Proceedings of the 8th European Lisp Symposium

Goldsmiths, University of London, April 20-21, 2015

Julian Padget (ed.)

Sponsors

We gratefully acknowledge the support given to the 8th European Lisp Symposium by the following sponsors:

Organization

Programme Committee

Julian Padget – University of Bath, UK (chair)

Giuseppe Attardi — University of Pisa, Italy
Sacha Chua — Toronto, Canada
Stephen Eglen — University of Cambridge, UK
Marc Feeley — University of Montreal, Canada
Matthew Flatt — University of Utah, USA
Rainer Joswig — Hamburg, Germany
Nick Levine — RavenPack, Spain
Henry Lieberman — MIT, USA
Christian Queinnec — University Pierre et Marie Curie, Paris 6, France
Robert Strandh — University of Bordeaux, France
Edmund Weitz — University of Applied Sciences, Hamburg, Germany

Local Organization

Christophe Rhodes – Goldsmiths, University of London, UK (chair)

Richard Lewis – Goldsmiths, University of London, UK
Shivi Hotwani – Goldsmiths, University of London, UK
Didier Verna – EPITA Research and Development Laboratory, France

Contents

Demonstrations

Message from the Programme Chair

Welcome to the 8th edition of the European Lisp Symposium!

We have a delightfully broad range of topics in the papers selected this year from systems, through domain-specific and visual languages to a wide variety of novel applications. This variety is also reflected in the invited talks, with an exploration of embedding a declarative computational model in Lisp, how to build (and maintain!) feature-rich Lisp systems and two talks focussing on the ever-present challenges arising from memory management.

My thanks to the programme committee not only for the detailed reviews, but their responsiveness and timeliness in handling the workload. I am particularly grateful to the local organizers – Christophe and Didier – for their guidance and support at every stage in the process, from planning the timeline, to distributing calls and providing organizational memory. Previous programme chairs, most especially Kent Pitman of ELS 2014, also provided helpful advice along the way. Finally, thanks to the authors, to all those that submitted contributions, and to you the participants for reminding us of the vibrancy of the community and keeping the Lisp flame alive.

Julian Padget, Bath, April 2015

Message from the Organizing Chair

Welcome to Goldsmiths!

Goldsmiths is situated in New Cross, Deptford in South-East London, which is perhaps historically most notorious for being where the playwright (and spy?) Christopher Marlowe met his end in a drunken dispute (or was it an assassination?). Goldsmiths was founded as a Technical and Recreative Institute in 1891, becoming part of the University of London in 1904 and receiving a Royal Charter in 1990. Today, Goldsmiths through its reaching and research serves London – fostering local artists and entrepreneurs – and the world – sharing the insights of research, working with businesses, charities, arts organizations and government.

Organizing a symposium is a substantial undertaking: too substantial for one person. I would like to thank our external sponsors for their generous support: Clozure Associates, EPITA, Franz Inc., Google and Lispworks Ltd. all help make an event like this possible. Julian Padget gamely accepted the challenge of organizing the programme, which he proceeded to accomplish in a marvellously unflappable way. Didier Verna handled the website, mass announcements, and payments; his voice of experience was soothing at stressful times. Shivi Hotwani and the Conference Services team at Goldsmiths were most helpful, coming up with ideas to fulfil our hosting duties, and modifying them to meet new constraints. Thanks also to the symposium's steering committee, who provided advice when it was needed, and encouraged me to offer Goldsmiths as the location for this year's event: I hope that their trust placed in me is vindicated, and that that you enjoy your time at the 8th European Lisp Symposium.

Christophe Rhodes, London, April 2015

Invited contributions

Quicklisp
On Beyond Beta

Zach Beane
Clozure Associates
`http://clozure.com/`

Quicklisp was released in 2010 as a public beta. Five years later, it's still in beta. How has Quicklisp (and Common Lisp) evolved in the past five years? What will it take for Quicklisp to go on beyond beta?

μKanren: Running the Little Things Backwards

Bodil Stokke
http://bodil.org/

Relational programming, or logic programming, is a programming paradigm that exhibits remarkable and powerful properties, to the extent that its implementation seems frightfully daunting to the layman. μKanren is a minimal relational language that seeks to strip the paradigm down to its core, leaving us with a succinct, elegant and above all simple set of primitives on top of which we can rebuild even the most powerful relational constructs.

In this talk, we will explore the μKanren language by implementing it from first principles in a simple functional programming language, going on to demonstrate how you can assemble these simple building blocks into a semblance of its richer parent, miniKanren, and maybe solve a logic puzzle or two to make sure it's working as advertised.

The μKanren paper, and the original μKanren implementation, were authored by Jason Hemann and Daniel P. Friedman.

Escaping the Heap

Ahmon Dancy
Franz Inc.
http://www.franz.com/

Common Lisp implementations provide great automatic memory management of data structures. These data structures are allocated from a memory area called the "heap". However, there are times when heap allocation is inadequate to satisfy the needs of the application. For example, sometimes data structures need to be persistent or shareable amongst separate processes. In these cases, alternatives to using the heap must be considered.

In this talk we will explore the motivations for out-of-heap data structures. We will discuss some of the out-of-heap data structures that we've created in the course of developing our database product, such as lists, hash tables, and arrays. We will describe the tools and mechanisms that we used to implement them, including memory-mapped files, foreign structs, aligned pointers and direct memory accesses. Finally we will discuss the downsides of out-of-heap data structures and the constant struggle between abstractions and performance.

Unwanted Memory Retention

Martin Cracauer
Google, Inc.
http://www.google.com/

This talk goes over numerous oddities in a Lisp-based system which led to unwanted heap memory retention and to constant resident memory growth over the uptime of the system. Issues covered include a mostly conservative but also paged garbage collector, the difficulty of clearing out data structures that are retainted as an optimization but that might hold on to large amounts of heap (and how that happens in C++, too) and how large intercollected and theoretically uprooted "clouds of heap debris" interact with stale pointers out of same. The most delicious pieces center around pointer staleness out of the saved (on-disk, but read-write mapped) part of the heap, which is not garbage collected, into anonymous memory backed heap and how you can create rootless but uncollected and "untraceable" object circles. Untraceable until you hack up the GC to help you…

Peer-reviewed papers

Efficient Applicative Programming Environments for Computer Vision Applications

Integration and Use of the VIGRA Library in Racket

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

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

ABSTRACT

Modern software development approaches, like agile software engineering, require adequate tools and languages to support the development in a clearly structured way. At best, they shall provide a steep learning curve as well as interactive development environments. In the field of computer vision, there is a major interest for both, general research and education e.g. of undergraduate students. Here, one often has to choose between understandable but comparably slow applicative programming languages, like Racket and fast but unintuitive imperative languages, like C/C++. In this paper we present a system, which combines the best of each approaches with respect to common tasks in computer vision, the applicative language Racket and the VIGRA C++ library. This approach is based on a similar Common Lisp module and has already proven to be adequate for research andr education purposes [12]. Moreover, it provides the basis for many further interesting applications. For this paper we demonstrate the use in one research and one educational case study. We also make suggestions with respect to the design and the needs of such a module, which may be helpful for the generic extension of applicative programming languages into other research areas as well.

Categories and Subject Descriptors

D.1.1 [**Programming Techniques**]: Applicative (Functional) Programming—*Allegro Common Lisp, SBCL, Racket*;
D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces, software libraries*;
I.4.8 [**Image Processing and computer vision**]: Scene Analysis

General Terms

Applicative Programming, Racket, Language Interoperability, Computer Vision, Image Processing

1. INTRODUCTION

Although applicative programming languages have a long tradition, they still do not belong to the scrap heap. Instead, they have proven to support state-of-the-art development approaches by means of an interactive development cycle, genericity and simplicity. The influence of applicative programming paradigms is even observable in modern languages, like Python, Dart and Go. However, there are some research areas, which are computationally of high costs and are thus currently less supported by applicative programming languages.

In this paper, we select the research field of computer vision and show how to connect applicative languages to a generic C++ computer vision library called VIGRA [5]. The interoperability is achieved by two similar modules, VIGRACL for Allegro and SBCL Common Lisp and VIGRACKET for Racket. Both are using a multi-layer architecture with a common C wrapper library. In contrast to [12], where we describe the architecture on the C/C++ and the Common Lisp side in more detail, we focus on the Racket extension in this paper. We also present useful applicative programming language additions for a seamless Racket integration.

Although C++ can lead to very efficient implementations, it is not capable of interactive modeling. Applicative programming languages like Lisp and derivatives on the other hand support symbolic processing and thus symbolic reasoning at a higher abstraction levels than typical imperative languages. Common Lisp has e.g. proven to be adequate for solving AI problems since decades. There are extensions for Lisp like e.g. description logics, which support the processes of computer vision and image understanding. Thus, the integration of computer vision algorithms has the potential to result in a homogenous applicative programming environment. Besides research tasks, the steep learning curve makes applicative programming languages interesting for educational purpose. To demonstrate this, we present two case studies for the application of the VIGRACKET module. The research case study shows the implementation of a state-of-the-art image processing algorithm. The other case study shows the use in an educational context by means of interpreting a board game from its image. The second case study has been performed with undergraduate students of computer science at the University of Hamburg.

2. RELATED WORK

The name VIGRA stands for "Vision with Generic Algorithms". Its main emphasis is on customizable generic algorithms and data structures (see [6], [7]). This allows an easy adaption of any VIGRA component to the special needs of computer vision developers without losing speed efficiency (see [5]). The VIGRA library was invented and firstly implemented by Dr. Ullrich Köthe as a part of his PhD thesis. Meanwhile, many people are involved to improve and extend the library. Moreover, the library is currently being used for various educational and research tasks in German Universities (e.g. Hamburg and Heidelberg) and has proven to be a reliable testbed for low-level computer vision tasks. The VIGRA library follows a current trend in computer vision. Instead of providing large (overloaded) libraries, smaller, generic and theoretically founded building blocks are defined and provided. These building blocks may then be connected, combined, extended and applied for each unique low-level computer vision problem. To demonstrate the need of an interactive and dynamic way of using this building block metaphor, VIGRA comes with (highly specialized) interactive Python-bindings. Other computer vision software assist the user by visualizing the building blocks metaphor and really let the user stack components together visually, e.g. MEVISLab [8].

Despite this trending topic, there are currently no competitive computer vision libraries for interactive development with Racket. For Common Lisp, few systems are still existing, but most of them are no longer maintained. Referring to the survey of Common Lisp computer vision systems in [12], beside VIGRACL [10] currently only one system "opticl" (the successor of ch-image) is still maintained [4]. In contrast to other systems, our proposed module is generic, light-weight, and offers advanced functionality like image segmentation or sub-pixel based image analysis.

Our aim is a generic interface to the VIGRA library, that allows the use of many other languages and programming styles. This generic approach is reflected in the programming languages (Racket, SBCL and Allegro Common Lisp) as well as in the platform availability (Windows, Linux or Mac). The only requirement on the "high-level" programming language is, that a foreign function interface needs to be existent. The interface has to support shared memory access of C-pointers and value passing. These interfaces, Common Lisp UFFI or Racket FFI, in conjunction with a C wrapper library called VIGRA_C [11], yield computationally demanding tasks to the compiled wrapper library. Figure 1 illustrates the scheme of each applicative library. Besides the Common Lisp and Racket bindings, we provide bindings for other interactive development environments, too.

3. DESIGN OF VIGRACKET

We will now present the embedding of the VIGRA algorithms into Racket. In contrast to [12], we will not discuss the multi-layer design of the extension, but focus on the specific extensions, which are needed for Racket and the use of the module in this language. Since Racket already has extended capabilities in visualizing and displaying graphics, we will also give an overview of the seamless integration of VIGRACKET [9] into Racket's GUI.

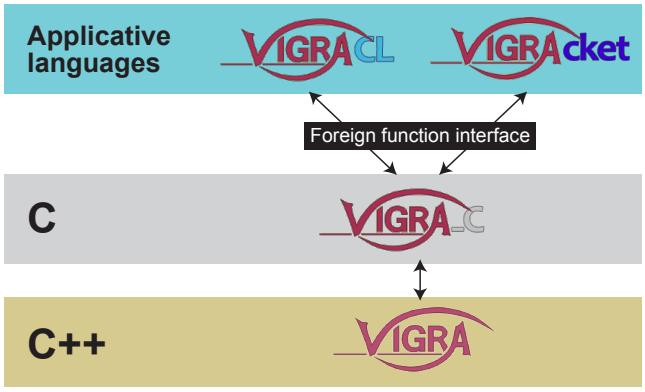


Figure 1: Schematic view of the connection between applicative languages and C++ by using FFI and a common C-library as applicative abstraction layer.

3.1 Data Structures for Images

At the lower layer of the VIGRACKET library, we need to select an appropriate data type for images. Since we assume digital images to be two-dimensional grid-aligned data, a two-dimensional array data structure is the first choice. Additionally, the array type needs to have access to a shared C-ordered memory space, where the computations of the image processing algorithms are carried out by means of the VIGRA_C wrapper library. Unlike Common Lisp, where the built-in 2d arrays are mostly capable of using a shared address space (cf. [12]), Racket provides only 1d arrays with shared memory: `cvector`. Thus, in a first step, we extend these `cvectors` to n dimensions, and rename this newly defined type `carray`. We also provide fast copying operations for `carrays`, constructors, accessors and a list conversion.

Since the VIGRACKET should be able to analyze multi-band color images, we construct an image based on the introduced data type as `carrays` of type `float`:

- For single-band (gray value) images:
  `(list gray_carray)`
- For multi-band images images:
  `(list ch1_carray ch2_carray ... chN_carray)`

This implementation favors genericity by means of a fast single-band access, but may yield in slower simultaneous access of one pixel's band values. For RGB-images, we get:

```
'(RED_carray GREEN_carray BLUE_carray)
```

The interface functions use the FFI included in Racket to pass the arguments as well as the `carray`'s pointers to the VIGRA_C library. The C wrapper mostly implements band-wise operations, which can be identified by the suffix `-band`. To work on images of any number of bands, most image processing functions use the `map` function to apply a band-defined operation on each band of an image. For instance, the gaussian smoothing of an image is defined of the gaussian smoothing of all the image's bands:

```
(define (gsmooth image sigma)
  (map (curryr gsmooth-band sigma) image))
```

3.2 Applicative Extensions

Besides the new data types and the interaction of the library with the VIGRA_C library through the Racket FFI, we provide a set of generic and flexible high order functions, which assist in the practical development of computer vision algorithms. These functions refer to both images and image bands. They can be seen as extensions to the well-known high-order functions, but tailored to the data types of images and image bands.

The first set of functions corresponds to the `map` function for lists. We define `array-map`, `array-map!`, `image-map` and `image-map!` for this purpose. These functions may be used to apply a function to one or more images (bands) to generate a result image. The functions with a bang at the end of the function name override the first given image instead of allocating new memory for the resulting image. Although this saves memory, it introduces side-effects and should only be used carefully. An an example for the applicative variants without side-effects, the absolute difference of two images may be computed by:

```
(image-map  (compose abs -) img1 img2)
```

We also define folding operations for bands and images: `array-reduce` and `image-reduce`. These functions can be used to derive single values from bands or a list of values for images, like the maximum intensity of an image:

```
(apply max (image-reduce max 0 img))
```

Here, the inner term derives the band-wise maximum by applying the maximum function to each band array. Since the result is a list of maximal values, we get the overall maximum by applying the `max` function to the result.

Further, we introduce image traversal functions, which further support the development of own algorithms:

- `array-for-each-index`,
- `image-for-each-index` and
- `image-for-each-pixel`.

These functions call any given function of correct signature at at position of the array or image. The signature is specific for each function. Examples of the use of these functions are given in the second case study.

3.3 Racket-specific Extensions

Unlike other applicative languages, Racket was designed to be an easy to learn beginner's language. One way to support the learning of a language is to learn programming by design (see [3]). To support the drawing and other GUI functionality, Racket offers a variety of different packages. However, the main package for the applicative programming of shapes, image creation and drawing is still `2hdtp/image`.

Since the VIGRACKET should benefit from the existing drawing capabilities, we provide conversion functions for the different formats of the `2hdtp/image` module (for 1-band gray- and 3-band RGB-images).

These conversion functions are defined as:

- `image->racket-image`
  for the conversion from shared memory to `2hdtp/image`,
- `racket-image->image`
  for the conversion from `2hdtp/image` to shared memory.

This conversion is often necessary, since it allows to present the (processed) image without saving it to disk. However, Racket's native interface only allows to read from or to write to a device context. This interface results in very slow conversions for moderate and large image sizes. In order to enhance the execution speed, we re-implemented this conversion at machine level on the shared memory VIGRA_C side. Instead of drawing onto a device context, we rearrange the list of `carrays` for display purpose to the byte pattern, which is needed for construction of an object of class `%bitmap` (ARGB order). Compiled in machine-code, this is performed within milliseconds. This allows us to switch from one side to the other whenever needed without notable delays.

3.4 Preliminaries and Automatic Installation

To make the VIGRACKET accessible to a wide range of researchers as well as teachers and (undergraduate) students, only few preliminaries exist. The Racket module has already proven to run stable on Windows, Linux and Mac OS under 32- and 64-bits, depending on the Racket version installed.

Since Windows is missing a powerful package manager, the necessary binaries are bundled inside the installation package, and thus no further preliminaries exist. Although this is a very efficient and simple approach, it is not favored for Mac OS and Linux, since they provide powerful package managers, like dpkg, rpm or macports [13]. Thus, under Mac OS and Linux, you need to have a C++ compiler installed as well as a current version of the VIGRA library. Note, that the Racket installation must be of the same architecture as the VIGRA installation (32bit or 64bit). After checking this, the only preliminary is, that the "vigra-config" script must be accessible from within your shell.

If all preliminaries are met, the VIGRACKET installation package needs to be unzipped and the "install.rkt" needs to be executed. This calls the automatic installation routine of the VIGRACKET module. This routine looks up the OS and the environment and builds the VIGRA_C wrapper library for Mac OS and Linux or copies the correct binaries for Windows. A typical output is:

```
Searching for vigra using 'vigra-config ...
-------------- BUILDING VIGRA-C-WRAPPER ...
cd bin && gcc -I/src `vigra-config --cpp ...
gcc `vigra-config --libs` `vigra-config ...
```

Afterwards, all needed files and the created or copied shared object or dynamic linked library of the VIGRA_C wrapper are copied into the user's collects directory. It may then be used like any other Racket module by calling:

```
(require vigracket)
```

The provided demos in "examples.rkt" may help in getting a first impression of this module.

Figure 2: Resulting images of the coherence enhancing shock filter using the parameters: $\sigma = 6$, $\rho = 2$, $h = 0.3$. From left to lower right: original image, result after 5, 10, and 20 iterations.

4. CASE STUDIES

To demonstrate the use of the VIGRACKET module, we choose two different scenarios: the first case study describes the use by means of implementing a state-of-the-art algorithm for anisotropic image diffusion. The second case study is an example of an undergraduate task during a Bachelor practice at the University of Hamburg.

4.1 Coherence Enhancing Shock Filter

In [14], Weickert et al. describe a coherence enhancing shock filters. Shock filters are a special class of diffusion filters, which correspond to morphological operations on images. They apply either a dilation or an erosion, depending on the local gray value configurations. Weickert et al. refer to these configurations as "influence zones", which either correspond to a minimum or a maximum of the image function. The aim of the filter is to create a sharp boundary (shock) between these influence zones. The main idea of [14] is to use the Structure Tensor approach [1] for determining the orientation of this flow field instead of the explicitly modeling the partial differential equations of the diffusion equation. The Structure Tensor at scale σ of an image is defined by:

$$ST_\sigma(I) = G_\sigma * \begin{pmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{pmatrix} \quad (1)$$

where I is the image function and I_x and I_y are the partial derivatives in x- and y-direction. G_σ is a Gaussian at scale σ and $*$ is the convolution operation. To derive the main direction, in which the filter should act, the eigenvalues and eigenvectors of the Structure Tensor need to be computed. Additionally the Hessian matrix of the second partial derivatives of the image needs to be computed:

$$H(I) = \begin{pmatrix} I_{xx} & I_{xy} \\ I_{xy} & I_{yy} \end{pmatrix} \quad (2)$$

According to [14], the following equation indicates whether a pixel will be eroded or dilated:

$$D = c^2 I_{xx} + 2cs I_{xy} + s^2 I_{yy} \quad (3)$$

where $w = (c, s)^T$ denotes the normalized dominant eigenvector of the Structure Tensor of I. The sign of $D(I)$ is then been used in a morphological upwinding scheme to determine if an erosion or a dilation has to be performed. This approach is designed in an iterative way. Fortunately, many of the mathematical operations are already included in the VIGRA and in the VIGRACKET module.

Besides the parameter σ, we need an additional parameter ρ to define the inner derivative of the Structure Tensor as well as a parameter h, which controls the intensity of the morphological operations. Thus, the implementation of the shock filter begins with:

```
(define (shock-image image sigma rho h iter)
  (if (= iter 0)
      image
```

This ensures that the image is given back after the last iteration. Otherwise the filtering needs to be performed. Thus, the following lines mainly show a straight-forward application of the former equations:

```
(let*
  ((st   (structuretensor image sigma rho))
   (te   (tensoreigenrepresentation st))
   (hm   (hessianmatrixofgaussian image sigma))
   (ev_x (image-map cos (third img_st_te)))
   (ev_y (image-map sin (third img_st_te)))
```

Note that the third element of the `tensoreigenrepresentation` function is the angle of the largest eigenvector. To derive the image D of Eq. 3, we use the `image-map` function:

```
   (d (image-map (lambda (c s I_xx I_xy I_yy)
                    (+ (* c c I_xx)
                       (* 2 c s I_xy)
                       (* s s I_yy)))
                 ev_x                    ; <= c
                 ev_y                    ; <= s
                 (first hm)              ; <= I_xx
                 (second hm)             ; <= I_xy
                 (third hm))))           ; <= I_yy
```

The resulting image D can now be used as the sign image in the unwinding morphological operation. Since this operation was not originally included in the VIGRA, it has been implemented by means of the VIGRA_C wrapper library and a corresponding Racket function was implemented, too. Thus, we end up with the recursive scheme for the last function call:

```
  (shock-image (upwindimage image d h)
               sigma rho h (- iter 1)))))
```

The results of the application of this filter to the famous Lenna image are shown in Fig. 2.

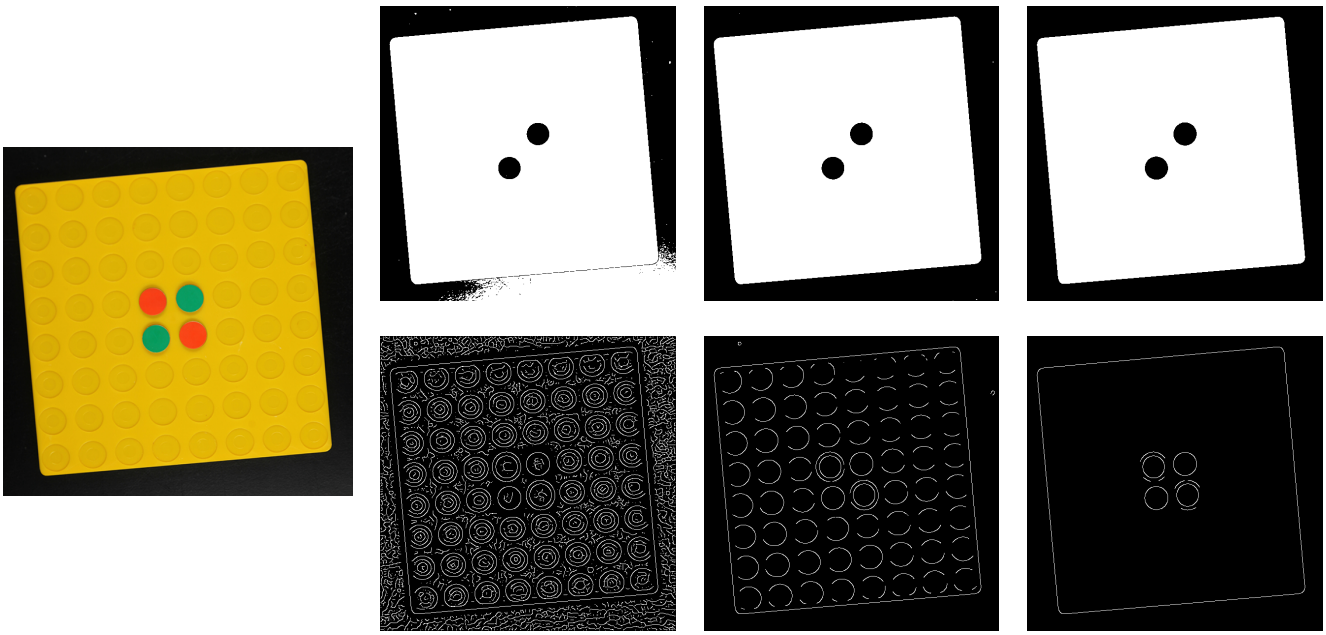


Figure 3: Detection of the game board from the image (left). Upper row, from left to right: thresholding results for $t = 25$, $t = 50$ and $t = 100$. Lower row, from left to right: results for the Canny algorithm at scale $\sigma = 3$ with (edge) thresholds $t = 0$, $t = 1$ and $t = 2$.

4.2 Board Game: Reversi

For the second case study, we selected the task of an undergraduate student practice. The results were achieved by a team of students during one term (13×4 hrs) in a Bachelor practice. The participants had few experience in applicative programming an no knowledge in image processing or computer vision. However, the aim of this practice is to teach both, applicative programming as well as computer vision at once. After the first two weeks, where the students participate in guided exercises, they select a board game. The selected game will be photographed at some states for each team. To pass the practice, the students have to perform the following tasks using DrRacket and VIGRACKET:

1. Retrieve the game state from the image,
2. Write a GUI to visualize the game state,
3. Implement the game logic and
4. Extend the GUI to continue the game.

Since we focus on the VIGRACKET integration in this paper, we present the retrieval of a game state from the image. The game is Reversi (also known as Othello), which can be considered as a prototypical example for this task. The retrieval may further be divided into the separation of the board from the background and the derivation of the game state from the sub-image.

Figure 3 (left) shows a typical image of the initial state. The board is slightly rotated and comparably brighter than the dark background. To determine the bounding box (Fig. 4, left) of the board, the boundaries between board and background need to be estimated.

One naive approach is to classify each pixel by its gray value (intensity), e.g. at the red band. This thresholding may be expressed by the following function:

```
(define (threshold v t)
  (if (< v t) 0.0 255.0))
```

To apply the threshold for $t = 50$ to the red band of `img`, we can use the `image-map` function:

```
(define img_gray (image->red img))

(define thresh_img50
  (image-map (curryr threshold 50)
             img_gray))
```

The application of different thresholds yields different results, which are shown in Fig. 3 (upper row). Note that the missing of the green game token does not influence the detection of the outer boundaries of the game board.

Another possibility is to use edge detection, since the strongest edges in the image may correspond to the boundaries of the game board. Here, we decided to use the Canny edge detector [2] at a scale of $\sigma = 3$ using various thresholds. To apply the detector for $t = 2$ to the red band of `img` and mark each edge pixel by 255.0, we can call the Canny implementation of the VIGRA directly:

```
(define canny_img2
  (cannyedgeimage img_gray 3.0 2.0 255.0)
```

The application of the Canny algorithm at different thresholds yields different results, which are shown in Fig. 3 (lower row). At low thresholds many edges are detected due to the image noise but vanish at a threshold of $t = 2$.

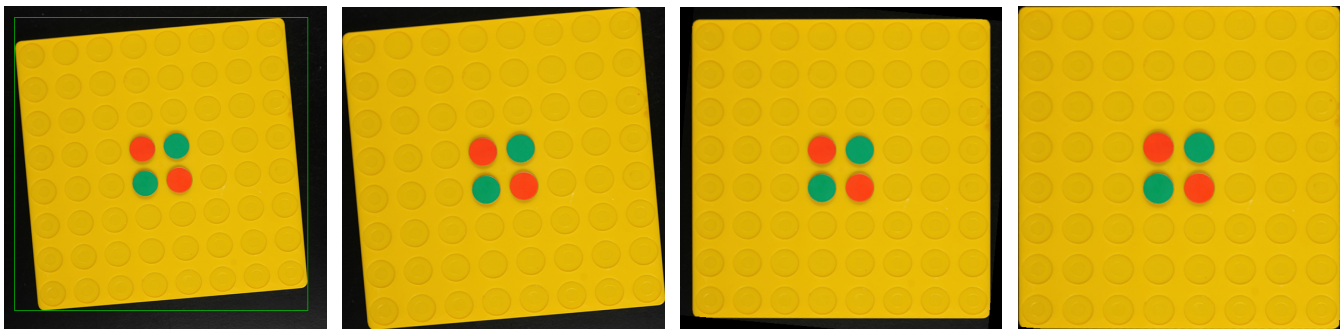


Figure 4: Extraction of the game board from the image. From left to right: estimated bounding box (green), cropped image, rotation corrected image, cropped rotation corrected image.

The next step is to derive the bounding box of the game board from either the threshold or the canny resulting image. We first define a bounding box as a four element vector (left, upper, right, and lower position). Then the derivation of the box at an image position ca be expressed as:

```
(define (findBBox x y col bbox)
  (when (>  (car col) 0.0)
    (begin
      (when (> x (vector-ref bbox 2))
            (vector-set! bbox 2 x))
      (when (< x (vector-ref bbox 0))
            (vector-set! bbox 0 x))
      (when (> y (vector-ref bbox 3))
            (vector-set! bbox 3 y))
      (when (< y (vector-ref bbox 1))
            (vector-set! bbox 1 y)))))
```

The initial state of the box depends on the update scheme of the above function and is given by:

```
(define bbox
  (vector (image-width img) (image-height img)
          0 0))
```

We can now use the iteration/traversal framework of the VIGRACKET to iterate over each pixel of the canny image and call the `findBBox` function at each coordinate `x`, `y` with the color list `col` at that position:

```
(image-for-each-pixel
              (curryr findBBox bbox)
              canny_img2)
```

This updates the bounding box to contain the minimal and maximal coordinates. For this example, we use the results of the canny approach to proceed with the next steps. Fig. 4 (left) shows the procedure after the determination of the first bounding box.

The next step it to crop, the image according to the bounding box `bbox`. Since VIGRACKET does not provide such a functionality, we use the Racket interface and import the `2htp/image` module. However, to resolve name conflicts, we have to restrict the `require` command by:

```
(require
  (rename-in 2htdp/image
             (save-image save-plt-image)
             (image-width plt-image-width)
             (image-height plt-image-height)))
```

After importing this module, we use the Racket's `crop` function to perform the cropping:

```
(define (cropimage img ul_x ul_y lr_x lr_y)
  (let ((w (- lr_x ul_x))
        (h (- lr_y ul_y)))
    (racket-image->image
      (crop ul_x ul_y w h
        (image->racket-image img)))))

(define cropped_img
  (cropimage img
             (vector-ref bbox 0)
             (vector-ref bbox 1)
             (vector-ref bbox 2)
             (vector-ref bbox 3)))
```

The cropped image is shown in Fig. 4 (second image). One can observe that the image is still not adequately cropped, since it is rotated around the new image center. To estimate this rotation, we search for the position of leftmost and rightmost marked pixel at the first line of the box of the edge image using a helper function:

```
(define (findFirstPixelInRow img x1 x2 row)
  (let ((intensity
         (apply max (image-ref img x1 row))))
    (if (= intensity 0)
        (if (= x1 x2)
            #f
            (findFirstPixelInRow
              img
              (+ x1 (sgn (- x2 x1))) x2 row))
        x1)))
```

To get the leftmost position relative to the beginning of the bounding box, we call the function and pass the extracted positions as parameters:

```
(define canny_left
  (- (findFirstPixelInRow canny_img2
                          (vector-ref bbox 0)
                          (vector-ref bbox 2)
                          (vector-ref bbox 1))
     (vector-ref bbox 0)))
```

To find the rightmost marked pixel, the second and third parameter of the `findFirstPixelInRow` need to be swapped. Both positions are then used to compute the arithmetic mean position on the first line.

Finally, due to the rounded borders of the game board, a constant correction factor is added. Let `pos` be the corrected position, `bbox_width` be the width of the bounding box. Then the rotation angle can be derived as:

```
(define angle
    (/ (* (atan  (- bbox-width pos) pos) 180)
       pi))
```

To correct the rotation of the cropped image, we use the rotation function provided by the VIGRACKET module. Its arguments are the angle (in degrees) and the degree of the interpolation function used (here: bilinear):

```
(define croped-rotated
   (rotateimage cropped_img (- angle) 1))
```

The result of this rotation correction is shown in Fig. 4 (third image). To crop this image, to get the final result, we repeat the former steps. Thus the edge image has to be cropped and rotated in the same manner as the image of the game board:

```
(define cropped_canny2
  (cropimage canny_img2
             (vector-ref bbox 0)
             (vector-ref bbox 1)
             (vector-ref bbox 2)
             (vector-ref bbox 3)))

(define cropped-rotated_canny2
   (rotateimage cropped_canny2 angle 1))
```

After these operations, the bounding box has to estimated again to crop the rotated image. This may be written as:

```
(define bbox2
  (vector (image-width  cropped-rotated_canny2)
          (image-height cropped-rotated_canny2)
          0 0))

(image-for-each-pixel  (curryr findBBox bbox2)
                       cropped-rotated_canny2)

(define cropped_img2
  (cropimage cropped-rotated
             (vector-ref bbox2 0)
             (vector-ref bbox2 1)
             (vector-ref bbox2 2)
             (vector-ref bbox2 3)))
```

The result of this application, the image `cropped_img2`, is shown in the rightmost image of Fig. 4. It does only contain the game board. The next step is to extract the game state from this cropped image. This is done by sampling the image at the center positions of every possible token location. Since there are 8×8 fields, the extraction of a color value for a place $x, y \in \{0, 1...7\}$ is performed by:

```
(define dx (/ (image-width  cropped_img2) 8))
(define dy (/ (image-height cropped_img2) 8))

(define (board_pos->color image x y)
  (image-ref image
    (inexact->exact
        (round (+ (/ dx 2) (* x dx))))
    (inexact->exact
        (round (+ (/ dy 2) (* y dy))))))
```

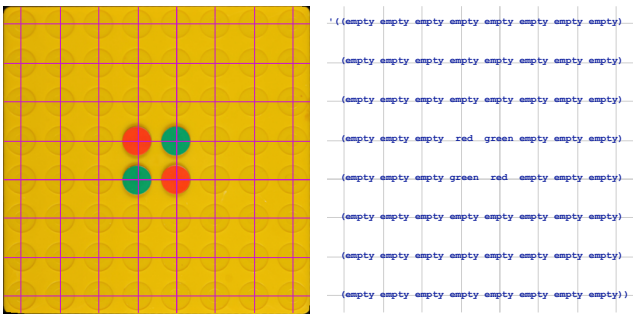


Figure 5: Sampling of the cropped image using an equally spaced rectangular grid. Left: grid superimposed to the cropped image, right: the resulting symbolic representation superimposed on the grid.

The sampling scheme is depicted in Fig. 5 (left). Depending on the extracted color, a classification to the tokens red, green or empty has to be made. If we assume, that red dominates the intensity for a red token, green dominates the intensity for green tokens but is comparably darker, we may express this as:

```
(define (classify-color col)
  (let* ((val (/ (apply + col) (length col))))
      (if (> (first col) (* 2 val))
            'red
            (if (> (second col) (* 1.5 val))
                  'green
                  'empty))))
```

Using these functions, we can now retrieve a list of lists to represent the game state from the image using two nested recursions, one for the rows and one for the columns of the locations of the tokens:

```
(define (board_rows image x y)
  (if (> y 7) '()
      (cons (board_cols image x y)
            (board_rows image x (add1 y)))))

(define (board_cols image x y)
  (if (> x 7) '()
      (cons (classify-color
                  (board_pos->item image x y))
            (board_cols image (add1 x) y))))

(define state (board_rows cropped_img2 0 0))
```

Since we have 8 positions per row and 8 rows, we end up with a list of 64 items in total. The resulting list is shown in Fig. 5 (right). Note that this is only one possible way of classification. Alternatively, one could switch from RGB into a HSV (hue, saturation, value) colorspace for classification.

Based on the extracted game state, the team members implemented a graphical user interface using the world framework, which is part of Racket's `2htdp/universe` module and wrote the game logic. As a result, they are able to continue the game, which was captured in the picture. Although this requires the knowledge of modeling graphical user interfaces and game logics, it can be implemented without using the VIGRACKET module.

5. CONCLUSIONS

We have motivated the need for interactive development methods in the field of computer vision w.r.t. research and education purposes. For many years, applicative programming has been used to solve higher AI tasks. But with a computer vision extension of such languages like Racket or Common Lisp, we are now able to offer a homogeneous, general, and highly interactive environment.

As a demonstration, we presented some of some functionalities of the VIGRACKET module in research and educational contexts. Since the extension uses a multi-layer architecture to grant access to the computer vision algorithms that are provided by the VIGRA library, a corresponding VIGRACL module (tested with Allegro Common Lisp and SBCL) is also available. We demonstrated the efficiency of the module in different aspects:

- An intuitive interface to images using shared memory,
- Generic approaches of the VIGRA, which provide great fundamental building blocks,
- High-order functions which support the development of own algorithms and
- A seamless integration into Racket by means of the built-in GUI and data types.

We have shown that the integrated high-order functions for images are really helpful in practice, since they provide wrappers for common tasks, like mapping a function on a complete image or traversing over an image in a clearly defined way. These functions in conjunction with the generic approach of the VIGRA and the easy automated installation routine make the VIGRACKET a valuable tool, not just for researchers but for teachers, too.

At the University of Hamburg, we use the VIGRACKET module as well as the VIGRACL module for research to experiment with low-level image processing tasks that have to be performed before the symbolic interpretation of the image's content.

The VIGRACKET module is also used and improved on a regular basis for a Bachelor practice at the University of Hamburg. Here, the steep learning curve and interactive experience with images and algorithms helps the undergraduate students to learn applicative programming in conjunction with computer vision during a single term. In the educational context, we found that the use of applicative programming encourages the students to understand the underlying algorithms better when compared with typical imperative low-level languages like C. This yields to an increased overall interest in computer vision.

This interest gaining of students has already resulted in many excellent Bachelor theses. In the last four years, a total of over 100 students successfully passed the practice and rated it A+. The only drawback, which has been mentioned by the students, is the limited performance of computer vision algorithms written in pure Racket.

It needs to be mentioned that, although the examples in chapter 4 where realistic, this paper cannot be more than an introduction into the interesting field of computer vision. Additionally, only a very small subset of the functionality of VIGRACKET module has been shown here. However, the examples clearly demonstrate how easy the functions of the VIGRA can be used within Racket or Common Lisp by means of the generic common VIGRA_C interface.

6. REFERENCES

[1] J. Bigün, G. Granlund, and J. Wiklund. Multidimensional orientation estimation with applications to texture analysis and optical flow. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(8):775 –790, aug 1991.

[2] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986.

[3] M. Felleisen. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, 2001.

[4] C. Harmon. opticl - an image processing library for common lisp: `https://github.com/slyrus/opticl`.

[5] U. Köthe. The vigra homepage: `https://github.com/ukoethe/vigra`.

[6] U. Köthe. *Reusable Software in Computer Vision*, pages 103–132. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[7] U. Köthe. *Generische Programmierung für die Bildverarbeitung*. Books on Demand GmbH, 2000.

[8] F. Ritter, T. Boskamp, A. Homeyer, H. Laue, M. Schwier, F. Link, and H.-O. Peitgen. Medical image analysis. *Pulse, IEEE*, 2(6):60–70, Nov 2011.

[9] B. Seppke. The vigracket homepage: `https://github.com/bseppke/vigracket`.

[10] B. Seppke. The vigracl homepage: `https://github.com/bseppke/vigracl`.

[11] B. Seppke. The vigra_c homepage: `https://github.com/bseppke/vigra_c`.

[12] B. Seppke and L. Dreschler-Fischer. Tutorial: Computer vision with allegro common lisp and the vigra library using vigracl. In *Proceedings of the 3rd European Lisp Symposium*, 2010.

[13] The MacPorts team. The macports project: `https://www.macports.org`.

[14] J. Weickert. Coherence-enhancing shock filters. In *Lecture Notes in Computer Science*, pages 1–8. Springer, 2003.

Keyboard? How quaint. Visual Dataflow Implemented in Lisp.

Donald Fisk
Barnet
Herts
England
hibou@onetel.com

ABSTRACT
Full Metal Jacket is a general-purpose, homoiconic, strongly-typed, pure visual dataflow language, in which functions are represented as directed acyclic graphs. It is implemented in Emblem, a bytecode-interpreted dialect of Lisp similar to, but simpler than, Common Lisp. Functions in Full Metal Jacket can call functions in Emblem, and vice-versa. After a brief language description, this paper describes the interpreter in detail, how iteration is handled, how the editor handles type checking interactively, and how it detects race conditions.

Categories and Subject Descriptors
D.3.2 [**Programming Languages**]: Language Classifications—*Data-flow languages*

General Terms
Languages

Keywords
Language design, dataflow, visual programming, Lisp

1. INTRODUCTION
In recent years, many new programming languages (e.g. GoLang, Swift, Rust, Hack, etc.) have been released. As none of these represents a radical departure from already existing languages, it is understandable that the announcement of the development of yet another programming language might be greeted with a degree of skepticism.

However, Full Metal Jacket *is* fundamentally different from any mainstream programming language (including Lisp), in both its outward appearance and its internal computational model. Instead of being expressed in text, its programs are directed acyclic graphs drawn on a two-dimensional canvas, and computations are run, whenever possible, in parallel, with no central flow control. It is strongly typed yet it does not have any variables, and has iteration but does not have any loops. Programmers do not have to learn "yet another syntax": its syntax is not much more complex than Lisp's, and it has an integrated editor which not only is simple to use, but also prevents syntax and type errors. Editing is mostly done with the mouse, by dragging and dropping program elements, and then joining them up. Early work on Full Metal Jacket is described in [5].

Work on Full Metal Jacket only recently resumed after a long break. In the meantime, the underlying Lisp dialect, Emblem, has, however, undergone substantial improvements which have made Full Metal Jacket implementation more straightforward. These include better event handling, X11 graphics programming, and object orientation. Changes to Emblem since [5] have, in general, moved it closer to Common Lisp, except that the object system has been simplified.

At present, Full Metal Jacket is interpreted. Ideally, it would compile onto a dataflow machine if a suitable one existed. (A prototype was built at the University of Manchester [6]. Others were designed by Kableshkov [9] and Papadopoulos [12].) Alternatively, it could be compiled onto a more conventional architecture, such as X86 or X86-64. Full Metal Jacket does go one step further than Lisp in facilitating compilation: not only is the parsing step trivial, optimizations based upon dataflow are also made more straightforward.

When a program runs, values can be thought of as flowing downwards along the edges which connect vertices, where they are transformed. Some vertices contain nested enclosures, in which data flows from ingates, down through vertices, towards outgates.

The intention is for the language to interoperate with Lisp, rather than to replace it. There is some evidence [8] that, although dataflow excels at coarse-grained parallelism, its high overhead makes it less suitable for parallelism at the instruction level.

In this article, following some examples of simple programs, the interpreter is described in some detail. Iteration, type inference, and race condition detection are also covered.

2. RELATED WORK
Full Metal Jacket is not the only visual dataflow language. Three others have been developed which have seen widespread use, namely Prograph [2] which is general-purpose, MAX/MSP [3] [4], which is designed for music and multimedia, and LabVIEW [10] which is designed for programming instruments and devices. Another system, Plumber [1], has been implemented in Lisp. [8] contains a recent survey of many such languages.

There are significant differences between Prograph and Full Metal Jacket: in Prograph, type checking is not done until run time; Prograph places more emphasis on object orientation; *then* and *else* clauses (and cases) are in separate windows; there is a 'syncro' edge which enforces execution

order; and garbage collection is by reference counting.

LabVIEW uses two separate windows: a front panel for user interface objects (these are equivalent to Full Metal Jacket's constants), and a block diagram for code. It also requires extra constructs for iteration and conditional code.

