The Design of a Functional Image Library

Ian Barland  
Radford University  
ibarland@radford.edu

Robert Bruce Findler  
Northwestern University  
robbie@eecs.northwestern.edu

Matthew Flatt  
University of Utah  
mflatt@cs.utah.edu
Goals

Wanted: A 2-D image library for racket, “2htdp/image”, usable by beginners on day one, yet rich enough to make interesting programs...
Goals

Wanted: A 2-D image library for racket, “2htdp/image”, usable by beginners on day one, yet rich enough to make interesting programs... i.e., video games.
Goals

Wanted: A 2-D image library for racket, “2htdp/image”, usable by beginners on day one, yet rich enough to make interesting programs... i.e., video games.

• Efficient equality checking for images

• Intuitive functions for overlaying images

• Include rotation, scaling, flipping, cropping.

We give an experience report on design decisions, and unexpected complications.
Roadmap

• API tour (w/ commentary)
• Issues with equality
• Implementation data definition
• Lessons Learned
Tour: atomic shapes

• \((\text{circle } 35 \ "\text{outline}\" \ "\text{black}\") \rightarrow \)
Tour: atomic shapes

- `(circle 35 "outline" "black")` → 
- `(isosceles-triangle 90 130 "solid" "lightseagreen")` →

![Diagram of a circle and an isosceles triangle]
Tour: atomic shapes (cont.)

- \((\text{regular-polygon } 25 \ 7 \ "\text{solid}" \ "\text{red}" ) \rightarrow \star\)

- \((\text{star-polygon } 25 \ 7 \ 2 \ "\text{outline}" \ "\text{blue}" ) \rightarrow \star\)

- \((\text{rhombus } 40 \ 60 \ "\text{outline}" \ "\text{black}" ) \rightarrow \)

Workshop on Scheme and Functional Programming, Montréal 2010; Barland, Findler, Flatt
Tour: atomic shapes (cont.)

- `(regular-polygon 25 7 "solid" "red") → □
- `(star-polygon 25 7 2 "outline" "blue") → □
- `(rhombus 40 60 "outline" "black") → □
- `(text "J'❤️ Montréal" 45 "olive") → J'❤️ Montréal
Tour: atomic shapes (cont.)

- `(regular-polygon 25 7 "solid" "red") → •
- `(star-polygon 25 7 2 "outline" "blue") → •
- `(rhombus 40 60 "outline" "black") → •
- `(text "J'❤ Montréal" 45 "olive") → J'❤ Montréal
- `(bitmap "plt-logo-small.png") → •
Tour: composite shapes

- \((\text{crop } 0 \ 30 \ 40 \ 40 \ (\text{circle } 40 \ "\text{solid}" \ "\text{orange}"))\) → ![Orange Circle]

- \((\text{rotate } 30 \ (\text{ellipse } 60 \ 30 \ "\text{solid}" \ "\text{blue}"))\) → ![Blue Ellipse]

N.B. no reference-point, for \textit{rotate}. 
Tour: composite shapes (cont.)

- \((\text{add-curve})\)
  \((\text{rectangle 200 50 "solid" "black"})\)
  \(10 40 30 1/2\)
  \(190 40 -90 1/5\)
  \((\text{make-pen "white" 4}
  "solid" "round" "round")) \rightarrow
Tour: composite shapes (cont.)

• \((\text{add-curve}
  \ (\text{rectangle} \ 200 \ 50 \ "\text{solid}\" \ "\text{black}\")
  \ 10 \ 40 \ 30 \ 1/2
  \ 190 \ 40 \ -90 \ 1/5
  \ (\text{make-pen} \ "\text{white}\" \ 4
   \ "\text{solid}\" \ "\text{round}\" \ "\text{round}\")\)) \rightarrow

\[
\begin{array}{c}
\end{array}
\]

• \((\text{above}
  \ (\text{star} \ 60 \ "\text{solid}\" \ "\text{firebrick}\")
  \ (\text{scale/xy}
   \ 1 \ 1/2
   \ (\text{flip-vertical}
    \ (\text{star} \ 60 \ "\text{solid}\" \ "\text{gray}\")\))))
Tour: composite shapes (cont.)

