8f/shader-entity.lisp for the CLOS part, and at https://github.com/Shirakumo/glsl-toolkit for the merger part.

A more in-depth discussion of the system can be found at https://reader.tymoon.eu/article/362.

REFERENCES

[1] Randi J Rost, Bill Licea-Kane, Dan Ginsburg, John Kessenich, Barthold Lichtenbelt, Hugh Malan, and Mike Weiblen. *OpenGL shading language.* Pearson Education, 2009.

[2] Matthias Trapp and Jürgen Döllner. Automated combination of real-time shader programs. In *Eurographics (Short Papers)*, pages 53–56, 2007.

[3] Michael D McCool, Zheng Qin, and Tiberiu S Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 57–68. Eurographics Association, 2002.

[4] Roland Kuck. Object-oriented shader design. In *Eurographics (Short Papers)*, pages 65–68, 2007.

[5] Inc. Kitware. Visualization toolkit. https://gitlab.kitware.com/vtk/vtk. [Online; accessed 2018.2.21].

[6] Jon Leech. Gl arb shading language include. https://tinyurl.com/y7er7d6d, 2013. [Online; accessed 2018.2.21].

[7] Chris Bagley. Varjo, a lisp to glsl compiler. https://github.com/cbaggers/varjo, 2016. [Online; accessed 2018.2.21].

Context-Oriented Algorithmic Design

Bruno Ferreira
Instituto Superior Técnico/INESC-ID
Lisbon, Portugal
bruno.b.ferreira@tecnico.ulisboa.pt

António Menezes Leitão
Instituto Superior Técnico/INESC-ID
Lisbon, Portugal
antonio.menezes.leitao@tecnico.ulisboa.pt

ABSTRACT

Currently, algorithmic approaches are being introduced in several areas of expertise, namely Architecture. Algorithmic Design (AD) is an approach for architecture that allows architects to take advantage of algorithms to produce complex forms for their projects, to simplify the exploration of variations, or to mechanize tasks, including those related to analysis and optimization of designs. However, architects might need different models of the same project for different kinds of analysis, which tempts them to extend the same code base for different purposes, typically making the code brittle and hard to understand. In this paper, we propose to extend AD with Context-Oriented Programming (COP), a programming paradigm based on context that dynamically changes the behavior of the code. To this end, we propose a COP library and we explore its combination with an AD tool. Finally, we implement a case study with our approach, and discuss the advantages and disadvantages.

CCS CONCEPTS

• **Software and its engineering** → *Object oriented languages*;

KEYWORDS

Context-Oriented Programming, Algorithmic Design, Racket, Generative Design

1 INTRODUCTION

Nowadays, Computer Science is being introduced in several areas of expertise, leading to new approaches in areas such as Architecture. Algorithmic Design (AD) is one of such approaches, and can be defined as the production of Computer-Aided Design (CAD) and Building Information Modeling (BIM) models through algorithms [9, 25]. This approach can be used to produce complex models of buildings that could not be created with traditional means, and its parametric nature allows an easier exploration of variations.

Due to these advantages, AD started to be introduced in CAD and BIM applications, which led to the development of tools that support AD programs, such as Grasshopper [26]. However, with the complexity of the models came the necessity of analyzing the produced solutions with analysis tools. For this task, the geometrical models are no longer sufficient, as analysis software usually requires special analytical models, that are different from geometrical models and can hardly be obtained with import/export mechanisms due to errors. These requirements lead to the production of several models, which have to be kept and developed in parallel, involving different development lines that are hard to manage and to keep synchronized. This complex workflow proves that current solutions are not sufficient [36].

Some tools like Rosetta [24] are already trying to address these issues by offering portable AD programs between different CAD applications and, more recently, BIM applications [7, 8] and analysis tools [22]. Nevertheless, this tool does not offer a unifying description capable of producing both the geometrical and the analytical models with the same AD program, which can lead to the cluttering of the current program in an effort to reduce the number of files to maintain.

To solve these problems, we propose the use of COP to develop a computational model capable of adapting itself to the required context, which in this case is defined by the requirements of modeling applications and analysis tools, allowing the production of different models with a change of context.

1.1 Context-Oriented Programming

COP was first introduced as a new programming approach that takes the context into account [10]. According to a more recent depiction of this approach, COP aims to give users ways to deal with context in their programs, making it accessible to manipulation with features that are usually unavailable in mainstream programming languages [16].

With this approach, users can express different behaviors in terms of the context of the system. The context is composed of the actors of the system, which can determine how the system is used, the environment of the system, which can restrict or influence its functionality, and the system itself, whose changes might lead to different responses.

Although there are different implementations of COP, which will be presented later in this paper, according to [16] necessary properties must be addressed by all of them. These are:

- behavioral variation: implementations of behavior for each context;
- layers: a way to group related context-dependent variations;
- activation and deactivation of layers: a way to dynamically enable or disable layers, based on the current context;
- context: information that is accessible to the program and can be used to determine behavioral variations;
- scoping: the scope in which layers are active or inactive and that can be controlled.

With these features, layers can be activated or deactivated dynamically in arbitrary places of the code, resulting in behaviors that fit the different contexts the program goes through during its execution. If analyzed in terms of multi-dimensional dispatch

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1.

[34], it is possible to say that COP has four-dimensional dispatch, since it considers the message, the receiver, the sender, and the context to determine which methods or partial method definitions are included or excluded from message dispatch.

These method definitions are used to implement behavioral variations in layers, which can be expressed differently in the several implementations of COP. In some, the adopted approach is known as *class-in-layer*, in which layers are defined outside the lexical scope of modules [2], in a manner similar to aspects from Aspect-Oriented Programming (AOP) [20]. In others, a *layer-in-class* approach is used, having the layer declarations within the lexical scope of the modules.

Although these concepts are used to define COP, each implementation of the paradigm might include additional features and concepts, depending on the programming language. Some of these differences will be discussed later in the paper.

1.2 Objectives

The main objectives of this paper are: (1) present and compare the different implementations for COP that have been proposed by the research community, and (2) present a simple case study that shows how COP can be applied to AD. The case study consists of a previously developed AD program that we re-implemented with our proposed solution and then used to produce different models according to different contexts. Finally, the results of our solution are compared to the ones obtained in the previous version of the code.

2 RELATED WORK

In this section, we introduce several paradigms that served as basis for COP, namely AOP, Feature-Oriented Programming (FOP), and Subject-Oriented Programming (SOP), as well as an overview of the several implementations of COP that have been proposed throughout the years.

2.1 Aspect-Oriented Programming

Most programming paradigms, such as Object-Oriented Programming (OOP), offer some way to modularize the concerns necessary to implement a complex system. However, it is common to encounter some concerns that do not fit the overall decomposition of the system, being scattered across several modules. These concerns are known as *crosscutting concerns*.

AOP was created to deal with these *crosscutting concerns*, introducing ways to specify them in a modularized manner, called *aspects*. *Aspects* can be implemented with proper isolation and composition, making the code easier to maintain and reuse [20]. Using AOP, it is possible to coordinate the *crosscutting concerns* with normal concerns, in well-defined points of the program, known as *join points*.

In addition to these concepts, AOP implementations, such as AspectJ, introduce more concepts, such as *pointcuts*, which are collections of *join points* and values at those *join points*, and *advice*, which are method-like implementations of behavior attached to *pointcuts* [21]. An *aspect* is then a module that implements a *crosscutting concern*, comprised of *pointcuts* and *advice*.

AspectJ also offers *cflow* constructs, that allow the expression of control-flow-dependent behavior variations, making it possible to conditionally compose behavior with *if pointcuts* [21]. If we compare this with the basic description of COP, we can see several similarities, making it possible to define behavior that depends on a given condition, which can, in itself, be considered a context. However, while AOP aims to modularize *crosscutting concerns*, this is not mandatory in COP, since the use of *layer-in-class* approaches scatters the code across several modules.

In addition, COP allows the activation and deactivation of layers in arbitrary pieces of code, while AOP triggers *pointcuts* at very specific *join points* that occur in the rest of the program [16]. This makes COP more flexible in dealing with behavioral variation.

2.2 Feature-Oriented Programming

Feature-Oriented Programming (FOP) is an approach that was originally proposed to generalize the inheritance mechanisms offered by OOP. With FOP, users create their objects by composing *features*, instead of defining traditional classes. This process is called *class refinement* [27].

The *features* introduced in FOP work in a very similar way to *mixins*, a mechanism that allows the specification of methods to be used by other classes [33]. *Mixin layers* can also be defined, consisting of a *mixin* that encapsulates another *mixin*, and whose parameters encapsulate all the parameters of the inner *mixin*.

With *mixins*, users can increment existing classes, which is similar to what *features* allow. However, with *features*, overriding is not used to define the functionality of a subclass, but it is used as a mechanism to resolve dependencies between features [27]. *Lifters* are also used for this purpose. They are functions that lift one *feature* to the context of another. Although this feature is similar to method overriding used in inheritance, *lifters* depend on two *features* and are implemented as a separate entity.

By combining *features*, it is possible to create objects with a very specific functionality based in already existing features, something that would have to be done with inheritance in traditional OOP. This combination provides higher modularity, flexibility, and reusability, since each *feature* is an entity in itself.

FOP also shares the idea of supporting behavioral variations of the original program with the composition mechanism, which is similar to how layers in COP work. However, while COP offers the possibility of using these variations dynamically, through the activation and deactivation of layers, FOP focuses on selecting and combining behavior at compile-time [29].

2.3 Subject-Oriented Programming

SOP is a programming paradigm that introduces the concept of *subject* to facilitate the development of cooperative applications. In this paradigm, applications are defined by the combination of *subjects*, which describe the state and behaviors of objects that are relevant to that *subject* [13].

The *Subjects*' goal is to introduce a perception of an object, such as it is seen by a given application. *Subjects* do so by adding classes, state, and behavior, according to the needs of that application. By doing this, each application can use a shared object through the operations defined for its *subject*, not needing to know the details of the object described by other *subjects* [13].

Subjects can also be combined in groups called *compositions*, which define a composition rule, explaining how classes and methods from different *subjects* can be combined. These subjects and compositions can then be used through *subject-activation*, which provides an executable instance of the *subject*, including all the data that can be manipulated by it.

Due to the introduction of all these concepts, SOP offers what is known as *Subjective Dispatch* [34]. *Subjective Dispatch* extends the dispatch introduced by OOP, by adding the sender dimension, in addition to the message and the receiver. This type of dispatch was later expanded by COP, which introduces a dimension for the context, as mentioned previously.

It is possible to see that, similarly to the other analyzed paradigms, SOP also supports behavior variations in the form of *subjects*. However, if we consider that each *subject* might have different contexts of execution, we need an extra dimension for dispatch, which is what COP offers.

2.4 Context-Oriented Programming Implementations

COP was proposed as an approach that allows the user to explore behavioral variations based on context. Concepts such as layers and contexts are present in all implementations of this approach. Nevertheless, each one can address the concepts differently, sometimes due to the support of the host language in which the COP constructs are implemented. In this section we present the different implementations available.

2.4.1 ContextL. ContextL was one of the first programming language extensions to introduce support for COP. It implements the features discussed previously by taking advantage of the Common-Lisp Object System (CLOS) [5].

The first feature to be considered is the implementation of layers, which are essential to implement the remaining features available in ContextL [6]. These layers can be activated dynamically throughout the code, since ContextL uses an approach called Dynamically Scoped Activation (DSA), where layers are explicitly activated and a part of the program is executed under that activation. The layer is active while the contained code is executing, becoming inactive when the control flow returns from the layer activation.

Regarding the activation of multiple layers, it is important to note that the approach introduced in ContextL, as well as in other implementations that support DSA, follows a stack-like discipline. Also, in ContextL this activation only affects the current thread.

By taking advantage of layers, it is then possible to define classes in specific layers, so that the classes can have several degrees of detail in different layers, introducing behavior that will only be executed when specific layers are activated. The class behavior can also be defined with *layered generic functions*. These functions take advantage of the generic functions from CLOS, and are instances of a generic function class named *layered-function* [6].

In addition, ContextL supports contextual variations in the definition of class slots as well. Slots can be declared as *:layered*, which makes the slot accessible through layered-functions. This feature introduces slots that are only relevant in specific contexts.

By looking at the constructs implemented in ContextL, it is possible to conclude that behavioral variations can be implemented in specific classes or outside of them. This means that ContextL supports both *layer-in-class* and *class-in-layer* approaches. The former allows the definition of partial methods to access private elements of enclosing classes, something that the latter does not support, since *class-in-layer* specifications cannot break encapsulation [2].

Finally, it should be noted that ContextL follows a library implementation strategy: it does not implement a source-to-source compiler, and all the constructs that support the COP features are integrated in CLOS by using the Metaobject Protocol [19].

2.4.2 PyContext and ContextPy. PyContext was the first implementation of COP for the Python programming language. Although it includes most of the COP constructs in a similar manner to the other implementations, PyContext introduces new mechanisms for layer activation, as well as to deal with variables.

Explicit layer activation is an appropriate mechanism for several problems but, sometimes, this activation might violate modularity. Since the behavioral variation may occur due to a state change that can happen at any time during the program execution, the user needs to insert verifications in several parts of the program, increasing the amount of scattered code. To deal with this problem, PyContext introduces *implicit layer activation*. Each layer has a method *active*, which determines if a layer is active or not. This method, in combination with a function *layers.register_implicit*, allows the framework to determine which layers are active during a method call, in order to produce the correct method combination [37].