Max/MSP differs from Full Metal Jacket in that it is explicitly object-oriented (with dataflow as message passing); allows feedback; does not comfortably support recursion; execution order is sensitive to program layout; and triggers are sometimes needed to guarantee execution order. Of these three systems, Max/MSP is the least similar to Full Metal Jacket.

3. A BRIEF DESCRIPTION OF THE LANGUAGE

The simplest program, shown in Figure 1, displays the constant "Hello, world!" in a dialog box.

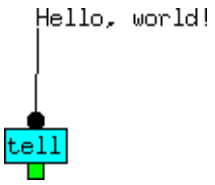


Figure 1: Hello, world!

In the program shown in Figure 2, $2.5sin(0.6) + 5.4$ is calculated.

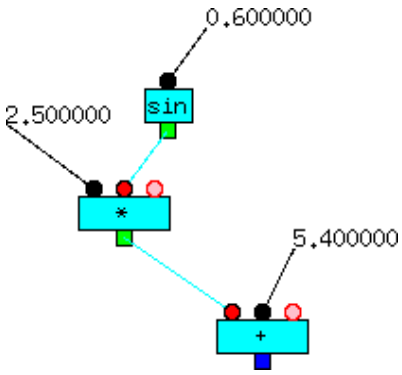


Figure 2: A simple calculation

The dark blue square indicates that the result of + should be output to a dialog box. Vertex inputs are shown as circles and are usually red, and vertex outputs are are shown as squares and are usually green. The edge colors are configurable, and give some indication of type. Here, *cyan* ≡ *Real*.

The third inputs of `*` and `+` are called extra inputs. An extra input allows additional arguments to be added by clicking on it, and can also be used like Common Lisp's `&rest` argument, in which case it will be the head of an edge.

Figure 3 shows the code for `myAppend`, the canonical recursive append. The surrounding square is an enclosure. Ingates are at the top and outgates are at the bottom.

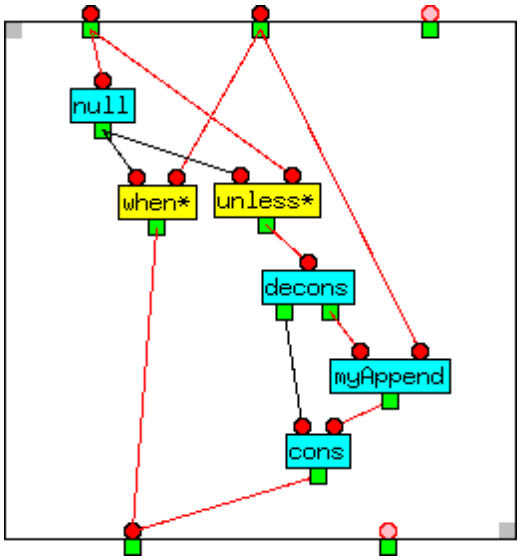


Figure 3: Recursive append.

`when*` outputs its second argument if its first is `T`, otherwise it does not output anything. `unless*` does the opposite. So, if `myAppend`'s first argument is `NIL`, `when*` outputs `myAppend`'s second argument. Otherwise, `unless*` outputs `myAppend`'s first argument, and `decons` splits it into its *car* and *cdr*. The *cdr* and `myAppend`'s second argument then become the inputs to `myAppend`, which is called recursively. The *car* and the result of the recursive call are then *cons*ed and returned.

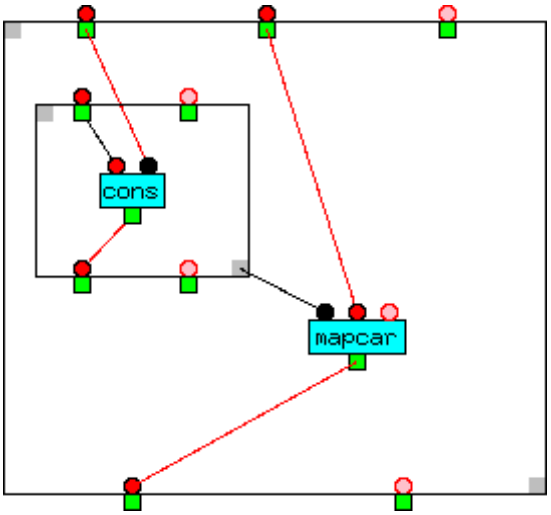


Figure 4: Use of a constant enclosure.

Figure 4 shows the code for `foo`, and is the equivalent of the Lisp

```
(defun foo (x y) (mapcar (lambda (z) (cons z x)) y))
```

The inner enclosure is a constant of the first argument of `mapcar`, and is called as often as the length of `foo`'s (and

`mapcar`'s) second argument, yet it captures `foo`'s first argument precisely once. To make it available for more than one call, it has been made sticky, which is why, like inputs with constants, it is shown in black. Sticky values behave like constants, but change when a new value reaches their input.

4. THE INTERPRETER

A simplified version of the interpreter code, written in Emblem, follows.

4.1 Interpreter Data Structures

Full Metal Jacket has four fundamental classes of objects: Vertex, Constant, Edge, and Enclosure. three other classes: Input, Output, and Gate, are used in constructing them.

A vertex has inputs, a function, and outputs. When a vertex has values with the same tag[1] on all of its inputs, it applies its function to those values. The values returned by the function are then sent with the same tag from the outputs.

```
(defclass Vertex (Any)
  (function type Fun accessor functionOfVertex)
  (inputs type List accessor inputsOfVertex)
  (outputs type List accessor outputsOfVertex))

(defclass Socket (Any)
  (edges type List accessor edgesOfSocket initForm '()))
```

An input usually has a queue of tagged values, but may instead have a sticky value, which is not consumed when the vertex's function is called, but changes when a different value is received, unless it is a constant.

```
(defclass Input (Socket)
  (vertexOrGate type Any accessor vertexOrGateOfInput)
  (taggedValueQueue type Queue accessor taggedValueQueueOfInput)
  (stickyValue type Any accessor stickyValueOfInput)
  (hasStickyValue type Bool accessor inputHasStickyValueP
                  initForm NIL))

(defclass Output (Socket))
```

A constant is a value attached to an vertex's input, which does not change between function calls.

```
(defclass Constant (Any)
  (input type Input accessor inputOfConstant))
```

An edge connects an output (its tail) to an input (its head). Values notionally flow along edges from output to input.

```
(defclass Edge (Any)
  (output type Output accessor outputOfEdge)
  (input type Input accessor inputOfEdge))
```

Enclosures, which are an exact analogue of Lisp's lambdas, are used in defining new functions, and also used within functions, to provide local scope. Each enclosure has ingates and outgates. Within an enclosure, the outputs of ingates are connected to inputs of vertices, and outputs of vertices are connected to the inputs of other vertices, or the inputs of outgates.

```
(defclass Enclosure (Any)
  (ingates type List accessor ingatesOfEnclosure
           initForm '())
  (outgates type List accessor outgatesOfEnclosure
            initForm '())
  (tag type Int accessor tagOfEnclosure initForm 0))
```

Ingates and outgates are gates.

```
(defclass Gate (Any)
  (enclosure type Enclosure accessor enclosureOfGate)
  (input type Input accessor inputOfGate)
  (output type Output accessor outputOfGate))
```

4.2 Interpreter Code

Vertices are executed asynchronously.

As soon as a vertex is ready to be executed by `executeVertex`, it is added to the task queue. In the current system, tasks are run by `runNextTask` in the order they appear on it.

```
(setf TASK_QUEUE (new Queue))


(defmacro makeTask (vertex tag args) `(list ,vertex ,tag ,args))


(alias vertexOfTask car)
(alias tagOfTask cadr)
(alias argsOfTask caddr)

(defun runNextTask ()
  (let ((task (takeOffQueue TASK_QUEUE)))
    (if task
        (executeVertex (vertexOfTask task)
                       (tagOfTask task)
                       (argsOfTask task))
      (write "TASK_QUEUE is empty!" $))))

(defun executeVertex (vertex tag argList)
  (do ((args (mvList (apply (functionOfVertex vertex)
                            argList))
             (cdr args))
       (outputs (outputsOfVertex vertex) (cdr outputs)))
      ((null outputs))
      (do ((edges (edgesOfSocket (car outputs))
                  (cdr edges)))
          ((null edges))
          (sendValueToInput (inputOfEdge (car edges))
                            tag
                            (car args)))))
```

Values must be tagged.

Values intended as arguments of a vertex's function are sent to its inputs asynchronously, and possibly out of order, by `sendValueToInput`, so it is essential to distinguish which values belong to which invocation of the function. This is achieved by accompanying each value intended for the same invocation with the same unique tag, which can be an integer. Because tagged values have to be queued at the input if

[1]Tags, which are used to distinguish different computations which share the same vertex, are explained in detail in Subsection 4.2.

the vertex is not ready to receive them, each input requires a queue for holding them. When every input is found by `everyInputHasAValueP` to have a value with the same tag, the vertex is ready to be executed. Then `extractValuesFromInputs` gets the input values, and `putOnQueue` adds a task to the task queue.

```
(defun sendValueToInput (inputOfDest tag value)
  ;; If the destination input has a sticky value, change it
  ;; to the new value.
  (if (inputHasStickyValueP inputOfDest)
      (setf (stickyValueOfInput inputOfDest) value)
    ;; Otherwise, tag the value and add it to end of
    ;; the input's tagged value queue.
    (putOnQueue (taggedValueQueueOfInput inputOfDest)
                (cons tag value)))
  ;; If the destination input belongs to a vertex,
  ;; rather than a gate, and it has values for all inputs
  ;; with the tag, schedule the vertex to be run
  ;; with those inputs.
  (let ((dest (vertexOrGateOfInput inputOfDest)))
    (when (and (instanceOf dest Vertex)
               (everyInputHasAValueP (inputsOfVertex dest)
                                     tag))
      (putOnQueue TASK_QUEUE
                  (makeTask dest
                            tag
                            (extractValuesFromInputs
                             (inputsOfVertex dest)
                             tag))))))

(defun everyInputHasAValueP (inputs tag)
  (every (lambda (input)
           (or (and (eq (classOf input) ExtraInput)
                    (null (edgesOfSocket input)))
               (inputHasStickyValueP input)
               (assoc tag
                      (elemsOfQueue (taggedValueQueueOfInput
                                     input)))))
         inputs))

;;; This should only be called after verifying
;;; that the values are actually present.   The use of mapcan
;;; permits the use of an extra arg.
(defun extractValuesFromInputs (inputs tag)
  (mapcan (lambda (input)
            (if (inputHasStickyValueP input)
                (list (stickyValueOfInput input))
              (let ((taggedValue
                     (assoc tag (taggedValueQueueOfInput input))))
                (cond (taggedValue
                       (deleteFromQueue (taggedValueQueueOfInput
                                         input)
                                        taggedValue)
                       (list (cdr taggedValue)))
                      ;; If there is no tagged or sticky value,
                      ;; no value should be extracted.
                      (T NIL)))))
          inputs))
```

Arguments must be imported into enclosures synchronously.

It might be thought that, if a vertex's function is defined as an enclosure, it would be possible to import individual input values into the enclosure through their corresponding ingate as soon as they arrive at the calling vertex, without waiting for the other argument values to arrive. However, in general, it is not possible, as when the function is recursively defined, such a value could be transmitted immediately to a vertex with the same enclosure as its definition. This value would then be imported into the enclosure via the same ingate and arrive at the vertex again, and again, and again. For example, this would occur in `myAppend` (Figure 3). Therefore, it is necessary to wait until all the arguments for a given invocation, and therefore with the same tag, are available before startng the enclosure call, although this might delay execution at a few vertices. As it also has to be done when vertex's function is encoded in Lisp, the same mechanism can be used in both cases.

```
(defun importArgsIntoEnclosure (enclosure tag args)
  (mapc (lambda (arg ingate)
          (do ((edges (edgesOfSocket (outputOfGate ingate))
                      (cdr edges)))
              ((null edges))
              (sendValueToInput (inputOfEdge (car edges))
                                tag
                                arg)))
        args
        (ingatesOfEnclosure enclosure)))
```

The tags used in an enclosure are a property of the enclosure and are unique to the enclosure's invocation.

An invocation results in data flowing through an enclosure, or outside any enclosure in the case of top-level computations (i.e. those taking place in the sandbox, Full Metal Jacket's equivalent of Lisp's `read-eval-print` loop.) Each invocation has its own tag for use in that enclosure (or sandbox), supplied on entry.

While values are flowing through an enclosure during a particular invocation, they must all have the same tag, unique to the invocation, in order for the vertices they encounter to execute each time with the correct input data.

During an invocation, the same function might be called from different vertices, each resulting in a different invocation of the same enclosure. In order to keep the computations for the different invocations separate, their values must be provided with different tags when the enclosure is entered, for use during their time in the enclosure. When the enclosure is exited and the value(s) returned to the calling vertex, the original tag on the data received by the vertex's inputs is restored.

As computations within different enclosures are physically separate, there is no problem if they occasionally have the same tag provided that tags are properties of their enclosure.

In `applyEnclosure`, each argument is given a new tag, unique to the invocation, then transmitted along each edge leading from its corresponding ingate.

`applyEnclosure` then waits until a value with the same tag appears at each outgate, before returning those values.

The values returned are then transmitted along each edge leading from the corresponding vertex output.

```
(defun applyEnclosure (enclosure args)
  (let ((newTag (incf (tagOfEnclosure enclosure))))
    (importArgsIntoEnclosure enclosure newTag args)
    ;; Wait for the values to appear at each outgate.
    (do ((inputs (mapcar inputOfGate
                         (outgatesOfEnclosure enclosure))))
        ((everyInputHasAValueP inputs newTag)
         (valuesList (extractValuesFromInputs inputs
                                              newTag)))
        ;; Not there?   Find something else to do.
        (runNextTask))))
```

Functions defined as enclosures in Full Metal Jacket can be called from Lisp.

When a tagged value is sent to a vertex's input, it is put on its tagged value queue. Then, if all the inputs either have an attached constant, or a value in their queue with the same tag, those values are removed from their respective queues to comprise, along with any constants, the argument list. The vertex's function is then applied to the argument list, by calling `apply` if the function was written in Lisp.

If the function was implemented as an enclosure, `applyEnclosure`, Full Metal Jacket's analogue of `apply`, is called instead. The arguments to `applyEnclosure` are the enclosure and the enclosure's inputs.

For example, the definition in Emblem of `myAppend`, generated automatically when the enclosure is saved, is

```
(defun myAppend (x y)
  (applyEnclosure (get 'myAppend 'enclosure) (list x y)))
```

5. ITERATION

`when*` and `unless*` are unusual because, unlike functions, they do not always output a value exactly once when invoked. This suggests that the function concept might be generalized to include, in addition to ordinary functions, not only boolean operators like `when*` and `unless*`, but also emitters, which can output more than once per invocation, and collectors, which can accept inputs more than once before outputting.

Figure 5 illustrates the function `iterReverse`, which reverses its first argument onto its second. If its inputs are `'(a b c)` and `NIL`, the values at various stages of the computation are shown in Table 1.

Output of `strip*`	2nd Input	3rd Input of `collectUntil*`	Accumulator	Output
`(a b c)`	`a`	`NIL`	`NIL` `(a)`	
`(b c)`	`b`	`NIL`	`(b a)`	
`(c)`	`c`	`T`	`(c b a)`	`(c b a)`

Table 1: Iteration in `iterReverse`

Emitters and collectors typically work in tandem, with emitters sending values and collectors receiving those values, or data computed from them, and accumulating them in some manner, until a boolean input changes value, at which point they output their result. Here, `strip*` repeatedly sends a list, stripping off one element at a time, and, after initializing its accumulator to `NIL`, `collectUntil*` receives the `car` of each of these lists, and `cons`es it onto its accumulator (a local storage area), outputting its value after `strip*` has sent all its outputs.

As tagged values are queued, when an emitter is connected to a collector, the collector will process values in the order they were sent by the emitter. This ensures that there is no problem in the collector repeatedly receiving values with the same tag, provided care is taken that it receives the same number of values on its other inputs, except when they are sticky.

The emitters and collectors available to the programmer are still, at present, not settled, and it will require further experimentation before a final selection can be made. One option is to make them compatible with Waters's Series or Curtis's Generators and Gatherers [13], to which they bear some similarity.

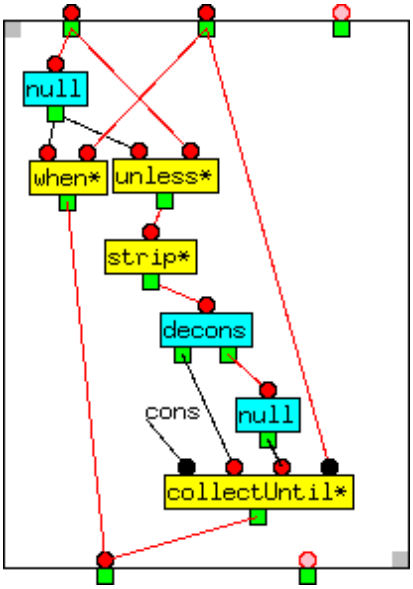


Figure 5: Iterative Reverse.

6. TYPES AND TYPE INFERENCE

Like ML and Haskell, Full Metal Jacket has a Hindley-Milner type system [7] [11], but types are inferred incrementally, and entirely interactively, when edges and constants are added or deleted in the editor, in the course of function definition.

At present (but see subsection 8.1), types in Full Metal Jacket must be defined in Emblem, and are not yet fully integrated into Emblem's object system. Arguments and return values of functions in Emblem, and therefore inputs and outputs of vertices in Full Metal Jacket, are typed.

Composite types can be parameterized, allowing their element types to be specified or inferred, e.g. a list of unknown items can be declared as `(List ?x)` and a list of integers as `(List Int)`.

Examples of type definitions are:

```
(deftype (List ?x) (or NIL (Pair ?x (List ?x))))
(deftype (AList ?x ?y) (List (Pair ?x ?y)))
(deftype (Bag ?x) (AList ?x Int))
(deftype (TaggedValueQueue ?x) (Queue (Pair Int ?x)))
```

The types of inputs and outputs are displayed in Full Metal Jacket's editor when the pointer is placed over them. Whenever an attempt is made to connect an output, or add a

constant, to an input, the types are checked, and unless the output's type matches the input's type, the programmer is prevented from making the connexion or adding the constant. Successful matches can result in type variables being bound (by means of a process similar to Prolog's unification), or types made more specific. Type matching and inference occur while the program is being edited, every time an edge is added, and also when one is deleted, in which case the type variable might again become undetermined.

6.1 Type Matching Algorithm

The basic type-matching algorithm matches an edge's output type to its input type, resulting in either a list of bindings for the variables contained in the types if the match is successful, or the symbol FAIL.

The cases are:

A. Match of two type constants. This results in either success (`NIL`) if the output type is equal to or a subtype of the input type, or failure (`FAIL`) otherwise.

B. Match with a type variable. The match results in a new binding unless the match is with the same variable, in which case `NIL` is returned.

C. Match of two function types. The match is then applied to the argument and return types.

D. Match with the same parameterized type. The parameters are matched recursively.

E. Match with a different parameterized type. If one is a subtype of the other, the match is repeated using the more specific type's definition, after the appropriate variable substitutions. Otherwise, the match fails.

Case	Output type	Input type	Bindings
A	`List`	`Str`	`FAIL`
A	`Str`	`Str`	`NIL`
B	`Str`	`?x`	`?x` → `Str`
B	`(List ?x)`	`?y`	`?y` → `(List ?x)`
B	`?x`	`?x`	`NIL`
B	`?x`	`?y`	`?y` → `?x`
D	`(List Int)`	`(List ?x)`	`?x` → `Int`
E	`(Bag Sym)`	`(List (Pair ?y ?x))`	`?x` → `Int` `?y` → `Sym`

Table 2: Type matching examples

The output of the matcher is the list of bindings for the type variables of the edge's output and input. (See Table 2.)

6.2 Type Inference

Type inference is performed whenever an edge or constant is connected to an output, as follows:

A. If there are any type variables shared between an edge's output and input, unique variables are substituted for them before matching. For example, if output type `(Bag ?x)` and input type `(List (Pair ?y ?x))` are matched, the type variables are renamed, giving (e.g.) `(Bag ?17)` and `(List (Pair ?18 ?19))`.

B. Matching is then performed. The bindings returned by the match algorithm described above are `?19` → `Int`, `?18` → `?17`.

C. The bindings for the output and input types are then separately extracted, giving `NIL` and `?19` → `Int`, `?18` → `?17` respectively.

D. Finally, the original names replace the substitute names, giving `NIL` and `?x` → `Int`, `?y` → `?x` respectively.

E. Type variables are shared among a vertex's input and output types, so if one becomes bound to a particular value on an input or output, it also becomes bound to the same value on all other inputs and outputs on the same vertex.

F. When the type of an input or output changes, type inference is applied along the edges connected to them.

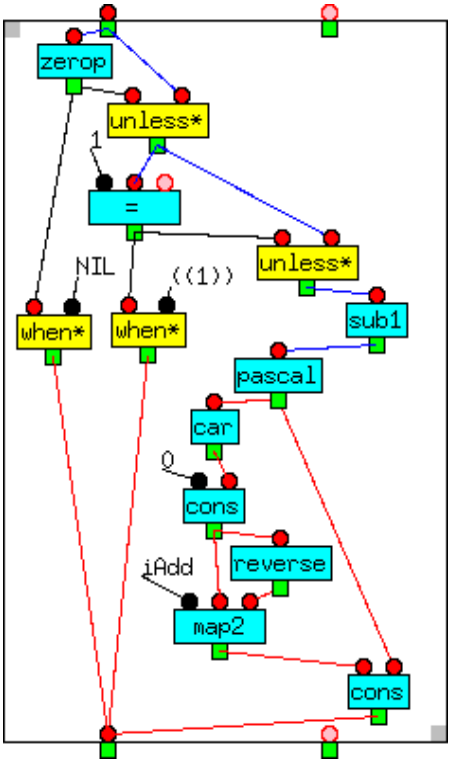


Figure 6: Code for Pascal's triangle.

6.3 An Example

`pascal`, shown in Figure 6, computes Pascal's triangle. For example, `(pascal 5)` returns `((1 4 6 4 1) (1 3 3 1) (1 2 1) (1 1) (1))`.

`pascal` takes an `Int` as input. Here, the types of the values are inferred statically.

Given the vertex input and output types shown in Table 3, the edge types to be matched will be those shown in Table 4. Together with the types of the constants (shown in Table

5), the types of all the variables can be inferred, and this is shown in Table 6.

Input Types	Function	Output Types
`Int`	`pascal`	`?a`
`(List ?b)`	`car`	`?b`
`?c` `(List ?c)`	$\texttt{cons}_a$	`(List ?c)`
`(List ?d)`	`reverse`	`(List ?d)`

`(?e ?f)` $\rightarrow$ `(?g)`
`(List ?e)`
`(List ?f)`
	`map2`	`(List ?g)`

`?h` `(List ?h)`	$\texttt{cons}_b$	`(List ?h)`

Table 3: Vertex types in `pascal`

Function	Output Type	Input Type	Function
`pascal`	`?a`	`(List ?b)`	`car`
`car`	`?b`	`(List ?c)`	$\texttt{cons}_a$
$\texttt{cons}_a$	`(List ?c)`	`(List ?d)`	`reverse`
$\texttt{cons}_a$	`(List ?c)`	`(List ?e)`	`map2`
`reverse`	`(List ?d)`	`(List ?f)`	`map2`
`map2`	`(List ?g)`	`?h`	$\texttt{cons}_b$
`pascal`	`?a`	`(List ?h)`	$\texttt{cons}_b$

Table 4: Edge types in `pascal`

Value	Type	Input Type	Fn.
`0`	`Int`	`?c`	$\texttt{cons}_a$
`iAdd`	`(Int Int)` $\rightarrow$ `(Int)`	`(?e ?f)` $\rightarrow$ `(?g)`	map2

Table 5: Constant types in `pascal`

Type Variable	Type
`?a`	`(List (List Int))`
`?b`	`(List Int)`
`?c`	`Int`
`?d`	`Int`
`?e`	`Int`
`?f`	`Int`
`?g`	`Int`
`?h`	`(List Int)`

Table 6: Inferred types in `pascal`

This, reassuringly, is consistent with the types returned in the other two paths through the `pascal` function, which return `NIL` and `'((1))`.

7. RACE CONDITION DETECTION

Race conditions occur when two or more values are written to the same memory location. The final value then depends on the order in which they are written, which is unsatisfactory. In Full Metal Jacket, if more than one edge is connected to the same input, a potential race condition occurs. Whether it is a true race condition can be detected fairly straightforwardly.

Data follows one out of one or more mutually exclusive *streams* through an enclosure. The stream containing a given vertex can be found by searching downstream from it along the edges leaving its outputs, and then back upstream at each vertex encountered, along *its* edges, marking the vertices as we go. All those vertices, and any edges connecting them, are then on the same stream. If data flow through a vertex, data also must be flowing through other vertices in the same stream on the same enclosure invocation. When two or more edges converge on the same input, and they are from vertices in the same stream, there is a race condition, and one of the edges should be disallowed.

The stream detection mechanism also solves a well-known problem with visual programming: how to remove clutter. Functions do not have to become very large before they become difficult to read, due to many edges crossing each other. Within the editor, clicking on a particular vertex hides all the vertices and edges except those in the same stream.

A further use for stream detection is in detecting gaps in unit test coverage.

8. CONCLUSIONS AND FUTURE WORK

Development of Full Metal Jacket is still incomplete. While it is already possible to implement and run some small programs containing nested function calls, others programs have been found to be difficult to implement without important features still absent from the language: in particular, the ability to extend the type system from within the language, and a more comprehensive set of emitters and collectors.

The ability to take advantage of the language's homoiconicity should also be added. Other features missing from Full Metal Jacket's environment include a compiler (initially, generating Emblem bytecode, for programmer-selected functions), and interactive debugging capabilities.

Learning to think in a dataflow language should not be considered a problem, but a worthwhile challenge, also present in any other programming paradigm switch.

8.1 Type and Class Hierarchy Extension

At present, the Emblem class hierarchy is displayed as a tree (Emblem has *single* inheritance), with each class displayed as a vertex, but this cannot yet be extended from inside Full Metal Jacket. Types and classes should ideally be merged, with the system capable of handling the two different ways of extending them: adding fields to objects, and abstracting and making types more specific. For example, the class hierarchy contains

`Any` $\rightarrow$ `Graphical` $\rightarrow$ `Shape` $\rightarrow$ `Rectangle`
`Any` $\rightarrow$ `Queue`

These are defined by adding extra fields. It should also be possible to define a type hierarchy, which would contain (see Section 6)

`Pair` $\rightarrow$ `List` $\rightarrow$ `AList` $\rightarrow$ `Bag`
`Queue` $\rightarrow$ `TaggedValueQueue`

The subtypes above share the underlying data structure of

their parent types, the `cons`-cell and `Queue` respectively, but a `List` requires its `cdr` also to be of type `List`, an `AList` requires all of its elements to be `Pair`s, and a `Bag` requires the `cdr` of each of its elements to be `Int`. Similarly, a tagged value queue is a `Queue` of `Pair`s, of which the `car` is an `Int`. (A `Queue` is an object containing a pointer to a list of elements, and a pointer to their last `cons`-cell. Alternatively, it could have been implemented by subtyping `Pair`.)

It will be noticed that Emblem's `typedef` macro (see Section 6) resembles Scheme's define macro when that is used to define functions, suggesting that types be defined graphically in Full Metal Jacket as enclosures, with more primitive types as vertices and type variables as gates. This hints at a deep equivalence between code and data.

8.2 Homoiconicity

Until now, Lisp and Prolog, and only those languages, have been meaningfully homoiconic, i.e. able to treat code written in them as data, transform it, and reason about it, and to treat data as code, and execute it. Lisp macros are the most widespread use of this, but more generally, code can be generated on the fly (data becomes code) or reasoned about (code becomes data).

A Full Metal Jacket program is a directed graph, which is the most general data structure. The intention is to exploit this feature, similar to how Lisp uses lists, and Prolog uses terms, to represent both programs and data. However, it is less straightforward: directed graph elements have a more complex structure, and are more specific to code. While this does not preclude the incorporation of *macros* into the language, use of the same structures for *data* might seem less natural. Vertices, edges and enclosures can, however, be generalized to objects without the program-specific fields (such as tagged value queues and types), and then specialized by restricting inputs and outputs each to one. This has already been done with classes, so the class hierarchy can be displayed as a tree.

9. REFERENCES

[1] Seika Abe. Plumber - A Higher Order Data Flow Visual Programming Language in Lisp *International Lisp Conference*, 2012.

[2] P.T. Cox. Prograph: a step towards liberating programming from textual conditioning. In *Proceedings of the 1989 IEEE Workshop on Visual Programming*, 1989.

[3] Peter Elsea. Max and Programming (July 2007) Retrieved 3rd March 2015 from `http://peterelsea.com/Maxtuts_advanced/Max&Programming.pdf`

[4] Peter Elsea. Messages and Structure in Max Patches (February 2011) Retrieved 3rd March 2015 from `http://peterelsea.com/Maxtuts_advanced/Messages%20and%20Structure.pdf`

[5] Donald Fisk. Full Metal Jacket: A Pure Visual Dataflow Language Built on Top of Lisp. *International Lisp Conference*, 2003.

[6] J.R. Gurd, C.C. Kirkham, and I. Watson. The manchester prototype dataflow computer. *Communications of the ACM*, 28:34–52, 1985.

[7] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[8] W. M. Johnston, J.R. Paul Hanna, and R.J. Millar. Advances in Dataflow Programming Languages. *ACM Computing Surveys, Vol. 36, No. 1*, March 2004.

[9] S. O. Kableshkov. Anthropocentric Approach to Computing and Reactive Machines. *John Wiley and Sons Ltd*, 1983.

[10] J. Kodosky. Visual programming using structured dataflow. In *Proceedings of the IEEE Workshop on Visual Languages*, 1991.

[11] Milner, R. A Theory of Type Polymorphism in Programming *Journal of Computer and System Science*, 17:348–374, 1978.

[12] G. M. Papadopoulos. Implementation of a General Purpose Dataflow Multiprocessor. LCS TR-432. *PhD thesis, MIT, Laboratory for Computer Science*, August 1988.

[13] Guy Steele. *Common Lisp: The Language. Second Edition Digital Press*, 1990.

P2R

Implementation of Processing in Racket

Hugo Correia
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
hugo.f.correia@tecnico.ulisboa.pt

António Menezes Leitão
INESC-ID, Instituto Superior Técnico
Universidade de Lisboa
Rua Alves Redol 9
Lisboa, Portugal
antonio.menezes.leitao@ulisboa.tecnico.pt

ABSTRACT

Processing is a programming language and development environment created to teach programming in a visual context. In spite of its enormous success, Processing remains a niche language with limited applicability outside the visual realm. Moreover, architects that have learnt Processing are unable to use the language with traditional Computer-Aided Design (CAD) and Building Information Modelling (BIM) applications, as none support Processing.

In the last few years, the Rosetta project has implemented languages and APIs which enable programmers to work with multiple CAD applications. Rosetta is implemented on top of Racket and allows programs written in JavaScript, AutoLISP, Racket, and Python, to generate designs in different CAD applications, such as AutoCAD, Rhinoceros 3D, or Sketchup. Unfortunately, Rosetta does not support Processing and, thus, is not available to the large Processing community.

In this paper, we present an implementation of Processing for the Racket platform. Our implementation allows Processing to use Rosetta's APIs and, as a result, architects and designers can use Processing with their favourite CAD application. Our implementation involves compiling Processing code into semantically equivalent Racket source code, using a compiler pipeline composed of parsing, code analysis, and code generation phases. Processing's runtime is implemented purely in Racket, allowing for greater interoperability with Racket code.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors

General Terms

Languages

Keywords

Processing; Racket; Compilers; Language implementation

1. INTRODUCTION

Many programming languages have been created to solve specific needs across a wide range of areas of expertise. Processing [1] is a programming language and development environment created to teach programming in a visual context. The language has grown over the years, creating a community where users are encouraged to share their artistic works. As a result, a wide range of Processing examples are freely available, making it easier for anyone with little, or even no programming knowledge, to experiment with Processing. Among the many benefits that Processing offers, are a wide range of 2D and 3D drawing primitives, and a simple Integrated development environment (IDE), that provides a basic development environment to create new designs.

Despite its enormous success, Processing is a niche programming language with limited applicability outside the visual realm. Architects, for instance, depend on traditional heavyweight CAD and BIM applications (i.e. AutoCAD, Rhinoceros 3D, Revit, etc), which provide APIs that are tailored for that specific CAD. Unfortunately, no CAD application allows users to write scripts in Processing. Therefore, architects that have learnt Processing cannot use their programming knowledge or any of the publicly available examples to program for their favourite CAD tool.

On the other hand, Racket is a descendent of Scheme, that has a wide range of applications, such as teaching newcomers how to program, developing web applications, or creating new languages. Racket encourages developers to tailor their environment to project-specific needs, by offering an ecosystem that allows for the creation of new languages, having direct interoperability with DrRacket and existing Racket's libraries. For instance, Rosetta [2], is Generative Design tool built on top of Racket, that encompasses Racket's philosophy of using different languages to solve specific issues. Rosetta allows programmers to generate 2D and 3D geometry in a variety of CAD applications, namely AutoCAD, Rhinoceros3D, Sketchup, and Revit, using several programming languages, such as JavaScript, AutoLISP, Racket, and Python.

Our implementation enables Processing to use Rosetta, therefore allowing architects to prototype designs in their favourite CAD application, using Processing. Our implementation involves compiling Processing code to semantically equivalent Racket source code, using Rosetta's modelling primitives and abstractions. Furthermore, Racket allows us to take advantage of language creation mechanisms [3], that simplify the language development process and its integration with DrRacket. Also, as Racket is our target language, Processing developers gain access to Racket libraries and vice versa. Lastly, as Racket encourages developers to use different languages within the Racket ecosystem, Processing developers

could potentially combine their scripts with other languages, such as Python [4].

The following sections describe in greater detail the Processing language and other language implementations that are relevant to our work. Additionally, we describe the main design decisions that were taken for our implementation and a sample of the results obtained so far.

2. PROCESSING

Processing was developed at MIT media labs and was heavily inspired by the *Design by Numbers* [5] project, with the goal to teach computer science to artists and designers with no previous programming experience. The language has grown over the years with the support of an academic community, which has written several educational materials, demonstrating how programming can be used in the visual arts. Also, an online community[1] has been created around the language, allowing users to share and discuss their works. The existence of an online community, good documentation, and a wide range of publicly available examples, has been a positive factor for the language's growth over the years.

The Processing language is built on Java. It is statically typed sharing Java's C-style syntax and implements a wide range of Java's features. The decision of developing Processing as a Java "preprocessor", was due to Java being a mainstream language, used by a large community of programmers. Moreover, Java has a more forgiving development environment for beginners when comparing with other languages used in computer graphics such as C++.

As Processing is meant for beginners, several features were introduced to simplify Java, and consequently, allow users to quickly test their design ideas. In Java, developers have to implement a set of language constructs to develop a simple example, namely a public `class` that implements public `methods` and a static `main` method. These constructs only bring verbosity and complexity to the program. As a result, Processing simplifies this by removing the these requirements, allowing users to write scripts (i.e. simple sequences of statements) that produce designs.

Processing also introduces the notion of a *sketch*, a common metaphor in the visual arts, acting as a sort of project that artists can use to organize their source code. Within a *sketch*, artists can develop their designs in several Processing source files, but that are viewed as a single compilation unit. A *sketch* can operate in one of two distinct modes: *Static* or *Active mode*. *Static mode* supports simple Processing scripts, such as simple statements and expressions. However, the majority of Processing programs are in *Active mode*, which allow users to implement their *sketches* using more advanced features of the language. Essentially, if a function or method definition is present, the *sketch* is considered to be in *Active mode*. Within each *sketch*, Processing users can define two functions to aid their design process: the `setup()` and `draw()` functions. On one hand, the `setup()` function is called once when the program starts. Here the user can define the initial environment properties and execute initialization routines that are required to create the design. On the other hand, the `draw()` function runs after the `setup()` and executes the code that draws the design. The control flow is simple, first `setup()` is executed, setting-up the environment; followed by `draw()` called in loop, continually rendering the sketch until stopped by the user.

[1] `http://openprocessing.org/`

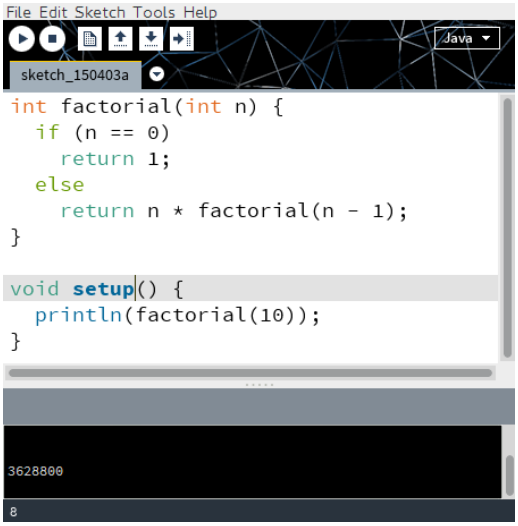


Figure 1: Processing Development Environment

Furthermore, Processing offers users a set of design and drawing tools that are specially tailored for visual artists, providing 2D and 3D drawing primitives along with different 2D and 3D rendering environments. Also, a set of built-in classes are provided, specifically tailored to help artists create their designs. For instance, the `PShape` class serves as a data type that enables users to easily create, manipulate, and reuse custom created design shapes throughout his *sketches*.

On top of being a programming language, Processing offers its users a development environment (presented in Fig 1) called PDE (Processing Development Environment). Users can develop their programs using this simple and straightforward environment, which is equipped with a tabbed editor and IDE services such as syntax highlighting and code formatting. Moreover, Processing users can create custom libraries and tools that extend the PDE with additional functionality, such as, Networking, PDF rendering support, color pickers, sketch archivers, etc.

3. RELATED WORK

Several different language implementations were analysed to guide our development. Our focus was on different implementations for the Processing environment, namely Processing.js, Ruby-processing, and Processing.py. Additionally, we analysed ProfessorJ due to the similarities that Java shares with Processing, and that Scheme shares with Racket.

3.1 Processing.js

Processing.js [6] is a JavaScript implementation of Processing for the web that enables developers to create scripts in Processing or JavaScript. Using Processing.js, developers

can use Processing's approach to design 2D and 3D geometry in a HTML5 compatible browser. Processing.js uses a custom-purpose JavaScript parser, that parses both Processing and JavaScript code, translating Processing code to JavaScript while leaving JavaScript code unmodified.

Moreover, Processing.js implements Processing drawing primitives and built-in classes directly in JavaScript. Therefore, greater interoperability is allowed between both languages, as Processing code is seamlessly integrated with JavaScript and Processing's data types are directly implemented in JavaScript. To render Processing scripts in a browser, Processing.js uses the HTML canvas element to provide 2D geometry, and *WebGL* to implement 3D geometry. Processing.js encourages users to develop their scripts in Processing's development environment, and then render them in a web browser. Additionally, Sketchpad[2] is an alternative online IDE for Processing.js, that allows users to create and test their design ideas online and share them with the community.

3.2 Ruby-processing & Processing.Py

Ruby-processing[3] and Processing.py[4] produce Processing as target code. Both Ruby and Python have language implementations for the JVM, allowing them to directly use Processing's drawing primitives. Processing.py takes advantage of Jython to translate Python code to Java, while Ruby-processing uses JRuby to provide a Ruby wrapper for Processing. Processing.py is fully integrated within Processing's development environment as a language mode, and therefore provides an identical development experience to users. On the other hand, Ruby-processing is lacking in this aspect, by not having a custom IDE. However, Ruby-processing offers *sketch* watching (code is automatically run when new changes are saved) and live coding, which are functionalities that are not present in any other implementation.

3.3 ProfessorJ

ProfessorJ [7, 8] was developed to be a language extension for DrScheme [9], providing a smoother learning curve for students that are learning Java and offering a set of language levels that progressively cover more complex notions of the language.

ProfessorJ implements a traditional compiler pipeline, that starts with a `lex` and `yacc` parsing phase, that produces an intermediate representation in Scheme. Subsequently, the translated code is analysed, generating target Scheme code by using custom defined functions and macro transformations. ProfessorJ implements several strategies to map Java code to Scheme. For instance, Java classes are translated into Scheme classes with certain caveats, such as implementing static methods as Scheme procedures or by changing Scheme's object creation to appropriately handle Java constructors. Also, Java has multiple namespaces while scheme has a single namespace, therefore name mangling techniques were implemented to correctly support Java's multiple namespaces in Scheme.

[2] `http://sketchpad.cc/`
[3] `https://github.com/jashkenas/ruby-processing`
[4] `http://py.processing.org/`

Moreover, Java's built-in primitive types and some classes are directly implemented in Scheme, while remaining classes are implemented in Java. Classes such as Strings, Arrays, and Exceptions are mapped directly to Scheme forms. Implementing these classes in Scheme is possible (with some constraints) due to similarities in both languages which, in turn, allow for a high level of interoperability between both languages.

Finally, ProfessorJ is fully integrated with DrScheme, providing a development environment that offers syntax highlighting, syntax checking, and error-highlighting for Java code. This is possible due to preserved source location information throughout the compilation pipeline.

4. SOLUTION

Although previously presented implementations are relevant for our solution, they do not fit entirely in our work's scope. Analysing Processing.js and Processing.py, we observe that having an IDE is a fundamental feature for Processing users. Also, both Processing.js and ProfessorJ implement their APIs directly in their target language, permitting greater interoperability between the source and target language. However, Processing.js, Processing.py, and Ruby-processing, do not allow designs to be visualized in a CAD, and, in spite of Java and Processing sharing many features, the differences require a custom tailored solution. Finally, Ruby-processing presents some relevant features that are useful for designs, namely live coding. Yet, as both Processing.py and Ruby-processing translate to the JVM, they are not relevant to our work.

Our proposed solution was to develop Processing as a new Racket language module, using Rosetta for Processing's visual needs, and integrating Processing with DrRacket's IDE services. The following sections explain how our compiler was developed and structured, presenting the main design decisions taken.

4.1 Module Decomposition

To better understand the main modules of our compiler, **Fig 2** illustrates the main Racket modules that are used, as well as the dependencies between them. We divided the modules in two major groups: the *Compiler* and *Runtime* modules. In the following paragraphs, we provide a detailed description of the most important modules.

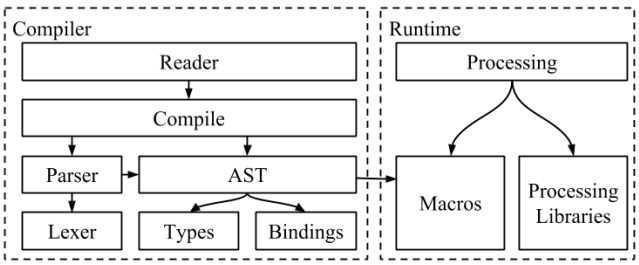


Figure 2: Main module decomposition and dependencies. The arrows indicate a uses relationship between modules - module A uses (⟶) module B

4.1.1 Compiler Modules

Reader Module. To add Processing as a new language module [10], a new specifically tailored `reader` is needed for Processing. This enables Racket to parse Processing source code and transform it to target Racket code. The `reader` must provide two important functions: `read` and `read-syntax`, and receive an `input-port` as input, differing in their return value. The former produces a list of S-expressions, while the latter generates `syntax-objects` (S-expressions with lexical-context and source-location information). The `reader` uses functions provided from the *Compile module*, to create and analyse an intermediate representation of the source Processing code, and to generate target Racket code.

Compile Module. The *Compile module* defines an interfacing layer of functions that connects the *Reader module* with the *Parse* and *AST* modules. The main advantage is to have a set of abstractions that manipulate certain phases of the compilation process. For instance, the *Compile module* provides functions that parse the source code, create an AST, check types, and generate Racket code.

Parser & Lexer Modules. The *Parse* and *Lexer* modules contain all the functions that analyse the syntactic and semantic structure of Processing code. To implement the lexer and parser specifications, we used Racket's `parser-tools` [11], adapting parts of ProfessorJ's lexer and grammar according to Processing's needs. The *Lexer* uses `parser-tools/lex` to split Processing code into tokens. To abstract generated tokens by the *Lexer module*, Racket's `position-tokens` are used, as they provide a simple way to save the code's original source locations. Processing's parser definition is implemented using Racket's `parser-tools/yacc`, which produces a LALR parser.