• \((\text{add-curve})\)
  \((\text{rectangle} \ 200 \ 50 \ "solid" \ "black")\)
  \(10 \ 40 \ 30 \ 1/2\)
  \(190 \ 40 \ -90 \ 1/5\)
  \((\text{make-pen} \ "white" \ 4 \ "solid" \ "round" \ "round")\) →

\[\begin{array}{c}
  \includegraphics[width=0.2\textwidth]{curve}
\end{array}\]

• \((\text{above})\)
  \((\text{star} \ 60 \ "solid" \ "firebrick")\)
  \((\text{scale/xy} \ 1 \ 1/2\)
  \((\text{flip-vertical} \ (\text{star} \ 60 \ "solid" \ "gray"))))\) →

\[\begin{array}{c}
  \includegraphics[width=0.2\textwidth]{star}
\end{array}\]
Tour: composite shapes (cont.)

• (beside (square 40 "solid" "blue")
  (ellipse 30 60 "solid" "green"))
Tour: composite shapes (cont.)

• \((\text{beside} \ (\text{square} \ 40 \ \text{"solid" \ "blue"}) \ (\text{ellipse} \ 30 \ 60 \ \text{"solid" \ "green"}))\) →
Tour: composite shapes (cont.)

- (beside (square 40 "solid" "blue")
  (ellipse 30 60 "solid" "green")) →

- (above/align "right"
  (star 30 "solid" "orange")
  (rectangle 120 20 "solid" "blue")
  (triangle 40 "solid" "red")) →
Tour: overlay

• \((\text{overlay} \ (\text{square} \ 30 \ "\text{solid}" \ "\text{orange}"))\) \((\text{square} \ 40 \ "\text{solid}" \ "\text{blue}")) \rightarrow \[\text{Figure} \]

• \((\text{overlay} \ \text{img1} \ \text{img2})\): “overlay \text{img1} on top of \text{img2}”, \textit{not} “\text{img1} is overlaid with \text{img2}”. 
Tour: overlay

• (overlay (square 30 "solid" "orange")
  (square 40 "solid" "blue")) → 

• (overlay img1 img2): “overlay img1 on top of img2”,
  not “img1 is overlaid with img2”.

• (overlay/xy (square 30 "solid" "orange")
  0 7
  (square 40 "solid" "blue")) →
Tour: overlay

- \((\text{overlay} \ (\text{square} \ 30 \ "\text{solid}" \ "\text{orange}")) \ (\text{square} \ 40 \ "\text{solid}" \ "\text{blue}")) \rightarrow \)

- \((\text{overlay} \ \text{img1} \ \text{img2})\): “overlay \ \text{img1} \ on \ top \ of \ \text{img2}”, not “\ \text{img1} \ is \ overlaid \ with \ \text{img2}”.

- \((\text{overlay/xy} \ (\text{square} \ 30 \ "\text{solid}" \ "\text{orange}")) \ 0 \ 7 \ (\text{square} \ 40 \ "\text{solid}" \ "\text{blue}")) \rightarrow \)

- Coordinate system: origin at top-left (as graphics), not lower-left (as math).
Tour: overlay’s relatives

- (underlay/xy (square 40 "solid" "seagreen")
  35 5
  (circle 10 "solid" "orange"))
Tour: overlay’s relatives

• \(\text{underlay/xy} \ (\text{square} \ 40 \ "\text{solid}\" \ "\text{seagreen}\")} \)
  \(35 \ 5\)
  \(\text{(circle} \ 10 \ "\text{solid}\" \ "\text{orange}\")})\)

→

• \underline{\text{underlay/xy}}, for the common task “put object onto a background at dx,dy”. 
Tour: overlay’s relatives

- `(underlay/xy (square 40 "solid" "seagreen")
  35 5
  (circle 10 "solid" "orange"))` →

- `underlay/xy`, for the common task “put object onto a background at dx,dy”.