Regarding variables, PyContext offers contextual variables, which can be used with a *with* statement in order to maintain their value in the dynamic scope of the construct. These variables are called *dynamic variables*. These variables are globally accessible, and their value is dynamically determined when entering the scope of a *with* construct [37]. In conjunction with specific getters and setters, it is possible to get the value of the variable, change it in a specific context, and then have it restored when exiting the scope of that context. It is important to note that this feature is thread-local.

As for the other features, PyContext does not modify the Python Virtual Machine, being implemented as a library. Layers are implemented using meta-programming, and layer activation mechanisms take advantage of Python's context handlers. As for the partial definition of methods and classes, PyContext follows a *class-in-layer* approach.

More recently, ContextPy was developed as another implementation of COP for the Python language. This implementation follows a more traditional approach to the COP features, offering DSA, using the *with* statement, which follows a stack-like approach for method composition. For partial definitions, ContextPy follows a *layer-in-class* approach, taking advantage of decorators to annotate base methods, as well as the definitions that replace those methods when a specific layer is active [17]. Finally, similarly to PyContext, ContextPy is offered as a library that can be easily included in a Python project.

2.4.3 ContextJ. ContextJ is an implementation of COP for the Java programming language, and one of the first implementations of this approach for statically typed programming languages. Before this implementation, there were two proof of concepts implemented in Java, namely ContextJ* [16] and ContextLogicAJ [1]. The first is a library-based implementation that does not offer all COP functionalities, while the second is an aspect-oriented pre-compiler that improves the features given by ContextJ* and offers new mechanisms, such as indefinite layer activation. Indefinite activation requires the user to explicitly activate and deactivate layers, in order to obtain the desired layer composition.

ContextJ is a sorce-to-source compiler solution that introduces all the concepts of COP in Java by extending the language with the *layer*, *with*, *without*, *proceed*, *before* and *after* terminal symbols [4]. Layers are included in the language as a non-instantiable type, and their definitions follows a *layer-in-class* approach. Each layer is composed of an identifier and a list of partial method definitions, whose signature must correspond to one of the methods of the class that defines the layer. Also, to use the defined layers, users must include a layer import declaration on their program, in order to make the layer type visible.

As for partial method definitions, they override the default method definition and can be combined, depending on the active layers. The *before* and *after* modifiers can also be used in partial method definitions, in order to include behavior that must be executed before and after the method execution. In addition, the *proceed* method can be used to execute the next partial definition that corresponds to the next active layer, allowing the combination of behavioral variations [4].

Regarding layer activation, ContextJ supports DSA by using a *with* block. Layers are only active during the scope of the block, and the activation is thread-local. *With* blocks can be nested, and the active layer list is traversed according to a stack approach. This approach, in combination with the *proceed* function, allows the user to compose complex behavior variations. In addition, it is possible to use the *without* block to deactivate a layer during its scope, in order to obtain a composition without the partial method definitions of that specific layer.

Finally, ContextJ also offers a reflection Application Programming Interface (API) for COP constructs. It includes classes for *Layer*, *Composition*, and *PartialMethod*, along with methods that support runtime inspection and manipulation of these concepts.

2.4.4 Other COP Implementations. The implementations described in the previous sections present some of the major strategies and features that are currently used with COP. Nevertheless, there are more implementations for other languages, which we briefly describe in this section.

Besides ContextJ, ContextJ*, and ContextLogicAJ, there are other implementations of COP for Java, namely: JCop [3] and EventCJ [18], which use join-point events to switch layers; cj [31], a subset of ContextJ that runs on an ad hoc Java virtual machine; and JavaCtx [28], a library that introduces COP semantics by weaving aspects.

There are also COP implementations for languages such as Ruby, Lua, and Smalltalk, namely ContextR [32], ContextLua [38], and ContextS [15] respectively. ContextR introduces reflection mechanisms to query layers, while ContextLua was conceived to introduce COP in games. ContextS follows the more traditional COP implementations, such as ContextL. ContextScheme[1], for the Scheme programming language, also follows an implementation similar to ContextL.

In addition, some implementations, such as ContextErlang, introduce COP in different paradigms, like the actor model [14]. ContextErlang also introduces different ways to combine layers, namely per-agent variation activation and composition [30].

Regarding layer combination and activation, there are also implementations that offer new strategies that differ from dynamic activation. One example is ContextJS [23] that offers a solution based on open implementation, in which layer composition strategies are encapsulated in objects. These strategies can add new scooping mechanisms, disable layers, or introduce a new layer composition behavior that works better with a domain-specific problem [23].

More recently, Ambience [12], Subjective-C [11], and Lambic [35] were developed. Ambience uses the amOS language and context objects to implement behavioral variations, with the context dispatch made through multi-methods. Subjective-C introduces a Domain Specific Language (DSL) that supports the definition of constraints and the activation of behaviors for each context. Finally, Lambic is a COP implementation for Common Lisp that uses predicate dispatching to produce different behavioral variations.

In the next section we present a comparison between all these implementations, as well as the advantages and disadvantages of using each one.

2.5 Comparison

Table 1 shows a comparison between the analyzed COP implementations.

As it is possible to see, most of the analyzed implementations are libraries, with source-to-source compilers being mostly used in statically typed programming languages. The library implementation has advantages when trying to add COP in an already existing project, since it does not change the language and uses the available constructs. On the other hand, source-to-source compilers, such as ContextJ, introduce new syntax that simplifies the COP mechanics, as well as possible advantages regarding performance.

As for layer activation, the most common strategy is DSA. However, to increase flexibility, some solutions introduce indefinite activation, global activation, per agent activation or, in the case of ContextJS, an open implementation, allowing users to implement an activation mechanism that best fits the problem they are solving. Although DSA is appropriate for most problems, other strategies might be best suited for multi-threaded applications or problems whose contexts depend on conditions that cannot be captured with the default layer activation approach.

Finally, regarding modularization, it is possible to see that most implementations use the *class-in-layer* or the *layer-in-class* approach. The first one allows users to create modules with all the concerns regarding a specific context, while the latter places all the behaviors on the class affected by the contexts. Hence, *class-in-layer* reduces code scattering, while *layer-in-class* can simplify program comprehension. There are implementations that support both approaches, such as ContextL, but usually the supported approach is restricted by the features of the language. Nevertheless, there are cases, such as ContextPy and PyContext, that take advantage

[1] http://p-cos.net/context-scheme.html

Table 1: Comparison between the COP implementations. DSA stands for Dynamically Scoped Activation, LIC for *layer-in-class*, and CIL for *class-in-layer*. Lambic uses predicate dispatching instead of layers, so the last two columns do not apply. Adapted from [29]

	Base Language	Implementation	Layer Activation	Modularization
ContextL	Common Lisp	Library	DSA	LIC, CIL
ContextScheme	Scheme	Library	DSA	CIL
ContextErlang	Erlang	Library	Per-agent	Erlang Modules
ContextJS	JavaScript	Library	Open Implementation	LIC, CIL
PyContext	Python	Library	DSA, Implicit Layer Activation	CIL
ContextPy	Python	Library	DSA	LIC
ContextJ	Java	Source-to-Source Compiler	DSA	LIC
JCop	Java	Source-to-Source and Aspect Compiler	DSA, declarative layer composition, conditional composition	LIC
EventCJ	Java	Source-to-Source and Aspect Compiler	DSA	LIC
JavaCtx	Java	Library and Aspect Compiler	DSA	LIC
ContextR	Ruby	Library	DSA	LIC
ContextLua	Lua	Library	DSA	CIL
ContextS	Smalltalk	Library	DSA, indefinite activation	CIL
Ambience	AmOS	Library	DSA, global activation	CIL
Lambic	Common Lisp	Library	-	-
Subjective-C	Objective-C	Preprocessor	Global Activation	LIC

of the same programming language but follow different principles regarding the COP concepts.

All these implementation support the COP paradigm, although they offer different variations of the relevant concepts. Choosing the most appropriate implementation requires a careful examination of their distinct features, and how they help in fulfilling the requirements of the problem at hand.

3 CONTEXT-ORIENTED ALGORITHMIC DESIGN

In this section, we propose to combine COP with AD, introducing what we call Context-Oriented Algorithmic Design. Since it is common for architects to produce several different models for the same project, depending on the intended use (e.g., for analysis or rendering), we define these different purposes as contexts. By doing this, it is possible to explicitly say which type of model is going to be produced.

In addition, we introduce definitions for the design primitives using COP as well. For each primitive, we can define different behavioral variations, depending on the model we want to produce. For example, since some analysis models require surfaces instead of solids, a primitive definition of a *Wall* would produce a box in a 3D context and a simple surface in an analysis context.

Finally, since COP allows the combination of layers, we can take advantage of that to combine additional concepts, such as Level of Detail (LOD) with the remaining ones. This combination allows more flexibility while exploring variations, since it not only supports the exploration in several contexts, but also the variation of LOD inside the same context. This might be useful, e.g., for architects that want to have less detail in certain phases to obtain quicker results.

3.1 Implementation

To test our solution we created a working prototype, taking advantage of Khepri, an existing implementation of AD, and ContextScheme to introduce the COP concepts. Khepri is a portable AD tool, similar to Rosetta [24], that allows the generation of models in different modeling back-ends, such as AutoCAD or Revit, and offers a wide range of modeling primitives for the supported applications. This tool offers a native implementation in Racket, a pedagogical programming language with support for COP, making it easier to extend.

As for the choice of the COP implementation to use, we decided for an implementation of ContextScheme that we adapted for Racket. This implementation is a library, making it easier to include in existing projects and tools, such as Khepri. Also, since AD programs are usually single-threaded, and we can easily indicate the scope to be affected by the context, DSA is a good approach to solve the problem. Both features are supported by ContextScheme.

Having these two components, we implemented a new library, that extends Khepri, and introduces design elements with contextual awareness. Since the behavioral variations have to produce results on the selected modeling tool, this new layer uses Khepri's primitive functions to do so. The functions to use depend on the context. For instance, the program uses those that produce surfaces in analysis contexts, and those that produce solids on 3D contexts. For each new modeling element of our library, we define functions with basic behavior and a variation for each possible context. The available contexts are implemented as layers in the library as well.

Listing 1 shows a simplified definition of the `wall` function. This definition has a default behavior, and variations for a 3D, 2D, and analysis contexts, which are identified by the layers used as parameters. In each of these functions, Khepri modeling functions are used, in order to produce the results in the modeling tools.

Listing 1: Definition of 3D and 2D walls.

```
(define-layered wall)

(deflayered (wall 3D)
  (lambda (...)
   (box ...)))

(deflayered (wall 2D)
  (lambda (...)
    (rectangle ...)))

(deflayered (wall analysis)
  (lambda (...)
    (surface ...)))
```

In the next section, we introduce a case study, that was produced with this new library, in order to evaluate our approach.

4 CASE STUDY

For the evaluation of our COP library we used a model of a shopping mall, originally used for evacuation simulations. The model was produced with an algorithmic solution, which we modified to include our library. This is a simple case study that uses few geometrical elements, namely doors, walls, and a floor, which the new library already supports.

We chose this case study because the original implementation required a plan view of the model, which had to be produced in addition to the usual 3D view. To take advantage of the same algorithm, the original developers included two implementations of the shop function, which produce each of the shops that compose the mall, and are used by the rest of the algorithm. One implementation is a 2D version that created lines, and the other is a 3D version that created solids.

In order to switch between them, the authors commented a couple lines of code and changed some variables as well. This approach has several disadvantages, namely the need to modify lines of code when it is necessary to change the type of model, and having to comment and uncomment several lines of code when we want to change from 2D to 3D. Both of these tasks are error prone, since developers might forget to do some of them.

Listing 2 shows a simplified definition of both 2D and 3D versions of the shop function, as well as the commented line of code correspondent to the 2D version, having the 3D version activated in the next line. The 2D version uses functions that generate 2D shapes, such as rectangles and lines, and the 3D version uses functions that generate solids.

Listing 2: Original version.

```
(define (shop-2d ...)
  (...
    (rectangle ...)
  ...)
)

(define (shop-3d ...)
  (...
    (right-cuboid ...)
  ...)
)

#;(define shop shop-2d)
(define shop shop-3d)
```

Our COP-based solution eliminates the aforementioned problems. By adding our library, we re-implemented parts of the algorithm, namely the shop function. Since we have different implementations for the elements, such as walls, available in different contexts, we do not require two versions of the same function, and can implement just one. Listing 3 shows a simplified definition of the COP version of the shop function, using the `wall` and `door` functions of our library.

Listing 3: COP implementation of the shop function.

```
(define (shop ...)
  (...
    ((wall) ...)
    ...
    ((door) ((wall) ...)
      ...)
  ...)
)
```

To switch between contexts, we eliminated the commented lines of code and introduced a `with-layers` construct, which receives the layer corresponding to the model we want to produce, and the expression that generates the entire shopping mall. For example, the `with-layers` can receive a layer corresponding to the 3D view and have a call to the `mall` function in its scope.