AST Module. Parsing the code produces a tree of `ast-node%`, that abstracts each language construct such as, statements, expressions, etc. These nodes are implemented as a Racket class, containing the original source locations and a common interface which allows the analysis and generation of equivalent Racket code. Each `ast-node%` provides the following methods:

- `->check-bindings`: traverses the AST populating the current scope with defined bindings and their type information;
- `->type-check`: checks each AST node for type errors, promoting types, if necessary;
- `->racket`: generates Racket code using custom defined functions and macros, wrapping them within a `syntax-object` along with the original source information.

Types and Bindings Module. The *bindings module* provides auxiliary data structures needed to store and manage different Processing bindings. We created a `binding%` class to abstract binding information (e.g. modifiers, argument and return types, etc), and a custom `scope%` class to handle Processing's scoping rules. Each `scope%` has a reference to its parent `scope%` and has a hash table that associates identifiers to `binding%` representation. The *types module* has all the necessary functions to check if two types are compatible or if they need to be promoted. As many of Processing's typing rules are similar to Java's, we adapted ProfessorJ's type-checking functions to work with our compiler.

4.1.2 Runtime Modules

The *runtime module* provides all the necessary macros, functions, and data types, that are required by generated Racket code. These functions are provided by the *Processing module* using the *Macros* and *Processing libraries* modules. The former contains necessary source transformations required to generate equivalent Racket code. The latter provides an interface that implements Processing's built-in classes and drawing primitives, using Rosetta to generate designs in several CAD backends.

4.2 Compilation Process

Our Processing implementation follows the traditional compiler pipeline approach (illustrated in Fig 3), composed by three separated phases, namely parsing, code analysis, and code generation.

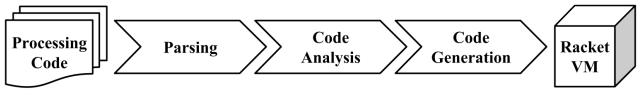


Figure 3: Overall compilation process

Parsing. The initial compilation process starts by the parsing phase, which is divided in two main steps. First, Processing source code is read from the `input-port` and transformed into tokens. Secondly, tokens are given to LALR parser, building an AST of Racket objects, that will be analysed in subsequent phases.

Code Analysis. Following the parsing phase, a series of checks must be made to the generated AST. This is due to some differences between Processing's and Racket's language definitions. For instance, Processing has static type-checking and has different namespaces for methods, fields, and classes, while Racket is dynamically typed and has a single namespace. As a result, custom tailored mechanisms were needed to solve compatibility issues, in order to generate semantically equivalent Racket code.

Initially, the AST is traversed by repeatedly calling `->check-bindings` on child nodes, passing the current `scope%`. When a new definition is created, be it a function, variable, or class, the newly defined binding is added to the current scope along with its type information. Each time a new scope is created in Processing code, a new `scope%` object is created to represent it, referring to the current `scope%` as its parent. These mechanisms are needed to implement

Processing scoping rules and type-checking rules. For example, to type-check a function call, the information of the return type, arity, and argument types is needed to correctly type-check the expression.

Secondly, the type-checking procedure runs over the AST by calling `->type-check` on the topmost AST node. As before, it repeatedly calls `->type-check` on child nodes until the full AST is traversed, using previously saved bindings in the current `scope%` to find out the types of each binding. During the type-checking procedures, each node is tested for type correctness and in some cases promoting types, if necessary. In the event that types do not match, a type error is produced, signalling where the error occurred.

Code Generation. After the AST is fully analysed and type-checked, semantically equivalent Racket code can be generated. To achieve this, every AST node implements `->racket`, which uses custom defined macros and functions to produce Racket code. This code is then wrapped in a `syntax-object` along with source information saved by the AST. Subsequently, these `syntax-objects` will be consumed by `read-syntax` at the reader level. Afterwards, Racket will expand the define macros and load the generated code into Racket's VM. By using macros, we can create human-readable boilerplate Racket code that can be constantly modified and tested.

Racket and Processing follow the same evaluation order on their programs, thus most of Processing's statements and expressions are directly mapped into Racket forms. However, other statements such as `return`, `break`, or `continue` need a different handling as they use control flow jumps. To implement this behaviour we used Racket's *escape continuations*, in the form of `let/ec`. Furthermore, Processing has multiple namespaces, which required an additional effort to translate bindings to Racket's single namespace. To support multiple namespaces in Racket, binding names were mangled with custom tags. For instance, `func` tag is appended to functions, so function `foo()` internally would be `foo-fn()`. The use of **'-'** as a separator allows us to solve the problem of name clashing with user defined bindings, as Processing does not allow **'-'** in names. Also, as we have function overloading in Processing, we append specific tags that represent the argument's types to the function's name. For instance, the following function definition: `float foo(gloat x, float y){ ... }` would be translated to `(define (foo-FF-fn x y)...)`.

An initial implementation of classes has been developed, by mapping Processing classes into Racket classes, reusing some of ProfessorJ's ideas. Instance methods are translated directly into Racket methods, therefore instance method `public void foo()` is translated to Racket's public methods, `(define/public (foo-fn)...)`. On the other hand, static methods are implemented as a Racket function, by appending the class name to the function's name. For example, a method `static void foo()` of class `Foo` will be translated to `(define (Foo.foo-fn)...)`. Also, there is an issue with constructors, as Processing can have multiple constructors. This problem is solved by adapting `new` to find the appropriate constructor to initialize the object.

To correctly support Processing's distinctions between *Active* and *Static mode* we used the following strategy. We added a custom check in the parser that signals if the code is in *Active mode*, i.e. if a function or method is defined. While in active mode, global statements are restricted, thus when generating code for global statements we check if the code is in *Active mode*, signalling an error if true.

4.3 Runtime

Our runtime is implemented directly in Racket, due to the necessity of integrating our implementation with Rosetta. Processing offers a set of built-in classes that provide common design abstractions that aid users during their development process. For instance, `PVector`, abstracts 2 or 3 dimensional vectors, useful to describe positions or velocities. These will be implemented directly in Racket, allowing for greater performance and interoperability.

However, this presents some important issues. First, as Racket is a dynamically typed language, the type-checker, at compile time, cannot know what are the types of Racket bindings. To solve this issue we introduced a new type in the type hierarchy, acting as an opaque super type that the type-checker ignores when type checking these bindings. On the other hand, as Processing primitives and built-in classes are implemented in Racket, we also have the problem of associating type information for these bindings. To solve this issue, we created a simple macro (**Fig 4**), that allows us to associate type information to Racket definitions, by adding them to the global environment, thus the type-checker can correctly verify if types are compatible. Alternatively, instead of using a custom macro, we could of written Processing's APIs in `typed/racket`, as types can be associated to Racket definitions. Yet, this alternative was not used due to possible type incompatibilities with Processing's and Racket's type hierarchy. Secondly, because the gain of using `typed/racket` [12] would not be significant due to Rosetta and the compiler being written in untyped racket.

```
(define-syntax-rule
  (define-types (id [type arg] ... -> rtype)
      body ...)
  (begin
    (add-binding! rtype 'id (type ...))
    (define (id arg ...) body ...)))
```

Figure 4: Macro that associates Processing types to a definition

Processing's drawing paradigm closely resembles OpenGL's traditional push/pop-matrix style. To provide rendering capabilities in our system, we use Rosetta, as it provides design abstractions that not only lets us generate designs in an OpenGL render, but also gives us access to several CAD back-ends. Custom interface adjustments are needed to implement Processing's drawing primitives in Racket, as not every Processing primitive maps directly into Rosetta's. Furthermore, Rosetta also enables us to supply Processing developers with different drawing primitives unavailable in Processing's core environment. Therefore we are able to augment Processing's core capabilities with additional drawing primitives and design approaches, that empower users to explore different designs.

4.4 Interoperability

Mapping Processing constructs directly into Racket's allows for greater interoperability between both languages. At the moment, each Processing module is translated to a Racket module. As a result, to use Racket code within a Processing module, a custom import mechanism was created. A `require` statement was introduced that maps into Racket's `require`, allowing Racket modules (or any other language of the Racket ecosystem) to be referenced within a Processing module. Nonetheless, this decision has a major issue in regard to Processing's identifiers, as they are not compatible with Racket's. Racket allows for identifiers to be composed of characters such as '?', '-', or '+', yet Processing does not. As result, we are unable to use these bindings in our Processing scripts. This issue can be solved by using two different approaches.

The first approach is to automatically rename all of provided bindings of the required module, by using a custom set of name renaming rules. For instance, Racket bindings separated with '-' characters are translated to camel case, i.e. `foo-bar` is converted to `fooBar`. The second approach is to force the developer to create his custom name mappings by creating a Racket module that does the name conversion. Clearly, the first approach provides a clearer and quicker way of using a Racket module. However, as renaming rules applied are debatable, the developer is free to create his custom mapping from our initial transformation.

On the other hand, we want to be able to use a Processing module in other languages of the Racket ecosystem. To correctly implement these mechanisms, Processing's modifiers (i.e. `public`, `private`, etc.) are used to provide bindings to other modules, mapping them into Racket's `provide`.

4.5 Integration with DrRacket

Processing developers are familiar with an IDE (the PDE) that offers them a set of common IDE tools, such as syntax highlighting or code formatting. DrRacket as a pedagogical IDE, shares some of the PDE's features, providing a similar development environment to Processing developers and allowing them to easily make the transition to our system. DrRacket's IDE services use source locations to operate, therefore by saving this information and passing it along through the compilation process, we can easily integrate our Processing implementation with DrRacket's features (ilustrated in Fig 5).

A relevant feature that Racket offers is a REPL, which is common in many Lisp descendants. However, currently no Processing implementation provides a REPL to its users. Therefore, having a REPL would be a major advantage to the PDE environment, as it would provide users a mechanism to test specific parts of their code, being a good mechanism for beginners to learn and immediately experiment new ideas.

Due to Racket's language development capabilities, this feature was easily implemented by creating a custom function to compile REPL interactions for Processing. However, as Processing is a statement based language, REPL interactions will not produce expressions. So we created a new parser rule to implement REPL interactions, adding it to the parser generator's start symbols. This way Racket's `parser-tools` produces different parsing procedures for each start symbol, which we can use according to the type of interaction we are manipulating. The interactions shown in **Fig 5** shows how we can use the REPL for Processing. For example, note that `println(factorial(5));` returns the result of factorial of 5 by producing a print side-effect, while in `factorial(5) + 100` the returned result is the actual expression that is produced by the add operator.

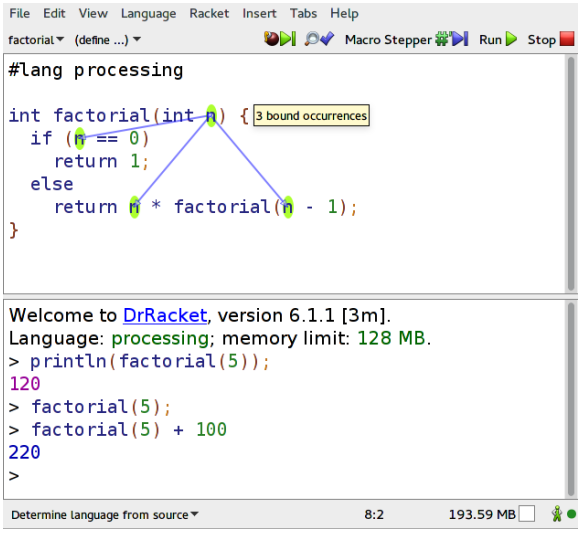


Figure 5: Processing in DrRacket

5. EXAMPLE

In this section, we illustrate an example of code that generates a double helix (**Fig 6**) using our system. Our current implementation is still a work in progress, hence the compilation results are subject to change. The code illustrated in **Fig 7** shows a Processing example that generates the helix illustrated in **Fig 6**.

The double helix is drawn by using a recursive function that repeatedly renders a pair of spheres connected by a cylinder, along a rotating axis. This example is a case of an *Active mode* sketch, as function definitions are present. Also, Processing's design flow is demonstrated by the use of `setup()` and `draw()`. In `setup()`, we use the `backend` function (provided by Rosetta) to define the rendering backend to use, which in this case is AutoCAD. On the other hand, `draw()` executes `helix()` to produce the design in AutoCAD. **Fig 8** presents the Racket code that is produced by our compiler.

The first point worth mentioning is that function identifiers are renamed to support multiple namespaces. We can see that `helix` identifier is translated to `helix-FF-fn`. The **F** is to indicate that the function has 2 arguments that are of type `float`. Also, we can see that `setup()` and `draw()` are mangled as well, by appending `fn` to their name. Functions and macros such as `p-mul`, `p-sub`, or `p-call`, are defined in the runtime modules, implementing Processing's semantics. Variable definitions are translated by using `p-declaration`, which is a macro that generates a Racket `define-values` form, using a sequence of `identifier` and `value` pairs. To

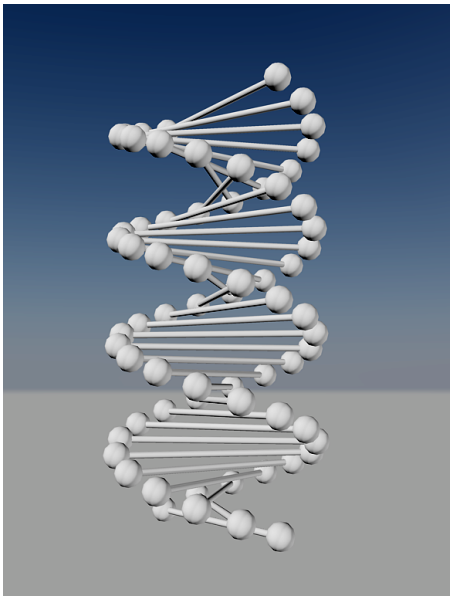


Figure 6: Double helix generated from Processing code using AutoCAD

declare variables (i.e. `float x,y;`), the `value` is stored using Racket's `undefined`. Mathematical operators (`p-mul`, `p-add`) are implemented as Racket functions, yet, do not overflow as Processing. On the other hand, the `sphere` and `cylinder` drawing primitives, are specially tailored to map into Rosetta's operators. For instance, `cylinder` is a good example of an operator that Rosetta provides, but that is not available in the current Processing environment.

At the moment, we observe that all function definitions have their body wrapped in a `let/ec` form. This is injected to support `return` statements within functions. Although performance will be limited by the chosen connection interface and rendering backend, the usage of `let/ec` is a clear example that brings additional performance overhead with no possible gains, as the return type is `void`. Thus an optimization is required to remove `let/ec`, for cases that jump statements are not present or when the last statement is a `return`.

```
float r =  15, height = 2;

void helix(float z, float ang) {
  float x1 = r*cos(ang), y1 = r*sin(ang);
  float x2 = r*cos(PI+ang), y2 = r*sin(PI+ang);

  sphere(x1, y1,  z, 2);
  cylinder(x1, y1,  z, 0.5, x2, y2, z);
  sphere(x2, y2,  z, 2);

  if(ang > 0) helix(z + height, ang - PI/8);
}
void setup() { backend(autocad); }
void draw()  { helix(0, 4 * PI); }
```

Figure 7: Processing code example of the Double Helix

```
(p-declaration (r 15.0) (height 2.0))

(define (helix-FF-fn z ang)
 (let/ec return
  (p-block
   (p-declaration
     (x1 (p-mul r (p-call cos-F-fn ang)))
     (y1 (p-mul r (p-call sin-F-fn ang))))
   (p-declaration
     (x2 (p-mul r (p-call cos-F-fn
                          (p-add PI ang))))
     (y2 (p-mul r (p-call sin-F-fn
                          (p-add PI ang)))))
   (p-call sphere-FFFI-fn x1 y1 z 2)
   (p-call cylinder-FFFFFFF-fn
           x1 y1 z 0.5 x2 y2 z)
   (p-call sphere-FFFI-fn x2 y2 z 2)
   (when (p-gt ang 0)
    (p-call helix-FF-fn
            (p-add z height)
            (p-sub ang (p-div PI 8.0)))))))

(define (setup-fn)
 (let/ec return
  (p-block (p-call backend-0-fn autocad))))

(define (draw-fn)
 (let/ec return
  (p-block (p-call helix-FF-fn 0 (p-mul 4.0 PI)
      ))))

(p-initialize))
```

Figure 8: Generated Racket code

Finally, a `p-initialize` macro is added to implement Processing's workflow semantics. This macro is responsible of ensuring that the `setup()` and `draw()` functions are called, if defined by the user. `p-initialize` is implemented by using Racket's `identifier-binding` to check if the `setup-fn` and `draw-fn` are bound in the current environment.

6. CONCLUSION

Implementing Processing for Racket benefits architects and designers, by allowing them to develop with Processing in a CAD environment. Also, the ability to provide new design paradigms offered by Rosetta is a strong reason for the architecture community to use our solution. The implementation follows the common compiler pipeline architecture, generating semantically equivalent Racket code and loading it into Racket's VM. Our strategy was to implement Processing's primitives directly in Racket to easily access Rosetta's features and allow a greater interoperability with Racket. Also, we have developed mechanisms to access Processing code from Racket and vice versa.

Currently, our development approach was to first fulfil the most basic needs of Processing users (the ability to write simple scripts) and present visual results in a CAD application. Afterwards, our goal is to build-upon our existing work, and progressively introduce more advanced mechanisms, such as implementing inheritance and interfaces in classes, support live coding, or adapt Processing's exception system to Racket. To provide a better environment for Processing developers, we plan to further adapt DrRacket by

creating an editor mode with better syntax highlighting for Processing. Also, adding visual support to REPL interactions would be a huge advantage to our implementation, as it would allow users to immediately visualize geometric shapes in the IDE without loading up the rendering backend. Finally, optimizations can be made to generated Racket code to improve the quality and performance of the generated Racket code.

7. ACKNOWLEDGEMENTS

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) with reference UID/CEC/50021/2013, and by the Rosetta project under contract PTDC/ATP-AQI/5224/2012.

8. REFERENCES

[1] Casey Reas and Ben Fry. Processing: programming for the media arts. *AI & SOCIETY*, 20(4):526–538, 2006.

[2] José Lopes and António Leitão. Portable generative design for cad applications. In *Proceedings of the 31st annual conference of the Association for Computer Aided Design in Architecture*, pages 196–203, 2011.

[3] Matthew Flatt. Creating languages in racket. *Communications of the ACM*, 55(1):48–56, 2012.

[4] Pedro Palma Ramos and António Menezes Leitão. An implementation of python for racket. *7 th European Lisp Symposium*, page 72, 2014.

[5] John Maeda. *Design by Numbers*. MIT Press, Cambridge, MA, USA, 1999.

[6] John Resig, Ben Fry, and Casey Reas. Processing. js, 2008.

[7] Kathryn E Gray and Matthew Flatt. Compiling java to plt scheme. In *Proc. 5th Workshop on Scheme and Functional Programming*, pages 53–61, 2004.

[8] Kathryn E Gray and Matthew Flatt. Professorj: a gradual introduction to java through language levels. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–177. ACM, 2003.

[9] Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. Drscheme: A programming environment for scheme. *Journal of functional programming*, 12(02):159–182, 2002.

[10] M Flatt and RB Findler. Creating languages: The racket guide.

[11] Scott Owens. Parser tools: lex and yacc-style parsing.

[12] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.

Constraining application behaviour by generating languages

Paul van der Walt
INRIA Bordeaux, France
paul.vanderwalt@inria.fr

ABSTRACT

Writing a platform for reactive applications which enforces operational constraints is difficult, and has been approached in various ways. In this experience report, we detail an approach using an embedded DSL which can be used to specify the structure and permissions of a program in a given application domain. Once the developer has specified which components an application will consist of, and which permissions each one needs, the specification itself evaluates to a new, tailored, language. The final implementation of the application is then written in this specialised environment where precisely the API calls associated with the permissions which have been granted, are made available.

Our prototype platform targets the domain of mobile computing, and is implemented using Racket. It demonstrates resource access control (*e.g.,* camera, address book, *etc.*) and tries to prevent leaking of private data. Racket is shown to be an extremely effective platform for designing new programming languages and their run-time libraries. We demonstrate that this approach allows reuse of an inter-component communication layer, is convenient for the application developer because it provides high-level building blocks to structure the application, and provides increased control to the platform owner, preventing certain classes of errors by the developer.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures—*domain-specific architectures, languages, patterns*

General Terms

Languages, Security

Keywords

Programming frameworks, functional programming, embedded domain-specific languages, sports equipment

1. INTRODUCTION

Among programming frameworks intended to be used by third party developers, we see a trend towards including mechanisms restricting access to certain features, or otherwise constraining behaviour of the application [4, 17]. In the case of platforms like Android [16], the aim is usually to protect the user's sensitive data (*e.g.,* contact list, physical location) from undesired use, while still giving applications access to the resources, whether hardware or data, needed to function correctly. For example, an email application legitimately requires access to the Internet, but for a calculator application this should raise suspicion. In the case of Android, these restrictions are enforced via run-time checks against a permissions file called the Manifest, which the user accepts at install-time. Other frameworks are also adopting such declarations, in various forms [3, 8].

1.1 Declaration-driven frameworks

Generally speaking, we identify a class of programming frameworks in widespread use, which we call *declaration-driven frameworks*. These frameworks are different to traditional static programming frameworks in that they have some form of declarations as input. Examples abound, including the Android SDK with its Manifest file, or the Facebook plugin SDK, both of which require permissions to be granted *a priori*. The declarations vary greatly in expressiveness, on a spectrum from simple resource permissions, *e.g.,* access to list of friends and the camera, to very expressive, *e.g.,* rules for the control flow of the application, a list of components to be implemented, *etc.* An example of the latter is DiaSuite [2], where the individual components of the application, as well as their subscription relations, are laid out in the declarations. Access to resources is also granted on a component level. These rich declarations encourage separation of components, and provide relative clarity for the user regarding potential information flow, when compared to a simpler list of permissions. Diagrams with potential information flow can be extracted from the specifications, and presented in graphical format, for example.

1.2 Problem

We identify a number of shortcomings with the systems mentioned above. Most frequently, the declarations are no more than a list of permissions checked at run-time, leaving the user guessing about the actual behaviour of the application [23]. The fact that these checks are dynamic also leads to the application halting on Android if a developer tries to access a forbidden resource. Existing static approaches that exist, on the other hand, generally try to solve different problems than resource access control, for example just checking that all required components are implemented [17].

These shortcomings are addressed by DiaSuite's approach allowing static checks on resource access, which is based on an external DSL. However, the current implementation of DiaSuite generates Java boilerplate code from the declarations, tailored to the specific application. With this generative approach, extending the declaration language would involve

modifying the standalone compiler, and in general, generated code tends to be difficult to debug and inconvenient to work with.

On the other hand, the language building platform provided by Racket allows simple implementation of an embedded DSL, with all the features of Racket potentially available to the application developer. Providing expressive constructs for specification and implementation of an application raises the level of abstraction, and allows us to implement static guarantees of resource access equivalent to DiaSuite. Furthermore, we need not maintain parser and compiler machinery in parallel with the framework infrastructure.

In this report we demonstrate the use of Racket's language extension system [21] allowing us to easily derive tailored programming environments from application declarations. This decreases effort for the application developer and gives more control to platform owners. To the best of our knowledge this is the first implementation of an EDSL which itself gives rise to a tailored EDSL in Racket. The code presented in this report is available from `http://people.bordeaux.inria.fr/pwalt`.

Outline. After giving a brief overview of related work in Sec. 2, we introduce the platform we have chosen as the basis of our prototype, as well as the example application to be implemented in Sec. 3. In Sec. 4 we show how a developer using our system would write the example application. Sec. 5 goes into detail about how the declarations unfold into a language extension, and finally in Sec. 6 we discuss strengths and weaknesses of the approach using Racket. Our conclusions are presented in Sec. 7.

2. RELATED WORK

The work most closely resembling ours is DiaSuite [1], since it is the inspiration for our approach. The relative advantages and disadvantages are thoroughly dealt with in Sec. 3.

Other than that, it seems there is not much literature on the generation of frameworks, although to varying degrees frameworks which depend on declarations are becoming ever more widely adopted [3,16]. These generally address demonstrated threats to user safety [6, 13, 15, 18, 22].

Many approaches have been proposed to address these leaks, such as parallel remote execution on a remote VM where a dynamic taint analysis is running [7]. This naturally incurs its own privacy concerns, as well as dependence on a connection to the VM. Another approach which bears similarity to our aim, is the work by Xiao *et al.* [23], which restrains developers of mobile applications to a limited external DSL based on TouchDevelop [14], from which they extract information flow via static analysis. This information is then presented to the user, to decide if the resource usage seems reasonable. This is a powerful and promising approach, but we believe that it is preferable to declare information flow paths *a priori* and constrain the developer, than having to do a heavy static analysis to extract that same information – especially since it means a developer cannot use a general-purpose language they are already familiar with, but must learn a DSL which is used for every aspect of the implementation.

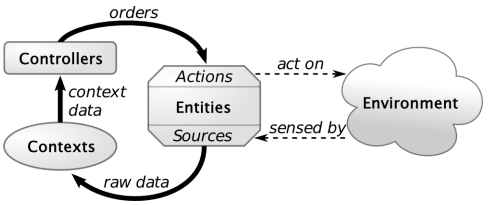


Figure 1: The *Sense/Compute/Control* paradigm. Illustration adapted from [2].

```
Declaration -> Resource | Context | Controller
Type        -> Bool | Int | String | ...
Resource -> (source srcName | action actName) as Type
Context  -> context ctxName as Type CtxtInteract
CtxtInteract -> when ( required GetData?
                     | provided (srcName | ctxName)
                        GetData? PublishSpec)
GetData       -> get (srcName | ctxName)
PublishSpec   -> (always | maybe) publish
Controller    -> controller ctrName ContrInteract
ContrInteract -> when provided ctxName do actName
```

Figure 2: Declaration grammar. Keywords are in bold, terminals in italic, and rules in normal font.

Compared to these alternatives, providing a "tower of languages"-style solution [11] seems be a good trade-off between restrictions on the developer and versatility of the implementation.

3. CASE STUDY

DiaSuite, the model for our prototype, is a declaration-driven framework which is dedicated to the *Sense/Compute/Control* architectural style described by Taylor *et al.* [19]. This pattern ideally fits applications that interact with an external environment. SCC applications are typical of domains such as building automation, robotics, avionics and automotive applications, but this model also fits mobile computing.

3.1 The Sense-Compute-Control model

As depicted in Fig. 1, this architectural pattern consists of three types of components: (1) *entities* correspond to managed[1] resources, whether hardware or virtual, supplying data; (2) *context components* process (filter, aggregate and interpret) data; (3) *controller components* use this information to control the environment by triggering actions on entities. Furthermore, all components are reactive. This decomposition of applications into processing blocks and data flow makes data reachability explicit, and isolation more natural. It is therefore well-suited to the domain of mobile computing, where users are entrusting their sensitive data to applications of dubious trustworthiness.

3.2 Declaration language

The minimal declaration language associated with DiaSuite is presented in Fig. 2. It is adapted from [2], keeping only essential constructs. An application specification is a list of

[1]Managed resources are those which are not available to arbitrary parts of the application, in contrast to basic system calls such as querying the current date.

`Declarations`. Resources (such as camera, GPS, *etc.*) are defined and implemented by the platform: they are inherent to the application domain. Context and controller declarations include interaction contracts [1], which prescribe how they interact. A context can be activated by either another component requesting its value (`when required`) or a publication of a value by another component (*i.e.,* `when provided component`). When activated, a context component may be allowed to pull data (denoted by the optional `get`). Note that contexts which may be pulled from must have a `when required` contract. Finally, a context might be required to publish when triggered (defined by `PublishSpec`). Note that `when required` contexts have no publish specification, since they are only activated by pulling, and hence return their values directly to the component which polled them. When activated, controller components can send orders, using the actuating interfaces of components they have access to (*i.e.,* `do actName`), for example printing text to the screen or sending an email.

DiaSuite compiles the declarations, written in an external DSL, into a set of Java abstract classes, one for each declared component, plus an execution environment to be used with the classes which extend them. The abstract classes contain method headers which are derived from the interaction contracts, and constrain the input and output of the developer's implementation of each component. Additionally, access to resources is passed in as arguments to these methods, so that the only way a developer may use a resource is via the capability passing method from the framework.

This approach allows advantages such as static checks by the Java compiler that the application conforms to the declarations. From these declarations it directly follows which sensitive resources components should have access to, giving the application developer a much more concise API to work with. For example, if a component only has access to the network, it need not have the API for dealing with the camera in scope. This is in contrast to Android, where all system API calls are always available, increasing the amount of information the developer must keep in mind. The disadvantage is that this is an external DSL, and thus requires a separate compiler to be maintained. It also implies less versatility, having to re-invent the wheel, and a symbolic barrier, as argued by Fowler [12].

3.3 Example application

As our running example, we use a prototype mobile application. We pretend that it is distributed for free, supported by advertisements. It allows the user to capture pictures and then view them with a colourful filter (see Fig. 3). An advertisement will be downloaded from the Internet, but we would like to prevent the developer from being able to leak the picture (which is private) to the outside world, whether intentionally or by using a malicious advertisement provider. It has been shown that this is in fact a threat: frequently, included third party advertisement libraries try to exfiltrate any private data to which they are able to get access [18].

From the specification it follows that it should be impossible for the picture to leak to the Web, since the bitmap processing component is separate from the advertisement component.

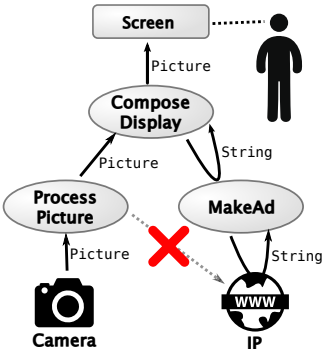


Figure 3: Simplified schematic of example application's design. We do not want the picture to be able to leak to the WWW. Note that pull requests (the curved arrows) are not parameterised, and are only used to return values.

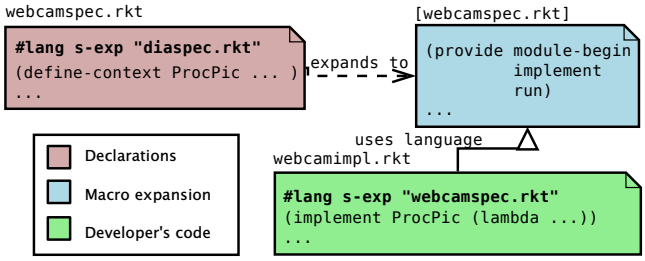


Figure 4: The prototype's architecture. Provided declarations are transformed into a tailored language for the implementation. The `implement` macro gets cases for each declared component.

4. IMPLEMENTATION OF EXAMPLE

Inspired by the DiaSuite approach, where a framework is generated from the specifications, the first step in our implementation is to provide an embedded DSL for writing specifications. It should include constructs for defining contexts and controllers, according to the grammar in Sec. 3. As illustrated in Fig. 4, when the specifications are evaluated, they in turn form a language extension which should be used to implement the application. The programming environment that is thus created provides the developer with tailored constructs for the application that is to be built, including an API precisely matched to what each component may do. In our prototype, we consider the advert developer and application developer as potentially the same, since we expect the advertisement library to provided in the form of a snippet of code that will be included along with the rest of the implementation code. This way, the advertisement code does need to be specially analysed, but is subject to the same constraints as any other code provided by the developer. That is, it can only access entities specified in the declarations.

4.1 Example specifications

The specifications as rendered in Racket, for our example application, are shown in Fig. 5. The syntax closely matches the DiaSuite declaration language previously introduced, and

```
;;; Specifications file, webcamspec.rkt
#lang s-exp "diaspec.rkt"
(define-context MakeAd String [when-required get IP])
(define-context ProcessPicture Picture
  [when-provided Camera always_publish])
(define-context ComposeDisplay Picture
  [when-provided ProcessPicture get MakeAd
     maybe_publish])
(define-controller Display
  [when-provided ComposeDisplay do Screen])
```

Figure 5: Complete declarations of the example application, in Racket prototype.

```
;;; Implementation file, webcamimpl.rkt
#lang s-exp "webcamspec.rkt"
(implement ComposeDisplay
  (lambda (pic getAdTxt publish nopublish)
    (let* ([canvas  (make-bitmap pic ..)]
           [adTxt   (getAdTxt)])
      (cond [(string=? "" adTxt) (nopublish)])
      ; ... do magic, overlay adTxt on pic
      (publish canvas))))
... ; the remaining implement-terms
```

Figure 6: The implementation of the `ComposeDisplay` context.

reflects the graphical representation of the application in Fig. 3.

4.2 Semantics of declarations

In this section, we explain the semantics of each term, from the point of view of the application developer.

The keywords `define-context` and `define-controller` are available for specifying the application, and upon evaluation, will result in a macro `implement`, for binding the implementations of components to their identifiers. For the developer this is convenient, since they only need to provide implementation terms while the framework takes care of inter-component communication as specified in the declarations. From the point of view of the framework, it provides more control over the implementation: before execution static checks can be done to determine if the terms provided by the application developer conform to the specifications.

Declaring a component C adds a case to the `implement` macro. Now, a developer can use the form `(implement` C f`)` to bind a lambda function f as the implementation of C. However, not just any f may be provided, as the arguments to `implement` are subject to a Racket function contract [5]. Unfortunately there is a name conflict between interaction contracts for components (as in DiaSuite) and function contracts in Racket, which are not the same thing. Function contracts in Racket are flexible annotations on definitions and module exports, which perform arbitrary tests at runtime on the input and output of functions. For example, a function can be annotated with a contract ensuring it maps integers to integers. If the function receives or produces a non-integer, the contract will trigger an error. The contract on f is derived from the interaction contracts of Fig. 5 as follows.

Activation conditions. These define the first argument to the function f.

`when-provided` *`x`*. First argument gets type of x. For *`ComposeDisplay`*, the contract starts with `(-> bitmap%? ...)`,[2] since it is activated by *`ProcessPicture`* publishing a bitmap image.

`when-required`. No argument added – the context was activated by pull.

Data sources and actions. These determine the (optional) next argument to the developer's function. This is a closure providing proxied (that is, surrounded by a run-time guard) access to the resource. This makes it convenient for a developer to query a resource, and allows the framework to enforce permissions. Actions for controllers are provided using the same mechanism.

`get` *`x`*. The contract of the closure becomes `(-> t?)` where t is the output type of x. Note that there is no parameter, just a return value. This means that a component requesting a value from another cannot exfiltrate data this way. The full contract so far is therefore `(-> ... (-> t?) ...)`.

`do` *`x`*. The contract of the closure becomes `(-> t? void?)` where t is the input type of x. The full contract is therefore `(-> ... (-> t? void?) void?)`. The final `void?` reflects that controllers do not return values.

Publication requirements. These determine the last arguments to the function contract of a context, corresponding to the output type of the context. Publishing is handled using continuations, to give us flexibility in the number of "return" statements provided.

`always_publish`. One continuation function corresponding to publication: the final contract becomes `(-> ... (-> t? void?) none/c)`, with t the expected return type.

`maybe_publish`. Two continuations to f, for publish and no-publish. The first has the contract `(-> t? void?)` with t the output type. The second continuation simply returns control to the framework. If the developer chooses not to publish, they use the second, no-publish continuation. The contract is therefore `(-> ... (-> t? void?) (-> void?) none/c)`.

The `none/c` contract accepts no values: this causes a run-time error if the developer does not use one of the provided continuations.

4.3 The implementation of the application

In Fig. 6, we show a developer's possible implementation of the context `ComposeDisplay`, which composes the modified image with the advertisement text. Essentially, a developer uses **`implement`** to bind their implementation to the identifier introduced in the specifications, *c.f.* Fig. 5. Their

[2] In reality, `bitmap%?` is shorthand for `(is-a?/c bitmap%)`, the contract builder which checks that a value is an object of type `bitmap%`.

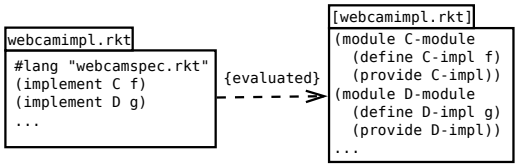


Figure 7: Separation of components using modules. The developer's code (left), and its expanded form (right). f in C cannot access D or g, because of lexical scoping.

```
1  (module webcamimpl "webcamspec.rkt"
2    (module ComposeDisplay-module racket/gui
3      (define/contract ComposeDisplay-impl
4        (-> bitmap%? (-> string?) (-> bitmap%? void?)
5            (-> void?) none/c)
6        (lambda (pic getAdTxt publish nopublish)
7          (let* ([canvas  (make-bitmap pic ..)]
8                 [adTxt   (getAdTxt)])
9            (cond [(string=? "" adTxt) (nopublish)])
10           ; .. do magic, overlay adTxt on pic
11           (publish canvas))))
12     (provide ComposeDisplay-impl))
13   ...)
```

Figure 8: The developer's code snippet is transformed into a submodule, as a result of evaluating Fig. 6. The shaded code is simply the term the developer provided in Fig. 6.

implementation should be a lambda term which obeys the contract resulting from the specification. For example, the `ComposeDisplay` context has the contract `(-> bitmap%? (-> string?) (-> bitmap%? void?) (-> void?) none/c)`. This is because it is activated by `ProcessPicture` publishing an image, it has **`get`**-access to the `MakeAd` component which returns a string, and it may optionally publish an image on account of its **`maybe_publish`** specification. The last two arguments correspond to publishing `(-> bitmap%? void?)` and not publishing `(-> void?)` continuations. The lambda function provided by the developer in Fig. 6 conforms to this contract. We see that if the advertisement component returns an empty string (line 7) the developer decides not to publish, but otherwise the string is overlaid on the picture and the developer publishes the composite image (line 9).

To prevent implementations of different components communicating outside of the condoned pathways, the `implement` macro wraps each f in its own submodule. As illustrated in Fig. 7, due to lexical scoping these do not have access to surrounding terms, but merely export the implementation for use in the top-level module. The result of this wrapping is shown in Fig. 8. The code in grey is precisely the term provided in Fig. 6, but it is now isolated from the implementations of the other components, preventing the developer from accessing them, which would constitute a leak.

Note that alongside this snippet, the rest of the implementations of the declared components must be provided in one module. This module must be implemented using the new `webcamspec.rkt` language – the one arising from the specifications we have written. The implementation module is checked before run-time to contain exactly one `(implement` C `...)` term for each declared C. However, we focus on this single context implementation to illustrate what transformations are done on the developer's code.

When the developer has provided implementations for each of the declared components, they can use the `(run)` convenience function which is also exported by the module resulting from the specifications. In the next section, we illustrate how these macros function.

5. THE FRAMEWORK AND RUN-TIME

Now that we have seen the user interface (*i.e.,* that which the application developer deals with) for our framework, we elucidate how the framework is implemented. This is broken down into a number of main parts: (1) the operation of the `define-context` and `define-controller` macros, (2) the expansion of the `implement` macro, and (3) how the run-time support libraries tie the implementations together to provide a coherent system. These mechanisms are explained globally here, though certain implementation details are elided. Notably, getting all the needed identifiers we had introduced to be available in the right transformer phases and module scopes was complicated. We invite the reader to experiment with the prototype code – the functionality for (1) and (2) is in the `diaspec.rkt` module, the run-time library can be found in the `fwexec.rkt` module.

5.1 What happens with the declarations?

Previously we saw that the first step for a developer is to declare the components of their application using the `define-context` and `define-controller` keywords. The specification should be provided in a file which starts with a **`#lang`** `s-exp "diaspec.rkt"` stanza, which causes the entire syntax tree of the specification to be passed to the function exported from `diaspec.rkt` as **`#%module-begin`**. This function does pattern matching on the specifications, and passes all occurrences of `define-` keywords to two handlers: (1) to compute and store the associated contracts, and (2) to instantiate a struct which will later store the implementation. The introduced identifiers are also stored as a list in the syntax transformer environment, *c.f.* the "Persistent effects" system presented in [20]. This compile-time storage will later be used to check implementation modules: have all components been implemented, and are all the identifiers used in the implementation declared in the specification?

To illustrate, Fig. 9 shows the expansion of the `ComposeDisplay` declaration, from line 6 of Fig. 5. Simplifications have been made, and module imports *etc.* have been omitted for brevity. Some elements which are not specific to this declaration term have been elided, namely a helper macro which transforms `(implement x ..)` terms into `(implement-x ..)`, to correspond with the generated macro in line 17, and a function which checks that all declared components have a corresponding `implement` term. Finally, we also omit the generated syntax for `module-begin-inner` from the specifications, since it is not particularly instructive. Note that it is this definition which allows the implementation module to use the specification module as its language, with the **`#lang`** `s-exp "webcamspec.rkt"` directive.

Line 1 marks the start of the implementation module, called

```
1 (module webcamspec "diaspec.rkt"
2  (define ComposeDisplay
3    (context 'ComposeDisplay
4      (interactioncontract ProcessPicture MakeAd
5        'maybePublish) 'pic))
6  (provide ComposeDisplay)
7  (module+ contracts
8    (define ComposeDisplay-contract
9        (-> bitmap%? (-> string?)
10           (-> bitmap%? void?) (-> void?) none/c))
11   (provide ComposeDisplay-contract))
12 (define-struct/contract ComposeDisplay-struct
13   ([spec   (or/c context? controller?)]
14    [implem (-> ...)])) ; contract from line 9
15 (provide ComposeDisplay-struct
16          implement-ComposeDisplay)
17 (define-syntax (implement-ComposeDisplay stx)
18  (syntax-case stx (implement-ComposeDisplay)
19   [(_ f)
20     #'(begin
21       (module ComposeDisplay-submodule racket/gui
22         (require (submod "webcamspec.rkt" contracts))
23         (provide ComposeDisplay-impl)
24         (define/contract ComposeDisplay-impl
25                          ComposeDisplay-contract f))
26       (require (submod "." ComposeDisplay-submodule))
27       (set-impl 'ComposeDisplay ; add to hashmap
28         (ComposeDisplay-struct ComposeDisplay
29            ComposeDisplay-impl)))]))
30
31 (provide run (rename-out
32                (module-begin-inner #%module-begin)))
33 (define-syntax (module-begin-inner stx2)
34    ... )) ;; omitted
```

Figure 9: The simplified expansion of the specifications, concentrating on `ComposeDisplay` from Fig 5. This code corresponds to the blue box in Fig. 4.

`webcamspec`. It still references `"diaspec.rkt"`, which is the language the specification was written in, *c.f.* Fig. 5. This leaves us with the code resulting from the `ComposeDisplay` context. In line 2, we see that a binding is introduced, using the name the developer chose for the component. Its value is a representation of the declaration, and is used to derive the contract. In line 7, a submodule is appended with the Racket contract the implementation is expected to adhere to. The **`module+`** keyword adds terms to a named submodule, creating the submodule if necessary [10]. Line 12 defines a tailored struct: it will hold the implementation of `ComposeDisplay`, in the field tagged with the corresponding contract. It becomes more interesting in line 17, where we see that the `implement` keyword wraps the developer's implementation in an independent submodule, as explained previously. This submodule will not have access to the surrounding scope, hence the need for the `contracts` submodule, which we import in line 22. As an aside, the **`#'`** form is shorthand for **`syntax`**, which is similar to **`quote`**, but produces a syntax object decorated with lexical information and source-location information that was attached to its argument at expansion time. Crucially, it also substitutes `f`, the pattern variable bound by **`syntax-case`** in line 19, with the pattern variable's match result, in this case the developer's implementation term. Next, in line 26, we have left the scope of the submodule. We **`require`** the submodule, bringing `ComposeDisplay-impl` into scope, which we add to a hash map (line 27). This hash map associates names of components to their implementations. Note how we are using the previously-defined struct, which forces the implementation term to adhere to its contract.

As a side-effect, `run` is only available to the developer if they manage to evaluate the implementation module without compile errors, which implies that only valid specifications and implementations allow the developer to execute the framework. Since the implementation of the framework run-time library is mundane, we do not discuss it here. To run this code, `racket-mode`[3] or DrRacket[4] [9] can be used. Simply load the `webcamimpl.rkt` file, and when it is loaded, evaluate `(run)` in the REPL.