- `(place-image (circle 10 "solid" "orange")
  35 5
  (square 40 "solid" "seagreen"))`
Tour: overlay’s relatives

- `(underlay/xy (square 40 "solid" "seagreen") 35 5 (circle 10 "solid" "orange"))`

  └───
   └───

- `underlay/xy`, for the common task “put object onto a background at dx,dy”.

- `(place-image (circle 10 "solid" "orange") 35 5 (square 40 "solid" "seagreen"))`

  └───
   └───

- `place-image`: as `overlay/xy` but crop, and place `img2`’s center (not origin); like `underlay` the coords are relative to background.
Tour: equality checking

- \((\text{equal?} \ (\text{circle} \ 20 \ "\text{solid}\" \ "\text{red}\") \ (\text{ellipse} \ 40 \ 40 \ "\text{solid}\" \ (\text{color} \ 255 \ 0 \ 0)))\)
  \[\rightarrow #t\]

- Critical for unit testing. For example:
  \[(\text{check-expect} \ (\text{draw-world} \ (\text{move-right initial-world})) \ (\text{overlay/xy player-image 5 0 initial-image}))\]
Related work

Many other functional image libraries:

- Functional Pictures (Henderson 1982)
- PIC (Kernighan 1991)
- MLGraph (Chaillous and Cousineau, 1992)
- Functional Postscript (Sae-Tan and Shivers, 1996)
- Pictures (Finne and Peyton Jones, 1995)
- Functional Images (Elliot 2003)
- ...and more.

Our work (a) is designed for use in an intro programming course and (b) emphasizes equality-checking.
Roadmap

• API tour (w/ commentary)
• **Issues with equality**
• Implementation data definition
• Lessons Learned
Image Equality: what does it mean?

Ideally, “Observationally Equivalent”: we want to say two images are equal iff they behave the same under any series of operations. This may not be the same as drawing identically at their current scale (e.g. a 0.8-pixel square vs a 0.9-pixel square).

We explore reasons why the ideal answer is both (too) difficult to compute, and is arguably not the correct pedagogic choice after all.
Consider the following four ways of representing a rectangle:

- (*rectangle* 10 20 "outline" "blue")
- (*rotate* 90 (*rectangle* 20 10 "outline" "blue"))
- a polygon connecting (0,0), (10,0), (10,20), (0,20).
- four entirely disjoint line segments and straight curves, rotated and placed above or beside each other to achieve the same rectangle.
Image Equality: difficulties

Consider the following four ways of representing a rectangle:

• \((\text{rectangle } 10 \ 20 \ "\text{outline}" \ "\text{blue}"))

• \((\text{rotate } 90 \ (\text{rectangle } 20 \ 10 \ "\text{outline}" \ "\text{blue}"))\)

• a polygon connecting \((0,0), (10,0), (10,20), (0,20)\).

• four entirely disjoint line segments and straight curves, rotated and placed above or beside each other to achieve the same rectangle.

One can think of ways to work around these, but there are more difficulties...
Image Equality: difficulties (cont)

What are different ways to construct an image equivalent to

```
• (beside (square 30 "solid" "orange")
  (square 30 "solid" "blue"))
• (overlay/align "left" "center"
  (rectangle 60 30 "solid" "blue")
  (square 30 "solid" "orange"))
• (overlay/xy ... 5 5 (circle 2 "solid" "orange"))
```
Image Equality: difficulties (cont)

What are different ways to construct an image equivalent to

• (beside (square 30 "solid" "orange")
  (square 30 "solid" "blue"))

• (overlay/align "left" "center"
  (rectangle 60 30 "solid" "blue")
  (square 30 "solid" "orange"))

• (overlay/xy ... 5 5 (circle 2 "solid" "orange"))

One can think of ways to work around these, but there are still more difficulties...
Image Equality: difficulties (cont)

• Cropping ellipses and curves can lead to complicated (disjoint) shapes; checking equality becomes difficult.

• Floating point error: rotating $30^\circ \times 3$ vs $45^\circ \times 2$. 
Image Equality: difficulties (cont)

• Cropping ellipses and curves can lead to complicated (disjoint) shapes; checking equality becomes difficult.

• Floating point error: rotating 30°×3 vs 45°×2.