Since we wanted to produce a plan view and a 3D model, and those correspond to layers that we support in the library, we could generate both of them by introducing `3D` or `2D` as arguments of the `with-layers` construct. The results can be seen in figures 1 and 2.

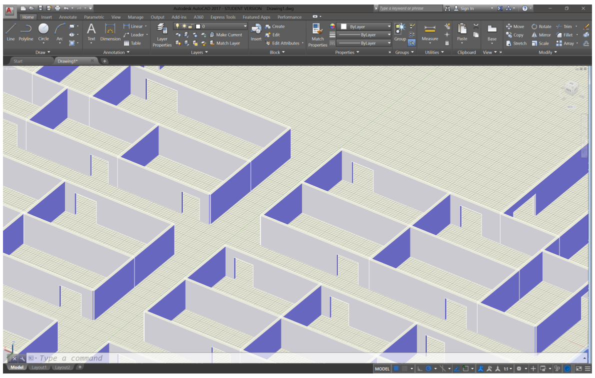


Figure 1: 3D model of the shopping mall, produced with COP in AutoCAD.

In addition, since we support a layer that produces only surfaces for analysis purposes, namely radiation analysis, we were able to produce another model simply by changing the context. By using `analysis` as argument for the `with-layers` construct, we

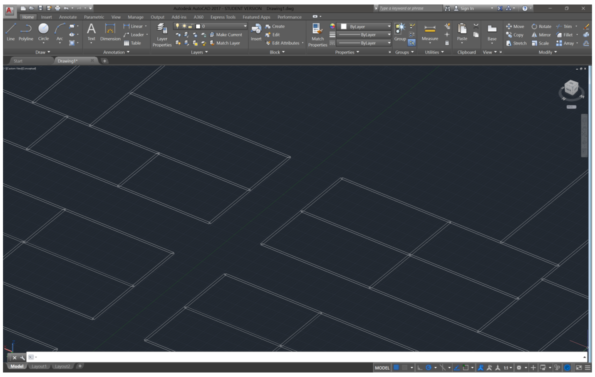
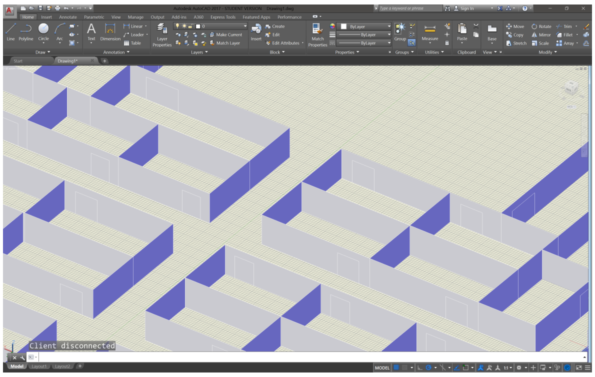


Figure 2: 2D model of the shopping mall, produced with COP in AutoCAD.



Client disconnected

Figure 3: Model of the mall produced for analysis, in AutoCAD. All the walls were replaced with surfaces.

produced a model for analysis (visible in Figure 3) without any changes to the algorithm.

With our solution, we were able to reduce the code that produces the models and introduce a more flexible way to both change the context and produce different views of the model. This did not required any additional functions, except the `with-layers` construct. In addition, by simply expanding the library and introducing new contexts, our algorithms are capable of producing new models without any changes, which would require additional code in the traditional approach.

Finally, there is another advantage in using our proposed solution, in comparison to the approach used by the original developers. In the original solution, one of the functions was chosen before the execution of the program, using the same function for all the generated elements, hence the same context. Changing the context would require the developer to stop the execution and change the code. On the other hand, with the COP version, layers are activated and deactivated dynamically, meaning that different parts of the program can be executed in different contexts. This feature offers more flexibility to developers, allowing the production of more complex models, where some elements can be represented in a simplified form, and others can be represented with more detail. This is useful in phases that only require further development of a group of elements, not needing detail in the remaining elements.

4.1 Evaluation

As it was possible to see in the previous section, ContextScheme and Racket allowed us to write context-dependent code that simplified the application, allowing the production of several models for different contexts. However, the resulting code can still be improved, making it easier to write, understand, and maintain.

For example, by examining listing 3, we can see an invocation of the `wall` and `door` functions before passing the actual arguments. This happens because ContextScheme uses higher-order functions that return the appropriate function for each context. However, if we chose ContextL instead, this would not be necessary, as the contextual functions can be used directly as normal functions, which simplifies the code.

Another relevant dimension is the way we interact with the contexts. In Ambience, context objects are implicit parameters and global activation is an option. This type of activation would allow us to activate the context and then write all the code we want to execute, instead of including it inside the scope of a `with-layers` construct. Nevertheless, this would require the user to deactivate and activate contexts explicitly in the code, so it is not clear if this option would simplify the experience of the user.

Finally, regarding performance, we have not yet done a comparison between the available COP implementations. However, due to the overheads involved in the creation of the geometric models, the impact of the COP implementation is negligible. For this reason, when used just for the implementation of the modeling operations, the COP implementation performance can be ignored.

5 CONCLUSIONS

Currently, algorithmic approaches are used in Architecture to create complex models of buildings that would otherwise be impossible to produce. Moreover, AD also simplifies and automates several tasks that were error-prone and time-consuming, and allows an easier exploration of variations. Nevertheless, when architects want to use analysis tools, or simply produce different views of the same model, they need additional algorithms, increasing the versions of code to maintain.

In this paper, we explore the combination of AD with COP, a paradigm that dynamically changes the behavior of the code depending on the active context, which is implemented with layers. There are several lines of research related to COP, which led to multiple implementations for different programming languages, namely ContextL, ContextJ, and ContextPy, among others, all of which have different features and advantages.

In our solution, we took advantage of a COP library that uses DSA, and a *class-in-layer* approach. All these features fit the needs of AD problems, and the use of Racket simplifies the introduction of COP in existing tools, such as Khepri.

To test our solution, we used our COP library in an existing AD program that produced the model of a shopping mall for simulation purposes. The program included multiple definitions of the same function to produce different views of the model, which were activated by commenting and uncommenting code. The program was re-implemented with COP, which eliminated the multiple definitions and the commented code. The use of our approach allowed the production of the model for several different contexts without additional changes in the program.

With this case study, we can conclude that COP can be combined with AD and it can be useful when exploring different views of the models, which require different behaviors from the same program.

6 FUTURE WORK

As future work, we will continue to expand our library with more building elements. We will also introduce more contexts, giving users more layers for the production of different kinds of models.

In addition, we will explore the combination of layers in order to obtain more sophisticated results. One idea is to explore a LOD layer in combination with the other layers, in order to produce simpler models in an exploration phase, and more complex ones in later stages of development.

ACKNOWLEDGMENTS

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013.

REFERENCES

[1] Malte Appeltauer, Robert Hirschfeld, and Tobias Rho. Dedicated programming support for context-aware ubiquitous applications. In *Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM'08. The Second International Conference on*, pages 38–43. IEEE, 2008.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, Jens Lincke, and Michael Perscheid. A comparison of context-oriented programming languages. In *International Workshop on Context-Oriented Programming*, page 6. ACM, 2009.

[3] Malte Appeltauer, Robert Hirschfeld, Hidehiko Masuhara, Michael Haupt, and Kazunori Kawauchi. Event-specific software composition in context-oriented programming. In *Software Composition*, volume 6144, pages 50–65. Springer, 2010.

[4] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. Contextj: Context-oriented programming with java. *Information and Media Technologies*, 6(2):399–419, 2011.

[5] DG Bobrow, L DeMichiel, RP Gabriel, G Kiczales, D Moon, and D Keene. Clos specification; x3j13 document 88-002r. *ACM-SIGPLAN Not*, 23, 1988.

[6] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: an overview of contextl. In *Proceedings of the 2005 symposium on Dynamic languages*, pages 1–10. ACM, 2005.

[7] Sofia Feist, Guilherme Barreto, Bruno Ferreira, and António Leitão. Portable generative design for building information modelling. In *Living Systems and Micro-Utopias: Towards Continuous Designing - Proceedings of the 21st International Conference of the Association for Computer-Aided Architectural Design Research in Asia, Melbourne, Australia*, 2016.

[8] Bruno Ferreira and António Leitão. Generative design for building information modeling. In *Real Time - Proceedings of the 33rd International Conference on Education and Research in Computer Aided Architectural Design in Europe, Vienna, Austria*, pages 635–644, 2015.

[9] Richard Garber. *BIM Design, Realising the Creative Potential of Building Information Modelling.* John Wiley and Sons: Hoboken, NJ, USA, 2014.

[10] Michael L Gassanenko. Context-oriented programming. *EuroForth'98*, 1998.

[11] S González, N Cardozo, K Mens, A Cádiz, JC Libbrecht, and J Goffaux. Subjective-c: Bringing context to mobile platform programming. intl. conf. on software language engineering. In *Proceedings of the International Conference on Software Language Engineering, Lecture Notes in Computer Science.* Springer-Verlag, 2010.

[12] Sebastián González, Kim Mens, and Alfredo Cádiz. Context-oriented programming with the ambient object system. *J. UCS*, 14(20):3307–3332, 2008.

[13] William Harrison and Harold Ossher. *Subject-oriented programming: a critique of pure objects*, volume 28. ACM, 1993.

[14] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245. Morgan Kaufmann Publishers Inc., 1973.

[15] Robert Hirschfeld, Pascal Costanza, and Michael Haupt. An introduction to context-oriented programming with contexts. In *Generative and Transformational Techniques in Software Engineering II*, pages 396–407. Springer, 2008.

[16] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object technology*, 7(3), 2008.

[17] Robert Hirschfeld, Michael Perscheid, Christian Schubert, and Malte Appeltauer. Dynamic contract layers. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 2169–2175. ACM, 2010.

[18] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. Designing event-based context transition in context-oriented programming. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, page 2. ACM, 2010.

[19] Gregor Kiczales, Jim Des Rivieres, and Daniel Gureasko Bobrow. *The art of the metaobject protocol.* MIT press, 1991.

[20] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. *ECOOP'97—Object-oriented programming*, pages 220–242, 1997.

[21] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An overview of aspectj. *ECOOP 2001—Object-Oriented Programming*, pages 327–354, 2001.

[22] António Leitão, Renata Castelo Branco, and Carmo Cardoso. Algorithmic-based analysis - design and analysis in a multi back-end generative tool. In *Protocols, Flows, and Glitches - Proceedings of the 22nd CAADRIA Conference, Xi'an Jiaotong-Liverpool University, Suzhou, China*, 2017.

[23] Jens Lincke, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. An open implementation for context-oriented layer composition in contextjs. *Science of Computer Programming*, 76(12):1194–1209, 2011.

[24] José Lopes and António Leitão. Portable generative design for cad applications. In *Proceedings of the 31st annual conference of the Association for Computed Aided Design in Architecture*, pages 196–203, 2011.

[25] Jon McCormack, Alan Dorin, and Troy Innocent. Generative design: a paradigm for design research. *Proceedings of Futureground, Design Research Society, Melbourne*, 2004.

[26] Andrew Payne and Rajaa Issa. The grasshopper primer. *Zen 'Edition. Robert McNeel & Associates*, 2009.

[27] Christian Prehofer. Feature-oriented programming: A fresh look at objects. *ECOOP'97—Object-Oriented Programming*, pages 419–443, 1997.

[28] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Javactx: seamless toolchain integration for context-oriented programming. In *Proceedings of the 3rd International Workshop on Context-Oriented Programming*, page 4. ACM, 2011.

[29] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Context-oriented programming: A software engineering perspective. *Journal of Systems and Software*, 85(8):1801–1817, 2012.

[30] Guido Salvaneschi, Carlo Ghezzi, and Matteo Pradella. Contexterlang: introducing context-oriented programming in the actor model. In *Proceedings of the 11th annual international conference on Aspect-oriented Software Development*, pages 191–202. ACM, 2012.

[31] Hans Schippers, Michael Haupt, and Robert Hirschfeld. An implementation substrate for languages composing modularized crosscutting concerns. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1944–1951. ACM, 2009.

[32] Gregor Schmidt. Contextr & contextwiki. *Master's thesis, Hasso-Plattner-Institut, Potsdam*, 39, 2008.

[33] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[34] Randall B Smith and David Ungar. A simple and unifying approach to subjective objects. *TAPOS*, 2(3):161–178, 1996.

[35] Jorge Vallejos, Sebastián González, Pascal Costanza, Wolfgang De Meuter, Theo D'Hondt, and Kim Mens. Predicated generic functions. In *Software Composition*, volume 6144, pages 66–81. Springer, 2010.

[36] Ramon van der Heijden, Evan Levelle, and Martin Riese. Parametric building information generation for design and construction. In *Computational Ecologies: Design in the Anthropocene - Proceedings of the 35th Annual Conference of the Association for Computer Aided Design in Architecture, Cincinnati, Ohio*, pages 417–429, 2015.

[37] Martin Von Löwis, Marcus Denker, and Oscar Nierstrasz. Context-oriented programming: beyond layers. In *Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007*, pages 143–156. ACM, 2007.

[38] Benjamin Hosain Wasty, Amir Semmo, Malte Appeltauer, Bastian Steinert, and Robert Hirschfeld. Contextlua: dynamic behavioral variations in computer games. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, page 5. ACM, 2010.