6. EVALUATION AND DISCUSSION

In the end, the application developer is presented with a reasonably polished and coherent system for implementing an application in two stages, which allows the platform to give the user more insight into what mischief an application could potentially get up to. This assumes that the specifications are distributed with the application, and presented to the user (optionally formatted like Fig. 3), and that the software is compiled locally, or on a server that the user trusts. This would ensure that the implementation does indeed conform to the specifications.

We observe that going beyond Racket the functional programming language, and using it as a language-building platform, is where it really shines. We can mix, match and create languages as best fits the niche, then glue modules together via the common run-time library provided by Racket. This allows great flexibility and control, since with Racket's

[3]Tested using MELPA version 20150330.1125 of Greg Hendershott's wonderful Racket mode for Emacs.
[4]Tested using DrRacket v6.1.1.

#lang mechanism, we can precisely dictate the syntax and semantics of our new languages. These two aspects therefore give Racket a lot of potential in the emerging domain of declaration-driven frameworks.

6.1 Limitations

Unfortunately, there are issues that would need to be resolved before the proposed approach would be feasible in the real world. One of the trickiest parts of ensuring no communication between components is that consequently we cannot allow a developer to use any external modules in their code. This is because if a developer could **require** any module, they could in effect execute arbitrary code. It could also be used as a communication channel, since modules have mutable state. Therefore, in the prototype, we chose not to allow any importing of modules, but for a realistic application this would probably not be acceptable – we could imagine needing to use a library for parsing JSON, or processing images, or any number of benign tasks. Perhaps this would be a decision for the platform provider to make: is a particular library "safe" and could it be white-listed?

Another potential leak could be the **eval** form. Using it, a developer could easily obfuscate any behaviour desired. In fact, arbitrary imports and calls would be possible that way. We therefore inspect a developer's implementation for such things as the use of **eval**, and reject them syntactically, but since the binding might be hidden or renamed, this approach is not necessarily robust. This highlights a need in Racket: allowing components or functions to be pure would solve this vulnerability. Perhaps Typed Racket [20] will offer a solution in the future – purity analysis is on the project to-do list.[5]

It was also rather finicky to implement all the macros as described above. Although conceptually simple, it turns out to be pretty difficult in practice to get all the identifiers to be available in the right syntax transformer phases. The macro debugger in DrRacket is quite powerful, but unfortunately still leaves a lot to be desired. For example, we failed to get it to show the completely expanded implementation module as it is presented in Fig. 9 – that code is largely worked out by hand. Finer control over the macro unfolding would therefore be beneficial.

The end result is not at all pessimistic, in spite of these shortcomings and difficulties. The prototype does demonstrate the power of a language such as Racket, which gives a programmer the capability to easily modify syntax and provide custom interpretations. This prototype also demonstrates that it is possible to cleanly separate concerns and enforce a certain structure on the final implementation.

6.2 Future work

There are a number of clear avenues for improving this work. Firstly, we note that the chosen platform and model are merely examples, it should be easy to build similar "active" specification DSLs for other domains. This modular approach is also very flexible: we could choose to use any Racket extension as the implementation language for the developer to use, whether it be FRTime or Typed Racket or any other of the many libraries. We could even decide to provide different languages for different modules – the changes would be minor. If for example Typed Racket were to support purity analysis in the future, this would be a very attractive option, allowing us to be confident that no unwanted communication between components is possible. As stated, though, before this approach could be introduced into the wild, a safe module importing mechanism should be devised.

Another aspect to be dealt with is a very practical one: how to integrate this approach into an application store, where users could download applications for use on their local platforms. As it stands, the developer would have to submit their specification and implementation modules as source code, and the application store would need to compile them together, to ensure that the contracts and modules have not been tampered with. The application store – which the user would have to trust – could then distribute compiled versions of the application which would be compatible with the run-time library locally available on users' devices. Clearly, this is not be desirable in all situations: most commercial application developers submit compiled versions of their software, which in our case could allow them to *e.g.,* modify the contracts, rendering the applications unsafe.

[5] The page "Typed Racket Plans" at https://github.com/plt/racket/wiki/Typed-Racket-plans gives us hope.

7. CONCLUSION

In conclusion, we have tried to address the problem of resource access control to protect privacy of end users' sensitive data. Taking inspiration from the DiaSuite approach, we demonstrate an embedded DSL for specifying applications, which itself unfolds to a programming environment, placing restrictions on the application developer. While the prototype is not a perfect solution to the problem, it does demonstrate a novel approach to resource control which is very versatile, by nature of being entirely composed of embedded DSLs. It also offers users more insight into what sensitive resources are used for, compared to currently widespread mobile platforms.

In the future, it would be interesting to explore the use of other Racket libraries, particularly Typed Racket, in the hope that we can achieve more reliable restrictions than currently possible. This might be an avenue to pursue in response to the vulnerability that the current prototype has, arising from evaluation of dynamically constructed expressions or allowing module importing.

Acknowledgements

Special thanks go to Ludovic Courtès, Camille Mañano, Andreas Enge, and Hamish Ivey-Law, who proofread early drafts of this work and provided invaluable comments. The constructive criticism provided by the anonymous reviewers is equally appreciated.

8. REFERENCES

[1] D. Cassou, E. Balland, C. Consel, and J. Lawall. Leveraging software architectures to guide and verify the development of Sense/Compute/Control applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 431–440, New York, NY, USA, 2011. ACM.

[2] D. Cassou, J. Bruneau, C. Consel, and E. Balland. Toward a tool-based development methodology for pervasive computing applications. *IEEE Trans. Software Eng.*, 38(6):1445–1463, 2012.

[3] Chrome developers. Developing Chrome extensions: Declare permissions. https://developer.chrome.com/extensions/declare_permissions, 2015. Accessed 2/2015.

[4] J. L. Dave Mark. *Beginning iPhone Development: Exploring the iPhone SDK*. Apress, 2009.

[5] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *ACM SIGPLAN Notices*, volume 46, pages 215–226. ACM, 2011.

[6] K. O. Elish, D. D. Yao, B. G. Ryder, and X. Jiang. A static assurance analysis of Android applications. *Virginia Polytechnic Institute and State University, Tech. Rep*, 2013.

[7] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: an information flow tracking system for real-time privacy monitoring on smartphones. *Communications of the ACM*, 57(3):99–106, 2014.

[8] J. Feiler. *How to Do Everything: Facebook Applications*. McGraw-Hill, Inc., New York, NY, USA, 1st edition, 2008.

[9] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: a programming environment for Scheme. *J. Funct. Program.*, 12(2):159–182, 2002.

[10] M. Flatt. Submodules in Racket: You want it when, again? In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 13–22, New York, NY, USA, 2013. ACM.

[11] M. Flatt, R. Culpepper, D. Darais, and R. B. Findler. Macros that work together. *Journal of Functional Programming*, 22:181–216, 3 2012.

[12] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[13] C. Gibler, J. Crussel, J. Erickson, and H. Chen. AndroidLeaks: Detecting privacy leaks in Android applications. Technical report, Tech. rep., UC Davis, 2011.

[14] R. N. Horspool and N. Tillmann. *TouchDevelop: Programming on the Go*. The Expert's Voice. Apress, 3rd edition, 2013. available at https://www.touchdevelop.com/docs/book.

[15] C. Mann and A. Starostin. A framework for static detection of privacy leaks in Android applications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, pages 1457–1462. ACM, 2012.

[16] R. Rogers, J. Lombardo, Z. Mednieks, and B. Meike. *Android Application Development: Programming with the Google SDK*. O'Reilly, Beijing, 2009.

[17] M. Snoyman. *Developing Web Applications with Haskell and Yesod – Safety-Driven Web Development*. O'Reilly, 2012.

[18] R. Stevens, C. Gibler, J. Crussell, J. Erickson, and H. Chen. Investigating user privacy in Android ad libraries, 2012.

[19] R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software architecture: foundations, theory, and practice*. Wiley Publishing, 2009.

[20] S. Tobin-Hochstadt and M. Flatt. Advanced macrology and the implementation of Typed Scheme. In *In Proc. 8th Workshop on Scheme and Functional Programming*, pages 1–14. ACM Press, 2007.

[21] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *ACM SIGPLAN Notices*, volume 46, pages 132–141. ACM, 2011.

[22] X. Wei, L. Gomez, I. Neamtiu, and M. Faloutsos. Permission evolution in the Android ecosystem. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 31–40. ACM, 2012.

[23] X. Xiao, N. Tillmann, M. Fähndrich, J. de Halleux, and M. Moskal. User-aware privacy control via extended static information-flow analysis. In M. Goedicke, T. Menzies, and M. Saeki, editors, *ASE*, pages 80–89. ACM, 2012.

Processing List Elements in Reverse Order

Irène Durand and Robert Strandh
University of Bordeaux
Science and Technology College
LaBRI, 351, Cours de la Libération
33405 Talence Cedex, France
robert.strandh@u-bordeaux.fr
irene.durand@u-bordeaux.fr

ABSTRACT

The Common Lisp sequence functions and some other functions such as `reduce` accept a keyword parameter called `from-end`. In the case of `count` and `reduce`, when the value of that parameter is *true*, it is required that the elements are processed in reverse order. Some implementations, in particular SBCL, CCL, and LispWorks, implement the reverse-order traversal of a list by non-destructively reversing the list and then traversing the reversed version instead. This technique requires $O(n)$ additional heap space (where n is the length of the list), and increases the amount of work required by the garbage collector.

In this paper, we present a technique that only uses additional *stack space*. To avoid stack overflow, our technique traverses parts of the list multiple times when the list has more elements than the available stack space can handle. We show that our technique is fast, in particular for lists with a large number of elements, which is also the case where it is the most important to avoid allocating heap space.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Code generation, Run-time environments*

General Terms

Algorithms, Languages

Keywords

Common Lisp, List processing

1. INTRODUCTION

The Common Lisp *sequence* functions are defined to work on lists as well as vectors. Furthermore, many of these sequence functions accept a keyword argument `from-end` that alters the behavior, in that elements toward the end of the sequence are favored over elements toward the beginning of the sequence. Other functions, in particular `reduce`, also accept this keyword argument.

Most sequence functions are not required to process the elements from the end of the sequence even though the value of the `from-end` keyword argument is *true*. For example, it is allowed for `find` to compare elements from the beginning of the sequence and return the *last* element that *satisfies the test*[1] even if the test has side effects. There is one exception, however: The `count` function is required by the standard to test the elements from the end of the sequence.[2] In addition to the function `count`, the function `reduce` also requires processing from the end of the list when `from-end` is *true*.

Processing list elements from the beginning to the end could, however, have a significant additional cost associated with it when processing from the end would require fewer executions of the test function, and the additional cost increases with the complexity of the test.

In this paper, we will concentrate on the functions that are required by the standard to process list elements from the end, and we will use only the function `count` in our test cases.

There are of course some very simple techniques for processing elements from the end of a list. One such technique would be to start by reversing the list[3] and processing the elements from the beginning in the reversed list. This technique is used by several implementations, including SBCL, CCL, and LispWorks. A major disadvantage of this technique is that it requires $O(n)$ additional heap space, and that it requires additional execution time by the memory allocator and the garbage collector.

Another simple technique would be to traverse the list *recursively* and testing the elements during the *backtracking*

ELS '15, April 20 - 21 2015, London, UK

[1]The phrase *satisfy the test* has a precise meaning in the Common Lisp standard as shown in section 17.2 in that document.
[2]Though if the test has no side effects and cannot fail, as is the case of functions such as `eq` or `eql`, testing from the beginning is arguably conforming behavior.
[3]By *reversing the list* we do not mean modifying the list as `nreverse` would do, but creating a new list with the elements in reverse order as `reverse` would do. The reason for excluding modifications to the list is that doing so might influence the semantics of other functions, including perhaps the test function or the view of the list by other threads.

phase of the recursion.[4] Again, $O(n)$ extra space is required, even though this time it is *stack space* rather than heap space, so that the memory allocator and the garbage collector are not solicited, at least in most implementations. Worse, many implementations have a fairly small call stack, in particular in multi-threaded implementations where each thread must have a dedicated stack. Aside from these disadvantages, this technique is however fairly efficient in terms of execution time, because a simple function call is quite fast on most modern processors. For that reason, we will use recursion as the basis of the technique described in this paper, but with fairly few recursive calls so that the additional extra space is modest.

Throughout this paper, we assume that the lists to be processed have a large number of elements, for several reasons:

- We do not want the list to be small enough to fit in the cache, because cache performance depends on other workload as well.
- For short lists, performance may be dominated by the overhead of calling a few functions, or by loop prologues and epilogues. By using long lists, we make sure that performance is dominated by traversing the list and computing the test.
- If the list is too short, it can be processed by a simple recursive technique. In order to avoid this possibility, we want the lists to have orders of magnitude more elements than the size of the stack.

Furthermore, throughout this paper, we will assume that the *test* to be performed on the elements of the list is the function `eq`. By making this assumption, we expose the worst case for our technique, because the execution time will then be dominated by the overhead of traversing the list, as opposed to by executing the test function.

It should be noted that the difficulty of processing list elements in reverse order is due to the way Common Lisp practically imposes the representations of such lists. Other representations are, of course, possible. For instance, Hughes [2] suggested a representation of lists as first-class functions. Similarly, in his talk on parallelism in 2009,[5] Guy Steele proposed a representation of lists for parallel processing, based on using the four operations `item`, `split`, `list`, and `conc`. Clearly, such alternative representations could be devised that facilitate processing elements in reverse order.

In this paper, we use the international convention [1] for writing logarithms. Hence, we write $\mathsf{lb}\, n$ for the logarithm in base 2. We use log only when the base is unimportant. Given a real number n, the notation $\lfloor n \rfloor$ represents the *floor* of n and $\lceil n \rceil$ *cieling*. For example, $\lfloor 1.5 \rfloor = 1$ and $\lceil 1.5 \rceil = 2$.

[4]Despite considerable research, we have not been able to find any original reference to this technique, and it seems to trivial for standard text books to even discuss. We must conclude that this technique must be so obvious that it was probably discovered independenlty by several people.

[5]http://groups.csail.mit.edu/mac/users/gjs/6.945/readings/MITApril2009Steele.pdf

2. PREVIOUS WORK

We will frequently refer to techniques used by SBCL because of its reputation as a high-performance implementation. We will however also use other high-performance implementations for comparison when we have information on the techniques used by those implementations, or when we can reasonably guess these techniques from other evidence.

For its implementation of `find`, SBCL takes advantage of the freedom given by the standard, by processing elements from the beginning, and remembering the last element that satisfies the test. For implementations where the technique is unknown, it suffices to write a test function that counts the number of times it is invoked and run it on a list where only the last element satisfies the test.

For its implementation of `count`, SBCL uses the simple technique of reversing the list first and then processing the elements of the reversed list from the beginning.

As we already mentioned, we use recursion as the basis of our technique, because it is quite fast. We devised the following test to verify this hypothesis:

```
(defun recursive-count (x list)
  (declare (optimize (speed 3) (debug 0) (safety 0)))
  (if (endp list)
      0
      (+ (recursive-count x (cdr list))
         (if (eq (car list) x) 1 0))))
```

On SBCL executing this function on a list with 50000 elements where no element satisfies the test takes around 4ns per element compared to around 1.5ns for an explicit loop from the beginning of the list, and around twice as fast as calling `count`. This result indicates that we should use recursion whenever the size of the stack allows it, though there is of course no portable way of testing how much stack space is available. However, each implementation might have a specific way, which would then be good enough.

3. OUR TECHNIQUE

3.1 Basic technique

To illustrate our technique, we first show a very simple version of it in the form of the following code:[6]

```
(defun count-from-end (x list)
  (labels ((aux (x list n)
             (cond ((= n 0) 0)
                   ((= n 1)
                    (if (eq x (car list)) 1 0))
                   (t (let* ((n/2 (ash n -1))
                             (half (nthcdr n/2 list)))
                        (+ (aux x half (- n n/2))
                           (aux x list n/2)))))))
    (aux x list (length list))))
```

This function starts by computing the length of the list and then calling the auxiliary function with the original arguments and the length. The auxiliary function calls `nthcdr`

[6]Throughout this paper, we rely on the left-to-right evaluation order mandated by the Common Lisp standard.

in order to get a reference to about half the list it was passed. Then it makes two recursive calls, first with the second half of the list and then with the first half of the list. The recursion terminates when the list has a single element or no element in it. When it has no element in it, clearly the count is 0. When it has a single element in it, the element is compared to the argument `x` and if they are the same, the value 1 is returned, otherwise 0 is returned.

The main feature of our technique is that it trades fewer recursive calls for multiple traversals of the list. The maximum number of simultaneous active invocations of this simple function is around $\mathsf{lb}\, n$, where n is the length of the list. The maximum value of this number is quite modest. On a 64-bit processor, it can never exceed 60 and it is significantly smaller in practice of course. The number of `cdr` operations can be approximately expressed as $n\,(1 + \frac{1}{2}\mathsf{lb}\, n)$. In Section 3.3 we analyze this result in greater depth.

The best case for this function is very efficient indeed. The worst case is unacceptably slow. Even for a list of some reasonable length such as a million elements, the execution time is a factor 6 slower than for the best case.

The remainder of this section is dedicated to ways of improving on the performance of the basic technique.

3.2 Using more stack space

By far the most important improvement to the basic technique is to take advantage of the available stack space to decrease the number of multiple list traversals required by the basic technique.

The following example illustrates this technique by using the simple recursive traversal if there are fewer than 10000 elements in the list.[7] If there are more elements, then it divides the list in two, just like the basic technique shown in Section 3.1.

```
(defun count-from-end-2 (x list)
  (labels ((recursive (x list n)
             (if (zerop n)
                 0
                 (+ (recursive x (cdr list) (1- n))
                    (if (eq x (car list)) 1 0))))
           (aux (x list n)
             (if (<= n 10000)
                 (recursive x list n)
                 (let* ((n/2 (ash n -1))
                        (half (nthcdr n/2 list)))
                   (+
                    (aux x half (- n n/2))
                    (aux x list n/2))))))
    (aux x list (length list))))
```

With this improvement, the number of `cdr` operations required can now be expressed as approximately

$$n\,(1 + \frac{1}{2}\mathsf{lb}\,\frac{n}{10000})$$

which is significantly better than the corresponding value for the basic technique.

However, there is no particular reason to divide the list into 2 equal-sized parts when there are too many elements for the basic technique. Section 4 gives a more complete explanation of the parameters involved and how they influence the execution time of the resulting code.

3.3 Analyses

In this section we give approximate formulas for the performance of our technique. The basic measure we are interested in is the number of `cdr` operations that must be performed as a function of the number of elements of the list. We will denote the number of elements of the list by N and the number of `cdr` operations required by $F(N)$. Since our technique always starts by traversing the entire list in order to compute N, we can always write $F(N)$ as $N + f(N)$, were $f(N)$ is the number of `cdr` operations required in the subsequent step.

For the basic technique where the list is divided into two equal-size sublists, we obtain the following recursive relation:

$$f(N) = \begin{cases} 0 & \text{if } N = 1 \\ \lfloor \frac{N}{2} \rfloor + f(\lfloor \frac{N}{2} \rfloor) + f(\lceil \frac{N}{2} \rceil) & \text{otherwise} \end{cases}$$

In order to obtain an approximate solution to this relation, we can solve for N being a power of 2, i.e., $N = 2^n$. In that case, for $N > 1$ we obtain:

$$f(N) = \frac{N}{2} + 2f(\frac{N}{2})$$

The details of the approximate resolution of this recursive equation is given in the appendix. This solution yields

$$f(N) = \frac{N}{2}\mathsf{lb}\ N + Nf(1) = \frac{N}{2}\mathsf{lb}\ N$$

Including the traversal to compute the number of elements of the list, we obtain:

$$F(N) = \frac{N}{2}\mathsf{lb}\ N + N = N(1 + \frac{1}{2}\mathsf{lb}\ N)$$

which is clearly $O(N\mathsf{log}\ N)$. More importantly, for a list with around 16 million elements (which fills the default heap of most implementations we have tested), we have $N \approx 2^{24}$ which gives $F(N) \approx 13N$ which is probably unacceptably slow.

Let us now consider what happens when we are able to handle more than a single element with the basic recursive technique, as shown in Section 3.2. We denote the number of

[7] The number 10000 was chosen to be a significant part of a typical per-thread default stack while still leaving room for stack space required by callers and callees of this function. In a real production implementation, the number would be chosen based on the remaining space left on the stack when the function is called.

elements that the basic recursive technique can handle by K, and again, in order to simplify the analysis, we assume that both N and K are powers of 2, i.e., $N = 2^n$ $K = 2^k$, and also that $N \geq K$. The recursion relation now looks as follows:

$$f(N) = \begin{cases} N - 1 & \text{if } N \leq K \\ \frac{N}{2} + 2f(\frac{N}{2}) & \text{otherwise} \end{cases}$$

The resolution of this equation is given in the appendix (Part B). It yields:

$$f(N) \approx N(1 + \frac{1}{2}\mathsf{lb}\ \frac{N}{K})$$

With the best portable version of our technique and a typical stack being able to handle $K = 2^{14}$ we are now looking at a performance for $N = 2^{24}$ of $F(N) \approx 6N$. Comparing this result to the technique of reversing the list, it is fair to say that the overhead of allocating and subsequently garbage-collecting a `cons` cell can very well be comparable to 6 times the time taken by the `cdr` operation. In other words, the performance of our portable version is already comparable to an implementation based on first creating a reversed copy of the list and then traversing that reversed copy.

Finally, instead of using more stack space for the base case, let us analyze what happens if we divide the original list into more than two parts. For this analysis, let us assume that we divide the list into M equal parts, and that M also is a power of 2 so that $M = 2^m$. We then obtain the following relation:

$$f(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - \frac{N}{M} + Mf(\frac{N}{M}) & \text{otherwise} \end{cases}$$

The resolution of this equation is given in the appendix (Part C). It yields:

$$F(N) \approx N(1 + \frac{\mathsf{lb}\ N}{\mathsf{lb}\ M})$$

While it may appear that we can get very good performance when M is chosen to be large, in practice, using large values of M introduces a different kind of overhead, namely large stack frames, making the gain less than what the formula suggests.

3.4 Implementation-specific solutions

So far, we have explored techniques that can mostly be implemented in portable Common Lisp. In this section, we explore a variation on our technique that requires access to the control stack of the implementation.

Recall that at the lowest level of our technique, there is a recursive function that is used for traversing the list when the number of elements is small compared to the stack size. At each invocation, this function does very little work.

With direct access to the control stack, we can convert the recursive function to an iterative function that pushes the elements of the list on the control stack, and then processes them in reverse order. This technique has several advantages:

- A single word is used for each element, whereas the recursive function requires space for a return address, a frame pointer, saved registers, etc. As a result, this technique can be used for lists with more elements than would be possible with the recursive technique, thereby further decreasing the number of times a list is traversed.
- There is no function-call overhead involved. The only processing that is needed for an element is to store it on the stack and then comparing it to the item.

We illustrate this technique in a notation similar to Common Lisp:

```
(defun low-level-reverse-count (item list length)
  (loop for rest = list then (cdr rest)
        repeat length
        do (push-on-stack (car rest)))
  (loop repeat length
        count (eq item (pop-from-stack))))
```

We implemented this technique in SBCL. In order not to have to recompile SBCL with our additional function, we used the implementation-specific foreign-function interface and wrote the function using the language C. Rather than pushing and popping the control stack, we used the built-in C function `alloca` to allocate a one-dimensional C array on the control stack to hold the list elements.

In SBCL, the default stack size is 2MBytes, or around 250k words on a 64-bit processor. We tested our technique using 100000 words on the stack. The result is that for a list with 10 million elements, our technique processes the list in reverse order as fast as an ordinary loop from the beginning of the list.

This surprising result can be explained by a few factors:

- Presumably in order to speed up the functions `car` and `cdr`, SBCL uses the same tag for `cons` cells and for the symbol `nil`. As a result, in order to traverse a list, SBCL must make *two* tests for each element, namely one to check whether the putative list is something other than a list altogether, and another to check whether it is a `cons` cell. When our technique traverses a list for which the number of elements is known, there is no need to make any additional tests, simply because when the length of the list is positive, the first element must be a `cons` cell.

- The SBCL compiler can not determine that the return value of `count` must always be a `fixnum`.[8] When the function is implemented in C, this problem disappears.

If we put this technique in the perspective of the analyses in Section 3.3, we can also see that the number of `cdr` operations remains quite modest, even for lists with a very large number of elements.

There are several variations on this implementation-specific technique. Some implementations might allocate a vector or a list declared to be `dynamic-extent` on the stack, thus giving essentially the same advantage as the version we implemented in C. However, such a technique would still be implementation specific, given that it is permitted for the compiler to ignore `dynamic-extent` declarations. In the case of SBCL, using such a declaration, we were able to obtain performance almost as good as our C version. However, as it turns out, SBCL only allocates a vector on the stack under certain circumstances thereby making this technique impossible to apply in general.

4. BENCHMARKS

We implemented ten different versions of `reverse-count`, a function that counts elements of a list from the end. The difference between these versions can be expressed in terms of two different numeric parameters, namely:

1. the minimum number of elements for which we apply the logarithmic method, consisting of dividing the list into two equal-size halves, and
2. into how many chunks do we cut the list when the number of elements is smaller than the first parameter, but larger than the number of elements that can be handled by the simple recursive technique.

In all of our versions, when the number of elements is less than 10000, we process the elements using the purely recursive technique where the element is processed in the backtracking phase.

For the purpose of this article, we have retained the one with the best experimental behavior (`v7`) and compared it to two more traditional versions (`v0` and `v1`).

These three versions can be characterized as follows:

0. Version `v0` uses the native `count` function called with the `from-end` keyword argument set to `t`,

```
;; standard version
(defun reverse-count-0 (x list)
  (count x list :from-end t :test #'eq))
```

1. Version `v1` is the naive version consisting in reversing the list before counting; this version uses the heap space and no stack space.

```
(defun reverse-count-1 (x list)
  (declare (optimize
            (speed 3) (debug 0) (safety 0)
            (compilation-speed 0)))
  (loop for e in (reverse list)
        count (eq x e)))
```

7. Version `v7` divides the list in 2 parts if it has more than one hundred million elements. Otherwise, if it has more than 10000 elements, it divides it into chunks that have 10000 elements each. Finally, if it has no more than 10000 elements, then it uses the standard recursive method.

   We think this method is faster than the others, at least for lengths no more than one hundred million elements, because then it is guaranteed to traverse the list at most 3 times + 1 time for computing the length. It could be improved for lengths greater than one hundred million by using a better division than 2 in this case, but we have not attempted that improvement. The code is given below.

```
(defun count-from-end-with-length-7 (x list length)
  (declare (optimize (speed 3) (safety 0) (debug 0)
            (compilation-speed 0)))
  (declare (type fixnum length))
  (labels (;; AUX1 is the recursive traversal
           ;; by CDR.
           (aux1 (x list length)
             (declare (type fixnum length))
             (if (zerop length)
                 0
                 (+ (aux1 x (cdr list) (1- length))
                    (if (eq x (car list))
                        1
                        0))))
           ;; AUX2 recursive traversal
           ;; by (NTHCDR 10000 ...).
           ;; used when the length of the list is
           ;; less than 100000000.
           (aux2 (x list length)
             (declare (type fixnum length))
             (if (<= length 10000)
                 (aux1 x list length)
                 (+ (aux2 x
                          (nthcdr 10000 list)
                          (- length 10000))
                    (aux1 x list 10000))))
           ;; AUX3 recursive traversal
           ;; by half the size of the list.
           ;; used for lists that have more than
           ;; 100000000 elements.
           (aux3 (x list length)
             (declare (type fixnum length))
             (if (< length 100000000)
                 (aux2 x list length)
                 (let* ((n (ash length -1))
                        (middle (nthcdr n list)))
```

[8] On a byte-addressed processor where n word-aligned bytes are needed to represent a `cons` cell, the number of elements in a list can be at most N/n where N is the maximum number of possible addresses. In a system that uses at most lb n tag bits for a fixnum, the value that `count` returns must be a fixnum. While some systems might use 8 tag bits, SBCL on a 64-bit platform uses a single tag bit for fixnums. As a consequence, `count` must then return a fixnum.

```
                  (+ (aux3 x middle (- length n))
                     (aux3 x list n))))))
    (aux3 x list length)))

(defun reverse-count-7 (x list)
  (count-from-end-with-length-7
   x list (length list)))
```

Thanks to the help of the Lisp community, we could test the behavior of these three versions on several implementations and architectures. In Figure 1, we summarize the results of tests that worked for a list of size up to 10^7. In many cases, the details of the implementation are unknown or not shown. However, the purpose of the Figure 1 is not to compare performance between different systems, but to compare the performance of different versions of `count` on each system. For that reason, the exact details of the system are unimportant; we are only interested in whether `v7` compares favorably to the other versions on most systems. Furthermore, for some implementations, we had to change the `optimize` settings and some other parameters in order to get our code to work.[9] For that reason, it is not possible to compare the performance on different implementations in Figure 1, even when the processor and the clock frequency are the same.

To get a better idea of the difference in performance between the three versions of the `count` function, we selected the table entry corresponding to the non-commercial implementation that resulted in the greatest advantage of our `v7` compared to the other versions (`Clozure CL 1.10-dev`), and we rendered the performance in the form of a graph. The result is shown in Figure 2.

As Figure 2 shows, the performance of `v7` is significantly better than that of the other versions. Furthermore, the fact that the curve for `v7` is smoother than the curves for other versions indicates that the performance of `v7` is more predictable. We attribute this behavior to the garbage collector, which occasionally has to run when heap allocation is required. Since `v7` does not require any heap allocation, the garbage collector is not solicited.

5. CONCLUSIONS AND FUTURE WORK

We have presented a general technique for processing elements of a list from the last element to the first. The implementation-specific version of our technique is comparable in speed to traversing the list from the first to the last element for all reasonably-sized lists. For very long lists, the performance of our technique degrades modestly and gracefully.

Even the implementation-independent version of our technique performs well enough that it is preferable to the existing technique of reversing the list that is used in some implementations.

We have presented our technique in the context of the function `count` because, together with `reduce`, processing the elements from the end is required by the HyperSpec, whereas, for other functions accepting the keyword argument `from-end`, it is explicitly allowed to process the elements from the beginning to the end.

However, our technique is potentially even more interesting to use with functions such as `find` and `position` for which it is not required to process the elements from the end to the beginning. The reason is that when elements are processed form the beginning to the end in these functions, *all* elements must be tested. When the combination of the `test` and the `key` functions has non-trivial computational cost, a significant amount of work may be wasted. However, when elements are processed in reverse order, a result can be returned when the test is satisfied the first time, thereby avoiding such wasted work.

Since the performance of our technique is not significantly worse than processing from the beginning to the end, it is very likely that our technique will be faster in almost all cases. Only when the last element of the list to satisfy the test is close to the beginning of the list will our technique apply the test as many times as when processing is done from the beginning to the end. We conjecture that, on the average, our technique will be faster whenever the cost of applying the test is at least that of a function call. If so, our technique should be used in all cases except for those using a very inexpensive test functions such as `eq`, and when the implementation then uses a special version of the sequence function where this test is inlined, so as to avoid a function call.

Further research is required in order to verify our conjecture. In order to determine the result with some accuracy, additional parameters have to be taken into account. In particular, the position of the last element in the list to satisfy the test must be taken into account, as well as the cost of calling the test function. As usual, benchmarks will have to be performed on a variety of implementations and processors, further complicating the verification of our conjecture.

6. ACKNOWLEDGMENTS

We would like to thank Pascal Bourguignon for reading and commenting on an early draft of this paper. We would also like to thank Alastair Bridgewater for helping us with the foreign-function interface of SBCL. Finally, we would like to thank Pascal Bourguignon, Alastair Bridgewater, James Kalenius, Steven Styer, Nicolas Hafner, and Eric Lind for helping us run benchmarks on platforms that are unavailable to us. Finally, we would like to thank Martin Simmons at LispWorks technical support for giving us the information we needed in order to explain the performance of our technique on the LispWorks Common Lisp implementation.

[9] In particular, LispWorks has a much smaller default stack than for instance SBCL (16k words, compared to 250k words) resulting in stack overflow of our benchmark with default parameters. For that reason, we ran the LispWorks benchmark with a smaller stack and with a higher value of `safety`. The combination of these factors is the likely explanation to the absence of performance improvement for LispWorks. However, we are told that on 32-bit LispWorks our technique gives a factor 10 improvement.

System characteristics				Time in seconds		
Implementation	Version	Processor	Frequency	v0	v1	v7
LispWorks	6.1.1	Intel Core	?	0.20	0.18	0.14
Clozure CL	1.10	Intel Xeon	3.33GHz	1.93	1.79	0.15
Clozure CL	1.10-dev	AMD FX	?	1.77	1.63	0.15
SBCL	1.2.8	Intel Xeon	3.33GHz	0.51	0.27	0.22
ABCL	1.3.1	Intel Xeon	3.33GHz	1.13	0.22	0.34
CLISP	2.49	X86_64	?	1.15	1.14	0.87
ECL	13.5.1	?	?	0.69	0.41	0.36
SBCL	1.2.7	Intel Core	2.53GHz	0.36	0.38	0.25

Figure 1: Performances of the three versions on several systems with a list of 10^7 elements

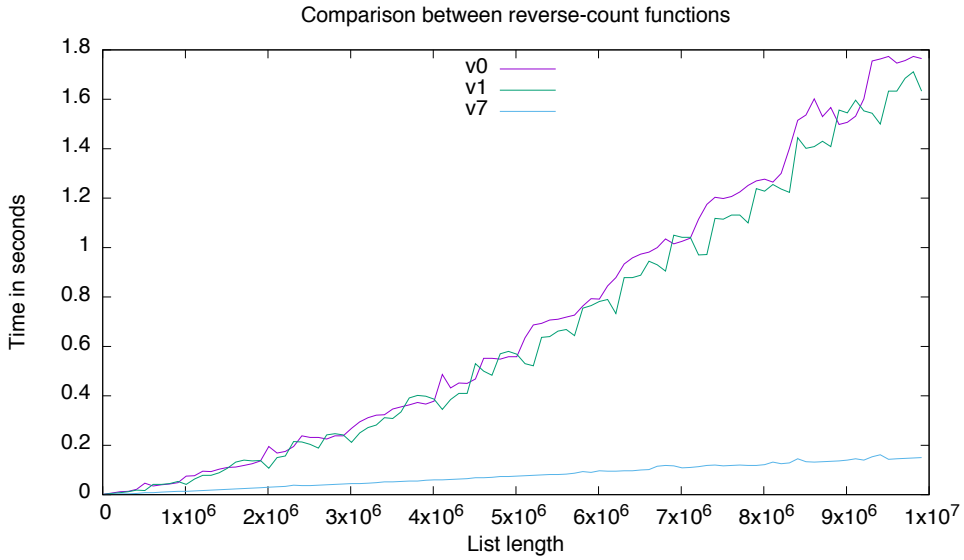


Figure 2: Comparison of the behavior of the three versions on a single system

APPENDIX

Part A

In this part, we develop the details of the approximate solution of the recursive equation defined in Section 3.3.

$$f(N) = \frac{N}{2} + 2f(\frac{N}{2})$$

Developing the second term of this equation one step, we obtain:

$$f(N) = \frac{N}{2} + 2f(\frac{N}{2}) =$$

$$= \frac{N}{2} + 2(\frac{N}{4} + 2f(\frac{N}{4})) =$$

$$= \frac{N}{2} + \frac{N}{2} + 4f(\frac{N}{4}) =$$

$$= 2\frac{N}{2} + 4f(\frac{N}{4})$$

Developing the second term of this equation one more step, we obtain:

$$f(N) = 2\frac{N}{2} + 4f(\frac{N}{4}) =$$

$$= 2\frac{N}{2} + 4(\frac{N}{8} + 2f(\frac{N}{8})) =$$

$$= 2\frac{N}{2} + \frac{N}{2} + 8f(\frac{N}{8}) =$$

$$= 3\frac{N}{2} + 8f(\frac{N}{8})$$

After developing the second term $p-1$ times, we obtain:

$$f(N) = p\frac{N}{2} + 2^p f(\frac{N}{2^p})$$

When $p = n = \mathsf{lb}\ N$, this equation turns into:

$$f(N) = \frac{N}{2}\mathsf{lb}\ N + Nf(1) = \frac{N}{2}\mathsf{lb}\ N$$

Part B

In this part, we develop the details of the approximate solution of the recursive equation defined in Section 3.3.

$$f(N) = \begin{cases} N-1 & \text{if } N \leq K \\ \frac{N}{2} + 2f(\frac{N}{2}) & \text{otherwise} \end{cases}$$

Since the second equation is the same as for the basic technique, developing the second equation $p-1$ times, we again obtain:

$$f(N) = p\frac{N}{2} + 2^p f(\frac{2^n}{2^p})$$

When $p = n - k = \mathsf{lb}\ \frac{N}{K}$ we get:

$$f(N) = \frac{N}{2}\mathsf{lb}\ \frac{N}{K} + \frac{N}{K}f(K)$$

Substituting $K-1$ for $f(K)$ and factoring out N, we obtain:

$$f(N) = N(\frac{1}{2}\mathsf{lb}\ \frac{N}{K} + \frac{K-1}{K})$$

Or:

$$f(N) = N(1 - \frac{1}{K} + \frac{1}{2}\mathsf{lb}\ \frac{N}{K})$$

Clearly, the term $\frac{1}{K}$ can be ignored, giving:

$$f(N) \approx N(1 + \frac{1}{2}\mathsf{lb}\ \frac{N}{K})$$

Part C

In this part, we develop the details of the approximate solution of the recursive equation defined in Section 3.3.

$$f(N) = \begin{cases} 0 & \text{if } N = 1 \\ N - \frac{N}{M} + Mf(\frac{N}{M}) & \text{otherwise} \end{cases}$$

Solving as before, after developing the last term once, we obtain:

$$f(N) = N - \frac{N}{M} + Mf(\frac{N}{M}) =$$

$$= N - \frac{N}{M} + M(\frac{N}{M} - \frac{N}{M^2} + Mf(\frac{N}{M^2})) =$$

$$= N - \frac{N}{M} + N - \frac{N}{M} + M^2 f(\frac{N}{M^2}) =$$

$$= 2(N - \frac{N}{M}) + M^2 f(\frac{N}{M^2})$$

Developing the last term a second time, we obtain:

$$f(N) = 2(N - \frac{N}{M}) + M^2 f(\frac{N}{M^2}) =$$

$$= 2(N - \frac{N}{M}) + M^2(\frac{N}{M^2} - \frac{N}{M^3} + Mf(\frac{N}{M^3})) =$$

$$= 2(N - \frac{N}{M}) + N - \frac{N}{M} + M^3 f(\frac{N}{M^3}) =$$

$$= 3(N - \frac{N}{M}) + M^3 f(\frac{N}{M^3})$$

After developing the last term $p-1$ times, we obtain:

$$f(N) = p(N - \frac{N}{M}) + M^p f(\frac{N}{M^p}) =$$

Setting $p = \frac{n}{m} = \frac{\mathsf{lb}\ N}{\mathsf{lb}\ M}$ so that $M^p = N$, we get:

$$f(N) = \frac{\mathsf{lb}\ N}{\mathsf{lb}\ M}(N - \frac{N}{M}) + Nf(1) = \frac{\mathsf{lb}\ N}{\mathsf{lb}\ M}(N - \frac{N}{M})$$

Factoring out N, we obtain:

$$f(N) = N(1 - \frac{1}{M})\frac{\mathsf{lb}\ N}{\mathsf{lb}\ M}$$

and thus:

$$F(N) = N(1 + (1 - \frac{1}{M})\frac{\mathsf{lb}\ N}{\mathsf{lb}\ M})$$

and again:

$$F(N) \approx N(1 + \frac{\mathsf{lb}\ N}{\mathsf{lb}\ M})$$

7. REFERENCES

[1] *ISO 80000-2:2009 Quantities and Units – part 2: Mathematical signs and symbols to be used in the natural sciences and technology.* International Organization for Standardization, 2009.

[2] R. J. M. Hughes. A novel representation of lists, and its application to the function "reverse". *Information Processing Letters*, 22(3):141–144, 1986.

Executable Pseudocode for Graph Algorithms

Breanndán Ó Nualláin
University of Amsterdam
bon@science.uva.nl

ABSTRACT

Algorithms are written in pseudocode. However the implementation of an algorithm in a conventional, imperative programming language can often be scattered over hundreds of lines of code thus obscuring its essence. This can lead to difficulties in understanding or verifying the code. Adapting or varying the original algorithm can be laborious.

We present a case study showing the use of Common Lisp macros to provide an embedded, domain-specific language for graph algorithms. This allows these algorithms to be presented in Lisp in a form directly comparable to their pseudocode, allowing rapid prototyping at the algorithm level.

As a proof of concept, we implement Brandes' algorithm for computing the betweenness centrality of a graph and see how our implementation compares favourably with state-of-the-art implementations in imperative programming languages, not only in terms of clarity and verisimilitude to the pseudocode, but also execution speed.

Categories and Subject Descriptors

G.2.2 [**Graph Theory**]: Graph algorithms; E.1 [**Data Structures**]: Graphs and networks; D.3.3 [**Language Constructs and Features**]: Patterns; D.2.3 [**Coding Tools and Techniques**]: Control Structures

General Terms

Algorithms, Design, Languages

Keywords

Graph algorithms, Lisp macros, Pseudocode, Verification

1. INTRODUCTION

Much effort is invested in ensuring that programs faithfully implement the algorithms on which they are based. Test-driven development [7], Software Verification [13] and a variety of other methodologies have been developed in efforts to achieve this goal.

But what could be better than a computer program that not only resembles the algorithm upon which it is based so closely as to inspire confidence in its implementation, but also runs with an efficiency competitive with more verbose implementations in lower-level programming languages?

We present a proof of concept of a Common Lisp library for programming in this manner and argue that it fulfils the above desiderata as well as having further advantages for pedagogy and experimentation in the field of algorithms.

Pseudograph is a Common Lisp library which provides this functionality permitting graph algorithms to be written in a manner similar to their pseudocode.

2. PSEUDOCODE

The *lingua franca* for presenting algorithms is pseudocode. Pseudocode is a jargon intended to be understood by practising professional programmers and computer scientists but lacking any formal semantics or standard. Students of data structures and algorithms learn pseudocode from their textbooks [10]. Computer scientists use it to publish descriptions of novel algorithms [12].

Pseudocode describes algorithms in a way which is programming-language independent. Machine-level implementation details are omitted, as are any consideration of data abstraction and error handling.

Although pseudocode is intended to be read by humans, not by machines, some attempts have been made to design programming languages which have more of a natural-language nature [16, 1]. Indeed, the syntax of the Python programming language [26] has been praised for its clarity and the natural way in which it can express algorithms [21].

3. LISP APPROACHES

In his book Practical Common Lisp [27, pp. 250–252], Peter Seibel presents an approach to using Common Lisp's `tagbody` and `go` special operators to "*translate*" algorithms from pseudocode into working Lisp code which is subsequently manually refactored into more natural Lisp code.

As an example he translates Donald Knuth's Algorithm S from the Art of Computer Programming [19, p. 142].

First the algorithm is translated into Lisp code which although "*not the prettiest*" is a verifiably "*faithful translation of Knuth's algorithm*". Subsequently the code is manually refactored, checking at each step, until it no longer resembles Knuth's recipe but still gives confidence that it indeed implements it.

This approach goes against the grain. Lisp is a programmable programming language that we should be able to bend to our will, shaping the language from the bottom up until it can express the problems we are tackling at the level at which they are expressed [17].

Lisp is the ultimate extensible programming language, deriving this power from the Lisp macro. Lisp macros allow us to create new binding constructs and control constructs, alter evaluation order, define data languages and improve code readability. Lisp macro programming is the extension of

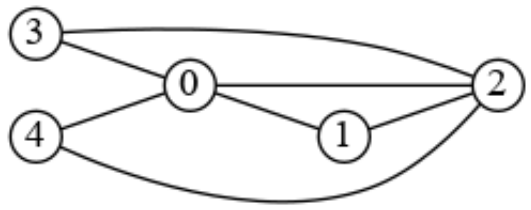


Figure 1: An undirected graph

the Lisp language by developing domain-specific languages (DSL) [15] embedded within Lisp itself.

Here we put macros to work to define such a DSL for expressing graph algorithms which are written in pseudocode. Graph algorithms can then be written in the DSL in their natural form once and for all, not requiring any further rewriting or refactoring.

