Programming Languages Final Project: "Design of a Functional Image Library"

Elizabeth Crowdus and Maxine King

Spring 2019

1 High-Level Overview

The 2010 paper 'Design of a Functional Image Library," details updates made to the image library in the functional programming language Racket [2]. The image library provides the infrastructure for introductory computer science students to program images.

The improvements to the image library focused on improving the experience of introductory-level computer science students who use the library. Specifically, these updates were intended to increase compilation speed, to make images act more like other values in Racket, and to make using the library more intuitive for beginners. This version of the image library increased the speed of checking whether two images image are the same, also known as checking image value equality. It also added support for rotation and reflection of images, made the process of overlapping images more intuitive, and adjusted how images were stored internally.

These changes aligned the library more closely with the philosophy of the Racket language as a whole. Racket is a functional language, which means that programs in Racket consist primarily of a series of function definitions, as opposed to a series of commands (which is the usual setup in languages like Python). Additionally, all values in Racket are un-changeable, or immutable, by default. In accordance with these principles, the image library of the paper treats images as standalone, unchangeable values, meant to undergo a series of transformations (functions) and then be displayed.

All of the authors of this paper taught introductory programming in Racket with images prior to publication, so the impetus for many of the individual updates came from their own classroom experiences. As a result, this paper gives intuition for the changes made but doesn't provide any scientific basis for their necessity.

2 Technical Overview

This paper details a serious update to the Racket image library. It modifies the original library, which is designed for introductory computer science students, with the following goals:

1. To make the library more intuitive for beginner students
2. To speed up run-time
3. To add missing features

2.1 Equality

One of the big changes made to the library in the update described in this paper was a change in how equality is calculated. In Racket, there are three functions for equality checking: `eq?`, `eqv?` and `equal?`. From reading through the Racket reference page [4], these all seem to act the same way; for any values `v1,v2`:
`eq? v1 v2` $\iff$ `eqv? v1 v2` $\iff$ `equal? v1 v2`

However, in their native forms, these functions are all defined slightly differently. `equal?` is intended to be modified to work more complexly for specific datatypes, and works better than `eq?`on complex datatypes in its default (naive) form. This is evident when looking at the use of `equal?` on user-defined structures. The Racket documentation page explains that "A generic `equal?` comparison automatically recurs on the fields of a transparent structure type, but `equal?` defaults to mere instance identity for opaque structure types" [4]. In Racket, "transparent" structure types allow you to see the details of internal aspects of the structure, whereas "opaque" types hide this from you. As a comparison, an "opaque" structure is essentially what you get by using the ">:" operator in SML.

Since shapes in 2htdp/image are represented in forms like

```
(rect 0 10 "solid" red")
```

we can determine that these are transparent types, so `equal?` will recurse on the internal fields to check equality.

In this paper's implementation of equality, shapes are first represented as "normalized shapes." Once this representation has been created, the authors' `equal?`implementation uses Racket's recursive equality checking, and can sometimes determine equality immediately. As written by the users, two identical but not `eq?` shapes could be the same original shape with the same transformations, just applied in a different order. As an example,

```
(define r1
    (rotate 30
        (scale 4
            (rectangle 10 10 "solid" "green"))))
```

```
(define r2
    (scale 4
       (rotate 30
           (rectangle 10 10 "solid" "green"))))
```

These two shapes are are clearly identical, even though the data structures representing them aren't exactly the same. So, they become the same normalized shape, and can then be checked for equality in a simple recursive manner.

However, this strategy does not work for all shapes, given that this paper aims to define two shapes as "equal" if they are observably the same (more formally, they are the same if all transformations on them will yield the same result). For example, define two shapes as follows:

```
(define r1
    (beside
        (rectangle 100 100 "solid" "blue")
        (rectangle 100 100 "solid" "blue")))

(define r2
    (rectangle 200 100 "solid" "blue"))
```

These structures are *not* the same with reordering; as data, they appear to be fundamentally different. However, basic arithmetic and geometry shows that the images produced should be the same. In this case, the implementation of the equality checker defaults to rendering and comparing the bitmaps.

The process of implementing more complex equality checking also involved adjusting the internal representation of the images. One of the changes the authors made was to adopt the new mantra "don't push cropping to the leaves." The previous implementation, which pushed the job of cropping a shape to children shapes of a given shape, had resulted in redundancy.

2.2 Overlay

The authors reversed the ordering of overlay, reversing the intuition behind the previous implementation. For example, consider the following code:

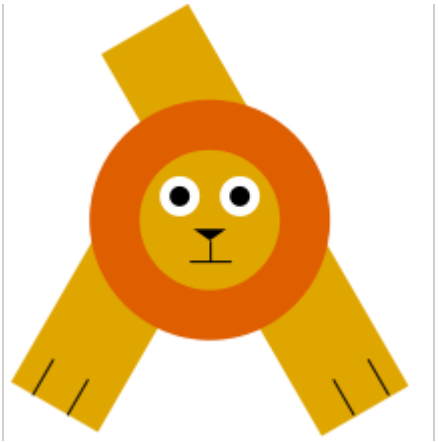
```
(overlay
    (rectangle 100 100 "solid" "blue")
    (rectangle 500 500 "solid" "orange"))
```

Previously, the library would have printed the orange rectangle on top of the blue rectangle. In the new implementation, the blue rectangle is printed on top of the orange one. In addition to this reversal, the authors implemented an `underlay` function, which does the exact opposite of the `overlay` function. The authors of the paper explained this change as aligning more with intuition, though they fail to cite any formal scientific reasoning for the change. We assume from their anecdotal evidence and citation of intuition that their experience in the classroom teaching introductory students influenced this design choice.