Session VI: Source Code Management, more Implementation

Clef Design

Thoughts on the Formalization of Program Construction

Klaas van Schelven
klaas@vanschelven.com

ABSTRACT

In *Expressions of Change* modifications to programs replace text files as the primary building blocks of software development. This novel approach yields structured historic information at arbitrary levels of program granularity across the programming toolchain. In this paper the associated questions of Programming Language Design are explored. We do so in the context of s-expressions, creating a modification-based infrastructure for languages in the Lisp family. We provide a framework for evaluation of the relative utility of different formalizations of program construction, which consists of the following: first, a requirement for completeness, meaning that a formalization of program construction should allow for the transformation of any valid program into any other. Second, a preference for succinctness over verbosity; succinctness of both of the formalization itself and typical expressions in the formalization. Third, a measure of the ability to clearly express intent. Fourth, a description of three ways in which the means of combination of the program itself and those of its means of construction may interact. Finally, we give a particular example of a formalization and use the provided framework to establish its utility.

CCS CONCEPTS

• **Software and its engineering → Development frameworks and environments**; *Software configuration management and version control systems*; *Software maintenance tools*;

KEYWORDS

program modification, programming language design, s-expressions

ACM Reference Format:
Klaas van Schelven. 2018. Clef Design: Thoughts on the Formalization of Program Construction. In *Proceedings of the 11th European Lisp Symposium (ELS'18).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

A large part of software development is concerned with modifying existing software, often over long time spans[6]. The associated need for historic record keeping is reflected in the popularity of *Version Control Systems.* In the mainstream approach, however, such systems are retrofitted rather than integrated: text remains the primary building block of program construction and the main interface shared across the tools in the development toolchain, such as editors and interpreters; the history is managed separately by the *VCS.*

An alternative approach is to take the modifications themselves as the primary building blocks. The set of allowed modifications to program structure is formalized, and such modifications are taken as the inputs and outputs by all tools in the programming toolchain. In short: program modification is reified. Experiments with this approach are bundled in a project called *Expressions of Change.*

We expect that the availability of well-structured historic information across the toolchain will prove invaluable when facing the typical challenges of program modification. As part of the project we have developed a prototype of an editor and a programming language, and early experiments with these indicate that the expected advantages will indeed materialize.

Setting out to reify program modification, one is immediately faced with the rather obvious question: *what does reified program modification look like?* What is the language with which a programmer can express, to the computer and other programmers alike, how a program can be constructed or modified? What are the *expressions of change*?

In more general terms, the question is: given some formalization of program structure, what should the accompanying formalization for program construction be? This is a non-trivial question, as the number of possible such formalizations is infinite, even for a single formalization of program structure, and any practical experiment with reified program modification must choose only a single one. This choice is thus a matter of programming language design.

This paper explores that design space for the single formalization of program structure of s-expressions. S-expressions make for a good initial testbed for this exploration for a number of reasons: the simplicity of their definition, the typical explicitness of their structure when pretty-printed, and the fact that the choice for s-expressions ensures some immediate relevance of any findings for languages in the Lisp family.

The relevance of this exploration for the project *Expressions of Change*, as well as projects which take a similar approach, is self-evident. As far as we know, questions of design of program modification have not been previously described, which is not surprising as they are directly tied to the original approach of reification of program modification itself. The contributions of this paper are the following:

- We develop a framework of criteria for comparison of different formalizations of program modification (section 3) and provide an overview of practical considerations in the design of such formalizations (section 4).
- We present a minimal formalization of program modification (section 5) and show that the chosen formalization ranks reasonably well given the criteria (sections 6 and 7).

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1.

2 THE DESIGN SPACE: S-EXPRESSIONS

In this paper we explore the question of design of formalizations of program construction in the context of a single formalization of program structure, namely that of s-expressions. An s-expression is recursively defined here as either:

- an atom, which is a symbol or number[1]
- a list of 0 or more s-expressions[2]

S-expressions are typically represented textually by printing the atoms as-is, and the lists between regular parentheses (and) with white space separating the elements of the list. Thus, the following is the textual representation of an s-expression:

```
(+ (* 6 9) 12)
```

2.1 Why s-expressions?

The choice for s-expressions as an object of study is motivated by a number of reasons.

First, their absolute minimalism: s-expressions can be both fully defined and illustrated with an example in some ten lines of text. Such a minimalistic definition of program structure allows for maximum focus on the subject at hand: that of program construction. Further, a minimal formalization of program structure is a prerequisite for a minimal formalization of program construction, because each special case of the program structure needs to be somehow accounted for. Smaller formalizations of program construction are preferable over larger ones in general (see section 3.2). For a first exploration of the design space, which this paper represents, this is even more strongly the case.

Second, the mapping between the structure of s-expressions and their visual representation is very direct. The more explicitly the structure is laid out on screen, the easier it is to understand modifications in terms of that structure. When modifications are put central, that is an important property.

Finally, the choice for s-expressions is practical, because s-expressions (or some extension thereof) form the basic syntax of many languages in the Lisp family, making it possible to use artifacts produced by *Expressions of Change* in the practical environment of an actual programming language. It also ensures relevance of the findings of the project for any existing languages in that family.

2.2 Intuitions for formalizations

Having formalized s-expressions, and thus program structure, we are ready to discuss formalizations of program construction. We start by developing an intuition, using a practical example of an expression of program construction in plain English:

Given the example expression given in section 2.1, do the following 3 things sequentially:

- remove the atom + from the root expression.
- insert a new atom, `hello-world`, as the first child of the root.
- remove the atom 6 from the sub-expression `(* 6 9)`.

The above list of bullets, although it is useful to provide an intuition for languages of program construction, is not sufficiently formal for our purposes. In particular, one of the explicit goals for such a language is that it can serve as an input for automated processes, i.e. as a shared interface across the development toolchain.

What are the elements that we might expect in any such formalization? Again, let's develop the intuition first. In general, we can at least expect:

- Support for a number of different kinds of modification, e.g. adding, updating, removing, copying and moving structures.
- Each kind of modification will be associated with further information that is specific to it, i.e. for deletions it is sufficient to specify a location only; for insertions we must also specify what must be inserted.
- Specific kinds of structure may be tied to specific mechanisms of construction. In the case of s-expressions: the ways we can modify atoms are not the same as the ways we can modify lists.
- A formalization of the order (if any) in which the operations must take place

Please note that the first two bullet points together form a good fit with Algebraic Data Types[7]: the different kinds can be represented using *Sum Types*, the different attributes using *Product Types*. We will occasionally use this fact as a notational shorthand below, independent of actual concerns of implementation.

Having established what we can expect in a formalization of program construction, let us develop an intuition for the associated design space, in which we need to make decisions such as:

- What kinds of modification should be supported? For example, is "updating" a special kind of operation, or is it enough to have access to a combination of adding and removing items?
- What are the relevant attributes for each kind of modification? For example, when deleting an item, what is the best way to formalize what is to be deleted? Should multiple such formalizations be catered for simultaneously?

Finally, let us establish the fact that this design space is infinite, by noticing that neither the space of kinds of modification, nor the set of relevant attributes has an upper bound. For example, to any formalization of program construction we can always add a kind of modification that inserts a particular s-expression. Since the number of s-expressions is infinite, we can thus create an infinite number of special kinds of modification.

2.3 Terminology and notation

Having established these intuitions, we shall introduce a minimal amount of terminology and notational convention. This is necessary because, in contrast to formalizations of program structure, in which terms such as *grammar*, *term* and *parser* have been well established, formalizations of program construction have not been well studied, and hence such terminology is not yet available. In choosing this terminology, we have taken inspiration from musical notation, being a real-world example of instructions for 'construction' rather than 'structure'. We introduce the following terms:

[1] There is no generally accepted single definition of what constitutes an s-expression. Instead, definitions vary, with support for a variety of possible atoms such as text, symbols, integers and floating point numbers. We restrict ourselves to printable symbols and numbers here without loss of generality.

[2] Lists are typically implemented using (nested) pairs; however, in this paper we shall make no assumptions about their implementation.

- A *clef* denotes a particular formalization of program construction. The analog in program structure is a grammar: just like a grammar describes what valid program syntax is, a clef describes what the valid means of construction are. The analog in natural language is a vocabulary: the clef defines the valid words. The musical metaphor is be understood as follows: just like a musical clef provides semantics for the notes in a score, a clef in program construction provides the semantics for its notes.
- A *note* denotes a specific kind of operation of construction, such as "adding", "removing" or "copying". Borrowing from the terminology of implementation: if a clef is implemented as an Algebraic Data Type, a note corresponds to a single data constructor. The respective analogs are: a term in a grammar, a word in a vocabulary. The term *note* is overloaded to also mean an instance of a *note*, with given values for all attributes, e.g. "delete the 2nd child".
- *To play* a note is to apply it to an existing structure, yielding a new structure.
- A *score* denotes a list of notes. To play a score is to play each note in turn, leading to the step-wise construction of some structure.

In this paper, we shall use s-expressions as a means of notation for notes. In particular, each note will be denoted using a list-expression, where the first element denotes its kind (corresponding to a data constructor in an ADT) and further elements denote the values of the attributes. The notation for a score is a list-expression of notes. Thus, the following denotes a score of 2 notes:

```
(
    (delete 3)
    (delete 5)
)
```

Please note that such a choice of notation poses no restrictions on implementation whatsoever, and is not mandated in any way by the fact that the structure under modification is itself an s-expression. It does, however, come with the benefit of directly suggesting a means of implementation in Lisps. An advantage of such an implementation is further that it enables self-applicability, i.e. the modification of notes in our clef in terms of that same clef.

The semantics of a clef are given in terms of a case-analysis on its notes. We use the imperative style, that is, a formulation in terms of how a given s-expression must be modified to reflect the playing of a particular note. If needed, an equivalent definition in the functional programming paradigm can be trivially derived.

3 A FRAMEWORK OF CRITERIA

In the previous section we have provided an overview of the infinitely many possible forms a clef may take. We shall now turn our attention to a comparison between these forms. The goal is to be able to pick a single clef which is shared as a common interface by tools in the programming toolchain[3].

Because different clefs are not equally useful in meaningfully expressing program modification, choosing a particular one is a matter of design. It stands to reason that we need some mechanism of evaluation of the utility of clefs, such that we may compare the utility of different approaches and choose the best one. In the below we present one such mechanism, i.e. a number of criteria that may be used for evaluation of clefs, as well as arguments pertaining to why these particular criteria are useful.

In the construction of this framework, we have in some cases taken inspiration from the evaluation of the relative utility of programming language features more generally, that is, outside the scope of program modification. Where this is the case, we make sure to highlight the aspects that are specific to program construction rather than structure.

3.1 Completeness

The first criterion for a successful clef is that it is *complete.* Given an arbitrary present structure it should be possible to reach an arbitrary desired structure in a finite number of steps. We relax this requirement somewhat in the context of structures that are defined in terms of sum types, stating that it is sufficient to be able to reach any structure defined using the same data constructor. That is, for s-expressions, it is enough to be able to construct any list-expression out of any list-expression, and any atom out of any atom. We assume the utility of this property to be self-evident.

3.2 Clef size

Second, with regards to the size of the clef's definition we make the observation that, all other things being equal, smaller is better.

First, the size of the clef is reflected in the cost of implementation of the automated processes programs that use it as its interface (editors, tools for program analysis, compilers). The larger this language, the larger the implementation-cost across the toolchain.

Second, larger clefs impose larger costs on their human users. In the approach of this project the notes from the clef form the primary building block of program construction. It follows that explicit exposure of the clef to the end-user, the programmer, is a design goal. If we want the programmer to be able to meaningfully interact with elements of the clef, they must understand these elements. The larger the clef's definition, the larger the mental burden of understanding it poses on the programmer.

Finally, a larger clef increases the risk of a distinction without a difference: that multiple equivalent mechanisms exist to construct the same result, even in cases when there is no meaningful underlying reason for this. Such distinctions serve only to confuse, and must be avoided.

The preference for small definition size is entirely analogous with the same preference in programming language design proper. However, it's worth noting that, with respect to text based programming languages, the clef introduces an additional layer of complexity on tools and programmers alike. Thus, the pressure to keep it small is increased.

3.3 Typical expressions' size

A clef that allows for concise expression of typical modifications to programs is to be preferred over one that does not. Please note that

[3] In fact, there being only a *single* clef is not a hard requirement, and future versions of the project may very well support a small set of somewhat related clefs, each one being used in a different part of the toolchain. For the utility of having a framework of evaluation of clefs this makes no difference: such a framework may then be used to select which clefs this small set consists of.

this is a separate concern from the one in the previous subsection, analogously to the cases of both programming language design and natural language, in which the size of the vocabulary is distinct from the size of sentences formed with that vocabulary, and in which the two are often inversely related.

In terms of automated tooling, larger expressions will typically incur some cost on storage and performance. However, this cost is expected to generally negligible and the size of typical expressions has no implication on implementation cost, which is tied strictly to definition size. The greatest cost of large expressions is thus incurred on the programmer, who will spend more time constructing, reading and understanding such expressions.

One important thing to note in the above criterion is that it assumes some knowledge of what "typical modifications to programs" are. That is, to a large degree, an empirical question.