4. GRAPH ALGORITHMS

Graph theory is a field of mathematics dealing with relations between objects [8]. An undirected graph is a set of nodes (or vertices) together with a set of edges (unordered pairs) on these nodes. Such a graph is usually depicted as in Figure 1; nodes as numbered circles and edges as links joining them.

Many other kinds of graphs can be considered such as directed graphs and graphs with weights or other attributes associated with their nodes and/or edges but for the purposes of this article we will limit ourselves to undirected graphs of a simple nature as in Figure 1.

In recent years graph theory has found application as the underlying model for the study of Complex Networks [23], such as social, computer and biological networks. With the recent increase in size of available networks and of the associated data sets, much research has been focusing on algorithms for computing properties of graphs.

As a case study, we consider one such graph property. The *betweenness centrality* of a node in a graph is a measure of the network load that passes through that node. It is the accumulated total number of messages passing through that node when every pair of nodes sends and receives a single message along each shortest path connecting the pair. This notion was first introduced in graph theory by Anthonisse in 1971 [4] who named it the "*rush*" of the graph. In the world of sociology, betweenness centrality was introduced by Linton Freeman in 1977 [14] and has become an important measure of the relative influence of members of social networks.

A straightforward algorithm based on all-pairs shortest paths can calculate the betweenness centrality in time $O(n^3)$ where n is the number of nodes in the graph. Several incremental improvements were proposed but a breakthrough came in 2001 in the form of Brandes' algorithm [9], which runs in $O(nm)$, where m is the number of edges in the graph. Brandes' algorithm consists of two phases: the computation of the lengths and numbers of shortest paths between all pairs; and then, the summing of all pair dependencies. Brandes' innovation was to recognise that the dependencies could be summed cumulatively by maintaining several attributes at each node of the graph.

5. EXECUTABLE PSEUDOCODE

In order to express Brandes and similar algorithms, we employ Common Lisp macros to build a DSL for graph algorithms. This DSL is PSEUDOGRAPH.

Central to PSEUDOGRAPH are two macros, `vlet` and `elet` which allow attributes on vertices and edges, respectively, to be declared, initialised, assigned and updated. Together with further macros for iteration over nodes and edges, they form the core of PSEUDOGRAPH.

In PSEUDOGRAPH the n vertices of a graph are represented by the consecutive integers $[0, n)$. Its undirected edges are represented by unordered pairs of vertices. PSEUDOGRAPH uses straightforward implementations of queues and stacks.

Brandes' algorithm as originally presented in his paper [9, p. 10] is shown in Figure 2. We have highlighted some sections in colour. These highlighted sections correspond to the similarly coloured sections in the PSEUDOGRAPH implementation of the algorithm in Figure 3. An explication of the usage in the various sections of the implementation follows.

The code sections which are not highlighted are merely Common Lisp code containing calls to underlying stack and queue libraries.

The sections highlighted in red make use of PSEUDOGRAPH macros for iteration over the nodes of a graph (`do-nodes`) and over the elements of stacks (`do-stack`) and queues (`do-queue`). Note how `do-stack` (respectively `do-queue`) pops an element from the given stack (queue) and binds a variable to that element for the body of the macro. This corresponds directly to Brandes' pseudocode for

while S not empty **do**
  pop w ← S;
  ...
end while

The three sections highlighted in blue are the header parts of uses of PSEUDOGRAPH's `vlet` macro (*"vertex let"*). `vlet` takes care of the initialisation of and assignment to the vertex attributes of a graph. Like Common Lisp's `let` special form, `vlet` binds values to the specified variables in the lexical scope of its body, `vlet` differing in that it binds a vector of (copies of) the given value to each variable. The size of these vectors is given by the first argument to `vlet`. For example, the `vlet` in the second blue block makes three vectors of size n, each element of the first vector containing (a copy of) an empty queue, each element of the second the `fixnum` 0 and each element of the third the value of the variable `unfound`. These values are then bound in parallel to the variables `pred`, `sigma` and `dist`, respectively.

As well as binding initial values to its variables, `vlet` defines a number of macros which have bindings local to the body of the `vlet`. These local macros permit operations assigning values to elements of the locally defined vectors such as updating, incrementing, enqueueing, *etc.*. Uses of these local macros are highlighted in yellow in the figures. Examples of the behaviour of these local macros appearing in Brandes' algorithm are as follows.

- The form `(dist v)` accesses the `v`th element of the vector `dist`
- The form `(sigma start = 1)` assigns the value `1` to the `start`th element of the vector `sigma`
- The form `(sigma w += (1+ (dist v)))` increments the `w`th element of the `sigma` vector by one plus the `v`th

Algorithm 1: Betweenness centrality in unweighted graphs

```
C_B[v] ← 0, v ∈ V;
for s ∈ V do
    S ← empty stack;
    P[w] ← empty list, w ∈ V;
    σ[t] ← 0, t ∈ V; | σ[s] ← 1;
    d[t] ← −1, t ∈ V; | d[s] ← 0;
    Q ← empty queue;
    enqueue s → Q;
    while Q not empty do
        dequeue v ← Q;
        push v → S;
        foreach neighbor w of v do
            // w found for the first time?
            if d[w] < 0 then
                enqueue w → Q;
                d[w] ← d[v] + 1;
            end
            // shortest path to w via v?
            if d[w] = d[v] + 1 then
                σ[w] ← σ[w] + σ[v];
                append v → P[w];
            end
        end
    end
    δ[v] ← 0, v ∈ V;
    // S returns vertices in order of non-increasing distance from s
    while S not empty do
        pop w ← S;
        for v ∈ P[w] do δ[v] ← δ[v] + (σ[v]/σ[w]) · (1 + δ[w]);
        if w ≠ s then C_B[w] ← C_B[w] + δ[w];
    end
end
```

Figure 2: Brandes' algorithm in pseudocode (from [9, p. 10], colouring added).

```
(defun brandes (G)
  (let ((n (node-count G))
        (unfound -1))
    (vlet n ((bc 0.0))
      (do-nodes (start n)
        (let ((S (make-stack))
              (Q (make-queue)))
          (vlet n ((pred  (make-queue))
                   (sigma 0)
                   (dist  unfound))
            (sigma start = 1)
            (dist start = 0)
            (enqueue start Q)
            (while ((v (qpop Q)))
              (push v S)
              (do-nodes (w (neighbours G v))
                (when (= (dist w) unfound)
                  (enqueue w Q)
                  (dist w = (1+ (dist v))))
                (when (= (dist w) (1+ (dist v)))
                  (sigma w += (sigma v))
                  (pred w enqueue v))))
            (vlet n ((delta 0.0))
              (do-stack (w S)
                (do-queue (v (pred w))
                  (delta v += (* (/ (sigma v) (sigma w))
                                 (1+ (delta w)))))
                (unless (= w start)
                  (bc w += (delta w))))))))
      (bc)))))
```

Figure 3: Brandes' algorithm in Common Lisp "executable pseudocode"

element of the `dist` vector.

- The form `(pred w enqueue v)` adds the value `v` to the queue in the `w`th element of `pred`. `vlet` allows the use of other operations similar to `enqueue` in its place permitting access and update of other data structures where required.
- Finally, the form `(bc)` returns the entire vector `bc`

5.1 Advantages

The PSEUDOGRAPH DSL provides a number of advantages over traditional implementations.

The local macros in the `vlet` body permit the expression of the operations in the pseudocode of Brandes' algorithm in a manner sufficiently similar to the original pseudocode to allow immediate, *at-a-glance* comparison. This gives confidence that the program is indeed a faithful implementation of the algorithm in pseudocode form.

The resulting code is succinct, a mere 30 lines. In fact the resulting code is almost line-for-line equivalent to the original pseudocode. As a comparison, the implementation of Brandes' algorithm in the Boost Graph Library [28] runs to over 600 lines of `C++`. Checking that this code faithfully implements the pseudocode is not a trivial task and certainly not as immediate as checking the PSEUDOGRAPH code.

Since the PSEUDOGRAPH code is executable, experimentation with graph algorithms can be carried out at the pseudocode level. A computer scientist wishing to investigate variants of a graph algorithm can immediately edit, execute and experiment without having to carry out intricate coding in a large code base. In this way, Lisp's advantage as a vehicle for rapid prototyping [25], with incremental development, program introspection and read-eval-print loop allowing short development cycles can be brought right to the algorithm level.

The PSEUDOGRAPH DSL provides a concise API surface area. The correct coding of a variety of graph algorithms on this DSL gives confidence in the correctness in the underlying Lisp code.

PSEUDOGRAPH can also be used for educational purposes. Krishnamurthi points out how "*Lisp programmers have long used macros to extend their language*" before continuing to lament the "*paucity of effective pedagogic examples of macro use*" [20]. Since PSEUDOGRAPH has a *common vocabulary* with text books on graph algorithms, students can readily implement and experiment with them at the pseudocode level leading to better understanding.

And finally, the process can be reversed and pseudocode in LaTeX format can be generated and pretty printed from the PSEUDOGRAPH implementation of an algorithm allowing the results of such experimentation to be included in documentation and reports with the guarantee that no errors have been introduced during transcription.

One could consider making the syntax available within `vlet` even more like the corresponding pseudocode by permitting expressions such as `(sigma[s] = 1)` for $\sigma[s] = 1$. We choose not to do this for the same reasons that we prefer the syntax of Jonathan Amsterdam's `iterate` package [3] over the syntax of Common Lisp's `loop` facility [29].

Where Lisp's `loop` facility provides a complete "*Pascalish*" sublanguage for carrying out iteration, the `iterate` package provides a more naturally Lisp-like syntax which is more easily embedded, extended and customised. In our view `loop` could be seen as `iterate` taken too far but we acknowledge this may be a matter of taste. We feel that our *executable pseudocode* is similar enough to the original pseudocode for *at-a-glance* comparison, while still enjoying all the advantages of Lisp syntax as enumerated by Amsterdam in [2].

5.2 Performance

With all of the advantages listed above, one might be forgiven for thinking that the resulting PSEUDOGRAPH code would be unable to compete for speed with carefully coded implementations in lower level imperative programming languages. However nothing is further from the truth.

Note, for example, that the initialisation of variables in the `vlet` macro also permits any keyword arguments accepted by `make-array` [29] such as type declarations, for example:

```
(vlet n ((cb   0.0     :element-type 'float)
         (dist unfound :element-type 'fixnum)
  ...
```

These particular declarations allow a Lisp compiler to introduce optimizations for the vertex attributes.

We performed initial comparisons of our PSEUDOGRAPH code for Brandes with implementations in NetworkX, igraph and Boost Graph Library.

NetworkX [22] is a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. It advertises itself as "*high-productivity software*" and it lives up to that billing by allowing the programmer to quickly set up and experiment with graphs of various sorts. However although Python shines in ease of use, it sacrifices performance. Comparison showed our implementation of Brandes in PSEUDOGRAPH to be between 2 and 3 times faster that the corresponding NetworkX implementation.

igraph [18, 11] is a collection of network analysis tools with the emphasis on efficiency, portability and ease of use. It has front ends for Python, C and R. Although the code appears fast, the igraph implementation of Brandes algorithm fails to run on graphs with 64 or more nodes due to data structure limitations. This precluded a meaningful comparison with PSEUDOGRAPH.

To get an indication of the performance of our PSEUDOGRAPH implementation of Brandes' algorithm we turned to the Boost Graph Library [28]. Boost is a collection of `C++` libraries that are open source and peer reviewed. Their libraries are widely regarded as being efficient and of high quality. In many areas Boost libraries are *de facto* standards and we take them to be the state-of-the-art yardstick against which we could get a first indication of the speed of our code.

We generated graphs of various sizes following the Newman-Watts-Strogatz model [24]. This is a model of so-called *small-world* graphs, a type of particular interest since it frequently occurs in social networks and other complex networks. The graphs are generated by connecting the nodes in a ring, then connecting each node to a number of its nearest neighbours, then rewiring each edge randomly with a fixed probability.

The comparison was carried out on a standard Linux 3.19.2 distribution on a 2.9 GHz Intel i7 processor. PSEUDOGRAPH ran on Steel Bank Common Lisp version 1.2.8 with compiler settings `(optimize (speed 3) (safety 0))`. Boost version 1.57 was compiled on `gcc 4.9.2` with the `-O3` compiler set-

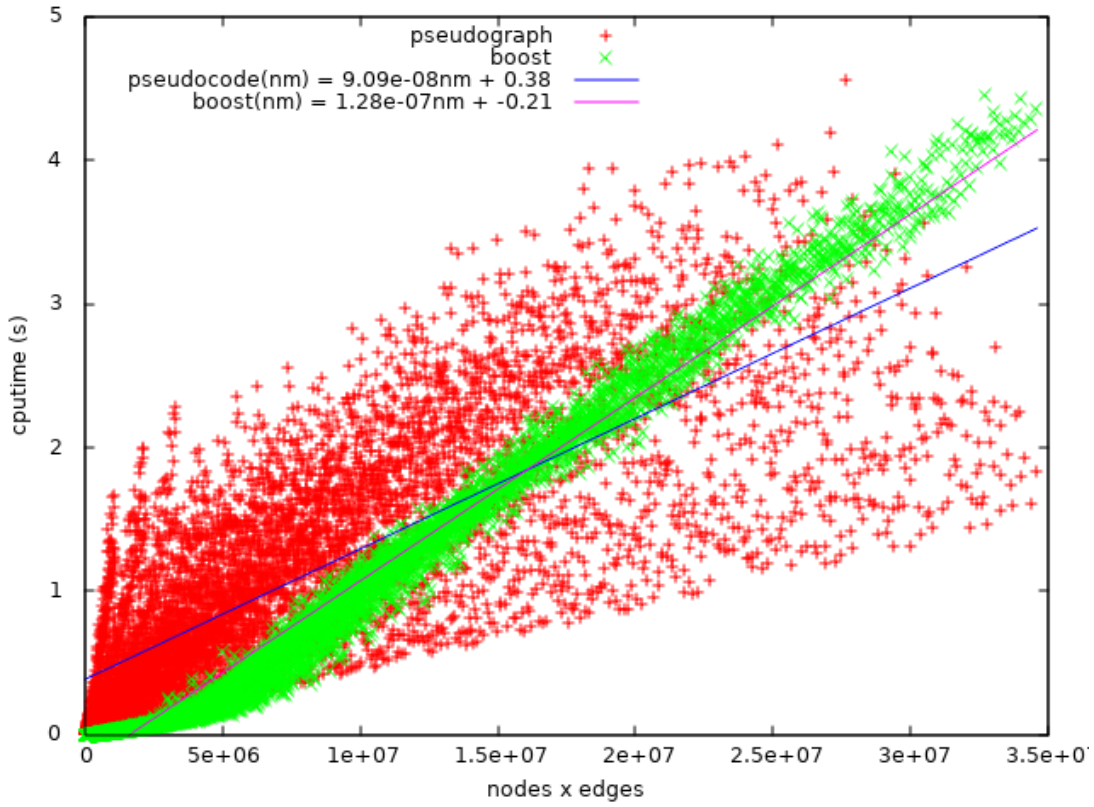


Figure 4: Comparison of pseudograph with Boost on small-world graphs.

ting. In both cases, these are the most aggressive compiler settings. The times shown are net run times after the graphs have been loaded into the native data structures.

The results are shown in the graph in Figure 4. On the x-axis is the product (nm) of the number of nodes and the number of edges of the graph. On the y-axis, the CPU time in seconds. Since Brandes is $O(mn)$, we expect the worst-case times on the graph to be roughly linear. Each data point is a run of either the PSEUDOGRAPH or Boost implementation of the betweenness centrality algorithm on an instance of a small-world graph generated according to the Newman-Watts-Strogatz model for $n \in \{10, 20, \ldots, 300\}$, $k \in \{3, 4, \ldots, n-1\}$, $p \in \{0.1, 0.2, \ldots, 0.9\}$. We make two observations on the basis of these results.

As can be seen, PSEUDOGRAPH is competitive with Boost, in some cases slower, in some faster. Linear regression on the two sets of data points (indicated by the two straight lines in Figure 4) shows the average PSEUDOGRAPH run time to be increasing more slowly than that of Boost with a crossover point at about $mn =$ `1.6e7`. This suggests that for larger graphs, PSEUDOGRAPH will continue to outperform Boost on average.

Further, it is noticeable that there is a greater variance in the PSEUDOGRAPH data points than in those of Boost. This might suggest that the performance of PSEUDOGRAPH is less predictable than that of Boost.[1]

In order to find an explanation, we look more closely at runs for a fixed number of nodes. Holding n fixed at 600 and varying only the parameter k, generates graphs with a fixed number of nodes but a varying number of edges. The run times of neither implementation was sensitive to the value of p so it was held fixed at 0.3. These results, shown in Figure 5, are typical of those for graphs of other sizes.

As we can see, the Boost and PSEUDOGRAPH implementations have differing performance characteristics for graphs which are sparse (having relatively few edges) and dense (having relatively many edges). While the run time of Boost increases uniformly with the density of edges, PSEUDOGRAPH's run time peaks before decreasing for very dense graphs. We have no explanation for this behaviour but speculate that it may be a property of `bitsets` a compact representation PSEUDOGRAPH uses for subsets of integers in a range $[0, n)$ [5]. This bears further investigation. In any case, it appears that this particular performance characteristic leads to the larger variation of PSEUDOGRAPH run times in Figure 4.

Given the many other advantages of PSEUDOGRAPH stated above, we regard the result that it can compete with the Boost `C++` library as very encouraging, especially since little attempt has been made at optimising the PSEUDOGRAPH code as yet. Currently, standard, straightforward representations are used for stacks and queues. The structure of undirected graphs is represented as adjacency lists of sets of nodes, which are represented as `fixnums`. These sets are represented as `bitsets`.

We emphasise however that this initial test, while encouraging, is by no means conclusive. We plan full tests and comparisons on other kinds of graphs and larger graphs.

[1] One might think that garbage collection is respobsible. However no garbage collection was triggered during the runs.

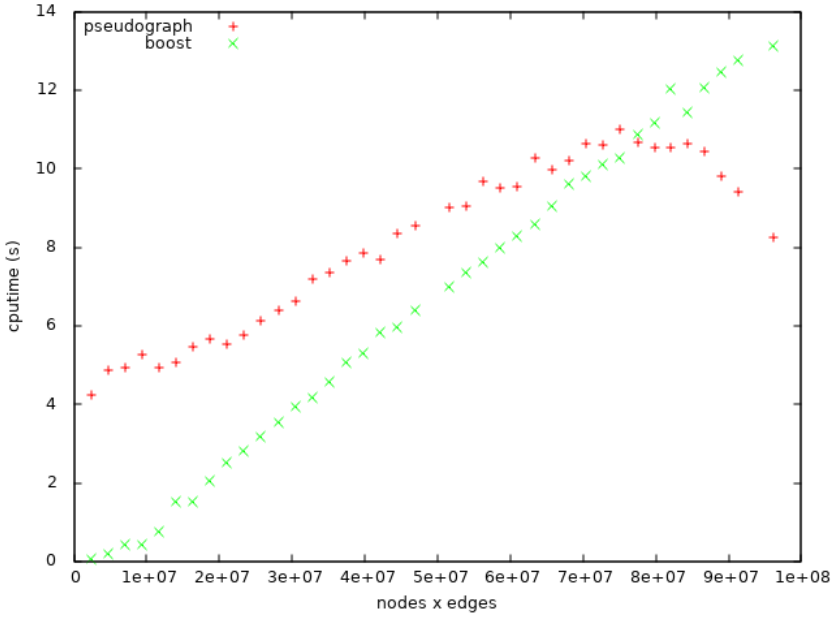


Figure 5: Comparison of pseudograph with Boost on small-world graphs, $n = 600$**,** $p = 0.3$

6. CONCLUSIONS AND FUTURE WORK

We have shown a proof of concept for our approach to programming graph algorithms by presenting an implementation of Brandes' algorithm which is not only comparable *at a glance* to the pseudocode of the original algorithm, but also competitive in speed with state-of-the-art code for the algorithm written in optimised `C++`.

To use the words of Krishnamurthi [20], we feel that this approach represents '*…a rare "sweet-spot" in the readability-performance trade-off.*' Despite the long-standing popular belief to the contrary, Lisp has been shown to be competitive with lower-level programming languages for scientific numerical computing [31, 30]. Our results now show that this competitive performance need not be limited to numerical computing.

The range of graph algorithms which we can express with the machinery of PSEUDOGRAPH as it stands is surprisingly large. As we have seen, Brandes' algorithm requires maintaining several vertex attributes. Other graph algorithms such as Floyd-Warshall requires the maintenance of edge attributes. This can be achieved using PSEUDOGRAPH's `elet` macro (*edge let*). Directed graphs can also be represented. Moreover, such is the flexibility of the approach that novel pseudocode constructs can readily be added to PSEUDOGRAPH.

We are continuing to implement other graph algorithms using this approach and experimenting with optimizations of the code *under the hood*. The bottom-up nature of the PSEUDOGRAPH code with clean interfaces makes it easy to vary such representations independently of each other.

We plan to release our work under an open-source license in the form of the PSEUDOGRAPH library which will also be packaged and submitted to Quicklisp [6].

7. ACKNOWLEDGMENTS

The author is grateful to Leen Torenvliet and Markus Pfundstein for helpful discussions.

8. REFERENCES

[1] Adobe. Lingo language support center. `http://www.adobe.com/support/director/lingo.html`.

[2] J. Amsterdam. Don't loop, iterate. Technical report, MIT Artificial Intelligence Laboratory, May 1990.

[3] J. Amsterdam. The iterate manual. Technical Report A. I. MEMO 1236, Massachusetts Institute of Technology, A.I. Lab., Cambridge, Massachusetts, Oct. 1990.

[4] J. Anthonisse. *The rush in a directed graph*. Stichting Mathematisch Centrum Amsterdam. Afdeling Mathematische Besliskunde, 1971.

[5] J. Arndt. *Matters Computational*. Springer, 2011. `http://www.jjj.de/fxt/#fxtbook`.

[6] Z. Beane. Quicklisp library manager for common lisp. `http://www.quicklisp.org/`.

[7] K. Beck. *Test-driven Development: By Example*. Addison-Wesley, 2003.

[8] B. Bollobás. *Modern Graph Theory*. Graduate texts in mathematics. Springer, 1998.

[9] U. Brandes. A faster algorithm for betweenness centrality. *Mathematical Sociology*, 25(2):163–177, 2001.

[10] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[11] G. Csárdi and T. Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 2006.

[12] DMTCS. Discrete mathematics & theoretical computer science. `http://www.dmtcs.org/dmtcs-ojs/index.php/dmtcs`.

[13] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[14] L. Freeman. A set of measures of centrality based on betweenness. *Sociometry*, 40:35–41, 1977.

[15] D. Ghosh. DSL for the uninitiated. *Commun. ACM*, 54(7):44–50, July 2011.

[16] D. Goodman. *The Complete HyperCard Handbook*. Bantam Books, 1987.

[17] P. Graham. *On Lisp: Advanced Techniques for Common Lisp*. An Alan R. Apt book. Prentice Hall, 1994.

[18] igraph. igraph. `http://igraph.org/`.

[19] D. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading, MA., 1981.

[20] S. Krishnamurthi. Educational pearl: Automata via macros. *J. Funct. Program.*, 16(3):253–267, 2006.

[21] S. McConnell. *Code Complete*. DV-Professional. Microsoft Press, 2009.

[22] NetworkX. Networkx. `http://networkx.github.io/`.

[23] M. Newman. *Networks: An Introduction*. Oxford University Press, March 2010.

[24] M. E. J. Newman and D. J. Watts. Renormalization group analysis of the small-world network model. *Physics Letters A*, 263:341–346, 1999.

[25] K. M. Pitman. Accelerating hindsight: Lisp as a vehicle for rapid prototyping. *ACM SIGPLAN Lisp Pointers*, VII(1–2):14–21, January 1994. `http://www.nhplace.com/kent/PS/Hindsight.html`.

[26] Python. The Python programming language. `https://www.python.org`.

[27] P. Seibel. *Practical Common Lisp*. Apress, Berkely, CA, USA, 1st edition, 2012.

[28] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley Professional, 2001.

[29] G. L. Steele, Jr. *COMMON LISP: the language*. Digital Press, 12 Crosby Drive, Bedford, MA 01730, USA, 2nd edition, 1990.

[30] D. Verna. Beating C in scientific computing applications. In *Third European Lisp Workshop*, Nantes, France, July 2006. `http://lisp-ecoop06.bknr.net/home`.

[31] D. Verna. How to make Lisp go faster than C. *IAENG International Journal of Computer Science*, 32(4), Dec. 2006.

lambdatalk

Alain Marty Engineer Architect
66180, Villeneuve de la Raho, France
marty.alain@free.fr

ABSTRACT

A wiki is a web application which allows collaborative modification, extension, or deletion of its content and structure. In a typical wiki, text is written using a simplified and rather limited markup syntax intended for an average user. My project was to build a wiki equiped with a true programming language which could be used in a collaborative mode by an author, a web designer and a coder. The result is a small Lisp-like language, **λ-talk**, built on a regexp based evaluator and embedded in a tiny wiki, **α-wiki**, working as a light overlay on any modern browser.

KEYWORDS

Lisp, Scheme, Javascript, Regular Expressions, CMS, Wiki.

INTRODUCTION

Web browsers parse data such as HTML code, CSS rules and Javascript code stored on the server side and display rich multimedia dynamic pages on the client side. Scripting languages such as PHP and Javascript allow strong interactions with these data leading to web applications. Hundreds of engines have been built, managing files on the server side and interfaces on the client side, such as MediaWiki, Wordpress, Drupal. The *de facto* standard *Markdown* syntax is widely used to simplify text markup, but is not intended to be used for an easy styling and even less for scripting. Works have been done to build unified syntaxes, for instance: *Scribe* by Erick Gallesio & Manuel Serrano [1], *Scribble* by Matthew Flatt & Eli Barzilay [2], *LAML* by Kurt Nørmark [3].

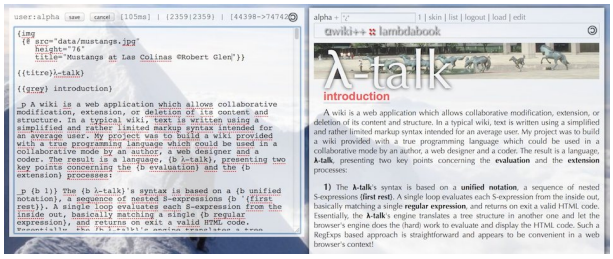
With the plain benefit of existing Scheme implementations, these projects make a strong junction between markup, styling and programming syntaxes. But these tools are definitively devoted to coders and not to web designers and even less to beginners. **α-wiki** and **λ-talk** have been conceived to give them a simple common environment where could be done a collective work.

The name of **λ-talk**, comes from the *Church's lambda calculus* which inspired the father of Lisp, *John McCarthy*, and the Apple's *Hypercard/Hypertalk* which inspired the father of the wiki concept, *Ward Cunningham*. The following document gives an introduction to the *λ-talk environment*, to the evaluation process of *words*, to the creation of *globals*, *functions* and *structures*, to the use of *events*, *scripts* and *libraries* and to some other things.

1. ENVIRONMENT

λ-talk is a small programming language embedded in a wiki, called **α-wiki**, a small interactive development tool working on top of any modern web browser running Javascript, independent of any external library and easy to install, (about 100kb), on a standard web hosting running PHP. **λ-talk** can be used at three levels: 1) with a handful of commands *any author* can easily create minimally structured documents made of words and pictures, 2) *any web designer* can find a full set of HTML tags and CSS rules to enrich theses documents, 3) and *any coder* can fit specific needs and make pages compositing more easy by extending the language's built-in functionalities. Everybody sharing a clear, unique and coherent notation allowing to produce rich and dynamical documents in a true collaborative work.

Here is a snapshot of the **α-wiki** interface with on the left the editor frame and on the right the display frame, both in a real-time interaction:



As it can be seen, there are no "**B, I, U**" fancy buttons for bold, italic or underline styles and other tools alike. Every enrichments are entered as special words, as commands to do something on other words. Even at the lowest level the user writes code and is ready to deal with higher levels. We are now going to briefly introduce the code *structure* and its *evaluation* process, then the *dictionary* and its *extension* process.

1) Structure: In **λ-talk** a *word* is defined as any sequence of characters except *spaces* and *curly braces*. The source code is a flattened tree (actually a forest ...), made of nested forms `{first rest}` - called s-expressions - where `first` is a word belonging to a dictionary and `rest` is any sequence of words recursively containing, or not, any `{first rest}` forms.

2) Evaluation: Following a simple hack shared by *Steven Levithan* [4], the interpreter evaluates the input code in realtime, in this single loop:

```
while (code != (code =
     code.replace( pattern , evaluate ))) ;
```

using a pattern built on this single Regular Expression:

```
/\{([^\s{}]*)(?:[\s]*)([^{}]*)\}/g
```

This pattern finds terminal forms `{first rest}` where `first` is a primitive (or a user defined) function belonging to a unique dictionary, and `rest` is any sequence of words without any `{first rest}` form. For each `{first rest}` terminal form, `first` is applyed to `rest` and the result replaces this form in the code. The loop exits when the initial code doesn't contain anymore `{first rest}` form. The output is the resulting HTML code sent to the web browser for evaluation and display. For instance the following code entered in the frame editor:

```
{b Hello {i brave {u new}} World:
   √(3{sup 2} +4{sup 2}) =
   {sqrt {+ {* 3 3} {* 4 4}}}.}
```

is progressively evaluated in 5 steps, from the leaves to the root:

```
1: {b Hello {i brave {u new}} World:
   √(3{sup 2}+4{sup 2}) =
   {sqrt {+ {* 3 3} {* 4 4}}.}
2: {b Hello {i brave < u>new< /u> World}:
   √(3² +4²) = {sqrt {+ 9 16}}.}
3: {b Hello < i> brave < u>new< /u>< /i>
World: √(3² +4²) = {sqrt 25}.}
4: {b Hello < i> brave < u>new< /u>< /i>
World:  √(3² +4²) = 5.}
5: < b> Hello < i> brave < u>new< /u>< /i>
World: √(3² +4²) = 5.< /b>
```

The browser's engine evaluates the HTML code and displays:

Hello *brave new* World: √(3² +4²) = 5.

---

3) Dictionary: The dictionary contains a basic set of words associated to the Javascript math operators and functions, [+,-,*,/,..., sqrt,...], to a wide set of HTML tags, [div, span, ..., input, script, style, canvas, SVG,...] and to some other ones specific to the wiki. For instance:

```
{sqrt {+ 1 1}} -> 1.4142135623730951
{span {@ style="color:red"}I'm red} -> I'am red
{b R{sub μν} - ½.R.g{sub μν} = T{sub μν}}
```
-> **$R_{\mu\nu} - ½.R.g_{\mu\nu} = T_{\mu\nu}$**

---

4) Extension: the dictionary can be extended beyond the set of primitive functions. User defined functions and structured data can be dynamically built using a minimal set of 3 special forms **[`if, lambda, def`]** processed before the evaluation loop. For instance we can define two new words:

```
{def shadow span
 {@ style="color:black; padding:0 5px;
           box-shadow:0 0 8px black;"}}
```
-> shadow

```
{def cute_add {lambda {:a :b}
  Yes mom, :a+:b is equal to {+ :a :b}!}}
```
-> cute_add

The parser adds the associated user defined functions to the dictionary and replaces in the input code the previous `{def ...}` expressions by the two words, `shadow` and `cute_add`. It's now possible to call these functions like this:

```
{{shadow}I am on a shadowed box}
```
-> I am on a shadowed box

```
{cute_add 1 1}
```
-> Yes mom, 1+1 is equal to 2!

We will now analyze more precisely words, globals, functions, structures, events, scripts, libraries, ...

2. WORDS

Words are the fundamental objects of **λ-talk**. Sequences of words are 99% of the content in a wiki context, they are quickly overflown by the parser, ignored and displayed as such.

---

1) Words don't need to be quoted like `strings` used in other languages:

```
Hello brave new World
```
-> Hello brave new World

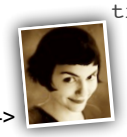
```
{del {i {u Hello {sup World}}}}
```
-> ~~*Hello* World~~

λ-talk has a wide set of primitives for handling in a standard and familiar way HTML tags and *all* CSS rules, for instance:

```
{span
 {@ style="color:white; background:black;"}
   White on Black}
```
-> White on Black

```
{a {@ href="http://www.pixar.com/"}PIXAR}
```
-> PIXAR

```
{img
 {@ id="amelie"
    src="data/amelie_sepia.jpg" height="50"
    title="Amélie Poulain"
    style="box-shadow:0 0 8px black;
           border:1px solid white;
           transform:rotate(-5deg);"
}}
```
->



λ-talk has a small set of primitives acting on sequences of words and chars:

```
{first Hello Brave World}
```
-> Hello

```
{rest Hello Brave World}
```
-> Brave World

```
{length Hello Brave World}
```
-> 3

```
{nth 1 Hello Brave World}
```
-> Brave

```
{chars Hello Brave World}
```
-> 17

```
{charAt 6 Hello Brave World}
```
-> B

---

2) Numbers are words evaluable as integer or real numbers by the primitives built on browser's Javascript operators and functions.

`-12345` -> -12345
`{+ 1.e3 1}` -> 1001
`{* 2 1e-3}` -> 0.002
`{- 100}` -> -100
`{/ 100}` -> 0.01
`{PI}` -> 3.141592653589793

Note that the last exemple shows that PI can be considered as a pointer to a value. Errors are quietly handled:

`{/ 1 0}` -> Infinity
`{+ 1 hello}` -> NaN // It's Not a Number
`{foo bar}` -> (foo bar) // simply unevaluated

---

3) Two words, `true` and `false`, are evaluated as booleans by the primitives built on the browser's Javascript boolean operators:

`{not true}` -> false
`{and true false}` -> false
`{or true false}` -> true
`{< 11 12}` -> true
`{= 1 1.00}` -> true // test equality between real numbers

Booleans are used to build *control structures* via the `if` special form: `{if bool_term then t_term else f_term}`:

`{if {< 1 2} then {+ 1 2} else {/ 1 0}}` -> 3

`if` is not a primitive function. In the pre-processing phase the `{if ...}` special form returns `{_if_ bool_term then 't_term else 'f_term}`, where `_if_` is a primitive function and the `t_term` and `f_term` are quoted, ie. hidden from evaluation. In the evaluation loop the `_if_` function evaluates the `bool_term`, then the corresponding true term and returns the result.

3. GLOBALS

Globals are user defined words added to the dictionary via the `def` special form: `{def name body}` where `name` is a word and `body` is a sequence of words and s-expressions.

---

1) Special words or numbers can be given a name:

`{def myPI 3.1416}` -> myPI
`myPI` -> myPI
`{myPI}` -> 3.1416

Note that `myPI` is a word and thus unevaluated; `{myPI}` is a function call returning the associated value 3.1416. This reminds spreadsheets in which "`PI`" is a word displayed as it is and "`=PI()`" is a function call returning 3.141592653589793.

Evaluable expressions can be given a name:

`{def PHI {/ {+ 1 {sqrt 5}} 2}}` -> PHI
`{PHI}` -> 1.618033988749895

Sequences of words can be given a name:

`{def sentence 12 34 56 Hello World}` -> sentence
`{sentence}` -> 12 34 56 Hello World

Thanks to primitives `first, rest, nth`, *named* sequences of words can be used as aggregate data structures (arrays, vectors, polynoms, complex numbers, ...). More in "**5. STRUCTURES**".

---

2) Using HTML attributes and CSS rules standard syntax inside a unique `{@ ...}` form was a design choice in order to avoid any pollution in the dictionary and to be easy to use by a web designer. This is how a global sequences of CSS rules can be defined and used:

`{def georgia span {@ style="font:italic 1.5em georgia; color:green;"}}` -> georgia
`{{georgia}I'm Georgia!}` -> *I'm Georgia!*

---

3) `def`s can be nested, but inner definitions are not locales (more in section "**4. FUNCTIONS**"). A good *old* practice is to prefix inner defined names with outer defined names, for instance following this pattern: `outer_name.inner_name`:

```
{def six
 {def six.two {+ 1 1}}
 {def six.three {+ {six.two} 1}}
 {* {six.two} {six.three}}
} -> six
{six} -> 6
```

λ-talk has no `set!` special form, globals are immutable.

4. FUNCTIONS

Beyond the set of primitive functions belonging to the dictionary, new functions can be defined by the user.

1) Functions are created with the `lambda` special form: `{lambda {:args} body}`, where `:args` is a sequence of words and `body` is any sequence of words and s-expressions. For instance:

```
{lambda {:a :b}
  :a+:b equals {+ :a :b}} -> lambda_1
```

The number postfixing the given name `lambda_` is generated by the parser for each new function and is unknown by the user. Such an *anonymous* function should be immediately used like this:

```
{{lambda {:a :b}
  :a+:b equals {+ :a :b}} 3 4} -> 3+4 equals 7
```

In this example the `lambda` replaces in the string "`:a+:b equals {+ :a :b}`" every occurrences of `:a` and `:b` by the two values `3` and `4` according to the pattern built with the arguments listed in `{:a :b}`, here the regular expressions `/:a/g` and `/:b/g`. And the resulting expression, `{+ 3 4}`, will be returned in the evaluation loop. Here is an overview of the replacement process inside the Javascript engine:

```
":a+:b equals {+ :a :b}"
      .replace( /:a/g, 3 ) -> 3+:b equals {+ 3 :b}
      .replace( /:b/g, 4 ) -> 3+4 equals {+ 3 4}
```

Note that, in order to avoid conflicts, arguments should be

prefixed by a distinctive char, e.g. a colon. In this example it obviously prevents the word `equals` to be replaced by `equ3ls`.

---

2) An anonymous function can be given a name and so added permanently to the dictionary:

```
{def add {lambda {:a :b}
         :a+:b equals {+ :a :b}}} -> add
```

It's now possible to call this function more than once:

```
{add 3 4} -> 3+4 equals 7
{add 1 1} -> 1+1 equals 2
```

---

3) It is not an error to call a function with a number of values greater than the function's arity. Extra values are just ignored. It is not an error to call a function called with a number of values less then its arity. Given values are stored and a new function is returned waiting for the missings:

```
{add 3} -> lambda_45    // stores 3 and returns a function
{{add 3} 4} -> 3+4 equals 7 // waiting for a second value
```

Here is an overview of the replacement process inside the Javascript engine. The first call stores the first value in the body:

```
":a+:b equals {+ :a :b}"
      .replace( /:a/g, 3 ) -> 3+:b equals {+ 3 :b}
```

Then a new function is created, waiting for the second value:

```
{lambda {:b} 3+:b equals '{+ 3 :b}} ->lambda_4
```

Thus, functions can return functions and can be passed as arguments to other functions. This powerful built-in capability will be used to create specialized functions and structured data, see section "**5. STRUCTURES**". For instance, it is easy to write the derivatives of any function, e.g. the function `log`:

```
{def D {lambda {:f :x}
  {/ {- {:f {+ :x 0.01}}
        {:f {- :x 0.01}} } 0.02}}}  -> D
{{D log} 1} -> 1.0000333353334772  // ≠ 1
{{D {D log}} 1} -> -1.0002000533493538  // ≠ -1
{{D {D {D log}}} 1} -> 2.0012007805464416  // ≠ 2
```

---

4) We have seen that defs can be nested but internal `defs` are global constants. Local variables will be created via lambdas. For instance, the area of any triangle (a,b,c) is given by:

triangle_area = √(s*(s-a)*(s-b)*(s-c)) where s=(a+b+c)/2

This function can be written like this:

```
{def triangle {lambda {:a :b :c}
 {{lambda {:a :b :c :s}
 {sqrt {* :s {- :s :a} {- :s :b} {- :s :c}}}
 } :a :b :c {/ {+ :a :b :c} 2}}}} -> triangle
{triangle 3 4 5} -> 6
```

The inner lambda's `:s` acts as a local variable avoiding computing 4 times the value `{/ {+ :a :b :c} 2}`. Note that the inner function has no access to the arguments of the outer function [`:a :b :c`] and must duplicate them. **λ-talk** doesn't know closures!

---

5) Functions can be recursive. Writing them tail recursive opens the way to effective iterative processes:

```
{def ifac {lambda {:acc :n}
  {if {<  :n 1} then :acc
   else {ifac {* :acc :n} {- :n 1}}}}} -> ifac
{ifac 1 21} -> 51090942171709440000
{ifac 1 22} -> 1.1240007277776077e+21
```

Recursive functions can be written without any `defs`:

```
{{lambda {:f :n :r} {:f :f :n :r}}
 {lambda {:f :r :n}
  {if {<  :n 1} then :r
   else {:f :f {* :n :r} {- :n 1}}
 }} 1 10} -> 3628800
```

---

6) In order to enlight the easy intermixing of words, numbers and lambdas, this is the quadratic equation `f(a,b,c) = a.x`^{2}`+b.x+c = 0`, defined with 3 inner nested lambdas. In the upper inner lambda, three arguments [`:d,:e,:f`] get the discriminant `b`^{2}`-4ac`, `-b` and `2a`. In the deeper inner lambdas, two arguments [`:g,:h`] get intermediate evaluations:

```
{def quadratic_equation
 {lambda {:a :b :c}
  {{lambda {:d :e :f} discriminant is :d, so
   {_if_ {> :d 0}
    then {{lambda {:g :h} 2 real roots:
    {br} x1 = {+ :g :h} {br} x2 = {- :g :h}
          } {/ :e :f} {/ {sqrt :d} :f}}
    else {_if_ {= :d 0}
    then {{lambda {:g :h} 1 root: x = :g
          } {/ :e :f} :d}
    else {{lambda {:g :h} 2 complex roots:
    {br} x1 = [:g,:h] {br} x2 = [:g,-:h]
          } {/ :e :f} {/ {sqrt {- :d}} :f}}
  }}} {- {* :b :b} {* 4 :a :c}}
     {- :b} {* 2 :a}} }} -> quadratic_equation
```

Then we call it and get these well formated result:

```
{quadratic_equation 1 -1 -1}
-> discriminant is 5, so 2 real roots:
 x1 = 1.618033988749895
 x2 = -0.6180339887498949
{quadratic_equation 1 -2 1}
-> discriminant is 0, so 1 root: x = 1
{quadratic_equation 1 1 1}
-> discriminant is -3, so 2 complex roots:
 x1 = [-0.5,0.8660254037844386]
 x2 = [-0.5,-0.8660254037844386]
```

---

7) λ-talk has 3 useful primitive: `serie, map` and `reduce`:

```
{def times {lambda {:a :b} {* :a :b}}}
```

-> times // artity is 2
`{map {times 3} {serie 1 9}}`
-> 3 6 9 12 15 18 21 24 27
`{reduce times {serie 1 50}}`
-> 3.0414093201713376e+64 // times is made variadic

The `{do something from start to end step}` control structure doesn't exist in **λ-talk**. Let's define one:

```
{def do {lambda {:f from :a to :b step :d}
  {map :f {serie :a :b :d}}}} -> do
{do {lambda {:x} {* :x :x}}
                 from 1 to 15 step 1}
```
-> 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225

Note that arguments [`from, to, step`] are nothing but *silent slots* without any occurrences in the function's body.