3 Constructive Engagement with the Work

Before this project, neither of us had experience with Racket (Maxine used Haskell in her introductory CS course, Elizabeth used Python). To engage with the work, we downloaded DrRacket and gave DrRacket's little green running person plenty of good workouts as we did an informal a crash course in Racket.

After playing around with Racket to the point of feeling comfortable, we started playing around with the image library described in the paper. We decided our personal goal for familiarizing ourselves with the image library was to create our new favorite class mascot, Lambda the Lion.



We chose our goal to implement lambda the lion because we hoped to engage with the authors' design choices regarding pinholing and rotation, scenes, and overlay, as highlighted in the paper.

4 Current Status of the Work/Related Work

The library described in this paper is currently in use by Racket, in all of Racket's sub-languages [4]. The authors of the paper stated that their image library was designed to be used with these teaching-focused introductory Racket languages. Throughout the paper the authors make references to the use of this library in classrooms.

To investigate related work, we looked at other instances of functional-first and image-first approaches to introductory to computer science. The textbook *The Fun of Programming*, for example, begins with a section on the beauty of the results of programming pictures and the beauty of the process of programming in functional languages [5]. The author remarks that these two are rarely combined; we create graphics in "ugly," code-heavy languages like Java, while we use "pretty," functional languages like SML, Racket, or Haskell primarily

for arithmetic or data processing. Stephen Bloch, the author of *Picturing Programs*, who teaches mostly non-CS majors, teaches programming with images first because he finds that his students are often scared off by introductory sequences that start with arithmetic [3]. Dr. Bloch's 2009 textbook starts with image manipulation at the very beginning of chapter 1 [3].

Despite the enthusiasm of a few functional programming enthusiasts, the functional-first trend seems not to have caught on. The University of Chicago's CS151 intro course still uses Racket and the 2htdp/image library. The intro sequence of Northwestern University does the same, though with minimal use of the 2htdp/image library. Both of these courses begin with arithmetic rather than images, and only UChicago's course makes extensive use of images in assignments. Of the other U.S. News-ranked top 10 universities and top 10 liberal arts colleges in the United States, no other introductory computer science class uses a functional language. Almost all other introductory courses at these schools use either Python or Java. A few start off with image-based lectures and projects: Yale University, The University of Pennsylvania, Bowdoin College, and Pomona College all use native image libraries and operate in an object-oriented environment. As a result, these schools' assignments ask students to hard-code drawing of certain shapes. These assignments do not ask students to write functions to create images with generalized characteristics. These requirements have the effect that Dr. Bloch aims for in his text, as students work on on creating images instead of computing factorials, sums, or Fibonacci sequences. However, the way the images are stored internally and the way functions on images operate have nothing to do with the assignments. Images are treated here as outputs, not values, and so none of these characteristics matter in context. Of these top colleges, then, the only intro sequence that is in any way benefiting from this library is ours (see tables below) [1].

Universities

Rank	College	Language	Image-first?
1	Princeton	Java	N
2	Harvard	C/Python/Web	N
3	Columbia	Java	N
3	MIT	Python	N
3	UChicago	Racket	Y
3	Yale	Java	Y
7	Stanford	Java or Python	N
8	Duke	Python	N
8	UPenn	Java	Y
10	JHU	Java	N
10	Northwestern	Racket	N

Liberal Arts Colleges

Rank	College	Language	Image-first?
1	Williams	Python	N
2	Amherst	Java	N
3	Swarthmore	Python	N
3	Wellesley	Python	Y
5	Bowdoin	Python	Y
5	Carleton	Python	N
5	Middlebury	Python	N
5	Pomona	Java	Y
9	Claremont McK	Python	N
10	Davidson	Python	?

Regarding the status of images-first pedagogy, one of the most popular programming languages for students younger than the high school level is Scratch. Scratch is a web-based program developed by researchers at the MIT Media Lab that allows students to program animated scenes by dragging command blocks into scripts [6]. In a sense, Scratch is doubly image-based: the goal of program-

ming in Scratch is to program a visual output, and the process of programming itself is image-based by nature of the block selection and dragging-into-script technique. In a 2009 paper explaining the pedagogical choices behind Scratch, the inventors of Scratch reflect that "there needs to be a shift in how people think about programming, and about computers in general. We need to expand the notion of 'digital fluency' to include designing and creating, not just browsing and interacting. Only then will initiatives like Scratch have a chance to live up to their full potential" [6] Per the authors of Scratch, using images in introductory computer science education is a technique designed to render computer science more accessible. Image-based programming is often seen as a way to sidestep the educational hurdle of math. The authors of the Scratch article argue that image-based introductory programming goes beyond serving as a mere substitute for math; they claim that it can reshape the narrative of what computer science itself means, weaving creativity into the narrative of computer science education and opening the door of computer science to students who might otherwise have been edged out. [6]

As for the paper itself, "The Design of Functional Image Libraries", ResearchGate lists it as having been cited once and viewed 30 times[**?**]. It seems unlikely that the paper has been read by many people since its publication a decade ago, and the library itself is used even less frequently than Racket (in other words, barely).

References

[1] Our summary of intro cs coursework; see submitted spreadsheet.

[2] Matthew Findler Robert Bruce Barland, Ian Flatt. The design of a functional image library, 2010.

[3] Stephen Bloch. Picturing programs, 2009.

[4] Robert Bruce Flatt, Matthew Findler. The racket guide.

[5] Jeremy Gibbon. The fun of programming, 2003.

[6] Mitchel Resnick et al. Scratch: Programming for all, 2009.