Aaagh!
Image Equality: difficulties (cont)

- Cropping ellipses and curves can lead to complicated (disjoint) shapes; checking equality becomes difficult.

- Floating point error: rotating $30^\circ \times 3$ vs $45^\circ \times 2$.

Aaagh! ...And if you do extensive work to handle all these, students (and others) might still be baffled when two identically-drawn images aren’t considered equal?
Two other issues:

• The \text{text} function might introduce ligatures or kerning; \text{text "Wifi"} might not draw the same as placing individual letters \text{beside} each other.
Two other issues:

• The `text` function might introduce ligatures or kerning; 
  \((\text{text} \ "Wifi")\) might not draw the same as placing 
  individual letters `beside` each other.
  \((2htdp/image\) patches this by passing each individual letter to 
  the underlying text-draw function and `besideing` the results.\)

• Zero-width rectangles can invisibly change the bounding box.
Image Equality: difficulties (cont)

Two other issues:

• The `text` function might introduce ligatures or kerning; `(text "Wifi")` might not draw the same as placing individual letters `beside` each other.
  `(2htdp/image` patches this by passing each individual letter to the underlying text-draw function and `beside`ing the results.)

• Zero-width rectangles can invisibly change the bounding box.
  `(2htdp/image` currently considers such images `non-equal`.)
Roadmap

• API tour (w/ commentary)
• Issues with equality
• Implementation data definition
• Lessons Learned
Data definition (part 1)

Images represented as a straightforward tree of structures.

\[
\text{(image}
\text{(bounding-box width height baseline))}
\text{(Rec Shape}
\text{(U Atomic-Shape ; includes ellipses,}
\text{; text, bitmaps, etc}
\text{Polygon-Shape ; includes rectangles,}
\text{; lines, curves, etc}
\text{(overlay Shape Shape))}
\text{(translate dx dy Shape)}
\text{(scale sx sy Shape)}
\text{(crop (Listof point) Shape))})
\]
Data definition (part 2)

When drawing or comparing images, normalize them first:

\[
(\text{Rec Normalized-Shape} \\
  (U (\text{overlay Normalized-Shape CN-Shape}) \\
   CN-Shape))
\]

\[
(\text{Rec CN-Shape} \\
  (U (\text{crop (Listof point) \\
          Normalized-Shape} \\
          (\text{translate num num Atomic-Shape} \\
           Polygon-Shape}))
\]

Overlay-of-overlays are linearized; translates, rotates, scales pushed down to leaves.

Time complexity of draw/equality-check is still linear.
Equality: resolution

Equality is implemented in two parts:

• First try checking structural equality of normalized shapes;
• if that can’t show them equal, fall back to comparing bitmaps.

This strategy well-suited for reasonable unit tests which pass.
Equality: resolution

Equality is implemented in two parts:

• First try checking structural equality of normalized shapes;
• if that can’t show them equal, fall back to comparing bitmaps.

This strategy well-suited for reasonable unit tests which pass.

<table>
<thead>
<tr>
<th>Library</th>
<th>Time</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original library</td>
<td>9346 msec</td>
<td></td>
</tr>
<tr>
<td>2htdp/image library, without fast path</td>
<td>440 msec</td>
<td>21x</td>
</tr>
<tr>
<td>2htdp/image library, with fast path</td>
<td>18 msec</td>
<td>509x</td>
</tr>
</tbody>
</table>

Figure 2: Timing a student’s final submission, run on a Mac Pro 3.2 GHz machine running Mac OS X 10.6.5, Racket v5.0.0.1
Roadmap

• API tour (w/ commentary)
• Issues with equality
• Implementation data definition
• **Lessons Learned** (three of ’em)
Lesson: Rotate is linear-time — example

Consider the following examples. Which vertices determine overall bbox?

```
(define r (rectangle 40 20 "solid" "red"))
(define (rot-above p)
  (above (rotate 40 p) r))

• (rot-above r) → [diagram of rotated rectangle]

• (rot-above (rot-above r)) → [diagram of double rotated rectangle]
```
Lesson: Rotate is linear-time — example