And a last exemple using `map` to compute the Euler's number:

```
{def euler
 {def euler.fac {lambda {:n}
   {if {<  :n 1} then 1
    else {* :n {euler.fac {- :n 1}}}}}}
 {lambda {:n} {+ {map {lambda {:n}
{/ 1 {euler.fac :n}}} {serie 0 :n}}}}}
```
-> euler
`{euler 17}` -> 2.7182818284590455 // ≠ E

---

8) λ-talk can be used to display mathematical expressions. This is the Dirac's equation displayed as an image:

$$i\hbar\frac{\partial\psi}{\partial t}(\mathbf{x},t) = \left(mc^2\alpha_0 - i\hbar c\sum_{j=1}^{3}\alpha_j\frac{\partial}{\partial x_j}\right)\psi(\mathbf{x},t)$$

We can forget (*so does Chrome*) the mathML tags, then build with **λ-talk** three user functions, [`quotient, sigma, paren`]:

```
{def quotient {lambda {:s :num :denom}
{table
 {@ style="width::spx; display:inline-
block; vertical-align:middle;
text-align:center;"}
 {tr {td {@ style="border:0 solid;
  border-bottom:1px solid;"}:num}}
 {tr {td {@ style="border:0 solid;"}:denom}}
}}} -> quotient
{def sigma {lambda {:s :one :two}
{table
 {@ style="width::spx; display:inline-
block;
  vertical-align:middle;
text-align:center;"}
 {tr {td {@ style="border:0 solid;"}:two}}
 {tr {td {@ style="border:0 solid;
  font-size:2em; line-height:0.7em;"}Σ}}
 {tr {td {@ style="border:0 solid;"}:one}}
}}} -> sigma
{def paren {lambda {:s :p}
 {span
  {@ style="font:normal :sem arial;
            vertical-align:-0.15em;"}:p}}}
```
-> paren

and write:

```
i{del h}{quotient 40 ∂ψ ∂t}(x,t) =
{paren 3 (}mc{sup 2}α{sub 0} - i{del h}c
{sigma 30 j=1 3} α{sub j}
{quotient 40 ∂ ∂x{sub j}}{paren 3 )} ψ(x,t)
```

to display as rich text, in any browser, the Dirac's equation:

$$i\hbar\frac{\partial\psi}{\partial t}(x,t) = \left(mc^2\alpha_0 - i\hbar c\sum_{j=1}^{3}\alpha_j\frac{\partial}{\partial x_j}\right)\psi(x,t)$$

5. STRUCTURES

User structured data can be created in several manners. The well known `cons`, `car` and `cdr` functions can be used to build useful structures like `pairs` and `lists`. Thanks to the fact that lambdas are first class citizens, `cons`, `car` and `cdr` were implemented in a previous version of **λ-talk** as user defined functions, following "Structure and Interpretation of Computer Programs", section "2.1.3 What Is Meant by Data?" [5]:

```
{def cons {lambda {:a :b :c}
          {if :c then :a else :b}}}
{def car {lambda {:c} {:c true} }}
{def cdr {lambda {:c} {:c false} }}
```

For obvious reasons of efficiency, these functions have been integrated as primitives in the dictionary.

---

1) A `pair` is made of 2 elements which are either a `word` or a `pair`. With two additional utility functions, `cons?` and `cons_disp` we can build, test and display `pairs`:

`{car {cons aa bb}}` -> aa
`{cdr {cons aa bb}}` -> bb
`{cons? {cons aa bb}}` -> true
`{cons? hello}` -> false
`{cons_disp {cons aa bb}}` -> (aa bb)
```
{cons_disp {cons {cons aa bb}
                 {cons cc dd}}}
```
-> ((aa bb) (cc dd))
```
{cons_disp
 {cons {cons {cons a a} {cons b b}}
       {cons {cons c c} {cons d d}}}}
```
-> (((a a) (b b)) ((c c) (d d)))
```
{cons_disp {cons {cons {cons a a} b}
           {cons c {cons d d}}}}
```
-> (((a a) b) (c (d d)))

Just a first and small step towards tree structures ...

---

2) A `list` is a `pair` whose `car` is any `word` and whose `cdr` is a `pair` or a terminal word arbitrarily chosen, say `nil`:

```
{cons 12 {cons 34 {cons 56 {cons 78 nil}}}}
```

On this definition a first set of user functions can be built to create and display `lists` and play with them: [`list_new, list_disp, first, butfirst, last, butlast, length, member?, duplicate, reverse , apply, insert, sort, ...`]. For instance `insert & sort`:

```
{def insert {lambda {:x :comp :l}
  {if {equal? :l nil}
   then {cons :x :l}
   else {if {:comp :x {car :l}}
   then {cons :x :l}
   else {cons {car :l}
        {insert :x :comp {cdr :l}}}}}}}
```
-> insert
```
{def sort {lambda {:comp :l}
  {if {equal? :l nil}
   then nil
   else {insert {car :l}
        :comp {sort :comp {cdr :l}}}}}}
```
-> sort
LIST: `{list_disp {LIST}}` ->
(99 61 22 64 71 75 7 93 32 63 61 67 93 38 88 26 2 90 nil)
`{list_disp {sort <  {LIST}}}` ->
(2 7 22 26 32 38 61 61 63 64 67 71 75 88 90 93 93 99 nil)
`{list_disp {sort > {LIST}}}` ->
(99 93 93 90 88 75 71 67 64 63 61 61 38 32 26 22 7 2 nil)

---

3) Arrays can be built as first class structures using lambdas. Example with 2DVectors:

```
{def V.new
 {lambda {:x :y :f} {if :f then :x else
:y}}} -> V.new
{def V.x {lambda {:v} {:v true}}} -> V.x
{def V.y {lambda {:v} {:v false}}} -> V.y
{def V.dsp {lambda {:v} [{V.x :v},{V.y
:v}]}} -> V.dsp
{def V.add
 {lambda {:v1 :v2}
  {V.new {+ {V.x :v1} {V.x :v2}}
         {+ {V.y :v1} {V.y :v2}}}}} -> V.add
```

Note that the add function returns a new vector, it's an internal operation allowing vector concatenations. Examples:

```
{V.dsp {V.new 123 456}} -> [123,456]
{V.x {V.new 123 456}} -> 123
{V.y {V.new 123 456}} -> 456
{V.dsp {V.add {V.new 123 456}
              {V.new 123 456}}} -> [246,912]
{V.dsp {reduce V.add
  {map {lambda {:z} {V.new 123 456}}
    {serie 1 100}}}} // reduce make V.add variadic
-> [12300,45600]
```

4) With the `first, rest, nth, length` primitives seen in section "**2. WORDS**", global defined sequence of words behave like arrays. It is a valuable alternative to define operations on complex numbers, rational numbers, polynomials, 2D/3DVectors. This is a simple example with 2DVectors:

```
{def Vx {lambda {:v} {nth 0 {:v}}}} -> Vx
{def Vy {lambda {:v} {nth 1 {:v}}}} -> Vy
{def Vdot {lambda {:v1 :v2}
  {+ {* {Vx :v1} {Vx :v2}}
     {* {Vy :v1} {Vy :v2}}}}} -> Vdot
{def Vnorm {lambda {:v}
  {sqrt {Vdot :v :v}}}} -> Vnorm
{def Vslope {lambda {:v}
  {{lambda {:a} {/ {* :a 180} {PI}}}
   {acos {/ {Vx :v} {Vnorm :v}}}}° }} -> Vslope
```

which can be simply used like this:

```
{def U 1.00 1.00} = [{U}] -> U = [1.00 1.00]
{Vnorm U} -> 1.4142135623730951
{Vslope U} -> 45.00000000000001°
```

---

5) This is a last example showing how to draw points along a Bézier cubic curve [6], and its control points out of any `canvas`:

With 2 user defined functions calling HTML tags and CSS rules:

```
{def bez_cubic
 {def bc.interp
  {lambda {:a0 :a1 :a2 :a3 :t :u}
   {round {+ {* :a0 :u :u :u}
             {* :a1 :u :u :t 3}
             {* :a2 :u :t :t 3}
             {* :a3 :t :t :t}}}}}
 {lambda {:p0 :p1 :p2 :p3 :t}
 {bc.interp {Vx :p0} {Vx :p1}
            {Vx :p2} {Vx :p3} :t {- 1 :t}}
 {bc.interp {Vy :p0} {Vy :p1}
            {Vy :p2} {Vy :p3} :t {- 1 :t}} }}
-> bez_cubic
{def dot
 {lambda {:x :y :r :bord :back}
  {span {@ style="
   position:absolute;
   left:{- :x {/ :r 2}}px;
   top:{- :y {/ :r 2}}px;
   width::rpx; height::rpx;
   border-radius::rpx;
   border:1px solid :bord;
   background::back;"}}}}
-> dot
```

Used in the following code displaying 2 Bézier cubic curves:

```
{dot {{def P0 70 90}} 20 black yellow}
{dot {{def P1 250 20}} 20 black yellow}
```

```
{dot {{def P2 70 320}} 20 black yellow}
{dot {{def P3 250 350}} 20 black yellow}
{map {lambda {:t} {dot
 {bez_cubic P0 P1 P3 P2 :t} 5 black cyan}}
  {serie -0.3 1.3 0.02}}
{map {lambda {:t} {dot
 {bez_cubic P0 P1 P2 P3 :t} 5 black red}}
  {serie -0.3 1.1 0.02}}
```

6. EVENTS, SCRIPTS & LIBRARIES

λ-talk provides functions to allow interaction with the user, with the underlying language, Javascript, and to build libraries of user defined functions and aggregate data.

---

1) This is a first very basic script interacting with the user via the `{input ...}` primitive associated to a `keyUp` event:

```
{input
 {@ id="smart_hello" type = "text"
    placeholder = "Please, enter your name"
     onkeyup =
  "getId('yourName').innerHTML = 'Hello '
     + getId('smart_hello').value + ' !'"}}
{h1 {@ id="yourName"}}
```

Entering your name in this text field

John & Ward

will display:

Hello John & Ward !

---

2) This is another one using the `{script ...}` primitive containing some Javascript functions:

```
{div {@ id="output" style="..."}time: }
{input {@ type="submit" value="start"
onclick="start()"}}
{input {@ type="submit" value="stop"
onclick="stop()"}}
{script
 function start() {
  document.chrono = window.setInterval(
   function() {
    getId('output').innerHTML = 'time: '
       + LAMBDATALK.eval_sexprs('{date}');
    }, 1000 );}
 function stop() {
  window.clearInterval( document.chrono ) }
}
```

which displays a digital stopwatch:

time: 2015 04 03 11 46 50

start stop

---

3) More complex scripts can be externalized (in a plugin folder) and called in the same way from any wiki page. The plugin folder contains a few basic tools for *painting, 2D/3D/fractal editing, spreadsheet*, and even a tiny but true Lisp console, **α-lisp**, following the *Peter Norvig*'s Lisp interpreter standard structure [7] and completely integrated in **λ-talk** via the `{lisp ...}` form:

`{lisp (* 1 2 3 4 5 6 7 8 9 10)}` -> 3628800

---

3) λ-talk works in a wiki-context (**α-wiki**) built as a stack of pages sharing the same style. A kind of **HyperCard/HyperTalk** on the web. Each page is isolated from the others, but data belonging to a page can be included in any other page via the `require` primitive. For instance, we assume that the page `amelie_lib` contains some informations (written as functions) about the wiki's mascot, **Amélie Poulain** :

```
{def name Amélie Poulain}
{def exact_born 1973 9 3 18 28 32}, age,...
```

These definitions can be called and used in this current page:

```
{require amelie_lib}
{picture}{name} is born the {nth 1
{exact_born}}{sup th} month
of {nth 0 {exact_born}},
and so, today, is {age exact_born date}.
```

Amélie Poulain is born the 9th month of 1973, and so, today, is 41 years old and 7 months.

8. MISCELLANEOUS

---

1) discarding multiline comments and displaying code:

```
°°° multiline comments are discarded °°°
°° {+ 1 2} {+ 3 4} °° -> {+ 1 2} {+ 3 4}
```

---

2) quoting forms:

```
{q {+ 1 2} {+ 3 4}} -> {+ 1 2} {+ 3 4}
'{+ 1 2} '{+ 3 4} -> {+ 1 2} {+ 3 4}
```

Note that `{q ...}` is a fourth special form beside the fundamental set `[if, lambda, def]` used for quoting sequences of s-expressions, coming with the quasi-equivalent `'{...}` simplified notation. Useful to display unevaluated s-expressions in a wiki page, they are not mandatory as language special forms beside `[if,lambda,def]` and could be forgotten.

---

3) Remembering that **λ-talk** must be easy to use by beginners, for some basic HTML tags defining blocks (`h1..h6, p, ul, ol`) which terminate with a carriage return (¬), the general `{first rest}` form can be replaced by an alternate one easier to write and read:

`{h1 Title level 1}` can be replaced by:
`_h1 Title level 1 ¬`
and alike for `h2, h3, h4, h5, h6`

`{p some paragraph...}` can be replaced by:
`_p some paragraph... ¬`

```
{ul
 {li unordered list item}
 {li unordered list item}
}
```
can be replaced by:
```
_ul unordered list item ¬
_ul unordered list item ¬
```
and alike for ordered lists `ol`

For links the general `{first rest}` form can be replaced by alternate forms easier to write and read, following the wiki standards:

`{a {@ href="website_URL"}website_name}`
can be replaced by `[[website_name|website_URL]]`
`{a {@ href="?view=page_name"}page_name}`
can be replaced by `[[page_name]]`

---

4) About evaluation speed: in **α-wiki**, each page is edited and evaluated in real-time, the entire evaluation process is started again at each keyboard entry. Tested on a MacBook Pro (2GHz InterCore i7) in the FireFox browser, on a small set of pages, the evaluation's time related to the number of characters before and after the evaluation and to the number of braces nested s-expressions `{}`, is detailed in the table below:

	page	chars before -> after	{}	time ms
1	start	16 532 -> 26 453	343	6
3	tutorial	15 931 -> 17 518	221	5
4	reference	18 557 -> 23 352	381	8
2	jules verne	1 230 422 -> 1 230 300	90	48
5	fibo_1	149 -> 52	12	10 000
6	fibo_2	200 -> 51	14	25

- `start`, `tutorial` and `reference` are representative of wiki pages of an intermediate size equivalent to a 8/10 PDF pages,
- `jules verne` contains in a "pre-wrap" div a long text, a 300 pages book. 62 tags (h1, h2, h3) have been added to mark the chapter's titles and activate a *TOC*,
- `fibo_1` and `fibo_2` contain fibonacci functions. The first one is built on a naive recursive algorithm and stops the browser's engine after the 28th fibonacci number (10.000ms). The second is built on a tail recursive algorithm and stays under 25 milliseconds when called for the 1000th fibonacci number.

This page has been written in **α-wiki**. The average CPU time to evaluate this page containing 43776->82616 characters and 2130 nested s-expressions {}, is around 100 milli-seconds on a Macbook (2Ghz Intel Core i7) and around 1500ms on an iPad 1st generation or on an iPhone 4s. Beyond such a rather approximative benchmark, it appears that on a recent computer the couple [**α-wiki** + **λ-talk**] allows a rather comfortable realtime edition of standard pages found in a wiki.

CONCLUSION

Commenting this work on the *reddit website [8]*, somebody wrote this: « Reminds me of John McCarthy's lament at the W3C's choice of SGML as the basis for HTML: "*An environment where the markup, styling and scripting is all s-expression based would be nice.*" » This is the goal of the **α-wiki & λ-talk** project.

α-wiki is free and under the GPL Licence, [9]. The present document has been created with **α-wiki** working on top of Firefox, using a specific style sheet in respect with the ACM SIGS guidelines, and exported as a 8 US pages PDF file directly from the browser.

REFERENCES

- [1] : Erick Gallesio and Manuel Serrano, http://www-sop.inria.fr/members/Manuel.Serrano/publi/jfp05/article.html#The-Skribe-evaluator
- [2] : M. Flatt and E. Barzilay, http://docs.racket-lang.org/scribble/
- [3] : Kurt Nørmark, http://people.cs.aau.dk/~normark/laml/
- [4] : Steven Levithan, http://blog.stevenlevithan.com/archives/reverse-recursive-pattern
- [5] : H. Abelson & GJ. Sussman, http://mitpress.mit.edu/sicp/
- [6] : http://marty.alain.free.fr/recherche/pformes/straightaway.pdf
- [7] : Peter Norvig, lis.py, http://norvig.com/lispy.html
- [8] : http://www.reddit.com/r/lisp/comments/1xfsd3/alphawiki/
- [9] : alphawiki++, http://epsilonwiki.free.fr/alphawiki_2/

Symbolic Pattern Matching in Clojure

S.C.Lynch
Teesside University
Middlesbrough
UK, TS1 3BA
(+44) 1642 218121
s.c.lynch@tees.ac.uk

ABSTRACT
This paper presents a symbolic pattern matcher developed for Clojure. The matcher provides new types of function definition, new conditional forms and new iterative structures. We argue that pattern matching and unification differ in significant ways that give them different semantics, both useful, and show that matcher capability is enhanced by allowing patterns to be dynamically created or embedded in data structures like rules and state-changing operators. We evaluate the matcher by experimentation, demonstrating that it can be used to simplify the specification of inference mechanisms as well as other types of code.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *control structures, patterns.*

General Terms
Design, Experimentation, Languages.

Keywords
Clojure, Pattern Matching, Rules, Inference.

1. BACKGROUND
We can consider four different types of matching:

- regular expression matching
- structural matching
- matching against types
- symbolic pattern matching

Regular expression matching operates at the level of strings and characters within strings and is a facility provided by most modern programming languages. Structural matching, offered by many languages (including Prolog, Haskell and Clojure), allow variables to be bound to data based on structural correspondence. In Clojure for example the variables X, Y and Z would be bound to 1, 2 and 3 respectively by the *let* expression:

```
(let [ [[X Y] Z] [[1 2] 3] ] …)
```

Structural matching is a feature of many of the more recent functional languages. Matching against types is also offered by some languages (Scala for example) where matching specifies type information and only succeeds if types correspond.

Symbolic pattern matching operates at the level of symbols and the (nested) structures within which they are contained. A distinction we make here is that *symbolic* matching permits literal values to be specified in patterns as well as variables. This allows patterns to specialise on data according to both its shape and its contents. The following patterns, for example, each bind variables X and Y but match against tuples describing different relations:

```
(on X Y)         ;; X is on top of Y
(next-to X Y)    ;; X is next to Y
(holds X Y)      ;; X is holding Y
```

Despite its long history, few programming languages provide symbolic pattern matching. In the early years of Artificial Intelligence, many systems, built in various dialects of Lisp, often used some form of symbolic matching. Typically this was built on an ad hoc "as needed" basis and, eventhough matchers would often deliver similar features and symbolic matching provided the core functionality for some systems, no standard emerged.

Examples of early systems based on symbolic pattern matching include SIR and STUDENT, perhaps culminating with Eliza and SHRDLU [1]; SIR used a small set of simple patterns to extract numeric and equality relations from simple English statements, STUDENT used patterns specifying conditional matching to process algebraic problems described textually (e.g. "If Joe has 4 times as many oranges as Mary…. How many oranges does Joe have"). Eliza (1966) engaged in dialogue, behaving as a non-directive psychotherapist but, while it produced some interesting conversation, did no semantic analysis. SHRDLU (1971) used patterns to parse English statements identifying commands to move blocks around in a simple world. Its capabilities exceeded those of many systems at that time but it targeted a very small micro-world of discourse and problem solving. Post Eliza and SHRDLU, the A.I. community considered that achieving more sophisticated results would not be accomplished by systems based exclusively on pattern matching and subsequently discussion of symbolic pattern matching, either as a basis for A.I. or as a topic in its own right, largely disappeared from academic literature.

Many modern languages provide regular expression matching and there is an increasing trend to provide capabilities associated with destructuring and variable assignment. Scheme and Racket provide macro-extensible matching but, outside the Lisp world, symbolic pattern matching is less common. POP-11 (a little dated now) is a notable exception, providing rich matching capabilities including match-iterators as well as destructuring [5] and more recently Scala has provided matching capability that facilitates some symbol matching [7].

Despite the lack of standardised *Symbolic* pattern matchers in modern programming languages they are often implicitly present in some software systems (e.g. expert systems and PDDL planning systems [4, 9]). We argue that a well featured symbolic pattern matcher provides many opportunities to simplify code; that appropriate matcher facilities allow inference engines and other systems to be constructed more concisely and with elegant code. We demonstrate this by example in the evaluation section of this

paper. The following sections outline different approaches to matching and present the key capabilities of the pattern matcher we have developed for Clojure.

2. INTRODUCTION

Broadly we aim to provide matching capabilities to simplify program code, to allow the definition of new types of functions which specialise on the structure of their arguments and can repeatedly apply patterns over larger data sets. We choose Clojure for this due to the nature of our applications (A.I. inference tools) which benefit from its semantics and its ability to integrate with Java. Clojure provides regular expression matching and some structural matching, there are Clojure libraries which provide tailored matching capabilities for specialised application areas (core.logic and core.unify [2]) and a partially specified matcher offering conditionals and function definition (matchure [10]). The matcher presented here is in part motivated and informed by these and other works but intentionally takes a different approach thereby offering alternative facilities. These, we suggest, can form the basis of a generalised symbolic matcher for Clojure.

2.1 Matching vs. Unification

While the difference between regular expression matching and symbolic matching is clear (regular expressions match at the level of strings and characters, symbolic matchers operate at the level of symbols and structures) the difference between matching and unification is more nuanced. Online forums suggest the difference between pattern matching and unification is only that unification is necessary/occurs if variables are allowed on both sides of a match expression. Here we accept there is some progression from simple pattern matching through to full unification but we consider the term "unification" to imply logical substitution. Specifically that a successful outcome of unification may leave variables unresolved in the sense that one or more variables may have more than one possible value or even an infinite set of possible values. For a wider discussion of unification see [3, 8].

We consider any process which simply associates one variable with one value to be matching. If variables are only permitted on only one side of a matching expression we term this "uni-directional" if variables are allowed on both sides we consider this "bi-directional".

We also consider examples where matching is uni-directional but still requires some level of unification (so unification does not fundamentally require variables on both sides of an expression). One example occurs when there are multiple patterns, with shared variables, which all need to be consistently satisfied.

A key aspect of any matching/unification algorithm is its policy for binding matched variables. The rest of this section explores some of the options. We assume that a function **f** takes two arguments and performs some matching or unification process on those arguments. **f** handles variables (denoted using a "?" prefix) and produces a mapping of variables to values if it succeeds.

An example of **f** using uni-directional matching:

```
f( [a ?x c], [a b c] ) → {?x ↦ b}
```

Bi-directional matching:

```
f( [a ?x c], [a b ?y] ) → {?x ↦ b, ?y → c}
```

The semantics of uni-directional matching are clear even when variables are bound more than once:

```
f( [a ?x ?x], [a b b] ) → {?x ↦ b}
f( [a ?x ?x], [a b c] ) → fail
```

However the semantics of bi-directional matching can become closer to some form of unification:

```
f( [a ?x c], [a ?y ?y] ) → {?x ↦ c, ?y ↦ c}
```

In this example there are two partial mappings (i) {?x ↦ ?y, ?y ↦ ?x} (ii) {?y ↦ c} which could unify in the following ways depending on the matching/unification algorithm:

(i)
```
{?x ↦ ?y, ?y ↦ ?x} ⊗ {?y ↦ c}
    → {?x ↦ ?y, ?y ↦ c} → {?x ↦ c, ?y ↦ c}
```

or:

(ii)
```
{?x ↦ ?y, ?y ↦ ?x} ⊗ {?y ↦ c}
    → {?x ↦ c, ?y ↦ ?x} ⊗ {?y ↦ c}
    → {?x ↦ c, ?y ↦ c}
```

Further considerations are necessary where values cannot be fully resolved during a single application of a pattern, this can occur for different reasons. For example, if variables can bind to 1 or more values (implied by the use of "??"):

```
f( [a ??x], [??x a] ) → {?x ↦ a...a}
```

Other variables may be unresolved or undefined:

```
f( [a ?x c], [a [?p ?q] c] )
→ {?x ↦ [?p ?q], ?p ↦ #\, ?q ↦ #\}
```

where #\ represents an undefined binding.

We could allow undefined bindings to propagate through to later expressions which would either successfully become unified or result in failure. The expectation in this case is that a fully developed (logic based) unification mechanism would be employed which would handle backtracking as necessary. This approach has its uses but can also present some limitations.

Consider a rule application mechanism which accepts rules of the form:

```
[Rule 5 (hairy ?x) => (mammal ?x)]
```

The rule application uses patterns in two stages (i) to deconstruct a rule and (ii) to work with its antecedent-consequent parts. In the first stage matching could be specified as follows:

```
f( [Rule ?id ?antecedent => ?consequent],
   [Rule 5 (hairy ?x) => (mammal ?x)] )

→ {?id ↦ 5, ?antecedent ↦ (hairy ?x),
   ?consequent ↦ (mammal ?x), ?x ↦ #\}
```

We then expect the ?x variable to be bound as part of the second stage. It is reasonable to expect that the rule application mechanism and the rules themselves are developed by different people and it is obvious practice to avoid any coupling between these specifications. However, when using the approach above, a small change in one of the patterns can have unwanted results because the same variable, ?x, is used on both sides:

```
f( [Rule ?x ?antecedent => ?consequent],
   [Rule 5 (hairy ?x) => (mammal ?x)] )

→ {?x ↦ 5, ?antecedent ↦ (hairy 5),
   ?consequent ↦ (mammal 5)}
```

This results in incorrect variable bindings for the second (rule application) phase. The situation could be avoided by requiring additional syntax for matching expressions, but adding syntactic notations has an impact on the usability of representations which is better avoided. In addition, while some of the matching forms (described below) use literally specified patterns and data, others allow patterns/data to be dynamically produced, e.g.

```
f( make-pattern1(),make-pattern2() )
```

The matcher could insist that all dynamically created patterns use some kind of name generator (gensym) for any dynamically created patterns but this approach has other drawbacks (it is harder to prime patterns with variables which are intended for sharing across pattern generators for example). Both cases above can be dealt with appropriately using a uni-directional approach (i.e.: assuming match variables are only used on one side of a match expression).

Even with uni-directional matching there may be a requirement for some level of unification, notably when multiple patterns, with shared variables, are to be consistently applied across data sets – a scenario which may also generate multiple possible matches. Consider matching a set of patterns $\{p_0, p_1 \ldots p_n\}$ over data $\{d_0, d_1 \ldots d_m\}$ where all variables need consistent values, for example:

```
f( { [?x ?y], [?y ?z] },
   { [a b], [q r], [c d], [m n], [p q] }

→ {?x ↦ a, ?y ↦ b, ?z ↦ c},
  {?x ↦ p, ?y ↦ q, ?z ↦ r}
```

In this case there are two valid mappings of matcher variables. While the function **f** may simply return these mappings we consider two other behaviors which may be preferred from a more fully developed matcher:

- to use any one of the matches found;
- to use each of the matches – either as arguments to some specified function or in some block of code which is called repeatedly for each valid match.

2.2 Design Principles

With the considerations outlined above, we aim to develop a matcher that provides the following:

- clean, unambiguous semantics which allow integration and nesting of different matcher forms (macros and functions) while consistently preserving their semantics and furthermore allows matcher forms to integrate and nest with other Clojure forms without disrupting the semantics of either;
- high-level matcher forms which abstract out the details of the matching processes themselves;
- pattern forms (unencumbered by unnecessary syntax) which allow matching to occur over nested lists, vectors and maps;
- a suitable mix of forms which take literally specified patterns (patterns specified in program text) and others which allow patterns to be dynamically specified (so patterns may be read, constructed or extracted from data at run-time);
- the ability to specify pattern groups with shared variables which implicitly require some type of unification (and by implication may require backtracking – see mfind* and mfor*;
- a namespace which (i) will extend (shadow) into lexically nested matcher forms and unwind out of these forms and (ii) may be captured in a data structure – to allow the state of successful matching to be saved and reinstated or provided (e.g. as an argument) to other functions and subsystems.

A key distinction between this matcher and those acknowledged earlier is that other matchers tend to operate only with static patterns and use normal (native) variable bindings. While this can provide some opportunity to improve performance it restricts the ability to store patterns as data or dynamically create them. The matcher presented below allows dynamic construction of patterns, an approach which significantly effects the matcher utility (as we demonstrate in the evaluation section).

3. MATCHER MACROS AND FUNCTIONS

This section describes key functions and macros developed to provide a symbolic pattern matcher for Clojure which addresses the issues discussed above. The matcher takes symbolic data structures and matches them against structured patterns. Patterns operate at the level of symbols and structures; they may contain literals and match variables. Match variables are prefixed with a "?" (or "??" – see later), symbols without a "?" prefix are literals so the pattern (?x ?y end) will match with any three element structure which contains 'end as its third element (binding match variables x and y to the first and second elements of the data). The pattern ((a b) {n ?x m ?y}) matches a nested structure binding the variables x and y to values for n and m held in a two-element map, nested within the data.

The most primitive form of matcher expression provided for general use is *mlet* (matcher-let), it is structured as follows:

```
(mlet [ pattern datum ] ...body... )
```

mlet operates as follows: if the pattern matches the datum, binding (zero or more) matcher variables as part of the matching process then *mlet* evaluates its body in the context of these bindings. If the pattern and datum do not match, *mlet* returns nil.

In the following example, the pattern (?x ?y ?z) matches the datum (cat dog bat) binding match variables "x", "y", "z" to 'cat, 'dog, 'bat respectively. The expression (? y) in the body of *mlet* retrieves the value of the match variable "y" from (pseudo) matcher name space.

```
(mlet ['(?x ?y ?z) '(cat dog bat)]
  (? y))
→ dog
```

mout (matcher-out) is a convenience form to build structured output from a mixture of literals and bound match variables:

```
(mlet ['(?x ?y ?z) '(cat dog bat)]
    (mout '(a ?x (a ?y) and a ?z)))
→ (a cat (a dog) and a bat)
```

mlet returns nil if matches fail:

```
(mlet ['(?x ?y ?z) '(cat dog bat frog)]
    (mout '(a ?x a ?y and a ?z)))
→ nil
```

Unbound matcher variables also have nil values as does the anonymous match variable "?_" which will always match with a piece of data but does not retain the data it matches against:

```
(mlet ['(?_ ?x) '(cat dog)]
    (list (? _) (? x) (? y)))
→ (nil dog nil)
```

Matcher variables are immutable so, once bound, a match variable cannot be implicitly re-bound and whilst the pattern (?x dog ?x) matches (cat dog cat) it will not match (cat dog bat) because this would result in an inconsistent/ambiguous binding for "?x". This approach also holds true with nested matcher forms, so given the data (dog bat) the following expression will return (cat dog bat) but with data (rat bat) it will return 'inner-match-failed:

```
(defn foo [data]
  (mlet ['(?x ?y) '(cat dog)]
    (or (mlet ['(?y ?z) data]
          (mout '(?x ?y ?z)))
      'inner-match-failed)
    ))

(foo '(dog bat)) → (cat dog bat)
(foo '(rat bat)) → inner-match-failed
```

In addition to *single element match directives* (prefixed with "?") the matcher supports *multiple match directives* which match against zero or more elements of data (these are prefixed with "??"). Multiple directives may also be used in matcher-out expressions, in which case their value is appended into the resulting structure:

```
(mlet ['(??pre x ??post)
       '(mango melon x apple pear berry)]
  (mout '(pre= ?pre post= ??post)))

→ (pre= (mango melon)
   post= apple pear berry)
```

All patterns may be structured, containing sequences, subsequences (and maps within sequences within maps within sequences, etc.), so it is possible to use patterns to extract data from nested data structures. The pattern used in the following example extracts the value from a *quantification* slot nested within an *actor* slot (which is also nested in the enclosing data structure)…

```
(mlet ['(??_ (actor ??_ [quant ?q] ??_) ??_)
       semantics ]
  (? q))
```

mlet has its uses but other forms (constructed on top of *mlet*) provide greater functionality. These other forms can be grouped into three families (i) switching and specialisation (ii) searching and selection (iii) iteration and collection.

3.1 Switching and Specialisation

mcond is the most general of the switching/specialisation forms, it can be used to specify a series of pattern based rules as follows:

```
(mcond [exp]
    ((?x plus ?y)  (+ (? x) (? y)))
    ((?x minus ?y) (- (? x) (? y)))
    )
```

The *mcond* form will attempt to match the data it is given (the value of *exp* in the example above) to the first pattern in its sequence of rules (?x plus ?y) then its second (?x minus ?y) until it finds a rule which matches; it then evaluates the body of that rule and returns the result. As with other matcher forms, *mcond* returns nil if it fails to find a match. The *mcond* form above will return 9 if exp has a value of (5 plus 4) or 1 if exp has a value of (5 minus 4). Note that *mcond* (and other forms) can optionally use additional symbols to make their rule-based structure more explicit, we recommend using ":=>" for example:

```
(mcond [exp]
    ((?x plus ?y)  :=> (+ (? x) (? y)))
    ((?x minus ?y) :=> (- (? x) (? y)))
    )
```

defmatch is similar in structure to *mcond*, wrapping an implicit *mcond* form with a function definition:

```
(defmatch math1 []
  ((?x plus ?y)  :=> (+ (? x) (? y)))
  ((?x minus ?y) :=> (- (? x) (? y)))
  )

(math1 '(4 plus 5))  → 9
(math1 '(4 minus 5)) → -1
(math1 '(4 times 5)) → nil
```

defmatch forms can take explicit arguments in addition to their implicit matched-data argument. The example below illustrates this and additionally uses an anonymous match variable to handle default cases:

```
(defmatch math2 [x]
  ((add ?y)  :=> (+ x (? y)))
  ((subt ?y) :=> (- x (? y)))
  ( ?_       :=> x)
  )

(math2 '(add 7) 12) → 19
(math2 '(subt 7) 12) → 5
(math2 '(times 7) 12) → 12
```

Due to the way patterns may be specified at the symbol level, *defmatch* forms can be used to specialise on keywords and thereby resemble some kind of dispatch, e.g.

```
(defmatch calcd [x y]
  (:add  :=> (+ x y))
  (:subt :=> (- x y))
  (:mult :=> (* x y))
  )

(calcd :add 5 4)  → 9
(calcd :mult 5 4) → 20
```

3.2 Searching and Selection

The searching and selection mechanisms apply patterns across collections of data, returning the first match found. These matcher forms are called *mfind* (which matches one pattern across a collection of data) and *mfind** (which consistently matches a

group of patterns across a collection of data). This is illustrated using the following data:

```
(def food
 '([isa cherry  fruit]   [isa cabbage veg]
   [isa chilli  veg]     [isa apple   fruit]
   [isa radish veg]      [isa leek    veg]
   [color leek  green]   [color chilli  red]
   [color apple green]   [color cherry  red]
   [color cabbage green] [color radish red]
  ))
```

Note that in this example we use vectors in our data, this is perhaps idiomatic but we sometimes prefer wrapping tuples as vectors (rather than as lists) and the matcher deals with either vectors or lists (or maps).

mfind takes one pattern, *mfind** takes multiple patterns:

```
(mfind ['[isa ?f veg] food] (? f))
→ cabbage

 (mfind* ['([isa ?f veg] [color ?f red])
          food]
    (? f))
→ chilli
```

3.3 Iteration and Collection

The matcher supports two forms to provide iteration and collection, these are called *mfor* and *mfor**. They iterate over sets of data using one pattern (*mfor*) or multiple patterns (*mfor**). The following examples use the food data presented above:

```
(mfor ['[isa ?f veg] food] (? f))
→ (cabbage chilli radish leek)

(mfor* ['([isa ?f veg] [color ?f red]) food]
    (? f))
→ (chilli radish)
```

3.4 Matcher Name Space

A *pseudo* matcher name space is maintained. This is not a Clojure name space but is a map (called *mvars*) which associates named matcher variables with their values. *mvars* is a lexically bound Clojure symbol accessible within the body of all matcher expressions.

with-mvars provides a simple way to inject variables into the matcher name space or shadow existing values, for example:

```
(with-mvars {'a (+ 2 3), 'b (- 3 4)}
  (println mvars)
  (with-mvars {'b 'bb, 'd 'xx, 'e 'yy}
    (println "  " mvars)
    (mlet ['(?a ?b ?d ?c) '(5 bb xx spam)]
      (println "    " mvars))
    (println "  " mvars))
  (println mvars))
```

output:

```
{b -1, a 5}
   {e yy, d xx, b bb, a 5}
     {c spam, :pat (?a ?b ?d ?c),
        :it (5 bb xx spam),
        e yy, d xx, b bb, a 5}
   {e yy, d xx, b bb, a 5}
{b -1, a 5}
nil
```

Note that the matcher adds the last datum that was match (called :it) and the last pattern :it matched against into the name space.

While direct reference to *mvars* is generally unnecessary, it is useful for writing new macros and it allows the results of successful matching operations to be saved for later processing or passed to other functions (in cases where the lexical scoping of matcher variables is found restrictive).

3.5 Implementation Notes

There are few basic building blocks to the matcher. The first is the core matches function which, in the context of any existing matcher variable bindings, performs the essential pattern matching process and builds a map of bindings for new matcher variables, e.g.

```
(matches '(a ?x ?y) '(a b c))
→ {y c, x b, :pat (a ?x ?y), :it (a b c)}

(with-mvars {'p 'ppp, 'q 'qqq}
    (matches '(a ?x ?y) '(a b c)))
→ {y c, x b, :pat (a ?x ?y),
     :it (a b c), p ppp, q qqq}
```

Other building blocks (*with-mvars* and *mlet*) use "*let*" forms to set up new lexical closures to shadow matcher name space when matching is successful which, in effect, provides lexical scope for matcher variables.

mcond (a macro) is specified as a series of *mlet* expressions and *defmatch* (also a macro) is specified in terms of *mcond*. *mfor*, *mfor**, *mfind* and *mfind** are also all specified as macros. *mfor* uses a function which recurses through an *mlet* form and *mfor** is specified in terms of *mfor*. *mfind* (like *mfor*) uses its own function to recurse through its own *mlet* form and *mfind** recurses through *mlet*.

In this way the expansion of nested matcher forms (defined as macros) produces a cascade of nested let forms where the pseudo matcher name space (*mvars*) is populated with match variables created by successful matches.

4. EVALUATION

We have evaluated the matcher from three different perspectives:

(i) an objective examination to assess whether matcher functions and macros operate as intended; whether they are semantically consistent when nested/interleaved with other matcher expressions and Clojure forms;

(ii) from the subjective view of Clojure programmers are the semantics of matcher forms appropriate and do their names (*mfind*, *mfor*, etc.) mnemonically suggest their semantics?

(iii) is the matcher useful? Does it simplify the construction and readability of Clojure code? Specifically (since this is the nature of much of our work) we are interested in simplifying the construction of inference engines – typically based on the application of rules and state changing operations.

The first approach to evaluation (above) is not described here, the matcher was constructed using a strict *test driven development* approach. The matcher presented here satisfies all tests.

User acceptance evaluation has been conducted using the matcher as a basis for student assessments and projects and also as a build tool for developing larger subsystems. Feedback from these user groups has influenced (i) the choice of names for matcher forms (macros and functions), (ii) the syntactic conventions used to specify macros and patterns and (iii) cases where the matcher semantics needed to be more clearly specified. Detailed analysis of this process is not discussed here, instead we focus on the third style of evaluation which considers the utility of the matcher as a tool for code construction.

4.1 Searching Sets of Tuples

For the first example we consider searching for objects in a set of tuples which describe the state of a micro-world. To put this in context: we receive object descriptions (and other forms) from language processing subsystem so, for example, the noun-phrase "red fruit" would produce:

```
(obj
  (quantifier all)
  (desc ((color red) (isa fruit))))
```

The phrase "a large red fruit" would produce:

```
(obj
  (quantifier any)
  (desc
    ((size large) (color red) (isa fruit))))
```

We store state information in the following form:

```
(def food
  '#{[isa chilli veg]     [isa cherry fruit]
     [isa radish veg]     [isa apple fruit]
     [isa leek veg]       [isa kiwi fruit]
     [color chilli red]   [color cherry red]
     [color radish red]   [color apple green]
     [color leek green]   [color kiwi green]
     [on chilli table]    [on cherry table]
     [on leek table]
     })
```

Our aim is to write code which, using the type of object descriptions from the language processing subsystem, can retrieve the relevant object names. Given the matcher facilities described in preceding sections we can use *mfor* to find the names of objects for a single type of fact/tuple. For example, the following form returns the names of all cubes:

```
(mfor ['(isa ?obj veg) food]
    (? obj))
→ (chilli leek radish)
```

It is possible to dynamically construct a suitable pattern for an mfor expression from the type of [relation value] pairs provided by the language processing subsystem. A match function provides a convenient way to extract the components of a [relation value] pair which can then be used in the mfor expression:

```
(defmatch find-all [tuples]
  ([?reln ?val]
    (mfor ['(?reln ?obj ?val) tuples]
      (? obj)
      )))

(find-all '(isa veg) food)
→ (chilli leek radish)
```

If the results of multiple find-all expressions are converted to sets multiple (relation value) pairs can be handled using set operators. So to find red vegetable from the food data:

```
(find-all '(isa veg) food)
→ (chilli leek radish)

(find-all '(color red) food)
→ (chilli radish cherry)

(intersection
    (set '(chilli leek radish))
    (set '(chilli radish cherry)))
→ #{radish chilli}
```

This processing can be captured in a function as follows:

```
(defn query
  [reduction pairs tuples]
  (reduce reduction
    (map #(set (find-all % tuples)) pairs))
  )

(query intersection
    '((isa veg)(color red)) food)
→ #{radish chilli}
```

The query function may also be used with union to return "or" combinations:

```
(query union
    '((isa veg)(color red)) food)
→ #{cherry radish chilli leek}
```

To satisfy our initial aim we therefore need the following:

```
(defmatch find-all [tuples]
  ([?reln ?val]
    (mfor ['(?reln ?obj ?val) tuples]
      (? obj)
      )))

(defn query
  [reduction pairs tuples]
  (reduce reduction
    (map #(set (find-all % tuples)) pairs))
  )
```

4.2 Application of Rules

The second example considers a rule-based, fact deduction or forward chaining mechanism. Facts are held as tuples and rules have antecedents and consequents. Some introductory texts for Artificial Intelligence provide example rules like:

```
IF (has fido hair) THEN (isa fido mammal)
```

While these serve to illustrate their discussion of rule-based inference, rules like this are of limited use because they are specific to object names ("fido" in this case) and take only a single antecedent and consequent. For practical purposes we need to extend this rule syntax – to allow rules to be flexible about the length of their antecedents/consequents and the objects they describe. Specifying rules in terms of match variables and writing a flexible rule application mechanism addresses this. For example:

```
(rule 15 (parent ?a ?b) (parent ?b ?c)
      => (grandparent ?a ?c))
```

can match against tuples like:

```
(def family
 '((parent Sarah Tom) (parent Steve Joe)
   (parent Sally Sam) (parent Ellen Sarah)
   (parent Emma  Bill)(parent Rob   Sally)))
```

A suitable rule application mechanism needs to split the rule into its constituent parts, search for all consistent sets of antecedents, ripple any antecedent variable bindings through to consequents and collect evaluated consequents for each rule every time it fires. In practice these requirements can be satisfied by using a match function to pull a rule apart, *mfor** to satisfy all possible antecedent combinations and *mout* to bind variables into consequents. This can be specified as follows:

```
(defmatch apply-rule [facts]
  ((rule ?n ??antecedents => ??consequents)
    :=> (mfor* [(? antecedents) facts]
          (mout (? consequents)))))

(apply-rule
  '(rule 15 (parent ?a ?b) (parent ?b ?c)
            => (grandparent ?a ?c))
  family)

→ ((grandparent Ellen Tom)
   (grandparent Rob Sam))
```

Notice that while the pattern for *defmatch* is literally specified, the patterns for *mfor** and *mout* must, necessarily, be generated dynamically. Furthermore these dynamically generated patterns are embedded in the rule structure pulled apart by *defmatch*'s literal pattern.