3.4 Preservation of intent

A fourth desirable quality in a clef is for it to allow for clear expression of programmer intent. Here, again we take inspiration from the design of computer programs and programming languages, in which clear communication of intent is a desirable quality[2, 4, 9]. In that context, the concept of *programmer intent* is more or less understood: it denotes what the programmer wants a particular part of the program to achieve, and the mechanisms for them to communicate this intent with others. But what do we mean when talking about programmer intent in the context of program construction?

We mean approximately the following: when a programmer modifies the program, they typically do not do so at random, but with the intent to achieve a particular desired result. More often than not, this is achieved in a number of steps — whereby each step has some meaning to the programmer. Such a step-wise approach might even be reflected in a *todo* file or a piece of paper on which the steps are crossed off one by one. Similarly, in a pair programming session, one programmer may explain to another what they need to do next in a number of steps. The more closely the expressions built using a particular clef resemble lines in a *todo* file or utterances in a pair programming session, the better they express original intent.

Why do we think this is important? A central hypothesis of *Expressions of Change* is that, by having access to ubiquitous well-structured historical information, programs can be more easily understood. Such understanding is much easier to achieve if those histories are expressed in ways that are close to the thought-process of the original programmer.

As an example, consider the programmer goal of moving the method `get_widget()` from class `Foo` to class `Bar`. A clef that allows for expression of this goal using a single-note called "move" reveals more about the original intent than one that expresses the same modification using two separate and unrelated actions called "insert" and "delete".

A final piece of evidence for the importance of expression of intent in the context of program modification is presented by the currently accepted best practices in Version Control Systems, which are in favor of expressing intent through both mechanisms. In the language of Version Control: "One commit, one change" and "Writing good commit messages".

We make a distinction here between two different mechanisms through which intent may be expressed. First, there is the greater or lesser ability to express intent directly in terms of the notes of the clef, as in the example above. Second, some clefs may allow for expression of intent through informal comments in natural language.

3.5 Means of combination

A final desirable quality in a clef is that it allows for meaningful means of combination. Here, again, we take inspiration from the evaluation of expressiveness of programming languages per se, which may be approached using the question "What are the means of combination?"[1, 10, p. 4]

The importance of this question is a direct consequence of human nature. Human beings, including programmers, can typically hold approximately 7 items in their mind in an active, readily available state at any given time[11]. Thus large problems are approached by dividing them into parts, and combining those parts, and programs are no exception. Examples of such means of division and combination are *modules*, *procedures*, *expressions*, *classes* and *methods*. It is in terms of such parts, and the way that they are combined, that programs are understood.

The central hypothesis of *Expressions of Change* is that programs may also be understood in their historical context. How do these two mechanisms of creating understanding interact? How does the concept of meaningful composition interact with clef-design? We distinguish 3 questions, for 3 different kinds of interaction.

First, what are the means of combination within the clef itself? Can its notes be meaningfully combined at all? Moreover, can the results be used as the elements in further combinations? That is, does this mechanism of combination form a closure, and thus the ability to form arbitrarily structured hierarchical histories?

The second question we can ask of a clef is the following: does its usage enable a meaningful relationship between the program composition and its history? As noted, programs are typically composed of modules, classes, procedures etc. Is the historical information available for each such part? If so, this enables the programmer to get a historic view at any level of the program hierarchy, such that they may get an answer to each of the questions "what's the history of my program?", "what's the history of this module?" and "what's the history of this expression?" equally.

Third, is there a meaningful relationship between the program's composition and the composition of the program's history? Do the histories of parts of the program relate to the histories of their sub-parts? If they do, a programmer may arbitrarily switch between 2 modes while navigating: that of program structure (space) and program history (time).

4 PRACTICALITIES OF DESIGN

Before introducing the clef proper, we present some further considerations of clef design. These cannot not included in the framework from the previous section, because the trade-offs associated with any particular approach do not obviously point towards a particular design. Nevertheless, they do represent design decisions and are therefore relevant to discuss.

4.1 Sum type structure

An s-expression, as defined in section 2, is precisely one of two things: an atom or a list, i.e. it is defined as a sum-type. The fact that the way we can modify each of these two things is different has immediate consequences for an approach to structured modification. For lists, for example, we can reasonably speak about the insertion of an element, but for an atom such a modification is meaningless because an atom has no elements[4].

For a clef of s-expressions, the practical result is that some of its notes will be playable exclusively on list-expressions, and others exclusively on atoms[5]. To play such notes on a structure of the wrong kind, e.g. to add a child to an atom, is not allowed by definition[6].

4.2 Initial notes

As noted in section 2.3, notes are typically defined in terms of modification of an existing s-expression. When a note is played as the first note in the score, this raises the question what the existing s-expression to modify would be. We present three possible answers:

- We choose a particular single structure, such as the empty list-expression `()`, as the initial structure as a matter of definition.
- We specify the initial structure explicitly as needed, i.e. when playing a score; we keep track of what the initial structure is in some location external to the score, i.e. as "metadata"
- We alter the definition of the semantics of notes slightly, such that special semantics may be assigned in case they are played as the initial note in a score. Further, we assign such special semantics to one or more notes in the clef. That is, the initial structure is defined to be some special sentinel value denoting nothingness, and one or more notes are defined to be construct a particular s-expression out of such nothingness.

All of these approaches have some drawbacks: the first elevates one particular kind of structure over the others by making it the initial one, even if no natural order exists. This lack of natural order applies to s-expressions: it isn't quite clear whether we should consider list-expressions or atoms to be the most natural initial s-expression. In the second approach, the task of identifying the initial structure is pushed out of the score, but we must still keep track of it somehow. In practice, this means we need to associate this information with all children-creating notes, i.e. push the information "one level up". In some sense this moves the problem elsewhere rather than solving it. The third approach introduces a degree of asymmetry in the clef: some notes, but not all, may be used as the initial note.

For clefs of s-expressions in particular, the drawbacks of the third approach seem to be most limited, hence we have chosen it. In the chosen approach we introduce one or more special notes for structure-creation, which leads to a final question of clef-design: should playing such notes as anything but the initial note be an error, or should it have the effect of transforming whatever previous structure there was into the initial structure of the designated kind? We have chosen to disallow such midway reinitializations, judging that there cannot be a meaningful historical connection between an atom and a list-expression. Again, this decision may or may not generalize to other types of structures than s-expressions.

5 A MINIMAL CLEF

In this section, we present a minimal clef for s-expressions. As noted in section 2.3, we use the imperative style for a description of the semantics. For each item, any expectations about the previous s-expression are noted first. Any non-defined behavior such as playing a note on the wrong kind of s-expression, playing an initial note non-initially and vice versa is considered disallowed by definition.

- `(become-atom` ⟨*atom*⟩`)` — initial note only — constructs the given atom out of nothing.
- `(set-atom` ⟨*atom*⟩`)` — playable on atoms — modifies the atom into the given atom.
- `(become-list)` — initial note only — constructs an empty list expression out of nothing.
- `(insert` ⟨*index*⟩ ⟨*score*⟩`)` — playable on list-expressions — constructs a new s-expression by playing all notes in the given score in order, and inserts this newly constructed s-expression as a new child element at the provided index[7].
- `(delete` ⟨*index*⟩`)` — playable on list-expressions — deletes the element at the index.
- `(extend` ⟨*index*⟩ ⟨*score*⟩`)` — playable on list-expressions — constructs a new s-expression by playing the score, using the child currently at the provided index as an initial s-expression, and replacing the child with the result.
- `(chord` ⟨*score*⟩`)` — playability depending on the first note of the provided score — plays the provided score sequentially.

5.1 Example usage

Consider the following score:

```
(
    (become-list)
    (insert 0 (
        (become-list)
        (insert 0 ((become-atom *)))
        (insert 1 ((become-atom 6)))
    ))
    (insert 0 ((become-atom +)))
    (insert 2 ((become-atom 12)))
    (extend 1 (
        (insert 2 ((become-atom 9)))
    ))
)
```

The stepwise construction of an s-expression according to this score is summarized in the table below.

[4] We take the indivisibility of atoms to be their defining property by definition (from Greek – the prefix "a" meaning *not* and the word "tomos" *to cut*). The fact that an atom may be represented as a string of textual characters is, in this view, an implementation detail that is encapsulated.

[5] This conclusion does not generalize to clefs for any structure which happens to be defined as a sum-type: if various kinds of structure are similar in the way they can be modified a single note may be applicable to more than a single kind of structure.

[6] This is not to say that any note for list-expressions is playable on any list-expression, as the set of playable notes may be constrained by properties of the list-expression its played on. For example: removal of the 3rd element of a list is only possible if the list has such an element.

[7] Indices are, of course, 0-based[3].

lines	note	result
2	`(become-list)`	`()`
3 - 7	`(insert 0 (... * ... 6 ...))`	`((* 6))`
8	`(insert 0 ((become-atom +)))`	`(+ (* 6))`
9	`(insert 2 ((become-atom 12)))`	`(+ (* 6) 12)`
10 - 12	`(extend 1 (... 9 ...))`	`(+ (* 6 9) 12)`

5.2 Chords

Chords may be used to hierarchically structure a score, as in the example below. Please note that grouping notes by means of chords does not alter the semantics of construction; as such, the below example may be flattened into a single score without altering what s-expression would be constructed as a result.

```
(
    (become-list)
    (chord (
        (insert 0 ((become-atom +)))
        (insert 1 ((become-atom 3)))
        (chord (
            (insert 2 ((become-atom 8)))
            (insert 3 ((become-atom 4)))
        ))
    ))
)
```

6 EVALUATION OF UTILITY

In this section we examine the provided clef in terms of the first four means of evaluation of the framework; that is in terms of completeness (3.1), clef size (3.2), succinctness of expression (3.3) and means of expression of intent (3.4).

We first note that the clef is complete, i.e. it may be used to construct arbitrary list-expressions from arbitrary list-expressions (although, because reinitializations have been disallowed as per section 4.2, it is not the case that arbitrary atoms can be created from arbitrary list-expressions and vice versa). A trivial, albeit inefficient, mechanism to do so is: first, construct the empty list-expression from the given list-expression by deleting the first element until no more child-elements exist. Then, from the empty list-expression, construct the desired list-expression by, for each child, creating the score that constructs it recursively, and inserting it. Arbitrary atoms can be created trivially by the clef's definition.

Regarding the size of the presented clef, we note that at present no alternative exists, which is to say that a quantitative comparison is hard to make. We thus content ourselves with the observations of some basic facts: the clef has a total of seven notes. Two of these are specific to atoms, four to list-expressions and one is a generic means of combination. With respect to the number of types of expressions (atoms and lists), this represents a factor 3.5. In any case, seven is by no means the minimal amount, which is one[8]. We must point out though, that such a reduction of clef-size comes at considerable cost in terms of the other criteria of evaluation.

The following two criteria, namely whether the clef can be used for succinct expression of typical program modification, and whether it allows for clear expression of programmer intent are discussed together below. As noted, those criteria imply further questions which can only be answered empirically: what is the nature of typical program modification, and what is the kind of intent that a programmer typically wants to reveal? To answer these questions, we have implemented, as part of the project "*Expressions of Change*", a prototype of an editor that implements the given clef. Informal experiments indicate that the presented clef scores quite well.

In addition to this empirical observation with an actual prototype, we make some observations about the general nature of editing. The most basic ways to interact with any kind of data, are to add, update and delete[9]. The presented clef provides direct support for all of these; that is: an edit-session comprising of such actions alone can be expressed succinctly, and without loss of intent. However, these are by no means the only operations available in typical (text) editors. We mention a few:

- moving pieces of text around
- copy-pasting
- search and replace
- advanced code refactoring

For such operations, no direct counterparts are available in the clef. However, annotated chords may be used to preserve, to some degree, expression of intent. For example, a *move* might be expressed as a single chord that deletes a sub-expression in one place, and inserts the score representing the moved sub-expression's method of construction elsewhere.

7 MEANS OF COMBINATION

Finally, we turn our attention to the means of combination. In section 3.5, we distinguished 3 questions about the interaction between means of combination of structure and construction. Here we shall evaluate the presented clef in terms of those 3 questions.

7.1 Combining notes with chords

First, what are the means of combination within the clef itself? The most straightforward combination is to take a linear sequence of changes. Such a sequence is part of our definition; we've called this a *score*.

A more profound means of combination is provided by the note `chord`: it combines multiple notes, in the form of a score, into a single note. Thus, it can be used to form arbitrarily structured hierarchical histories.

The practical use of this ability is to structure a stream of changes into time-wise "chapters" and "sub chapters" for the human reader, i.e. to express intent. For example, the introduction of a new feature may require the refactoring of a certain part of the program, which is in turn an operation that consists of a number of further restructurings. Chords allow for a programmer to express precisely such a hierarchical structuring of history.

[8] The clef with the single note `become`, which takes an s-expression and changes the entire s-expression under consideration into the given s-expression is an example of such a 1-note clef.

[9] Reflected, for example, in the acronym CRUD[8]. (Please note that the *R* in that acronym, for *Read*, is not a data-altering operation and has therefore no relevance to a clef).

7.2 Combining structure and construction

Postponing the discussion of the second question for a moment, we turn our attention to the third: is there a meaningful relationship between the program's composition and the composition of the program's history?