Consider the following examples. Which vertices determine overall bbox?

```scheme
(define r (rectangle 40 20 "solid" "red"))
(define (rot-above p)
  (above (rotate 40 p) r))

• (rot-above (rot-above r)) →

• (rot-above (rot-above (rot-above r))) →
```
Lesson: Rotate is linear-time — example

Consider the following examples. Which vertices determine overall bbox?

\[
\text{(define } r \text{ (rectangle 40 20 "solid" "red"))}
\]
\[
\text{(define } \text{(rot-above } p \text{)} \quad \text{(above (rotate 40 } p \text{) } r))\)
\]

\[
\bullet \text{(rot-above}
\quad \text{(rot-above (rot-above } r)\text{)) } \rightarrow \\
\]

\[
\bullet \text{(rot-above}
\quad \text{(rot-above}
\quad \text{(rot-above (rot-above } r)\text{))} \text{) } \rightarrow \\
\]
Lesson: Don’t push cropping to the leaves

Initially, cropping (like rotating, translating) was pushed to leaves. But this (with \texttt{overlay}) leads to quadratic blowup when normalizing.

\[
\text{(crop r1 (crop r2 (crop r3 (overlay s1 s2))))}
\]
Lesson: Don’t push cropping to the leaves

Initially, cropping (like rotating, translating) was pushed to leaves. But this (with overlay) leads to quadratic blowup when normalizing.

\[
\begin{align*}
(crop \ r1) &\quad (overlay \ (crop \ r1) \ (overlay \ (crop \ r2) \ (crop \ r3) \ s1))) \\
(crop \ r1) &\quad (overlay \ (cow \ r1) \ (overlay \ (crop \ r2) \ (crop \ r3) \ s2)))
\end{align*}
\]
Lesson: Don’t push cropping to the leaves

Initially, cropping (like rotating, translating) was pushed to leaves. But this (with *overlay*) leads to quadratic blowup when normalizing.

\[
\begin{align*}
(crop \\
r1 \\
(crop \\
r2 \\
(crop \\
r3 \\
\text{(overlay s1 s2)}))) & \Rightarrow \quad (crop \ r1 \\
(crop \ r2 \\
\text{(overlay s1 s2)}))) & \Rightarrow \quad (crop \ r1 \\
(crop \ r2 \\
(crop \ r3 \ s1))) \\
\end{align*}
\]

Resolution: in a normalized shape, nodes are overlays or crops.
Lesson: filled-pixels vs lines

Consider drawing a solid 3x3 rectangle: lines from (0,0), (0,3), (3,3), (3,0). But isn’t that off-by-one?
Lesson: filled-pixels vs lines

Consider drawing a solid 3x3 rectangle: lines from (0,0), (0,3), (3,3), (3,0). But isn’t that off-by-one?

No — coordinates reside at the corner of the square pixel.
Lesson: filled-pixels vs lines

Consider drawing a solid 3x3 rectangle: lines from (0,0), (0,3), (3,3), (3,0). But isn’t that off-by-one?

No — coordinates reside at the corner of the square pixel.

But when drawing outline rectangles: Drawing between pixels with a pen will appear as a 2-pixel-wide line.
Lesson: filled-pixels vs lines

Consider drawing a solid 3x3 rectangle: lines from (0,0), (0,3), (3,3), (3,0). But isn’t that off-by-one?

No — coordinates reside at the corner of the square pixel.

But when drawing outline rectangles: Drawing between pixels with a pen will appear as a 2-pixel-wide line.

Solution: offset all polygon outline coordinates by (0.5,0.5).
Thanks

Thanks to Stephen Bloch, Carl Eastlund, Matthias Felleisen, Guillaume Marceau, and Jean-Paul Roy for helpful discussions, work and ideas.
Thanks

Thanks to Stephen Bloch, Carl Eastlund, Matthias Felleisen, Guillaume Marceau, and Jean-Paul Roy for helpful discussions, work and ideas.

Questions?

Ian Barland
Radford University
ibarland@radford.edu

Robert Bruce Findler
Northwestern University
robb@eecs.northwestern.edu

Matthew Flatt
University of Utah
mflatt@cs.utah.edu