To investigate this rule deduction example further we use a richer set of facts and rules where the consequences of some rules trigger the antecedents of others (we choose a "toy" example to illustrate this).

```
(def facts
  '((mineral pebble)  (small pebble)
    (mineral boulder) (large boulder)
    (small daisy)     (light daisy)
    (on boulder daisy)
     ))

(def rules1
  '((rule 0
       (dangerous ?x)(fragile ?y)(on ?x ?y)
           => (broken ?y))
     (rule 1 (heavy ?x) => (dangerous ?x))
     (rule 2 (large ?x) => (heavy ?x))
     (rule 3 (small ?x)(light ?x)
           => (portable ?x)(fragile ?x))
    ))
```

Given these definitions it is possible to develop a function to apply all rules once:

```
(defn apply-all [rules facts]
  (reduce concat
    (map #(apply-rule % facts) rules)
    ))

(apply-all rules facts)
→ ((hard pebble)   (hard boulder)
   (fragile daisy) (portable daisy))
```

For simplicity in combining the output of rules we use sets which necessitates modifying the apply-all function to:

```
(defn apply-all [rules facts]
  (set (reduce concat
         (map #(apply-rule % facts) rules)
         )))
```

A forward chaining/fact deduction function which continues to operate while it is generating new facts can then be defined:

```
(defn fwd-chain [rules facts]
  (let [new-facts (apply-all rules facts)]
    (if (subset? new-facts facts)
      facts
      (recur rules (union facts new-facts))
      )))

(fwd-chain rules (set facts))
→ #{(light daisy)  (mineral boulder)
    (hard boulder) (fragile daisy)
    (small daisy)  (heavy boulder)
    (broken daisy) (small pebble)
    (mineral pebble) (hard pebble)
    (portable daisy) (on boulder daisy)
    (large boulder)}
```

As with the previous example, the matcher performs most of the processing (in this case using a *defmatch* construct and *mfor** in apply-rule) while other functions collate results, etc.

```
(defmatch apply-rule [facts]
  ((rule ?n ??antecedents => ??consequents)
    :=> (mfor* [(? antecedents) facts]
          (mout (? consequents)))))

(defn apply-all [rules facts]
  (reduce concat
    (map #(apply-rule % facts) rules)
    ))

(defn fwd-chain [rules facts]
  (let [new-facts (apply-all rules facts)]
    (if (subset? new-facts facts)
      facts
      (recur rules (union facts new-facts))
      )))
```

4.3 Application of Operators

In this example we consider how to apply the kind of state changing operators that are used in some planning systems. Broadly we adapt a representation borrowed from PDDL [4, 9] for use with a STRIPS [6] style solver. The operators are specified in terms of their preconditions and their effects. As with the earlier examples, we use tuples to capture state information. The following tuples, for example, describe a simple state in which some (animated) agent (R) is at a table, holding nothing and a book is on the table.

```
#{(at R table)         (on book table)
  (holds R nil)        (path table bench)
  (manipulable book)   (agent R) }
```

In order to generalise an operator (so it can be used with different agents, objects and in various locations) it is necessary to specify it using variables, in this case matcher variables. An operator which describes a "pickup" activity for an agent and which can be

used to produce a new state (new tuples) can be described as follows:

```
{:pre ((agent ?agent)
       (manipulable ?obj)
       (at ?agent ?place)
       (on ?obj   ?place)
       (holds ?agent nil)
       )
 :add ((holds ?agent ?obj))
 :del ((on ?obj   ?place)
       (holds ?agent nil))
 }
```

The operator is a map with three components (i) a set of preconditions which must be satisfied in order for the operator to be used (ii) a set of tuples to add to an existing state when producing a new state and (iii) a set of tuples to delete from an existing state.

To apply this kind of operator specification we extract patterns from the operator then use *mfind**

```
(defn apply-op
  [state {:keys [pre add del]}]
  (mfind* [pre state]
    (union (mout add)
      (difference state (mout del))
      )))

 (apply-op state1 ('pickup ops))
 →  #{(agent R) (holds R book)
      (manipulable book)
      (path table bench) (at R table)}
```

As with the previous examples, the patterns used by *mfind** are provided dynamically when apply-op is called. Furthermore, in this example, the patterns themselves define the semantics of the operators.

Collections of operators are conveniently held in a map and ordered sequences of operator applications can be formed by chaining apply-op calls, e.g.

```
(def ops
  '{pickup {:pre ((agent ?agent)
                  (manipulable ?obj)
                  (at ?agent ?place)
                  (on ?obj   ?place)
                  (holds ?agent nil)
                  )
            :add ((holds ?agent ?obj))
            :del ((on ?obj   ?place)
                  (holds ?agent nil))
            }
    drop    {:pre ((at ?agent ?place)
                   (holds ?agent ?obj))
             :add ((holds ?agent nil)
                   (on ?obj   ?place))
             :del ((holds ?agent ?obj))
             }
    move    {:pre ((agent ?agent)
                   (at ?agent ?p1)
                   (path ?p1 ?p2)
                   )
             :add ((at ?agent ?p2))
             :del ((at ?agent ?p1))
    }})
```

```
(-> state1  (apply-op ('pickup ops))
            (apply-op ('move   ops))
            (apply-op ('drop   ops)))

→ #{(agent R) (manipulable book)
    (on book bench) (holds R nil)
    (at R bench) (path table bench)}
```

We can further develop this example so apply-op (or some similar function) works with a search process or a STRIPS-style planner to generate sequences of moves in order to reach a goal state.

5. SUMMARY & CONCLUSION

This paper has argued that pattern matching can be used to simplify some programming tasks, facilitating the production of concise, well-formed code with precise semantics. We have presented a symbolic pattern matcher (now available under "clojure resources" at www.agent-domain.org) which binds immutable match variables and provides matcher functions/macros to support pattern-based conditional statements, function definitions and iterative/mapping forms. The matcher has some forms which take literal patterns but, importantly, has others which allow their patterns to be retrieved from data structures or to be constructed at run-time. This provides increased flexibility in pattern production and use; allowing some types of rules and state-change operators to have their semantics described in terms of patterns. These in turn facilitate the construction of inference engines which apply these structures. In the evaluation section we have presented three sample problems (searching tuples, applying rules and using operators) and demonstrated how the matcher can be employed to solve these problems.

6. REFERENCES

[1] Barr, A., Feigenbaum, E.A., The Handbook of Artificial Intelligence, (1981) vol 1, II F

[2] Fogus, M., Houser, C., Joy of Clojure (2nd ed.) ch. 16, (2014)

[3] Franz, B. and Snyder. W., "Unification Theory." *Handbook of automated reasoning 1* (2001): 445-532.

[4] Ghallab, M., Knoblock, C., Wilkins, D., Barrett, A., Christianson, D., Friedman, M., ... & Weld, D. (1998). PDDL-the planning domain definition language.

[5] Jones, T., "Artificial Intelligence: a systems approach" Jones & Bartlett, ISBN 978-0-7637-7337-3 (2009)

[6] Lekavý, M., Návrat, P. Expressivity of STRIPS-Like and HTN-Like Planning, Agent and Multi-Agent Systems: Technologies and Applications (2007), LNCS 4496, 121-130

[7] Odersky, M.: The Scala Language Specification (2008)

[8] Oliart, A., and Snyder, W., "Fast Algorithms for Uniform Semi-Unification," Journal of Symbolic Computation 37 (2004) 455-484.

[9] De Weerdt, M., T. Mors, A.T., Witteveen, C., Multi-agent planning: An introduction to planning and coordination In: Handouts of the European Agent Summer (2005), pp. 1-32

[10] Younger, B., "Matchure", (2013) https://clojars.org/brendanyounger/matchure

Quantum Physics Simulations in Common Lisp

Miroslav Urbanek
Faculty of Mathematics and Physics
Charles University in Prague
Ke Karlovu 3
Prague, Czech Republic
miroslav.urbanek@mff.cuni.cz

ABSTRACT

Simulations based on tensor networks have become popular in computational quantum physics recently. They allow to predict the behavior of quantum models with a large number of particles. While most of the software for tensor-network simulations is written in MATLAB or C++, implementations in higher languages have many advantages. They allow to extend simulations to more complex models. In this article I present TEBDOL, a program using the time-evolving block decimation algorithm (TEBD) to simulate time evolution of one-dimensional chains of atoms in optical lattices. I discuss the advantages and disadvantages of using Common Lisp for high-performance computing, and report on strong and weak scaling performance of the program.

Categories and Subject Descriptors

J.2 [**Physical Sciences and Engineering**]: Physics; I.6.8 [**Simulation and Modeling**]: Types of Simulation—*Parallel*

General Terms

Languages, Performance

Keywords

Common Lisp, High-performance computing, Quantum physics, Tensor network

1. INTRODUCTION

The world of scientific high-performance computing (HPC) is a world firmly rooted in programming languages Fortran, C, and C++. Tools and libraries provided by the system vendors usually have bindings to these languages only. With growing complexity of numerical simulations there is a need to use higher-level languages that provide better abstractions. At the same time, huge problem sizes require good numerical performance.

In this article I report on the experience with using Common Lisp for high-performance computing. There are many popular implementations of the language available. We use Steel Bank Common Lisp (SBCL) [5] specifically. Most of our code is written in ANSI Common Lisp, but there are several areas where we use extensions available in SBCL only. We decided for this approach based on observations that a) SBCL has a permissive open-source license, b) it is actively developed and has a vibrant community, c) it runs on x86_64, and d) it generates reasonably fast code. We did not use a portable libraries like CFFI [6] because portability was not a priority, and each library or dependency brings additional complexity. However, it should be a fairly easy to port the program to a different implementation.

TEBDOL uses tensor-network algorithms to simulate time-evolution of ultracold atoms in one-dimensional optical lattices. It has good numerical performance, and has been parallelized using Message Passing Interface (MPI). It was used to investigate phase revivals in binary mixtures of ultracold atoms [8].

Time evolution of a system is simulated using the time-evolving block decimation (TEBD) algorithm [9, 10]. The program uses BLAS and LAPACK libraries for low-level numerical algebra and MPI library for communication between parallel processes. Up to 1024 CPU cores were used in one simulation run. The program was tested on several supercomputers.

Raw performance is not our only goal. The program uses external libraries for most of the heavy numerical lifting anyway. We are instead interested in a good balance between speed and convenience. After successfully investigating a single physical model, our focus usually shifts to a different model. This requires additional non-trivial programming. We therefore prefer a convenient and high-level programming language that allows to constantly reshape the code. Our first version was developed in MATLAB. As the program grew more complex, it became clear that MATLAB does not provide language constructs we would like to use. We considered several languages, and we chose Common Lisp because it provides a nice balance of required features. There is other software for tensor network manipulation available, namely ITensor [3] written in C++. It represents an example of different approach to similar problems in C++ in comparison with Common Lisp.

SBCL supports symmetric multiprocessing, but we currently do not use it. There are two reasons for that. First, calculations with tensors networks consume huge amounts of memory. It is common that TEBDOL consumes tens of gigabytes on a single node. While many supercomputers include a few *fat* nodes, which have hundreds of gigabytes of available memory, most of the nodes are equipped with a somewhat conservative amount of 32-128 GB RAM. We therefore decided to parallelize TEBDOL with MPI for distributed-memory architectures. With this approach we are able to

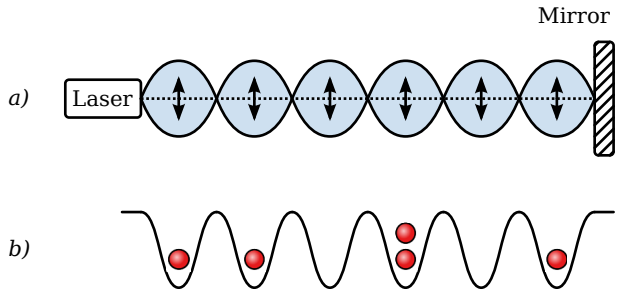


Figure 1: Optical lattice confines ultracold neutral atoms in a periodic structure. a) A reflected laser beam interferes with the original beam. The resulting standing wave creates a periodic energetic potential for the atoms. b) The atoms are confined to potential minima, which constitute lattice sites. They can hop between sites, and multiple atoms at a single site repulse each other.

scale the program beyond the memory limits of a single node and thus simulate larger and more difficult systems. Second reason is that the BLAS/LAPACK libraries are usually parallelized using OpenMP. Difficult simulations spend most of the time in routines from these libraries. The program therefore utilizes additional parallel processing in each MPI process.

2. MANY-BODY QUANTUM PHYSICS

An optical lattice is an experimental device used to investigate fundamental quantum physics. A laser beam, reflected by a mirror, interferes with the original beam, and both beams create a standing electromagnetic wave (Fig. 1). The standing wave provides a tiny energy difference between its nodes and anti-nodes for a gas of ultracold neutral atoms. Atoms then stay in places of minimal energy. These spatial points constitute the lattice sites.

A state of a system in quantum physics is represented by a complex unit vector, and all physical quantities correspond to linear operators, which are represented by matrices. Numerical quantum simulations employ mainly linear algebra. Typical questions asked are a) what is the energy spectrum, i.e., what are the eigenvalues of the energy operator, and b) how the system evolves in time. The energy operator is usually called a Hamiltonian.

The physics of low-energetic atoms loaded into an optical lattice is described by the Bose-Hubbard model with Hamiltonian

$$H = -J \sum_{\langle i,j \rangle} b_i^\dagger b_j + \frac{U}{2} \sum_i n_i \left(n_i - 1\right). \qquad (1)$$

Terms in the sums above and also the Hamiltonian itself are matrices. The integers i and j index lattice sites. We consider a system of length L and impose the open boundary conditions, therefore $0 \leq i, j < L$. The angle brackets denote sum over adjacent sites, because we take into account interactions between neighboring sites only. The operators b_i and $b_i^\dagger$ are so-called annihilation and creation operators, and $n_i = b_i^\dagger b_i$ are particle number operators. A term $b_i^\dagger b_j$ moves a particle from the site i to the site j. The parameter J describes the amount of tunneling between the sites and the parameter U denotes the intensity of the on-site repulsion.

A state of the system is represented by a vector in an abstract space. The matrices above act on this space. All quantum-mechanical calculations can be reduced to operations with these vectors and matrices. Depending on the number of lattice sites and particles, the space has a certain dimension.

This dimension grows exponentially with the system size. For example, let us assume that each lattice site can be occupied by a single particle only, i.e., each site can be in a vacant state or in an occupied state. A single site then represents a two-state quantum system. The vector space of this system has a dimension of $D_1 = 2$. With two lattice sites, there are two possible states at each site, and the state space has a dimension of $D_2 = 2 \times 2 = 4$. The space dimension becomes $D_L = 2^L$ for a system with L lattice sites. For a moderate system of $L = 20$ sites, this gives a dimension of $D_{20} = 1048576$. Each observable quantity in this system is represented by a $D_{20} \times D_{20}$ matrix. Such matrices are very difficult to handle numerically. The problem can be somewhat simplified because the total number of particles is usually fixed, and the calculations can be restricted to a subspace of the above vector space. Additionally, relevant matrices are usually represented by sparse matrices, which make the problem more tractable. However, mesoscopic systems with hundreds of particles, which are interesting from the experimental point of view, are unreachable with this approach.

It has been found that there exists a better description of a quantum system using *tensor networks*. This representation is especially useful for studying one-dimensional systems, where it produces very precise results, while being tractable on current computers.

3. TENSOR NETWORKS

A tensor is a multidimensional array. A rank of a tensor specifies its number of dimensions. A rank-0 tensors is a scalar, a rank-1 tensor is a vector, a rank-2 tensor is a matrix, and so on. A common tensor operation is a tensor contraction. It is a generalization of matrix multiplication. Let $A_{i_1 i_2 i_3 \ldots}$ and $B_{j_1 j_2 j_3 \ldots}$ be two tensors of an arbitrary rank. A tensor contraction over indices i_a and j_b of equal dimension is an operation that produces a new tensor C defined by

$$C_{i_1 \ldots i_{a-1} i_{a+1} \ldots j_1 \ldots j_{b-1} j_{b+1} \ldots} = \sum_k A_{i_1 \ldots i_{a-1} k i_{a+1} \ldots} B_{j_1 \ldots j_{b-1} k j_{b+1} \ldots}. \qquad (2)$$

A convenient graphical representation of a tensor is a ball with several legs that represent its indices. A tensor contraction can be depicted by joining legs of two tensors. A tensor network is a set of tensors and their contractions (Fig. 2).

Besides the tensor contraction, there are other important tensor operations, namely a tensor decomposition, a tensor permutation, and a tensor fusion (Fig. 3).

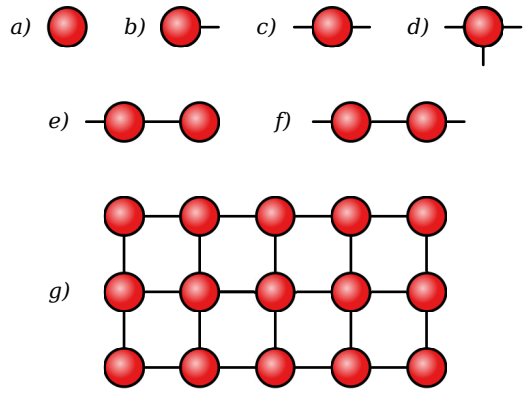


Figure 2: A tensor can be graphically represented as a ball with several legs representing its indices. The number of its legs is the same as the tensor rank. A leg connecting two tensors represents a tensor contraction. A tensor network is then a graph whose vertices represent tensors and whose edges represent tensors contractions. The figure shows an example of a) a scalar, b) a vector, c) a matrix, d) a rank-3 tensor, e) a matrix-vector multiplication, f) a matrix-matrix multiplication, and g) a large tensor network that results in a scalar after performing all its contractions.

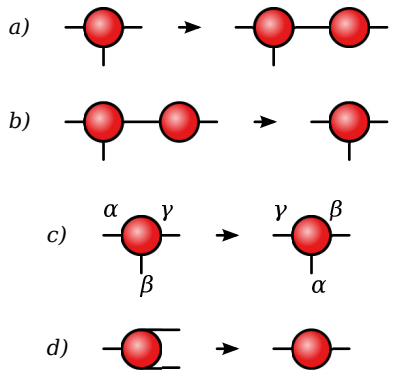


Figure 3: a) A tensor decomposition splits a single tensor into two. It is a generalization of the singular value decomposition. b) A tensor contraction joins two tensors into one. c) A tensor permutation reshuffles the order of indices. d) A tensor fusion combines several indices into one.

A numerical analysis of certain quantum models becomes more tractable when the system state and its operators are represented by tensor networks instead of vectors and matrices. The full system description still requires the same amount of information. However, under reasonable assumptions many elements can be ignored because they contribute to the physical behavior only a little. Approximation of the full system with a tensor network containing only relevant elements produces very precise results.

TEBDOL uses tensors as its basic building blocks. Tensors are organized in tensor networks that represent a system state, a Hamiltonian, and observables (Fig. 4). Tensors dimensions are truncated during simulations, and only states of high physical probability are kept in memory. This ensures that the problem stays tractable.

Many interesting physical models exhibit symmetries and thus conserve certain physical quantities. The model (1) conserves the total number of particles. Particles cannot suddenly appear in the system or disappear from it. We take advantage of this fact in our tensor representation. It turns out that it is not necessary to work with a full tensor, because some parts of it are guaranteed to be filled with zero elements only. TEBDOL therefore works with *symmetric* tensors, and for each tensor stores only sectors which can contain non-zero elements. Each sector is stamped with the number of particles it represents. This approach allows us to further reduce the computational costs.

4. PARALLELIZATION

A parallel implementation of the TEBD algorithms is conceptually simple. A full tensor networks is divided among MPI processes. Each process calculates time evolution of a part of the network it manages. After each time step it is necessary to perform an exchange of boundary tensors between processes managing adjacent parts of the network. Communication costs are small in comparison with costs of numerical routines. MPI provides a major speedup in the calculation time.

Additional parallelization comes from BLAS/LAPACK libraries. We benchmarked several implementations during the initial phase of the project. Commercial implementations provided by vendors, for examples Inter Math Kernel Library (MKL) [2] for Intel processors, offer great performance. Among the open-source implementations, GotoBLAS [1] stands out both with its superb performance and its clean design. It is remarkable that it was developed by a single person, Kazushige Goto, whose hand-written optimizations could surpass optimizations generated by compilers. The last architecture GotoBLAS is optimized for is Intel Nehalem. OpenBLAS is a project originating from GotoBLAS that provides optimizations for current processors, including Intel Sandy Bridge architecture. We achieved performance similar to Intel MKL with OpenBLAS.

5. COMMON LISP BENEFITS

Programming in ANSI Common Lisp brings many practical benefits. Expressive power of the language reduces development time. At the same time, the code produced by SBCL is very fast. Interactive nature of the SLIME development environment allows the programmer to design and immediately test program functions. In the following I describe several program areas that benefit from using Common Lisp.

5.1 Data structures

Common Lisp includes a rich set of built-in data types and allows to easily create complex data structures. The basic building block of TEBDOL is a symmetric tensor. The tensor data structure contains a list of indices and list of non-vanishing sectors. Each index is a list of segments, where each segment is defined by a list of particle numbers and a segment dimension. Each sector contains lists of particle numbers and a complex double-float array. Following the

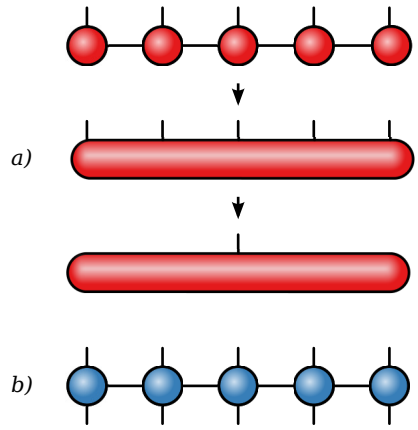


Figure 4: a) A state of a one-dimensional quantum system is represented by a tensor network consisting of rank-3 tensors in the bulk and rank-2 tensor at the edges. A transformation from a tensor-network representation to the standard vector representation consists of performing all tensor contractions and fusing all tensor indices into a single index. b) A one-dimensional quantum operator is represented by a tensor network consisting of rank-4 tensors in the bulk and rank-3 tensors at the edges.

theory of symmetric tensors, a sector is non-vanishing only if its lists of particle numbers sum to a list of zeros. As an illustration, the following listing shows a rank-2 tensor with 0 to 2 particles on each index, equivalent to a 3x3 identity matrix:

```
#s(tensor
   :indices
    ((#s(segment :numbers (0) :dimension 1)
      #s(segment :numbers (1) :dimension 1)
      #s(segment :numbers (2) :dimension 1))
     (#s(segment :numbers (-2) :dimension 1)
      #s(segment :numbers (-1) :dimension 1)
      #s(segment :numbers (0) :dimension 1)))
   :sectors
    (#s(sector
        :numbers ((0) (0))
        :array #2a((#c(1.0d0 0.0d0))))
     #s(sector
        :numbers ((1) (-1))
        :array #2a((#c(1.0d0 0.0d0))))
     #s(sector
        :numbers ((2) (-2))
        :array #2a((#c(1.0d0 0.0d0))))))
```

Tensor operations transform input tensors into output tensors and thus have to access and manipulate tensors in an intricate way. For example, in a tensor contraction only sectors with matching particle numbers are combined into a new sector. TEBDOL uses hash tables with lists as keys to find matching sectors of the input tensors. All these operations are conveniently expressed in Common Lisp. Moreover, data structures are allocated and collected automatically. Common Lisp ability to create and manipulate complex structures is its key characteristic. It makes possible to deliver a functioning software in a realistic time frame.

5.2 Multidimensional indices

The elements of multidimensional arrays are accessed using multidimensional indices. A multidimensional index is a list of subscripts. Common Lisp offers a simple expression to access an array element using a multiindex:

```
(apply #'aref array multiindex)
```

Operations with multiindices are similarly simple. For example, the following function implements a sum of multiindices:

```
(defun multi-index-+ (&rest args)
  (apply #'mapcar #'+ args))
```

Working with multidimensional arrays and indices is thus very convenient. Functions to manipulate, compare, and iterate over multiindices are easy to express.

5.3 Array permutations

A common operation in array manipulations is a permutation of array indices. It corresponds to a matrix transpose. We have found that an abstract implementation of index permutations for arrays of arbitrary rank is rather slow. On the other hand, SBCL can generate an efficient machine code for a fixed permutation implemented using nested loops. TEBDOL therefore uses a macro to generate an expression with nested loops that performs a particular permutation. Fixed permutation functions are generated, compiled and stored in a cache during the program runtime. The array-permutation function works in the following way:

1. If there already exists a permutation function for the requested array rank and permutation description in the cache, it is called.
2. Otherwise, such function is generated using the permutation macro.
3. The generated function is compiled. SBCL produces optimized code for nested loops.
4. The compiled function is stored in the cache. The cache is implemented using a hash table with permutation descriptions as keys.
5. The compiled function is called to perform the array permutation.

The above strategy preserves the high-level interface, where the permutation-function parameters are only the array itself and the permutation description. At the same time, SBCL produces highly optimized and fast code. This strategy would be very difficult to implement in the most of the other programming languages.

There are several types of tensor contraction performed during a single program run. We could achieve a similar performance without using macros by implementing each contraction type individually. Extending the algorithm, for example to two-dimensional models, would be a challenging task. We would need to implement new contraction types. Therefore the main advantage of our solution is its generality and its ease of use. Common Lisp allowed us to achieve it without compromising on the performance.

5.4 Array contractions

Similarly to the array permutations, the array contraction function makes use of a generating macro and a function cache. The performance benefits are not as pronounced as for the permutation function. However, this strategy ensures that the contraction function is as fast as possible. It also helps in profiling, as each generated function shows up independently in the profiler statistics.

Array contractions in TEBDOL utilize the BLAS library. First, TEBDOL permutes array indices to ensure that the contraction indices are either the leading or the ending indices of both arrays. Then it calls the BLAS matrix multiplication routine. Here, the program uses SBCL functions `array-storage-vector` and `vector-sap` that allows to pass a pointer to Lisp typed arrays to external routines. As a result, the contraction function has a high-level interface that supports arrays of any rank and contractions over multiple indices. At the same time, the code is very fast thanks to the optimized BLAS library.

5.5 Data serialization

Distributed-memory parallel programs have to transfer data between computing nodes. The MPI standard provides several data sending and receiving routines. These routines are designed to transfer C and Fortran arrays and structures. TEBDOL exchanges symmetric tensors between the nodes. A symmetric tensor consists of lists, numbers, typed arrays, and a few Lisp structures. To conveniently transfer these tensors, TEBDOL uses custom data serialization and deserialization routines.

The routines are implemented using macros. For each data type, a custom function to serialize and deserialize the type is defined. For example, the following expressions define these routines for the type `fixnum`:

```
(defser fixnum (obj buf pos)
  (if (plusp (length buf))
      (pack obj +fixnum-size+ buf pos))
  +fixnum-size+)

(defdes fixnum (buf pos)
  (values (unpack +fixnum-size+ buf pos)
          +fixnum-size+))
```

The macro `defser` generates a function that first stores a one-byte type identifier into the output buffer `buf`. This ensures that a correct function, generated by the `defdes` macro, is dispatched during the deserialization. The functions `pack` and `unpack` store and load representations of object `obj` to and from a byte array `buf` at position `pos`, respectively. When the serialization routine is called with a buffer of length zero, the routine does not store any data, and only calculates the required size of the output array. The routines for more complex data structures recursively store and load each structure member.

The data transfer in the sending process works as follows:

1. The required buffer length for the transferred object in bytes is calculated.
2. A byte array with the calculated length is created.
3. The object is serialized into the created array.
4. Data in the array are sent as the MPI type `MPI_BYTE`.

In the receiving process:

1. The length of the incoming data is obtained using the function `MPI_Probe`.
2. An input buffer with the required length is created.
3. Data are received into the created array as the MPI type `MPI_BYTE`.
4. Data are deserialized into an object.

The serialization and deserialization routines provide a powerful method to transfer any complex Lisp object with MPI. They are fast enough for our application. A calculation of new tensors during a time-evolution step takes an order of magnitude longer than the exchange of tensors between MPI processes.

6. COMMON LISP ISSUES

There are several areas that are difficult to handle in Common Lisp. Although the hindrances are not fundamental, they have to be taken into account during program development. Most of the following issues are specific to SBCL, but they illustrate a class of problems one encounters when interacting with real systems.

6.1 Profiling

Profiling is a necessary technique for developing a high-performing program. SBCL offers two profiling packages, a deterministic profiler `sb-profile` and a statistical profiler `sb-sprof`.

`sb-profile` does not profile foreign functions and therefore is not suitable for use with TEBDOL. `sb-sprof` works better, it covers foreign functions and also prints a call graph. Its performance hit is small enough and can be decreased by adjusting the sampling frequency. However, the profiler does not provide a complete performance picture. Its results does not show the overhead of internal routines, especially time spent in the garbage collector.

Another approach is to use the Linux system-wide profiler `perf`. It provides accurate and detailed statistics using hardware counters. However, it does not distinguish between individual Lisp functions. `perf` supports dynamic languages as well. The program has to create a file with debug information in the form of function addresses. It should be straightforward to create a library providing this debug information for SBCL.

There exist powerful tools for code profiling and analysis geared towards HPC users. A package underpinning several of these tools is `Score-P` [4]. It provides instrumentation for programs written in C/C++ and Fortran. Using a compiler wrapper, `Score-P` automatically inserts calls to trace functions during a compilation. The trace data generated in a program run can be analyzed to obtain detailed report about program bottlenecks. `Score-P` is easy to use in C/C++ and Fortran, but there is no support for Common Lisp. The only way to use it is to explicitly call trace functions. While this approach is possible, the difficulties in low-level interaction with the library make it a cumbersome solution. Profiling cluster performance is thus quite limited in Lisp. A tool that could automatically provide `Score-P` instrumentation for Lisp programs would be a great improvement.

A good profiler should have low overhead, profile foreign functions, produce a call graph, and include garbage collection overhead. `sb-sprof` is currently closest to achieving these objectives. Profiling Lisp code with `perf` or `Score-P` would provide a better overview of overall performance, but has not been developed yet.

6.2 Garbage collection

TEBDOL is a memory-bound program. It can consume any amount of computer memory to accurately simulate interesting physical models. Garbage collection is thus a major factor influencing both the maximal accessible model sizes and calculation times. The behavior of the garbage collector is hard to predict.

SBCL uses a generational garbage collector. Its design is based on a hypothesis that the most recently allocated objects are ephemeral, i.e., they are also the most probable to be collected soon. The collector allocates objects in generations and collects the most recent generations more often. The oldest generation is collected only rarely.

The memory usage pattern of TEBDOL is quite different. The existing tensors are used to calculate new tensors in each time step. The program thus allocates huge data structures in every step. The old structures can be released when they are not referenced anymore. However, they are usually kept in older generations and the collector does not collect them immediately. Old huge tensors therefore easily fill up the available dynamic space. This usage pattern, where the oldest allocated objects are released often, does not satisfy the hypothesis above.

With a standard collector settings, the accessible model sizes are limited because of insufficient memory. To solve this problem, TEBDOL performs an explicit full garbage collection after each partial time-evolution step. This strategy keeps the allocated memory compact and allows us to use much higher tensor dimensions. On the other hand, a full garbage collection consumes processor time. Another possible strategy would be adjusting the collector settings, for example, decreasing the number of generations or decreasing the thresholds for collecting older generations.

6.3 Groveller

The groveller, provided by the SBCL package `sb-grovel`, is a standard method to create Common Lisp interface to libraries implemented in C. However, it is often easier to define required foreign structures and constants by hand. The following example illustrates such a case.

The MPI library defines a type `MPI_Comm`, which describes a communicator handle. A default instance of this type is a communicator `MPI_COMM_WORLD`, which is defined as a preprocessor macro in a C header file. The actual definitions are very different in OpenMPI and MPICH, the two standard implementations of the MPI specification. In OpenMPI, `MPI_Comm` is a pointer and `MPI_COMM_WORLD` is a pointer to a library object. The following definitions provide the required type and constant in SBCL:

```
(define-alien-type nil
  (struct ompi-communicator-t))

(define-alien-type mpi-comm
  (* (struct ompi-communicator-t)))

(define-alien-variable "ompi_mpi_comm_world"
  (struct ompi-communicator-t))

(defparameter *mpi-comm-world*
  (addr ompi-mpi-comm-world))
```

The version for MPICH is simpler. The type is an integer and the constant is just a magic value:

```
(define-alien-type mpi-comm int)
(defparameter *mpi-comm-world* #x44000000)
```

It would be great if the groveller could provide these definitions automatically in both cases. Another strategy would be to write a wrapper library in C that would call respective MPI functions. Common Lisp code would call functions from the wrapper library only. The wrapper library would encapsulate the differences between MPI implementations.

6.4 Foreign libraries

TEBDOL uses the foreign libraries BLAS, LAPACK, and MPI. There are several popular implementations of them. Their optimized versions are usually supplied by the vendor of a computer cluster. TEBDOL is being developed with OpenBLAS, which provides both BLAS and LAPACK, and OpenMPI.

While all implementations use the same API, there are major differences in their behavior. The differences usually do not affect programs in C and Fortran. However, they can

influence the SBCL runtime environment. It is thus necessary to test every implementation for incompatibilities with SBCL.

For example, IBM MPI installs a signal handler for the signal SIGSEGV. At the same time, SBCL uses SIGSEGV for its memory management. As the MPI library is loaded dynamically, the MPI handler takes over the SBCL handler. When the operating system delivers the signal, the MPI handler does not call the SBCL handler, and aborts the program instead. The best solution for this problem would be to patch the IBM MPI library. It is not easy, because the library is a proprietary software with no source code available. Another solution would be to reinstall the SBCL handler after calling the function `MPI_init`, but that requires a change to SBCL internals.

Implementation details of foreign libraries like in this example represent a difficulty in making the Common Lisp software portable in high-performance computing environments.

7. BENCHMARKS

In this section I present the scaling performance of TEBDOL. There are two notions of scaling, *strong scaling* refers to a speedup with a fixed problem size, and *weak scaling* refers to performance increase when the problem sizes grows with the number of processes.

7.1 Strong scaling

The first presented benchmark simulates time evolution of 65 particles in a lattice with 65 sites and with a hopping parameter $J = U/10$. In this case, tensor dimensions grow quickly. I was able to run all calculations with the maximal tensor dimension of $D = 2000$. The second benchmark simulates a long lattice with 513 particles and 513 sites and with a hopping parameter $J = U/25$. Due to memory requirements, I chose the maximal tensor dimension of $D = 1000$ to be able to finish the calculation even with a single MPI process.

The scaling results for both calculations are presented in Fig. 5. TEBDOL shows good strong scaling performance.

7.2 Weak scaling

The second benchmark simulates a lattice with a varying number of lattice sites L and a varying number of particles N. In each calculation $L = N$. The results in Fig. 6 represent constant scaling performance. With the increased problem size and with appropriately increased resources, the calculation time is approximately constant.

7.3 Large dimensions

The last example shows a recalculation of a single state-of-art result from Ref. [7]. Initially, atoms occupy even sites in a one-dimensional lattice and do not hop between sites. After a sudden change in lattice parameters, atoms start to move around. The system then relaxes into equilibrium due to their strong mutual interactions.

The calculation was performed in 32 MPI processes on 32 nodes. Each node was equipped with 16 CPU cores and

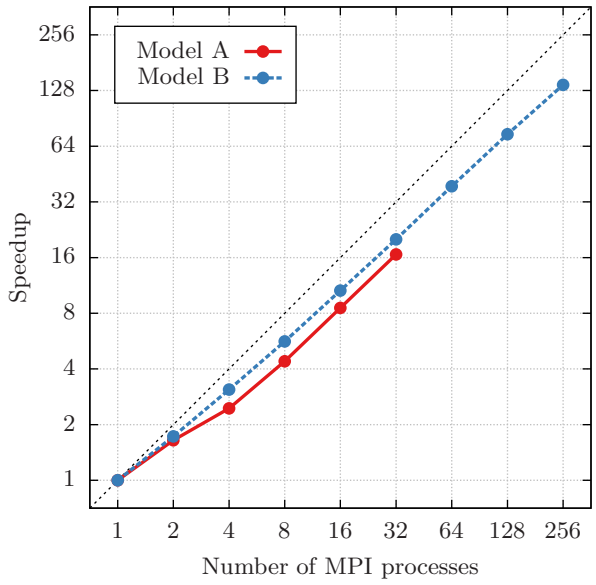


Figure 5: Strong scaling of TEBDOL for two one-dimensional models. Model A has 65 lattice sites and 65 particles, while model B has 513 sites 513 particles. Maximal allowed tensor dimension was $D = 1000$ and $D = 2000$, respectively.

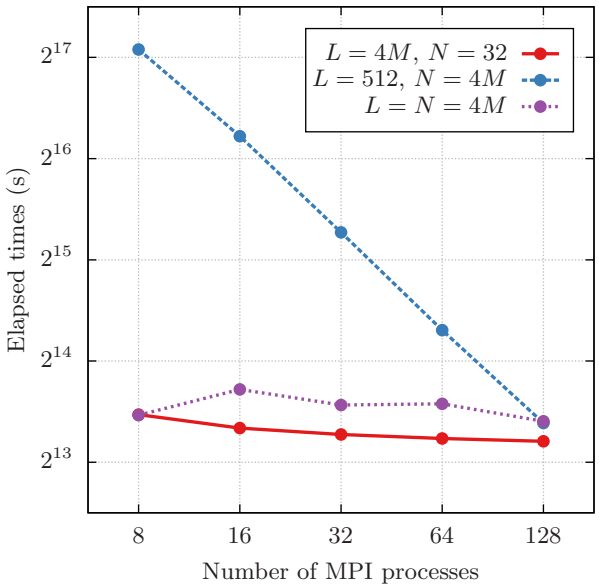
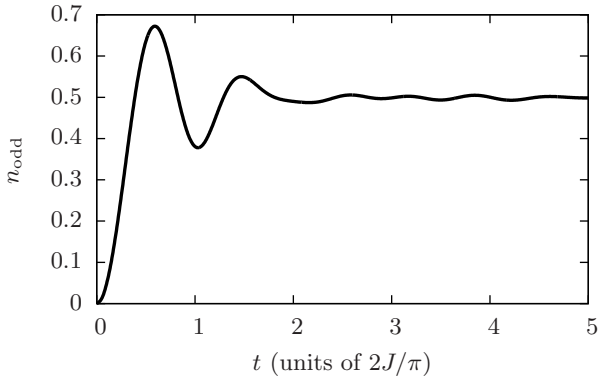


Figure 6: Weak scaling of TEBDOL for three definitions of a problem size. The number of lattice sizes L, the total number of particles N, and both parameters at the same time were increased with the number of MPI processes M. The results show approximately constant weak scaling with the number of lattice sites. Maximal allowed tensor dimension was $D = 2000$.

Figure 7: Relaxation towards equilibrium in a one-dimensional optical lattice. The calculation uses a model from Ref. [7]. There are 43 particles on a lattice with 121 sites. Particles initially occupy even sites near the lattice center. They start to move after a change of parameters at the time $t = 0$. n_{odd} is the particle density at odd sites. Maximal allowed tensor dimension was $D = 8000$.



64 GB RAM. The MPI processes consumed about 32 GB of memory at their peaks. The calculation took about 38 hours with the maximal tensor dimension of $D = 8000$. The result in Fig. 7 shows a change in the occupation of odd sites with time. The example illustrates that our implementation is competitive with other state-of-art implementations.

8. CONCLUSIONS

In this paper I discussed experience with using Common Lisp in the field of computational quantum physics. Common Lisp equipped with external numerical libraries is a suitable programming language for this domain. While there are some disadvantages with regards to existing libraries and tools that focus on C and Fortan, the overall experience has been positive. The power of language constructs in Common Lisp will be beneficial in extending the current program to higher-dimensional models.

A major improvement over the current state of things would be a better groveller. Many programs access foreign libraries. Several libraries provide a clean API on the source-code level only. A groveller should parse library headers and provide Common Lisp definitions. The tools we worked with are quite limited in this scope, and cannot provide proper definitions for OpenMPI. I am not aware of any tool that have the required capabilities. A better groveller would accelerate the development and would also provide access the non-fundamental but useful libraries like the Score-P profiling library.

Most of TEBDOL is written in ANSI Common Lisp. We use extensions specific to SBCL mainly for calling foreign functions. It should be easy to port the program to other implementations after rewritting this code to use CFFI. Other non-portable code deals with garbage collection, but it is not critical to the program operation. However, we do not plan to port the program, because SBCL suits our needs.

9. ACKNOWLEDGMENTS

I acknowledge support of the Czech Science Foundation - GAČR (Grant no. P209/15-10267S). I also acknowledge support by the IT4Innovations Centre of Excellence project (CZ.1.05/1.1.00/02.0070), funded by the European Regional Development Fund and the national budget of the Czech Republic via the Research and Development for Innovations Operational Programme, as well as Czech Ministry of Education, Youth and Sports via the project Large Research, Development and Innovations Infrastructures (LM2011033). Furthermore I acknowledge PRACE for awarding us access to resource SuperMUC based in Germany at LRZ and to resource Curie based in France at CEA.

10. REFERENCES

[1] GotoBLAS. `https://www.tacc.utexas.edu/research-development/tacc-software/gotoblas2`. (Accessed April 3, 2015).

[2] Intel Math Kernel Library. `https://software.intel.com/en-us/intel-mkl`. (Accessed April 3, 2015).

[3] ITensor. `http://itensor.org/`. (Accessed April 3, 2015).

[4] Score-P. `http://www.vi-hps.org/projects/score-p/`. (Accessed April 3, 2015).

[5] Steel Bank Common Lisp. `http://www.sbcl.org/`. (Accessed April 3, 2015).

[6] The Common Foreign Function Interface. `http://common-lisp.net/project/cffi/`. (Accessed April 3, 2015).

[7] S. Trotzky, Y.-A. Chen, A. Flesch, I. P. McCulloch, U. Schollwöck, J. Eisert, and I. Bloch. Probing the relaxation towards equilibrium in an isolated strongly correlated one-dimensional bose gas. *Nat. Phys.*, 8(4):325–330, 2012.

[8] M. Urbanek and P. Soldán. Matter-wave revival of binary mixtures in optical lattices. *Phys. Rev. A*, 90:033610, Sep 2014.

[9] G. Vidal. Efficient classical simulation of slightly entangled quantum computations. *Phys. Rev. Lett.*, 91:147902, Oct 2003.

[10] G. Vidal. Efficient simulation of one-dimensional quantum many-body systems. *Phys. Rev. Lett.*, 93:040502, Jul 2004.

First-class Global Environments in Common Lisp

Robert Strandh
University of Bordeaux
351, Cours de la Libération
Talence, France
robert.strandh@u-bordeaux1.fr

ABSTRACT

Environments are mentioned in many places in the Common Lisp standard, but the nature of such objects is not specified. For the purpose of this paper, an environment is a mapping from *names* to *meanings*. In a typical Common Lisp implementation the *global environment* is not a first-class object.

In this paper, we advocate first-class global environments, not as an extension or a modification of the Common Lisp standard, but as an implementation technique. We state several advantages in terms of bootstrapping, sandboxing, and more. We show an implementation where there is no performance penalty associated with making the environment first class. For performance purposes, the essence of the implementation relies on the environment containing *cells* (ordinary `cons` cells in our implementation) holding bindings of names to functions and global values that are likely to be heavily solicited at runtime.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, Packages*; D.3.4 [**Programming Languages**]: Processors—*Code generation, Run-time environments*

General Terms

Design, Languages

Keywords

CLOS, Common Lisp, Environment

1. INTRODUCTION

The Common Lisp standard contains many references to environments. Most of these references concern *lexical* environments at *compile time*, because they are needed in order to process forms in non-null lexical environments. The standard does not specify the nature of these objects, though in CLtL2 [5] there is a suggested protocol that is sometimes supplied in existing Common Lisp implementations.

When it comes to *global environments*, however, the standard is even more reticent. In section 3.2.1 (entitled Compiler Terminology) of the Common Lisp HyperSpec, the distinction is made between the *startup environment*, the *compilation environment*, the *evaluation environment*, and the *runtime environment*. Excluding the runtime environment, the standard allows for all the others to be identical.

In a typical Common Lisp implementation, these global environments are not first-class objects, and there is typically only one global environment available as specifically allowed by the standard. In some implementations, part of the environment is contained in other objects. For instance, it is common for a symbol object to contain a *value cell* containing the global value (if any) of the variable having the name of the symbol and/or a *function cell* containing the definition of a global function having the name of the symbol. This kind of representation is even implicitly suggested by the standard in that it sometimes uses terminology such as *value cell* and *function cell*[1], while pointing out that this terminology is "traditional".

In this paper, we argue that there are many advantages to making the global environment a first-class object. An immediate advantage is that it is then possible to distinguish between the *startup environment*, the *compilation environment* and the *evaluation environment* so that compile-time evaluations by the compiler are not visible in the startup-environment. However, as we show in this paper, there are many more advantages, such as making it easier to create a so-called *sandbox* environment, which is notoriously difficult to do in a typical Common Lisp implementation. Another significant advantage of first-class global environments is that it becomes unnecessary to use temporary package names for bootstrapping a target Common Lisp system from a host Common Lisp system.

In order for first-class global environments to be a viable alternative to the traditional implementation method, they must not incur any performance penalty, at least not at runtime. We show an implementation of first-class global environments that respects this constraint by supplying *cells* that can be thought of as the same as the traditional value

[1]See for instance the glossary entries for *cell*, *value cell*, and *function cell* in the HyperSpec.