Please note that the means of combination for the program structure under consideration, s-expressions, are list-expressions: list-expressions combine s-expressions into new s-expressions. As an instance of this, consider the s-expression constructed in section 5.1. It is a list-expression consisting of the further s-expressions `+`, `(* 6 9)` and `12`.

As we have seen in the previous section, the most basic mechanism for composition of program construction is the score. In the presented clef scores show up as an attribute to the 3 notes `insert`, `extend` and `chord`.

What, then, is the relationship between the scores of list-expressions, and the scores of the s-expressions that they are composed of? In terms of the provided example: what is the relationship between the score for the construction of `(+ (* 6 9) 12)` and the respective scores for the construction of its parts? In particular: can the listing in section 5.1, which corresponds to the score of the expression as a whole, be used to reconstruct the scores for that expression's sub-expressions, such as `(* 6 9)`?

Indeed it can be. The key observation is that all mechanisms that affect sub-expressions, namely `insert` and `extend`, are expressed in terms of a `score` that describes modifications to the sub-expression. In general, the process for extracting a lower-level score is to extract the relevant scores from the higher level expressions and concatenate them.

For the atoms `+` and `12`, the result is somewhat trivial: their histories consist solely of those atoms coming into being. The history of the list-expression `(* 6 9)` is more interesting; it can be constructed by concatenating the scores as found on lines 4 – 6 and 11, resulting in the following score[10]:

```
(
    (become-list)
    (insert 0 ((become-atom *)))
    (insert 1 ((become-atom 6)))
    (insert 2 ((become-atom 9)))
)
```

This is precisely what we were looking for: a meaningful relationship between the program's composition and the composition of the program's history.

Incidentally, this relationship also implies a positive answer to the second question: it enables the programmer to get a historic view at any level of the program hierarchy. Thus, our rather minimal clef has meaningful composition on all 3 levels.

8 FUTURE WORK

The central hypothesis of *Expressions of Change* is that structured historic information at arbitrary levels of program granularity is extremely useful in the software development practice. As part of the project we have built a prototype of an editor and a small interpreter of a subset of *Scheme*. Experiments with this editor in an informal setting confirm the practical applicability of the findings in this paper. However, to substantiate the central claim, much work remains to be done. In this work, we may distinguish between production and consumption of the clef. That is, first further experimentation with the editor to ensure a fluent editing experience, and second, experimentation with the utilization of this structured information in the rest of the toolchain. An example of the latter is to approach various forms of static analysis from the perspective of program construction, that is: incrementally.

In this paper we have explored formalization of program construction for a single particular formalization of program structure, namely that of s-expressions. Whether it is possible to define clefs of practical utility for arbitrarily complex formalizations of program structure is an open question, although there are some reasons to believe that it isn't. In particular, the arguments in favor of s-expressions as an object of study, presented in section 2.1, do not generally extend to arbitrary definitions of program structure. Our conclusion is that future languages should be designed with structured modification in mind.

In the prototype of the editor, user actions correspond directly to notes in the clef, and its output is a score representing the actual edit-session. However, programming is to some degree an exploratory activity and an actual edit-session may contain many dead ends. A log including all those dead ends is not the clearest possible way to communicate intent. Thus, there is likely a need for an extra step, that is analogous with a *commit* in a VCS, in which some of reality's details are hidden in the interest of clarity. We imagine that various automated tools will be usefull in this step, e.g. to automatically detect such dead ends and prune them as required.

The properties of formalizations of program construction in the context of a collaborative programming effort have not yet been researched. Briefly, we can say the following: one key question when collaborating is how the diverging work of multiple programmers can be joined together with some degree of automation (that is: "merging"). With respect to the mainstream approach, Version Control of text files, our approach creates both advantages and challenges. On the one hand, the availability of fine-grained well-structured information makes automated merges easier, and error messages more precise. On the other hand, the introduction of a historical dimension raises new questions, because merging needs to take place on the level of program construction as well as program structure.

9 RELATED WORK

The approach to program construction presented in this paper presupposes a structured approach to program editing. Examples of recent projects that take this approach are *Lamdu* and *Unison*, both with a program syntax inspired by Haskell and a strong focus on static typing, and *Cirru*, which is a structured editor of s-expressions. Program *construction*, however, has not been given a central role in those projects. To our knowledge, the only other project which has an explicitly defined semantics of program construction is

[10] The fact that the notes (insert 0 ...) and (extend 1 ...) are indeed modifying the same sub-expression might not be immediately apparent because the indices differ. This shift in indices is caused by an intermediate insertion at index 0 on line line 8. In the UI of practical applications, the connection between such apparently unrelated notes may be clarified by using some unique and unchanging identifier, such as the order of creation of a child.

Hazelnut[13], a project that focusses on the interaction between program construction and static typing.

Mechanisms of recording program history are wide-spread in mainstream software development, most notably in the form of Version Control Systems. However, such systems typically operate on unstructured text. A more structured approach to diffing is taken by Miraldo et. al.[12]. Our approach differs because it puts program modification central in the design, rather than trying to extract information about program modification after the fact.

Finally, and most generally, some ideas analog to those presented in this paper are captured in the design pattern of *Event Sourcing*, described by Martin Fowler[5]. In that pattern, the idea is to capture all changes to an application state as a sequence of events. That is, the pattern captures the idea of construction-over-structure in the domain of Enterprise Application Architecture rather than programming.

10 CONCLUSIONS

Following from the observation that computer programs will be changed, and that managing those changes themselves forms a large part of a computer programmer's work, *Expressions of Change* presents a novel approach: to put the modifications themselves central in the programming experience.

The results of the first step in that approach have been presented in this paper: how to approach the design of the formalization of programming construction? We have shown that important goals in such design are: to enable intent-revealing primitives and succinct expression, while keeping a minimal footprint. Further, we have shown various possible levels of combination, both within the formalization of program construction and in its interaction with program structure.

Finally, we have shown a particular formalization of program construction for s-expressions, and how this minimal mechanism allows for expression of programmer intent, and all 3 levels of possible means of combination.

ACKNOWLEDGMENTS

We would like to thank Wouter Swierstra and the anonymous reviewers for their helpful and constructive comments that greatly contributed to improving the final version of the paper.

REFERENCES

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 2nd edition, 1996. ISBN 0262011530.

[2] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice-Hall, Upper Saddle River, NJ, 1997.

[3] Edsger W. Dijkstra. Why numbering should start at zero. circulated privately, 8 1982. URL http://www.cs.utexas.edu/users/EWD/ewd08xx/EWD831.PDF.

[4] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-48567-2.

[5] Martin Fowler. Event sourcing, capture all changes to an application state as a sequence of events, 2005. URL https://martinfowler.com/eaaDev/EventSourcing.html.

[6] Robert L. Glass. Frequently forgotten fundamental facts about software engineering. *IEEE Softw.*, 18(3):112–111, 2001. ISSN 0740-7459. doi: 10.1109/MS.2001.922739. URL http://dx.doi.org/10.1109/MS.2001.922739.

[7] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-7. doi: 10.1145/1238844.1238856. URL http://doi.acm.org/10.1145/1238844.1238856.

[8] James Martin. *Managing the Data Base Environment*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1983. ISBN 0135505828.

[9] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1 edition, 2008. ISBN 0132350882, 9780132350884.

[10] Erik Meijer. "composition is the essence of programming", 23 November 2014. URL https://twitter.com/headinthebox/status/536665449799614464.

[11] George A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information. *The Psychological Review*, 63(2):81–97, March 1956. URL http://www.musanim.com/miller1956/.

[12] Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. Type-directed diffing of structured data. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development*, TyDe 2017, pages 2–15, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5183-6. doi: 10.1145/3122975.3122976. URL http://doi.acm.org/10.1145/3122975.3122976.

[13] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. Hazelnut: A bidirectionally typed structure editor calculus. *CoRR*, abs/1607.04180, 2016. URL http://arxiv.org/abs/1607.04180.

Partial Inlining Using Local Graph Rewriting

Irène Durand *
Robert Strandh
irene.durand@u-bordeaux.fr
robert.strandh@u-bordeaux.fr
LaBRI, University of Bordeaux
Talence, France

ABSTRACT

Inlining is an important optimization technique in any modern compiler, though the description of this technique in the literature is informal and vague. We describe a technique for inlining, designed to work on a *flow graph* of instructions of intermediate code.

Our technique uses *local graph rewriting*, making the semantic correctness of this technique obvious. In addition, we prove that the algorithm terminates.

As a direct result of the preservation of the semantics of the program after each local rewriting step, the algorithm can stop after any iteration, resulting in a *partial inlining* of the called function. Such partial inlining can be advantageous in order to avoid the inlining of code that is not performance critical, in particular for creating arguments and calls to error-signaling functions.

CCS CONCEPTS

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

KEYWORDS

Common Lisp, Compiler optimization, Portability, Maintainability, Graph rewriting

ACM Reference Format:
Irène Durand and Robert Strandh. 2018. Partial Inlining Using Local Graph Rewriting . In *Proceedings of the 11th European Lisp Symposium (ELS'18).* ACM, New York, NY, USA, 8 pages.

1 INTRODUCTION

Inlining represents an important optimization technique in any modern compiler. It avoids the overhead of a full function call, and it allows further optimization in the calling function in the form of type inference, loop optimizations, and more.

While the advantages of inlining are well known and well documented, inlining also entails some disadvantages. It increases the size of the code, with a possible negative impact on processor cache performance. It also increases pressure on register allocation, possibly making it necessary to spill registers to the stack more often. Most importantly, though, as Ayers et al. point out [1, 2], since many optimization algorithms do not have linear-time complexity in the size of the code, inlining can have a serious impact on the execution time of the compiler.

Some authors distinguish between *procedure integration* and *inline expansion* [7]. Both techniques are often referred to with the abbreviated form *inlining*. Our use of *inlining* corresponds to *procedure integration*.

Most literature sources define inlining as "replacing a call to a function with a copy of the body of the called function" (see e.g., [3, 4, 8]). This definition suggests that inlining is an all-or-nothing transformation. In this paper, we present a technique that allows for *partial* inlining. More precisely, it allows for a *prefix* of the callee to be copied into the caller. We obtain this property by using *local graph rewriting* at the level of instructions in intermediate code. A single instruction is inlined in each step, preserving the overall semantics of the program, and thereby allowing us to stop the process at any time.

The traditional definition of inlining is too vague for our purpose. It suggests that the sole purpose of inlining is to avoid overhead in the function-call protocol. However, on modern processors, this overhead is insignificant. For the purpose of this paper, we would also like to avoid the creation of a local *environment* that would normally be necessary for each invocation of the callee. This additional requirement poses additional restrictions as to when inlining is appropriate.

In this paper, we discuss only the inlining technique itself. We do not consider the policy to determine when it is advantageous to perform the technique, and, although our technique allows for partial inlining, we also do not consider the policy of when inlining should stop.

2 PREVIOUS WORK

Before inlining was applied to so-called "structured programming languages", the technique was applied to languages such as Fortran, that do not allow recursion, and therefore do not need for subroutines to allocate their own environments upon entry. And it was then referred to as "open-coding of subroutines". Scheifler [8] is probably one of the first to apply inlining to more modern programming languages. The language used by Scheifler is CLU [6].

Ayers et al [1] consider the benefit of inlining consisting of the elimination of the overhead of a procedure call to be a "side benefit", and we agree. They cite the main benefit as the opportunity for

[*] This author was supported by the French National Research Agency (ANR project GraphEn / ANR-15-CE40-0009).

ELS'18, April 16–17 2018, Marbella, Spain

ACM ISBN 978-2-9557474-2-1.

more optimizing code transformations when the code of the called function is exposed in the context of the calling function.

In their paper, they also mention *cloning* as an alternative to inlining, i.e., the duplication and specialization of the called function according to the context of the calling function. However, they consider inlining to be strictly superior to cloning in terms of the possible additional optimizations made possible, so they recommend cloning only as a means to avoid too large an increase in the code size, which could slow down subsequent non-linear optimizations. Cloning, and especially the specialization of the cloned code in the context of the caller, is one technique used in *partial evaluation* [5]. Inlining, however, whether total or partial, is not a technique of partial evaluation. Inlining may of course enable such techniques by exposing the code of the called function in the context of the caller.

Most existing work is concerned with determining when inlining is to be performed, based on some analysis of the benefits as compared to the penalties in terms of increased compilation time in subsequent optimization passes. The inlining technique itself is considered trivial, or in the words of Chang and Hwu ([3, 4]) "The work required to duplicate the callee is trivial". Inlining might be trivial in the context of purely function programming, in that it suffices to replace occurrences of local variables in the called function by the argument expressions in the function call. However, for a language such as Common Lisp that allows for assignments to lexical variables, inlining can be non-trivial. Consider the following example:

```
(defun f (x y) (setq x y))

(defun g (a) (f a 3) a)
```

If simple renaming is applied, we obtain the following code which does not preserve the semantics of the original code:

```
(defun g (a) (setq a 3) a)
```

The use of continuation-passing style for compiling Common Lisp programs often requires a priori elimination of side effects by confiding these side effects to updates on *cells*. Such a conversion transforms the program so that it respects a purely functional style, making inlining trivial as indicated above. However, such a conversion has a significant impact on program performance, especially in the context of modern processors, where memory access are orders of magnitude more expensive than register operations.