ELS '15, April 20 - 21 2015, London, UK Copyright is held by the author.

cells and function cells, except that they are dislocated so that the are physically located in the environment object as opposed to being associated with a symbol.

2. PREVIOUS WORK

2.1 Gelernter et al

The idea of first-class environments is of course not new. Gelernter et al [1] defined a language called "Symmetric Lisp" in which the programmer is allowed to evaluate expressions with respect to a particular first-class environment. They suggest using this kind of environment as a replacement for a variety of constructs, including closures, structures, classes, and modules. The present paper does not have this kind of ambitious objective, simply because we do not know how to obtain excellent performance for all these constructs with our suggested protocol.

2.2 Miller and Rozas

Miller and Rozas [2] describe a set of extensions for the Scheme programming language. In their paper, Miller and Rozas also claim that their proposed first-class environments could serve as a basis for an object-oriented system. Like the present work, Miller and Rozas are concerned with performance, and a large part of their paper is dedicated to this aspect of their proposal.

In their paper, they show that the extensions incurs no performance penalty for code that does not use it. However, for code that uses the extension, the compiler defers accesses to the first-class environment to the *interpreter*, thereby imposing a performance penalty in code using the extension.

The basic mechanism of their proposed extension is a *special form* named `make-environment` that creates and returns an environment in which the code in the *body* of that special form is executed. The operators `lexical-reference` and `lexical-assignment` are provided to access bindings in a first-class environment.

From the examples in the paper, it is clear that their first-class environments are meant to be used at a much finer level of granularity than ours.

2.3 Queinnec and Roure

Queinnec and de Roure [3] proposed a protocol for first-class environments in the context of the Scheme programming language. Their motivation is different from ours, in that their environments are meant to be part of the user-visible interface so as to simplify sharing of various objects. However, like the present work, they are also concerned with performance, and they show how to implement their protocol without serious performance degradation.

The environments proposed by Queinnec and de Roure were clearly meant to allow for *modules* as collections of *bindings* where the bindings can be shared by other modules through the use of the new operators `export` and `import`. In contrast, the first-class environments proposed in the present paper are not meant to allow such sharing of bindings, though our proposal might allow such sharing for function and variable bindings.

The paper by Queinnec and de Roure contains a thorough survey of other work related to first-class environments that will not be repeated here.

2.4 Garret's lexicons

Ron Garret describes *lexicons*[2] which are said to be first-class global environments. However, the concept of a lexicon is very different from the concept of a first-class global environment as defined in this paper.

For one thing, a lexicon is "a mapping from symbols to bindings", which excludes a per-environment set of packages, simply because package names are not symbols, but strings.

Furthermore, a clearly stated goal of lexicons is to create a different Lisp dialect targeted to new users or to users that have prior experience with languages that are different from Lisp.

One explicit goal of lexicons is to replace Common Lisp packages, so that there is a single system-wide symbol with a particular name. In contrast, the first-class environments presented in this paper do not in any way affect the package system. It should be noted that with our first-class environments, *symbols are still unique*, i.e., for a given package P and a given symbol name N, there is at most one symbol with the name N in P; independently of the number of first-class environments in the system.

Garret discusses the use of lexicons as modules and shows examples where functions defined in one lexicon can be imported into a different lexicon. The use of the operator `use-lexicon` imports all the *bindings* of an explicitly-mentioned lexicon into the current one. In the present work, we do not emphasize the possibility of sharing bindings between first-class environments. However, since functions and global values of special variables are stored in indirections called *cells* in our environment, such sharing of bindings would also be possible in the first-class environments presented in this paper.

3. OUR TECHNIQUE

We suggest a CLOS-based *protocol* defining the set of operations on a first-class environment. This protocol contains around 40 generic functions. The details of the proposed protocol can be found in the appendix of this paper. The protocol has been implemented as part of the SICL project.[3]

Mainly, the protocol contains versions of Common Lisp environment functions such as `fboundp`, `find-class`, etc. that take an additional required `environment` argument.

For a simple example, consider the SICL implementation of the standard Common Lisp function `fboundp`:

```
(defun fboundp (name)
  (sicl-genv:fboundp
    name
```

[2]Unpublished document. A PDF version can be found here: http://www.flownet.com/ron/lisp/lexicons.pdf.
[3]See https://github.com/robert-strandh/SICL.

```
(load-time-value (sicl-genv:global-environment))))
```

In this example `sicl-genv` is the nickname for the package named `sicl-global-environment` which contains the symbols of the protocol defined in this paper. In each global environment, the function `global-environment` in that package returns the value of the environment itself. When the definition in the example above is *loaded*, either as source or from a previously compiled file, the value of the `load-time-value` form will therefore be the global environment in which the definition is loaded, thereby permanently *linking* this definition to that global environment.

In addition to these functions, the protocol contains a set of functions for accessing *cells* that in most implementations would be stored elsewhere. Thus, a binding of a function name to a function object contains an indirection in the form of a *function cell*. The same holds for the binding of a variable name (a symbol) to its *global value*. In our implementation, these cells are ordinary `cons` cells with the `car` containing the value of the binding, and the `cdr` containing `nil`. The reason for using ordinary `cons` cells is that they are already supported in any Common Lisp implementation. The only possible reason for choosing a different representation for cells would be to save one word in each cell, since the `cdr` slot in each of our `cons` cells is wasted. However, the saved space would probably be more than consumed by the space occupied by specialized inspector functionality for dealing with custom cell representations.

These cells are created as needed. The first time a reference to a function is made, the corresponding cell is created. Compiled code that refers to a global function will have the corresponding cell in its run-time environment. The cost of accessing a function at run-time is therefore no greater in our implementation than in an implementation that accesses the function through the symbol naming it, hence our claim that there is no performance penalty for accessing this information at run-time.

The SICL compiler translates a reference to a global function (say `foo`) into something similar to this code:

```
(car
  (load-time-value
    (sicl-genv:function-cell
      'foo
      (sicl-genv:global-environment+))))
```

except that what is shown as `car` is not the full Common Lisp function, because the argument is known to be a `cons` cell. When the code containing this reference is loaded, the resulting machine code will refer to a local variable containing the `cons` cell of the current global environment that is permanently assigned to holding the function definition of `foo`.

Our technique does, however, incur a performance penalty for functions such as `fdefinition` and `symbol-value` with an argument that is computed at run-time[4] compared to an implementation in which each symbol contains slots for these objects. However, even in a high-performance implementation such as SBCL, these values are *not* contained in symbol slots.

The performance penalty incurred on these functions depends on the exact representation of the environment. The representation of the environment is outside the scope of this paper, however. Here, we only consider the *protocol* for accessing it. However, it is not hard to devise a reasonable implementation. In SICL, we use a hash table for each namespace with the keys being the corresponding *names*[5] of the entities in that namespace.

While it is *possible* for the application programmer to create new global environments, it would not be a common thing to do, at least not for the applications of first-class global environments that we have considered so far. For that reason, we have not streamlined any particular technique for doing so. The difficulty is not in *creating* the environment per se, but rather in filling it with useful objects. For the purpose of bootstrapping, we currently fill environments by loading code into it from files.

4. BENEFITS OF OUR METHOD

4.1 Native compilation

The Common Lisp standards suggests that the *startup environment* and the *evaluation environment* may be different.[6] Our method allows most evaluations by the compiler to have no influence in the startup environment. It suffices to *clone* the startup environment in order to obtain the evaluation environment.

With the tradition of the startup environment and evaluation environment being identical, some evaluations by the compiler would have side effects in the startup environment. In particular, the value cells and function cells are shared. Therefore, executing code at compile time that alters the global binding of a function or a variable will also be seen in the startup environment.

As an example of code that should not be evaluated in the startup environment, consider definitions of macros that are only required for the correct compilation of some program, as well as definitions of functions that are only required for the expansion of such macros. A definition for this purpose might be wrapped in an `eval-when` form with `:compile-toplevel` as the only *situation* in which the definition should be evaluated. When the evaluation environment and the startup environment are identical, such a definition will be evaluated in the startup environment, and persist after the program has been compiled.

[4]When the argument is a constant, a suitable *compiler-macro* can turn the form into an access of the corresponding cell.

[5]Functions are named by symbols and lists; variables are named by symbols; packages are named by strings; classes are named by symbols; etc.

[6]Recall that the startup environment is the global environment as it was when the compilation was initiated, and that the evaluation environment is the global environment in which evaluations initiated by the compiler are accomplished.

4.2 Bootstrapping

For the purpose of this paper, we use the word *bootstrapping* to mean the process of building the executable of some implementation (the *target* system) by executing code in the running process of another implementation (the *host* system). The host and the target systems may be the same implementation. In this context, a *cross compiler* is a compiler that executes in the host system while generating code for the target system.

When a *host* Common Lisp system is used to bootstrap a *target* Common Lisp system, the target system needs its own definitions of many standard Common Lisp features. In particular, in order to compile code for the target system in the host system, the cross compiler needs access to the target definitions of standard Common Lisp macros, in particular the defining macros such as `defun`, `defmacro`, `defgeneric`, `defvar`, etc.

It is, of course, not an option to replace the host versions of such macros with the corresponding target versions. Doing so would almost certainly break the host system in irreparable ways. To avoid that the system might be damaged this way, many Common Lisp systems have a feature called *package locks*[7] which prevents the redefinition of standard Common Lisp functions, macros, etc.

To deal with the problem of bootstrapping, some systems, in particular SBCL, replace the standard package names by some other names for target code, typically derived from the standard names in some systematic way [4]. Using different package names guarantees that there is no clash between a host package name and the corresponding target package name. However, using non-standard package names also means that the text of the source code for the target will not correspond to the target code that ends up in the final system.

As an alternative to renaming packages, first-class global environments represent an elegant solution to the bootstrapping problem. In a system that already supports first-class global environments, creating a new such environment in which the target definitions are allowed to replace standard Common Lisp definitions is of course very simple. But even in a host system that does not a priori support first-class global environments, it is not very difficult to create such environments.

Making the cross compiler access such a first-class global environment is just a matter of structuring its environment-lookup functions so that they do not directly use standard Common Lisp functions such as `fboundp` or `fdefinition`, and instead use the generic functions of the first-class global environment protocol.

4.3 Sandboxing

It is notoriously hard to create a so-called *sandbox environment* for Common Lisp, i.e., an environment that contains a "safe" subset of the full language. A typical use case would be to provide a Read-Eval-Print Loop accessible through a web interface for educational purposes. Such a sandbox environment is hard to achieve because functions such as `eval` and `compile` would have to be removed so that the environment could not be destroyed by a careless user. However, these functions are typically used by parts of the system. For example, CLOS might need the compiler in order to generate dispatch code.

The root of the problem is that in Common Lisp there is always a way for the user of a Read-Eval-Print Loop to access every global function in the system, including the compiler. While it might be easy to remove functions that may render the system unusable *directly* such as functions for opening and deleting files, it is generally not possible to remove the compiler, since it is used at run-time to evaluate expressions and in many systems in order to create functions for generic dispatch. With access to the compiler, a user can potentially create and execute code for any purpose.

Using first-class global environments solves this problem in an elegant way. It suffices to provide a restricted environment in which there is no binding from the names `eval` and `compile` to the corresponding functions. These functions can still be available in some other environment for use by the system itself.

4.4 Multiple package versions

When running multiple applications in the same Common Lisp process, there can easily be conflicts between different versions of the same package. First-class global environments can alleviate this problem by having different global environments for the different applications causing the conflict.

Suppose, for instance, that applications A and B both require some Common Lisp package P, but that P exists in different *versions*. Suppose also that A and B require different such versions of P. Since the Common Lisp standard has no provisions for multiple versions of a package, it becomes difficult to provide both A and B in the same Common Lisp process.

Using first-class global environments as proposed in this paper, two different global environments can be created for building A and B. These two environments would differ in that the name P would refer to different versions of the package P.

4.5 Separate environment for each application

Taking the idea of Section 4.4 even further, it is sometimes desirable for a large application to use a large number of packages that are specific to that application. In such a situation, it is advantageous to build the application in a separate global environment, so that the application-specific packages exist only in that environment. The main entry point(s) of the application can then be made available in other environments without making its packages available.

Using separate first-class global environments for this purpose would also eliminate the problem of choosing package names for an application that are guaranteed not to conflict

[7] The name of this feature is misleading. While it does make sure that the protected package is not modified, it also makes sure that functions, macros, etc., with names in the package are not redefined. Such redefinitions do not alter the package itself, of course.

with names of packages in other applications that some user might simultaneously want to install.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have advocated first-class global environments as a way of implementing the global environments mentioned in the HyperSpec. We have seen that this technique has several advantages in terms of flexibility of the system, and that it greatly simplifies certain difficult problems such as bootstrapping and sandboxing.

An interesting extension of our technique would be to consider *environment inheritance*.[8] For example, an environment providing the standard bindings of the Common Lisp language could be divided into an immutable part and a mutable part. The mutable part would then contain features that can be modified by the user, such as the generic function `print-object` or the variable `*print-base*`, and it would inherit from the immutable part. With this feature, it would only be necessary to clone the mutable part in order to create the evaluation environment from the startup environment as suggested in Section 4.1.

We also think that first-class global environments could be an excellent basis for a multi-user Common Lisp system.

In such a system, each user would have an initial, private, environment. That environment would contain the standard Common Lisp functionality. Most standard Common Lisp functions would be shared between all users. Some functions, such as `print-object` or `initialize-instance` would not be shared, so as to allow individual users to add methods to them without affecting other users.

Furthermore, functionality that could destroy the integrity of the system, such as access to raw memory, would be accessible in an environment reserved for system maintenance. This environment would not be accessible to ordinary users.

6. ACKNOWLEDGMENTS

We would like to thank Alastair Bridgewater, David Murray, Robert Smith, Nicolas Hafner, and Bart Botta for providing valuable feedback on early versions of this paper.

APPENDIX

A. PROTOCOL

In this appendix we present the generic functions making up the protocol for our first-class global environments. The definitions here should be considered *preliminary*, because there are some aspects of this protocol that need further consideration. As an example, consider the function `function-lambda-list`. We have not made up our minds as to whether this function should be part of the protocol, or just a function to be applied to function objects.

In order for our definitions to fit in a column, we have abbreviated "Generic Function" as "GF".

`+global-environment+` [*Constant*]

In each global environment e, the value of this constant variable is e.

`global-environment` [*Function*]

In each global environment e, this function takes no arguments and returns e.

`fboundp` *fname env* [*GF*]

This generic function is a generic version of the Common Lisp function `cl:fboundp`.

It returns true if *fname* has a definition in *env* as an ordinary function, a generic function, a macro, or a special operator.

`fmakunbound` *fname env* [*GF*]

This generic function is a generic version of the Common Lisp function `cl:fmakunbound`.

Makes *fname* unbound in the function namespace of *env*.

If *fname* already has a definition in *env* as an ordinary function, as a generic function, as a macro, or as a special operator, then that definition is lost.

If *fname* has a `setf` expander associated with it, then that `setf` expander is lost.

`special-operator` *fname env* [*GF*]

If *fname* has a definition as a special operator in *env*, then that definition is returned. The definition is the object that was used as an argument to `(setf special-operator)`. The exact nature of this object is not specified, other than that it can not be `nil`. If *fname* does not have a definition as a special operator in *env*, then `nil` is returned.

`(setf special-operator)` *new-def fname env* [*GF*]

Set the definition of *fname* to be a special operator. The exact nature of *new-def* is not specified, except that a value of `nil` means that *fname* no longer has a definition as a special operator in *env*.

If a value other than `nil` is given for *new-def*, and *fname* already has a definition as an ordinary function, as a generic function, or as a macro, then an error is signaled. As a consequence, if it is desirable for *fname* to have a definition both as a special operator and as a macro, then the definition as a special operator should be set first.

`fdefinition` *fname env* [*GF*]

This generic function is a generic version of the Common Lisp function `cl:fdefinition`.

If *fname* has a definition in the function namespace of *env* (i.e., if `fboundp` returns true), then a call to this function succeeds. Otherwise an error of type `undefined-function` is signaled.

If *fname* is defined as an ordinary function or a generic function, then a call to this function returns the associated function object.

If *fname* is defined as a macro, then a list of the form `(cl:macro-function` *function*`)` is returned, where *function*

[8] We mean *inheritance* not in the sense of subclassing, but rather as used in section 3.2.1 of the HyperSpec.

is the macro expansion function associated with the macro.

If *fname* is defined as a special operator, then a list of the form `(cl:special` *object*`)` is returned, where the nature of *object* is currently not specified.

`(setf fdefinition)` *new-def fname env* [*GF*]
This generic function is a generic version of the Common Lisp function `cl:fdefinition`.

new-def must be an ordinary function or a generic function. If *fname* already names a function or a macro, then the previous definition is lost. If *fname* already names a special operator, then an error is signaled.

If *fname* is a symbol and it has an associated `setf` expander, then that `setf` expander is preserved.

`macro-function` *symbol env* [*GF*]
This generic function is a generic version of the Common Lisp function `cl:macro-function`.

If *symbol* has a definition as a macro in *env*, then the corresponding macro expansion function is returned.

If *symbol* has no definition in the function namespace of *env*, or if the definition is not a macro, then this function returns `nil`.

`(setf macro-function)` *new-def symbol env* [*GF*]
This generic function is a generic version of the Common Lisp function `(setf cl:macro-function)`.

new-def must be a macro expansion function or `nil`. A call to this function then always succeeds. A value of `nil` means that the *symbol* no longer has a macro function associated with it. If *symbol* already names a macro or a function, then the previous definition is lost. If *symbol* already names a special operator, that definition is kept.

If *symbol* already names a function, then any proclamation of the type of that function is lost. In other words, if at some later point *symbol* is again defined as a function, its proclaimed type will be `t`.

If *symbol* already names a function, then any `inline` or `notinline` proclamation of the type of that function is lost. In other words, if at some later point *symbol* is again defined as a function, its proclaimed inline information will be `nil`.

If *fname* is a symbol and it has an associated `setf` expander, then that `setf` expander is preserved.

`compiler-macro-function` *fname env* [*GF*]
This generic function is a generic version of the Common Lisp function `cl:compiler-macro-function`.

If *fname* has a definition as a compiler macro in *env*, then the corresponding compiler macro function is returned.

If *fname* has no definition as a compiler macro in *env*, then this function returns `nil`.

`(setf compiler-macro-function)` *new-def fname env* [*GF*]
This generic function is a generic version of the Common Lisp function `(setf cl:compiler-macro-function)`.

new-def can be a compiler macro function or `nil`. When it is a compiler macro function, then it establishes *new-def* as a compiler macro for *fname* and any existing definition is lost. A value of `nil` means that *fname* no longer has a compiler macro associated with it in *env*.

`function-type` *fname env* [*GF*]
This generic function returns the proclaimed type of the function associated with *fname* in *env*.

If *fname* is not associated with an ordinary function or a generic function in *env*, then an error is signaled.

If *fname* is associated with an ordinary function or a generic function in *env*, but no type proclamation for that function has been made, then this generic function returns `t`.

`(setf function-type)` *new-type fname env* [*GF*]
This generic function is used to set the proclaimed type of the function associated with *fname* in *env* to *new-type*.

If *fname* is associated with a macro or a special operator in *env*, then an error is signaled.

`function-inline` *fname env* [*GF*]
This generic function returns the proclaimed inline information of the function associated with *fname* in *env*.

If *fname* is not associated with an ordinary function or a generic function in *env*, then an error is signaled.

If *fname* is associated with an ordinary function or a generic function in *env*, then the return value of this function is either `nil`, `inline`, or `notinline`. If no inline proclamation has been made, then this generic function returns `nil`.

`(setf function-inline)` *new-inline fname env* [*GF*]
This generic function is used to set the proclaimed inline information of the function associated with *fname* in *env* to *new-inline*.

new-inline must have one of the values `nil`, `inline`, or `notinline`.

If *fname* is not associated with an ordinary function or a generic function in *env*, then an error is signaled.

`function-cell` *fname env* [*GF*]
A call to this function always succeeds. It returns a `cons` cell, in which the `car` always holds the current definition of the function named *fname*. When *fname* has no definition as a function, the `car` of this cell will contain a function that, when called, signals an error of type `undefined-function`. The return value of this function is always the same (in the sense of `eq`) when it is passed the same (in the sense of `equal`) function name and the same (in the sense of `eq`) environment.

`function-unbound` *fname env* [*GF*]
A call to this function always succeeds. It returns a func-

tion that, when called, signals an error of type `undefined-function`. When *fname* has no definition as a function, the return value of this function is the contents of the `cons` cell returned by `function-cell`. The return value of this function is always the same (in the sense of `eq`) when it is passed the same (in the sense of `equal`) function name and the same (in the sense of `eq`) environment. Client code can use the return value of this function to determine whether *fname* is unbound and if so signal an error when an attempt is made to evaluate the form `(function` *fname*`)`.

`function-lambda-list` *fname env* [*GF*]

This function returns two values. The first value is an ordinary lambda list, or `nil` if no lambda list has been defined for *fname*. The second value is true if and only if a lambda list has been defined for *fname*.

`boundp` *symbol env* [*GF*]

It returns true if *symbol* has a definition in *env* as a constant variable, as a special variable, or as a symbol macro. Otherwise, it returns `nil`.

`constant-variable` *symbol env* [*GF*]

This function returns the value of the constant variable *symbol*.

If *symbol* does not have a definition as a constant variable, then an error is signaled.

`(setf constant-variable)` *value symbol env* [*GF*]

This function is used in order to define *symbol* as a constant variable in *env*, with *value* as its value.

If *symbol* already has a definition as a special variable or as a symbol macro in *env*, then an error is signaled.

If *symbol* already has a definition as a constant variable, and its current value is not `eql` to *value*, then an error is signaled.

`special-variable` *symbol env* [*GF*]

This function returns two values. The first value is the value of *symbol* as a special variable in *env*, or `nil` if *symbol* does not have a value as a special variable in *env*. The second value is true if *symbol* does have a value as a special variable in *env* and `nil` otherwise.

Notice that the symbol can have a value even though this function returns `nil` and `nil`. The first such case is when the symbol has a value as a constant variable in *env*. The second case is when the symbol was assigned a value using `(setf symbol-value)` without declaring the variable as `special`.

`(setf special-variable)` *value symbol env init-p* [*GF*]

This function is used in order to define *symbol* as a special variable in *env*.

If *symbol* already has a definition as a constant variable or as a symbol macro in *env*, then an error is signaled. Otherwise, *symbol* is defined as a special variable in *env*.

If *symbol* already has a definition as a special variable, and *init-p* is `nil`, then this function has no effect. The current value is not altered, or if *symbol* is currently unbound, then it remains unbound.

If *init-p* is true, then *value* becomes the new value of the special variable *symbol*.

`symbol-macro` *symbol env* [*GF*]

This function returns two values. The first value is a macro expansion function associated with the symbol macro named by *symbol*, or `nil` if *symbol* does not have a definition as a symbol macro. The second value is the form that *symbol* expands to as a macro, or `nil` if symbol does not have a definition as a symbol macro.

It is guaranteed that the same (in the sense of `eq`) function is returned by two consecutive calls to this function with the same symbol as the first argument, as long as the definition of *symbol* does not change.

`(setf symbol-macro)` *expansion symbol env* [*GF*]

This function is used in order to define *symbol* as a symbol macro with the given *expansion* in *env*.

If *symbol* already has a definition as a constant variable, or as a special variable, then an error of type `program-error` is signaled.

`variable-type` *symbol env* [*GF*]

This generic function returns the proclaimed type of the variable associated with *symbol* in *env*.

If *symbol* has a definition as a constant variable in *env*, then the result of calling `type-of` on its value is returned.

If *symbol* does not have a definition as a constant variable in *env*, and no previous type proclamation has been made for *symbol* in *env*, then this function returns `t`.

`(setf variable-type)` *new-type symbol env* [*GF*]

This generic function is used to set the proclaimed type of the variable associated with *symbol* in *env*.

If *symbol* has a definition as a constant variable in *env*, then an error is signaled.

It is meaningful to set the proclaimed type even if *symbol* has not previously been defined as a special variable or as a symbol macro, because it is meaningful to use `(setf symbol-value)` on such a symbol.

Recall that the HyperSpec defines the meaning of proclaiming the type of a symbol macro. Therefore, it is meaningful to call this function when *symbol* has a definition as a symbol macro in *env*.

`variable-cell` *symbol env* [*GF*]

A call to this function always succeeds. It returns a `cons` cell, in which the `car` always holds the current definition of the variable named *symbol*. When *symbol* has no definition as a variable, the `car` of this cell will contain an object that indicates that the variable is unbound. This object is the return value of the function `variable-unbound`. The return value of this function is always the same (in the sense of `eq`) when it is passed the same symbol and the same environment.

`variable-unbound` *symbol env* [*GF*]

A call to this function always succeeds. It returns an object that indicates that the variable is unbound. The `cons` cell returned by the function `variable-cell` contains this object whenever the variable named *symbol* is unbound. The return value of this function is always the same (in the sense of `eq`) when it is passed the same symbol and the same environment. Client code can use the return value of this function to determine whether *symbol* is unbound.

`find-class` *symbol env* [*GF*]

This generic function is a generic version of the Common Lisp function `cl:find-class`.

If *symbol* has a definition as a class in *env*, then that class metaobject is returned. Otherwise `nil` is returned.

`(setf find-class)` *new-class symbol env* [*GF*]

This generic function is a generic version of the Common Lisp function `(setf cl:find-class)`.

This function is used in order to associate a class with a class name in *env*.

If *new-class* is a class metaobject, then that class metaobject is associated with the name *symbol* in *env*. If *symbol* already names a class in *env* than that association is lost.

If *new-class* is `nil`, then *symbol* is no longer associated with a class in *env*.

If *new-class* is neither a class metaobject nor `nil`, then an error of type `type-error` is signaled.

`setf-expander` *symbol env* [*GF*]

This generic function returns the `setf` expander associated with *symbol* in *env*. If *symbol* is not associated with any `setf` expander in *env*, then `nil` is returned.

`(setf setf-expander)` *new-expander symbol env* [*GF*]

This generic function is used to set the `setf` expander associated with *symbol* in *env*.

If *symbol* is not associated with an ordinary function, a generic function, or a macro in *env*, then an error is signaled.

If there is already a `setf` expander associated with *symbol* in *env*, then the old `setf` expander is lost.

If a value of `nil` is given for *new-expander*, then any current `setf` expander associated with *symbol* is removed. In this case, no error is signaled, even if *symbol* is not associated with any ordinary function, generic function, or macro in *env*.

`default-setf-expander` *env* [*GF*]

This generic function returns the default `setf` expander, to be used when the function `setf-expander` returns `nil`. This function always returns a valid `setf` expander.

`(setf default-setf-expander)` *new-expander env* [*GF*]

This generic function is used to set the default `setf` expander in `env`.

`type-expander` *symbol env* [*GF*]

This generic function returns the type expander associated with *symbol* in `env`. If *symbol* is not associated with any type expander in `env`, then `nil` is returned.

`(setf type-expander)` *new-expander symbol env* [*GF*]

This generic function is used to set the type expander associated with *symbol* in `env`.

If there is already a type expander associated with *symbol* in `env`, then the old type expander is lost.

`find-package` *name env* [*GF*]

Return the package with the name or the nickname *name* in the environment *env*. If there is no package with that name in *env*, then return `nil`. Contrary to the standard Common Lisp function `cl:find-package`, for this function, *name* must be a string.

`package-name` *package env* [*GF*]

Return the string that names *package* in *env*. If *package* is not associated with any name in *env*, then `nil` is returned. Contrary to the standard Common Lisp function `cl:package-name`, for this function, *package* must be a package object.

`(setf package-name)` *new-name package env* [*GF*]

Make the string *new-name* the new name of *package* in *env*. If *new-name* is `nil`, then *package* no longer has a name in *env*.

`package-nicknames` *package env* [*GF*]

Return a list of the strings that are nicknames of *package* in *env*. Contrary to the standard Common Lisp function `cl:package-nicknames`, for this function, *package* must be a package object.

`(setf package-nicknames)` *new-names package env* [*GF*]

Associate the strings in the list *new-names* as nicknames of *package* in *env*.

B. REFERENCES

[1] D. Gelernter, S. Jagannathan, and T. London. Environments as first class objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 98–110, New York, NY, USA, 1987. ACM.

[2] J. S. Miller and G. J. Rozas. Free variables and first-class environments. *Lisp and Symbolic Computation*, 4(2):107–141, Mar. 1991.

[3] C. Queinnec and D. de Roure. Sharing code through first-class environments. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming*, ICFP '96, pages 251–261, New York, NY, USA, 1996. ACM.

[4] C. Rhodes. Self-sustaining systems. chapter SBCL: A Sanely-Bootstrappable Common Lisp, pages 74–86. Springer-Verlag, Berlin, Heidelberg, 2008.

[5] G. L. Steele, Jr. *Common LISP: The Language (2nd Ed.)*. Digital Press, Newton, MA, USA, 1990.

Demonstrations

Woo: a fast HTTP server for Common Lisp

Eitaro Fukamachi
Somewrite Co., Ltd.
Tokyo, Japan
e.arrows@gmail.com

1. INTRODUCTION

HTTP (Hypertext Transfer Protocol) is a successful protocol that transfers hypertext requests and information between servers and browsers. It was originally intended for the World Wide Web, however, these days it is commonly used for activities that do more than just send HTML documents between servers.

In the context of increasing information exchange, the importance of HTTP is raising with each passing day. As the scope of HTTP gets wider, more robust servers becomes a necessity. One such server is *Woo*.

Woo is a high-performance HTTP server written in Common Lisp. It is designed to be fast and to be able to handle vast amounts of simultaneous connections.

2. HOW FAST?

Figure 1 is a benchmark graph of several HTTP servers.

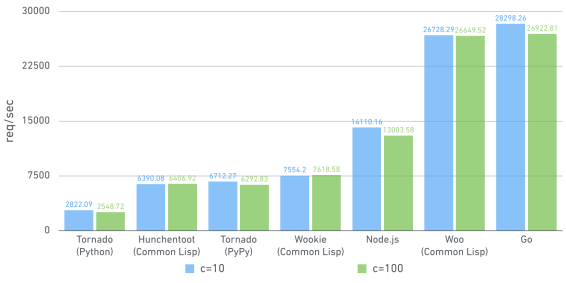


Figure 1: Benchmark

The y-axis represents a number of requests processed in 1 second when responding with a "Hello, World" to every client, and the variable "c" on x-axis tells the number of simultaneous connections the server is holding at the moment.

Among the HTTP servers, *Woo* stands second, only next to *Go* in performance. It's approximately 3.5 times faster than Wookie[1] and 4 times faster than Hunchentoot[2].

The graph indicates that not only Woo can respond faster than many other HTTP servers but also that the performance hardly deteriorates even when the number of simultaneous connections increase.

See `https://github.com/fukamachi/woo#benchmark` for the detail.

3. WHY FAST?

3.1 Better architecture

Roughly speaking, all HTTP servers can be classified into 2 groups by their architecture – "*Multi-process I/O model*" and "*Evented I/O model*".

3.1.1 Multi-process I/O

"Multi-process I/O model" is an architecture that uses multiple OS processes (or threads) to manage HTTP clients (Figure 2). Each process handles 1 client at a time. Although this model is a simple and traditional one, its performance could drop massively when many client requests arrive at the server simultaneously, because each process will be blocked during the process of reading an HTTP request and sending acknowledging it with a response.

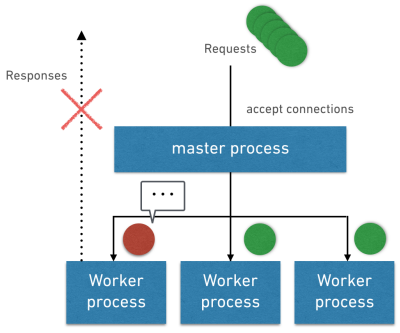


Figure 2: Slow client problem in multi-process I/O model

Hunchentoot, which was also written in Common Lisp, has this problem. It spawns a thread for each request, but there is a limit on the thread count, which means, there's a restriction on the number of client requests the Hunchentoot server can handle at a time.

The default limit is 100. Such a number could be easily exceeded if there are a lot of slow clients (such as 3G smartphones). If the web application uses interactive connections like WebSocket, the number of connected clients would increase. Besides that, Hunchentoot is vulnerable to DoS attacks like Slowloris[3]. Although the limit can be raised to 1,000,000, we also naturally have a limit on the number of CPUs. The performance would decrease quickly when the thread count is higher than a number of processors.

3.1.2 *Evented I/O*

"Evented I/O model" is another choice for handling a multitude of client request simultaneously (Figure 3). Each process has an "event-loop" which allows asynchronous I/O. Therefore, this kind of HTTP servers can handle more clients at the same time than multi-process I/O model. As clients don't block each other, the server performance doesn't go down much even when many requests are sent by numerous clients at the same time. Slowloris attack isn't an issue with this model.

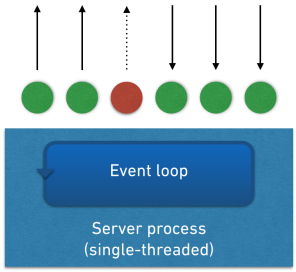


Figure 3: Evented I/O model

3.2 Fast HTTP parsing

As HTTP parsing is executed for every request, it can result in a bottleneck situation, especially when the HTTP server is having many parse requests in queue. Woo uses fast-http[4] for HTTP request parsing. It's 5 times faster than http-parser[5], a C equivalent used in Node.js.

Both *fast-http* and *http-parser* don't buffer data and parse HTTP requests asynchronously. The difficulty here is that an HTTP chunk that a parser gets could end in the middle. For handling such a case, most asynchronous HTTP parsers have an internal state for tracking where it is reading.

Common HTTP parser like *http-parser* looks like this:

```
(do ((i 0 (1+ i)))
    ((= i end))
  (let ((byte (aref data i)))
    (case (parser-state parser) ;; <- executed for every char
      (:parse-method
         (cond
           ((= byte (char-code #\G))
            (setf (parser-state parser) :parse-method-get))
           ((= byte (char-code #\P))
            (setf (parser-state parser) :parse-method-post))
           ...))
      (:parse-uri ...) ...)))
```

It is a character-level state machine. It's simple and easy to implement, however, if the performance really matters, the model would be worse because the "case" condition is executed for every character. The *fast-http* also has the state; it saves it for each line and not for each character. It has a big advantage of reducing the condition matching though it would backtrack when a chunk ends in the middle of a line, because most HTTP request-lines and headers arrive in a single packet anyway.

The speed of *fast-http* is a result of not just its architecture, but also of its many optimization techniques such as – less memory allocations, choosing the right data types and type declarations. Common Lisp programs can sometimes be faster than C programs if they have been written in an optimized manner.

3.3 The libev event library

The libev[6] is a wrapper of Linux's epoll, BSD's kqueue, POSIX select and poll. It enables us to read HTTP requests and send responses asynchronously. It is thin and fast when compared to other similar libraries such as – libevent2 and libuv. Though its Windows support is poor, it's not a serious problem since few people want to run an HTTP server on Windows in production environment. When people want to run their web applications on Windows, they can use others like Hunchentoot via Clack[7] without any changes since Woo is compatible with Clack.

When I mention the use of libev in Woo, some people would ask "is it fast because of libev, a C library?". I would say that it's not the point. Since libev is only a thin wrapper of OS system calls, it doesn't mean that its speed is due to the C library. The point is that choosing the right event library sometimes would result in great differences of speed.

APPENDIX

A. REFERENCES

1. http://wookie.beeets.com
2. http://weitz.de/hunchentoot/
3. http://ha.ckers.org/slowloris/
4. https://github.com/fukamachi/fast-http
5. https://github.com/joyent/http-parser
6. http://software.schmorp.de/pkg/libev.html
7. http://clacklisp.org

B. RESOURCES

1. https://github.com/fukamachi/woo
2. http://www.slideshare.net/fukamachi/woo-writing-a-fast-web-server
3. http://www.slideshare.net/fukamachi/writing-a-fast-http-parser

Clasp - A Common Lisp that Interoperates with C++ and Uses the LLVM Backend

Christian E. Schafmeister
Chemistry Department
Temple University
1901 N. 13th Street
Philadelphia, PA
meister@temple.edu

ABSTRACT

Clasp is an implementation of Common Lisp that interoperates with C++ and uses LLVM as its backend. It is available at `github.com/drmeister/clasp`. The goal of Clasp is to become a performant Common Lisp that can use C++ libraries and interoperate with LLVM-based tools and languages. The first sophisticated C++ library with which Clasp interoperates is the Clang C/C++ compiler front end. Using the Clang library, Common Lisp programs can be written that parse and carry out static analysis and automatic refactoring of C/C++ code. This facility is used to automatically analyze the Clasp C++ source code and construct an interface to the Memory Pool System garbage collector. It could also be used to generate automatically Foreign Function Interfaces to C/C++ libraries for use by other Common Lisp implementations.

Categories and Subject Descriptors

D.2.12 [**Software and its engineering**]: Interoperability; D.3.4 [**Software and Programming Languages**]: Incremental compilers

Keywords

Common Lisp, LLVM, C++, interoperation

1. INTRODUCTION

C++ is a popular, multi-paradigm, general-purpose language. In order to support sophisticated C++ language features (classes, methods, overloading, namespaces, exception handling, constructors/destructors, etc.), the syntax and application binary interface (ABI) of C++ has grown to be quite complex and difficult for other programming languages to interoperate with. In past decades, many software libraries were written in C++ and it would be valuable to make these libraries available to Common Lisp.

2. PREVIOUS WORK

Clasp is inspired by, and borrows code from, Embedded Common Lisp (ECL)[1], a Common Lisp implementation that is both written in and interoperates with C.

3. THE CLASP SOURCE CODE

Clasp consists of about 100,000 logical source lines of C++ code, 50,000 logical source lines of C++ header code, and 33,000 logical source lines of Common Lisp. About 26,000 lines of the Clasp Common Lisp source code are derived from the ECL Common Lisp source code. About 10% of the Clasp C++ code is translated from ECL C code.

4. AUTOMATED ANALYSIS OF CLASP C++ CODE FOR COMPACTING GC

Clasp makes it easy for Common Lisp to interoperate with complex C++ libraries and it uses this capability to integrate a sophisticated compacting garbage collector within Clasp. In normal operation, garbage collection is carried out in Clasp by the compacting Ravenbrook Memory Pool System (MPS) garbage collector library[2]. MPS is a copying garbage collector that treats pointers conservatively on the stack and precisely on the heap. MPS continuously moves objects and compacts memory, however, moving of objects in memory is challenging to reconcile with C++ which provides access to C-style pointers and pointer arithmetic. In order to allow the MPS library to work with Clasp's C++ core, every pointer to every object that will move in memory needs to be updated whenever that object is moved. Clasp fully automates the identification of C++ pointers that need to be updated by MPS, using a static analyzer written in Clasp Common Lisp. The static analyzer uses the Clang AST (Abstract Syntax Tree) and ASTMatcher C++ libraries. The static analyzer uses the Clang C++ compiler front end to parse the 173 C++ source files of Clasp and uses the Clang ASTMatcher library to search the C++ Abstract Syntax Tree in order to identify every class and global pointer that needs to be managed by MPS. It generates about 18,000 lines of C++ code that provides a functional interface to the MPS library to update all of these C++ pointers every time the MPS library carries out garbage collection. The static analyzer generates C++ code that updates pointers for 295 C++ classes and 2,625 global variables that represent Common Lisp types and symbols.

5. EXPOSING C++ TO COMMON LISP

Clasp was designed to ease interoperation of Common Lisp with foreign C++ libraries. To facilitate interoperation, Clasp incorporates a C++ template library (called *clbind*) that allows C++ library functions and classes to be exposed within Clasp by binding Common Lisp symbols to C++ functions, classes, and enumerated types. This approach is very different from the typical Foreign Function Interface (FFI) approach found in other Common Lisp implementations. Binding a C++ function to a Common Lisp symbol requires one C++ function call at Clasp startup, providing the name of the global Common Lisp symbol to bind and a pointer to the C++ function to bind to it. A C++ wrapper function that performs argument and return value conversions is automatically constructed by the C++ template library. The programmer can add additional value conversion functions. C++ pointer ownership is controlled by additional template arguments to `def`. The *clbind* library is based on the *boost::python*[3] and *luabind*[4] C++ template libraries.

To illustrate how a C++ class, constructor, method and function are exposed to Clasp Common Lisp, within the package `VEC`, the following example is provided.

```
// Exposing a C++ class and function to Clasp
#include <stdio.h>
#include "clasp/clasp.h"
class Vec2 {
public:
  double x, y;
  Vec2(double ax, double ay) : x(ax), y(ay) {};
  double dotProduct(const Vec2& o) {
    return this->x*o.x+this->y*o.y;
  };
};
void printVec(const std::string& s, const Vec2& v) {
  std::cout <<s<<"("<<v.x<<","<<v.y<<")";
}
extern "C" {
  void CLASP_MAIN() {
    using namespace clbind;
    package("VEC") [
      class_<Vec2>("vec2",no_default_constructor)
      .  def_constructor("make-vec2"
               ,constructor<double,double>())
      .  def("dot-product",&Vec2::dotProduct)
      ,def("print-vec2",&printVec)
    ];
  }
}
```

A sample Clasp Common Lisp session that uses the exposed C++ class and function is shown below:

```
>;;; A sample Common Lisp session.
>;;; "els.bundle" is the library built
>;;; from the source file above.
> (load "els.bundle")
T
> (defvar *a* (vec:make-vec2 1.0 2.0))
*A*
> (defvar *b* (vec:make-vec2 4.0 5.0))
*B*
> (vec:print-vec2 "a" *a*)
a(1,2)
> (vec:dot-product *a* *b*)
14
```

Notice how two Vec2 GC managed instances are created. The *a* object is printed and then the dot-product is calculated between the two vectors. An interface constructed this way between Clasp and a C++ library does not require either Clasp or the C++ library to be recompiled.

6. C++ RAII AND NON-LOCAL EXITS IN COMMON LISP

C++ code makes heavy use of a technique called "Resource Acquisition Is Initialization" (RAII). RAII requires that stack unwinding be accompanied by the ordered invocation of C++ destructors. To support interoperation with C++ and RAII, C++ exception handling is used within Clasp to achieve stack-unwinding and to implement the Common Lisp special operators: `go`, `return-from`, and `throw`. C++ exception handling is "zero runtime cost" when it is not invoked but can be quite expensive when used. For this reason the Clasp compiler uses exception handling only when necessary.

7. CONCLUSIONS AND FUTURE WORK

Towards the goal of developing Clasp as a performant Common Lisp compiler, the Cleavir[5] compiler, which is part of the SICL Common Lisp implementation developed by Robert Strandh, has been incorporated into Clasp. We are currently extending Cleavir/Clasp to generate source location debugging information (as DWARF metadata) and to incorporate optimizations that enhance performance of the generated code. We are also working on utilizing immediate tagged pointers for `fixnum`, `character`, and `cons` Common Lisp types and garbage collection of LLVM generated code.

8. LICENSE

Clasp is currently licensed under the GNU Library General Public License version 2.

9. ACKNOWLEDGMENTS

Thanks to Robert Strandh for providing Cleavir and for many fruitful discussions.

10. REFERENCES

[1] *Attardi, G. (1994)* The embeddable Common Lisp. In Papers of the fourth international conference on LISP users and vendors (LUV '94). ACM, New York, NY, USA, 30-41. `http://doi.acm.org/10.1145/224139.1379849`

[2] *Richard Brooksby. (2002)* The Memory Pool System: Thirty person-years of memory management development goes Open Source. ISMM02.

[3] *Abrahams, D.; Grosse-Kunstleve, R.W. (2003)* Building Hybrid Systems with Boost.Python. C/C++ Users Journal

[4] http://www.rasterbar.com/products/luabind.html

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

Colophon

The papers collected here were formatted according to the style guidelines for ACM Special Interest Group Proceedings. These proceedings were typeset using XeLaTeX, including the typeset papers using the pdfpages package. The typefaces used for this collection are the Linux Libertine and Linux Biolinum typefaces from the Libertine Open Fonts Project, and the `Computer Modern Teletype` typeface originally designed by Donald Knuth. The front cover image is a photograph of the Ben Pimlott Building, while the back image is the main entrance to the Richard Hoggart Building at Goldsmiths.

8th European Lisp Symposium, April 20-21 2015, Goldsmiths, University of London