Because of issues such as this one, this paper discusses only a technique for inlining in the context of arbitrary Common Lisp code that might contain such side effects. It does not discuss the more complex issue of determining a strategy for when inlining should or should not be applied.

Although the paper by Ayers et al explains that their technique is applied to intermediate code, just like the technique that we present in this paper, their paper contains little information about the details of their technique.

3 OUR TECHNIQUE

The work described in this paper is part of the Cleavir compiler framework. Cleavir is currently part of the SICL project[1], but we may turn it into an independent project in the future.

In our compiler, source code is first converted to an *abstract syntax tree*. In such a tree, lexical variables and lexical function names have been converted to unique objects. When a globally defined function F is inlined into another function G, we incorporate the abstract syntax tree of F as if it were a local function in G. No alpha renaming is required. Notice that this step in itself does not count as inlining. The function F is still invoked using the normal function-call protocol at this stage.

In the second phase, the abstract syntax tree is translated to intermediate code in the form of a flow graph of instructions. Our inlining technique is designed to work on this intermediate representation.

There are several advantages of using this intermediate representation over higher-level ones such as source code or abstract syntax trees, as we will show in greater detail below, namely:

- Each iteration of the algorithm defined by our technique is very simple, and we can be shown to preserve the semantics of the program.
- Because each iteration preserves the semantics, the process can be interrupted at any point in time, resulting in a *partial* inlining of the called function.

Furthermore, this intermediate code representation is similar to the one used in many traditional compiler optimization techniques, making it possible to reuse code for similar transformations.

One potential drawback of this representation is that operations on programs represented this way are inherently imperative, i.e. they modify the structure of the flow graph. The use of techniques from functional programming is therefore difficult or impractical with this representation. Moreover, the flow graph resulting from some arbitrary number of iterations of our technique does not necessarily have any correspondence as Common Lisp source code.

3.1 Intermediate code

The intermediate code on which our technique is designed to work is called *High-level Intermediate Representation*, or *HIR* for short. This representation takes the form of a *flow graph* of *instructions* as used by many traditional compiler optimization techniques. The main difference between HIR and the intermediate representation used in compilers for lower-level languages is that in HIR, the only data objects that the instructions manipulate are Common Lisp objects. Arbitrary computations on addresses are exposed in a later stage called *Medium-level Intermediate Representation*, or *MIR*.

Most HIR instructions correspond directly to Common Lisp operators such as the ones in the categories described below. Notice that, although the names of the instructions often resemble the names of Common Lisp operators, the instruction typically requires more precise objects than the corresponding Common Lisp operator does. Thus, the `car` instruction requires the argument to be a `cons` object, and the `funcall` instruction requires its first argument to be a function. The following such categories exist:

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

- Low-level accessors such as car, cdr, rplaca, rplacd, aref, aset, slot-read, and slot-write.
- Instructions for low-level arithmetic on, and comparison of, floating-point numbers and fixnums.
- Instructions for testing the type of an object.
- Instructions such as funcall, return, and unwind for handling function calls and returns.

Two of the HIR instructions are special in that they do not have direct corresponding Common Lisp operators, and in that they are essential to the inlining machinery described in this paper:

- The enter instruction. This instruction is the first one to be executed in a function, and it is responsible for creating the initial local lexical environment of the function from the arguments given by the calling function. This initial environment is typically augmented by temporary lexical variables during the execution of the function. Variables may also be eliminated from the local environment when they are no longer accessible by any execution path.
- The enclose instruction. This instruction takes the *code* of a nested function (represented by its enter instruction) and creates a *callable function* that may be a *closure*.

3.2 Algorithm

The algorithm that implements our technique maintains a *worklist*. An item[2] of the worklist contains:

- A funcall instruction, representing the call site in the calling function.
- An enter instruction, representing the called function.
- The successor instruction of the enter instruction, called the *target instruction*, or *target* for short. The target instruction is the one that is a candidate for inlining, and it is used for generic dispatch.
- A mapping from lexical variables in the called function that have already been duplicated in the calling function.

In addition to the contents of the worklist items, our algorithm maintains the following global information, independent of any worklist item:

- A mapping from instructions in the called function that have already been inlined, to the corresponding instructions in the calling function. This information prevents an instruction from being inlined more than once. Without this information, and in the presence of loops in the called function, our inlining algorithm would go into an infinite computation.
- Information about the ownership of lexical variables referred to by the called function. This ownership information indicates whether a lexical variable is created by the called function itself, or by some enclosing function. When an instruction to be inlined refers to a variable that is created by some enclosing function, the reference is maintained without modification. When the reference is to a variable created by the function itself, the inlined instruction must refer to the corresponding variable in the calling function instead.

Prior to algorithm execution, assignment instructions are inserted before the funcall instruction, copying each argument to a temporary lexical variable. These lexical variables represent a copy of the initial environment of the called function, but allocated in the calling function. The pair consisting of the funcall and the enter instruction can be seen as transferring this environment from the calling function to the called function. The variable correspondences form the initial lexical variable mapping to be used in the algorithm.

Initially, the worklist contains a single worklist item with the following contents:

- The funcall instruction representing the call that should be inlined.
- A *private copy* of the initial enter instruction of the function to inline.
- The successor instruction of the initial enter instruction, which is the initial target.
- The initial lexical variable mapping described previously.

In each iteration of the algorithm, a worklist item is removed from the worklist, and a generic function is called with four arguments, representing the contents of the worklist item. Each iteration may result in zero, one, or two new worklist items, according to the mappings and ownership information, and according to the number of successors of the target instruction in this contents.

When the generic function is called in each iteration, one of the following four rules applies. As we show in Section 3.4, each of the following rules preserves the overall operational semantics of the code:

(1) If the target instruction has already been inlined, i.e. it is in the mapping containing this information as described previously, then replace the funcall instruction by the inlined version of the target. There are two ways of doing this replacement. Either the predecessors of the funcall instruction are redirected to the inlined version of the target instruction, effectively making the funcall instruction unreachable, or else, the funcall instruction is replaced by a no-operation instruction with the inlined version of the target instruction as its successor. When this rule applies, no new item is added to the worklist.
(2) If the target instruction is a return instruction, then replace the funcall instruction by one or more assignment instructions mapping inputs of the funcall instruction to outputs of that same instruction. Again, in this case, no new item is added to the worklist.
(3) If the target instruction has a single successor, insert a copy of the next instruction before the funcall instruction, and make the enter instruction refer to that successor. Update the mappings, the inputs of the funcall instruction, and the outputs of the enter instruction as described below. In this case, the funcall instruction, the enter instruction, the new successor of the enter instruction, and the updated lexical variable mapping are inserted as a new item on the worklist for later processing.
(4) If the target instruction has two successors, insert a copy of the target instruction before the funcall instruction, and

[2] In the code, an item also contains an enclose instruction, but we omit this instruction from our description, in order to simplify it.

replicate the funcall instruction in each branch. Also replicate the enter instruction so that each replica refers to a different successor of the original instruction. Update the mappings, the inputs of the funcall instruction, and the outputs of the enter instruction as described below. In this case, two new items are inserted on the worklist for later processing. Each item contains a funcall instruction, an enter instruction, the successor of the enter instruction, and a lexical variable mapping, corresponding to each successor branch of the inlined instruction.

For rules 3 and 4, when a new instruction is inlined, the mappings, the inputs to the funcall instruction, and the outputs of the enter instruction are updated as follows:

- An entry is created in the mapping from instructions in the called function to instructions in the calling function, containing the inlined instruction and its copy in the calling function.
- If some input i to the inlined instruction is present in the lexical variable mapping (mapping to (say) ii in the calling function) and in the outputs of the enter instruction, but i is no longer live after the inlined instruction, then the entry ii - i is eliminated from the mapping, i is eliminated from the outputs of the enter instruction, and ii is eliminated from the inputs to the funcall instruction. It would be semantically harmless to leave it intact, but it might harm performance if the inlining procedure is stopped when it is still partial. Notice that, when an instruction with two successors is inlined, variable liveness may be different in the two successor branches.
- If some output o of the inlined instruction is a new variable that is created by that instruction, then we proceed as follows. Let I be the instruction in the called function that has been inlined, and let II be the copy of I in the calling function. We create a new variable oo in the calling function that takes the place of o in II. We add oo as an input to the funcall instruction, o as an output of the enter instruction, and we add oo - o to the lexical variable mapping. Again, if the inlined instruction has two successors, the lexical variable mapping may have to be updated for one or the other or both of the successors.

3.3 Example

As an example of our technique, consider the initial instruction graph in Figure 1. On the left is the calling function. It has three lexical variables, namely x, a, and y. The variable a is referenced by the called function, but it is owned by the calling function. The called function has a single variable named z in its initial lexical environment. A temporary variable w is created as a result of the execution of one of the instructions in the called function.

Before the inlining procedure is started, we create temporary variables in the calling function for the variables in the initial environment of the called function. We also create a private copy of the enter instruction so that we can mutate it during the inlining procedure. The result is shown in Figure 2.

As we can see in Figure 2, an assignment instruction has been created that copies the value of the lexical variable x into a variable zz that mirrors the initial lexical variable z in the called function. We also see that there are now two identical enter instructions. The one labeled enterA is the private copy.

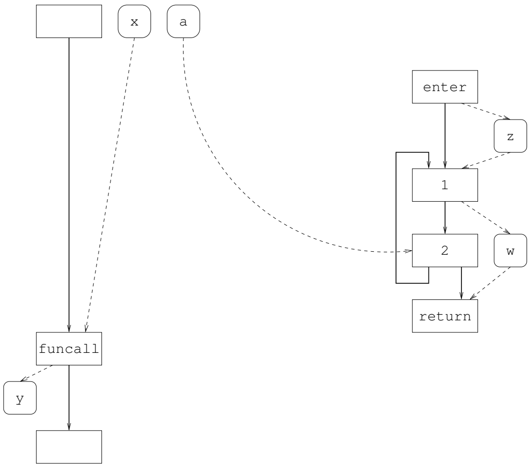


Figure 1: Initial instruction graph.

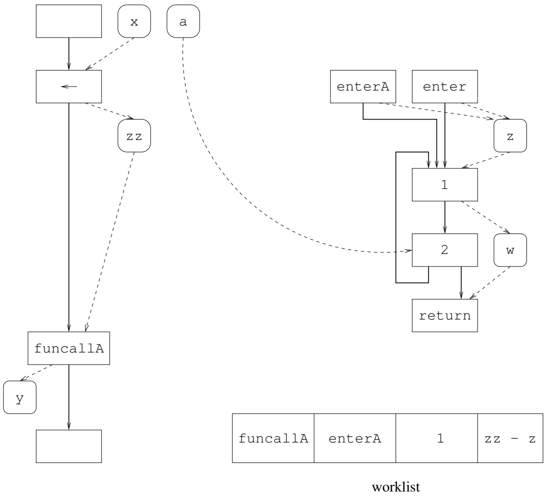


Figure 2: Instruction graph after initialization.

Step one of the inlining procedure consists of inlining the successor of our private enter instruction, i.e. the instruction labeled 1 in Figure 2. That instruction has a single successor, and it has not yet been inlined. Therefore, rule 3 applies, so we insert a copy of that instruction before the funcall instruction. Furthermore, since the input to the original instruction is the lexical variable z, and that variable is mapped to zz in the calling function, the inlined instruction receives zz as its input. The output of the original instruction is the temporary variable w that is not in our lexical variable mapping. Therefore, a temporary variable ww is created in the calling function, and an entry is created in the mapping that translates w to ww. The private enter instruction (labeled enterA) is modified so that it

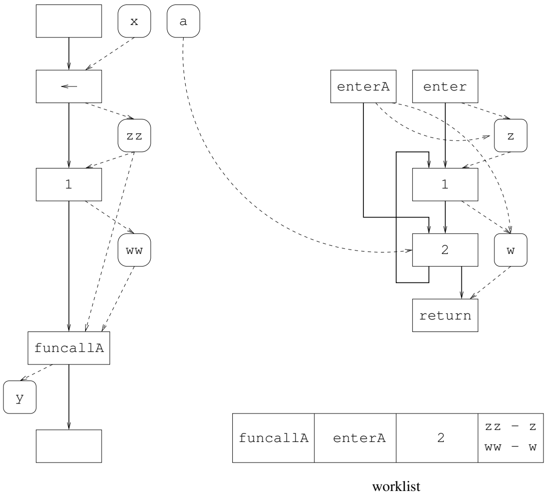


Figure 3: Instruction graph after one inlining step.

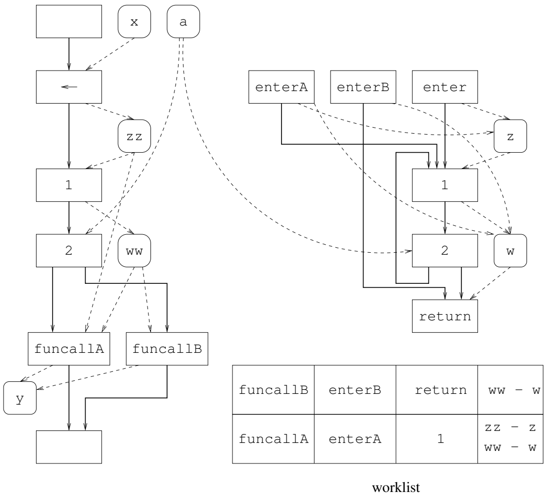

x
a
←
zz
1
2
ww
funcallA
funcallB
y
enterA
enterB
enter
z
1
2
w
return
funcallB | enterB | return | ww - w
funcallA | enterA | 1 | zz - z ww - w
worklist

Figure 4: Instruction graph after two inlining steps.

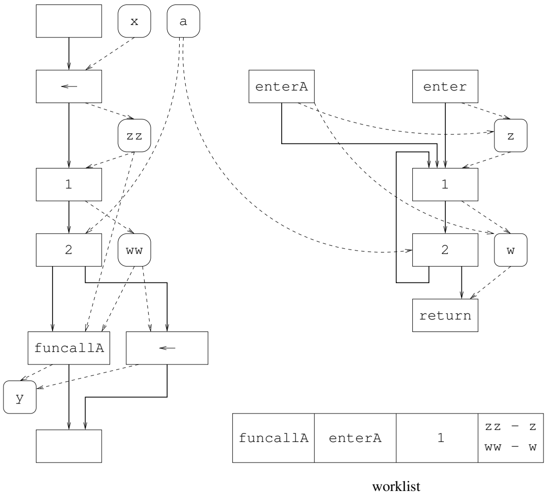


Figure 5: Instruction graph after three inlining steps.

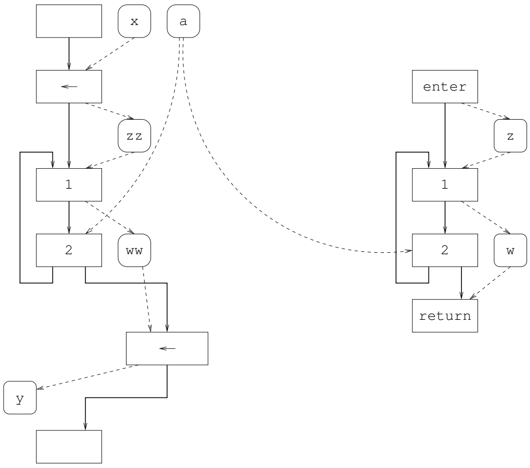


Figure 6: Instruction graph after four inlining steps.

now refers to the next instruction to be considered as a target. The result of this step is shown in Figure 3.

In step two of the inlining procedure, we are considering inlining an instruction with two successors, i.e. the one labeled 2 in Figure 3. It has not yet been inlined, so rule number 4 applies. As rule number 4 stipulates, we must replicate both the `enter` instruction and the `funcall` instruction. The result is shown in Figure 4.

In Figure 4, the `funcall` instruction labeled `funcallA` is paired with the `enter` instruction labeled `enterA` and the `funcall` instruction labeled `funcallB` is paired with the `enter` instruction labeled `enterB`.

In step three of the inlining procedure, we consider the `funcall` instruction labeled `funcallB`. The corresponding `enter` instruction has a `return` instruction as its successor, so rule number 2 applies. We must therefore replace the `funcall` instruction by an assignment instruction, assigning the value of the variable `ww` to the variable `y`. The result of this operation is shown in Figure 5.

In step four of the inlining procedure, we consider the `funcall` instruction labeled `funcallA` in Figure 5 and the corresponding `enter` instruction. The successor of the `enter` instruction is the instruction labeled 1, and that instruction has already been inlined, so rule number 1 applies. We therefore remove the `funcall` and redirect its predecessors to the inlined version of the instruction labeled 1. The result is shown in Figure 6, and that completes the inlining procedure.

After some minor reorganization of the instructions in Figure 6, we obtain the final result shown in Figure 7. Clearly we have an inlined version of the called function now replicated in the calling function.

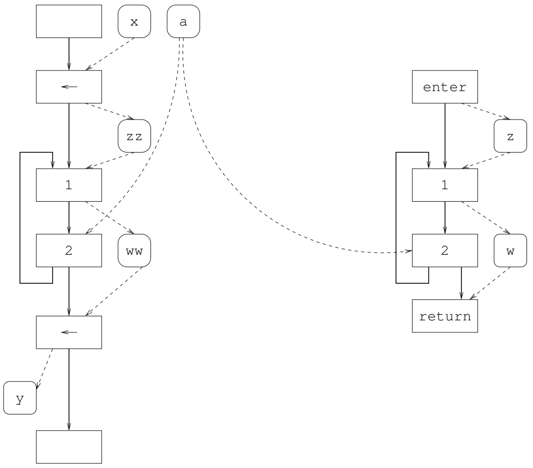


Figure 7: Final instruction graph.

3.4 Correctness of our technique

In order to prove total correctness of our technique, we must show that two conditions hold:

(1) Partial correctness, i.e. the technique must preserve the semantics of the program.
(2) Termination.

3.4.1 Partial correctness. Our technique preserves a very strong version of the semantics of the program, namely the *operational* semantics. This fact makes it unnecessary to create a precise definition of the program semantics, as might have been the case for some weaker type of semantics. Instead, we only need to show that the exact same operations are performed before and after each inlining step.

After a copy of the initial environment of the called function has been made in the environment of the calling function, we can see a pair of funcall/enter instructions as defining a morphism σ, mapping the copy of this environment in the calling function to its original version in the called function. The inputs of the funcall instruction are mapped to the outputs of the enter instruction. The lexical variable mapping used in our technique is simply the inverse, i.e σ^{-1} of this morphism. Similarly, a pair of return/funcall instructions can be seen as defining a morphism τ, mapping the environment in the called function to the environment in the calling function. The inputs of the return instruction are mapped to the outputs of the funcall instruction. These morphisms are illustrated in an example of an initial situation in Figure 8.

Applying rule 3 or rule 4 copies one instruction from the called function to the calling function, applying the morphism σ^{-1} to its inputs and outputs. Two applications of rule 3 from the initial situation are illustrated in Figure 9 and Figure 10. Applying rule 4 is a bit more involved, but the same mechanism is used. As we can see from these figures, thanks to the morphism, the instructions operate the same way whether inlined or not. The semantics are thus the same in both cases.

When rule 2 is applied, the return instruction is not copied. Instead, a number of assignment instructions are created in the calling function. Together, these assignment instructions define the composition of the two morphisms τ and σ, i.e. $\tau \circ \sigma$. Applying this rule therefore does not alter the semantics of the program. It merely maps the returned values to their copies in the calling function. Applying this rule is illustrated in Figure 11.

Finally, applying rule 1 merely avoids the control transfer from the calling function to the called function, by replacing the funcall instruction by an existing copy of the instruction that would have been inlined by rule 3 or rule 4. The existing copy obviously already operates in the environment of the calling function.

3.4.2 Termination. In order to prove termination, we invent a metric with the following properties:

- It has a lower bound on its value.
- Its value decreases with each iteration of our inlining procedure.

The metric we have chosen for this purpose is called *remaining work*, and it is represented as a pair $r = (I, F)$ where I is the number of instructions that have yet to be inlined, and F is the number of funcall instructions that have yet to be processed as part of the worklist items. Clearly, it has a lower bound on its value, namely $r_{min} = (0, 0)$.

Initially, the remaining work has the value $r_0 = (N, 1)$ where N is the number of instructions in the called function. We consider the metric to be lexicographically ordered by its components, i.e. $(I_1, F_1) < (I_2, F_2)$ if and only if either $I_1 < I_2$ or $I_1 = I_2$ and $F_1 < F_2$. We show that each step yields a value that is strictly smaller than before the step.

Consider some iteration k of our inlining procedure, so that $r_k = (I_k, F_k)$ is the remaining work before the iteration, and $r_{k+1} = (I_{k+1}, F_{k+1})$ is the remaining work after the iteration.

- If rule number 1 applies, then one funcall instruction is eliminated in the iteration, so that $I_{k+1} = I_k$ and $F_{k+1} = F_k - 1$. Clearly, $r_{k+1} < r_k$ in this case.
- If rule number 2 applies, then again one funcall instruction is eliminated in the iteration, so that $I_{k+1} = I_k$ and $F_{k+1} = F_k - 1$. Again, $r_{k+1} < r_k$.
- If rule number 3 applies, then another instruction is inlined, but the number of funcall instructions remains the same, so that $I_{k+1} = I_k - 1$ and $F_{k+1} = F_k$. Again, $r_{k+1} < r_k$.
- Finally, if rule number 4 applies, then another instruction is inlined, but the number of funcall instructions increases by 1, so that $I_{k+1} = I_k - 1$ and $F_{k+1} = F_k + 1$. Again, $r_{k+1} < r_k$.

4 CONCLUSIONS AND FUTURE WORK

We have presented a technique for inlining local functions that uses local graph rewriting techniques. We have proved our technique to be correct in that it preserves the semantics of the original program, and it is guaranteed to terminate.

Although our iterative technique can be stopped at any point, thus giving us *partial inlining*, there are some practical aspects of such partial inlining that still need to be investigated:

- When the inlining is not complete, the called function has multiple entry points. Many optimization techniques described in the literature assume that a function has a single entry point. We plan to investigate the consequences of such

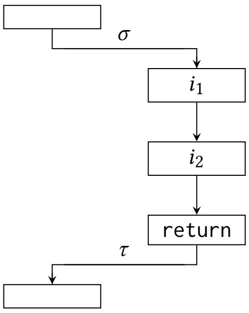


Figure 8: Initial situation.

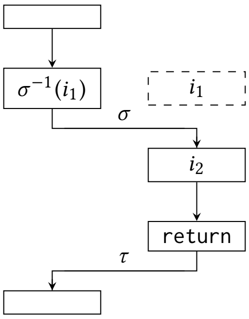


Figure 9: Situation after one application of rule 3.

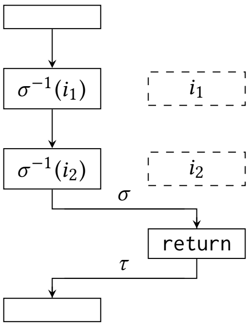


Figure 10: Situation after two applications of rule 3.

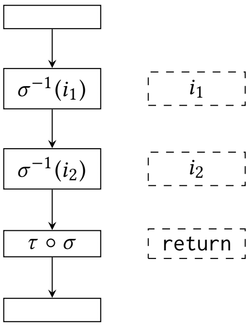


Figure 11: Situation after an applications of rule 2.

multiple entry points on the optimization techniques that we have already implemented, as well as on any optimization techniques that we plan to incorporate in the future.

- In our intermediate code, we treat multiple values with an unknown number of values as a special type of datum. It is special in that it must store an arbitrary number (unknown at compile time) of values. During the execution of our inlining procedure, such a datum may become part of the mapping between variables of the called function and the calling function. When the inlining procedure continues until termination, such a datum will be handled in the calling function in the same way that it is handled in the called function. However, if the inlining procedure is stopped with such a datum in the mapping, we would somehow need to transmit it as an argument to the called function. Doing so may require costly allocation of temporary memory and costly tests for the number of values that would not be required when the procedure continues until termination. However, it is rare that code needs to store intermediate multiple values. It only happens in a few cases such as when `multiple-value-prog1` is used. Therefore, one solution to this problem is to avoid inlining in this case. Another possible solution is to let the inlining procedure continue until termination for these cases.

As presented in this paper, our technique handles only functions with very simple lambda lists. It is probably not worth the effort to attempt to inline functions with lambda lists containing keyword arguments, but it might be useful to be able to handle optional arguments. We intend to generalize our technique to such lambda lists.

We have implemented the technique described in this paper, but have yet to implement a decision procedure for determining whether this technique could and should be applied. The details of this decision procedure are currently being investigated.

5 ACKNOWLEDGMENTS

We would like to thank Bart Botta, Jan Moringen, John Mercouris, and Alastair Bridgewater for providing valuable feedback on early versions of this paper.

REFERENCES

[1] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 134–145, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. doi: 10.1145/258915.258928. URL http://doi.acm.org/10.1145/258915.258928.

[2] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. *SIGPLAN Not.*, 32(5):134–145, May 1997. ISSN 0362-1340. doi: 10.1145/258916.258928. URL http://doi.acm.org/10.1145/258916.258928.

[3] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming Language Design and Implementation*, PLDI '89, pages 246–257, New York, NY, USA, 1989. ACM. ISBN 0-89791-306-X. doi: 10.1145/73141.74840. URL http://doi.acm.org/10.1145/73141.74840.

[4] P. P. Chang and W.-W. Hwu. Inline function expansion for compiling c programs. *SIGPLAN Not.*, 24(7):246–257, June 1989. ISSN 0362-1340. doi: 10.1145/74818.74840. URL http://doi.acm.org/10.1145/74818.74840.

[5] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5.

[6] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction Mechanisms in CLU. *Commun. ACM*, 20(8):564–576, August 1977. ISSN 0001-0782. doi: 10.1145/359763.359789. URL http://doi.acm.org/10.1145/359763.359789.

[7] Steven S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997. ISBN 1-55860-320-4.

[8] Robert W. Scheifler. An analysis of inline substitution for a structured programming language. *Commun. ACM*, 20(9):647–654, September 1977. ISSN 0001-0782. doi: 10.1145/359810.359830. URL http://doi.acm.org/10.1145/359810.359830.