Haven

Scalable Vector Graphics for Haskell with GCJNI and Java2D

Antony Courtney

Yale University

Part I: GCJNI

GreenCard + Java Native Interface

Java Native Interface (JNI)

- **What is the JNI?**
  - A set of programming conventions for writing C code which can be invoked by Java programs (as *native methods*).
  - A C Language API that allows native code to create and access objects, methods and fields in a Java VM.

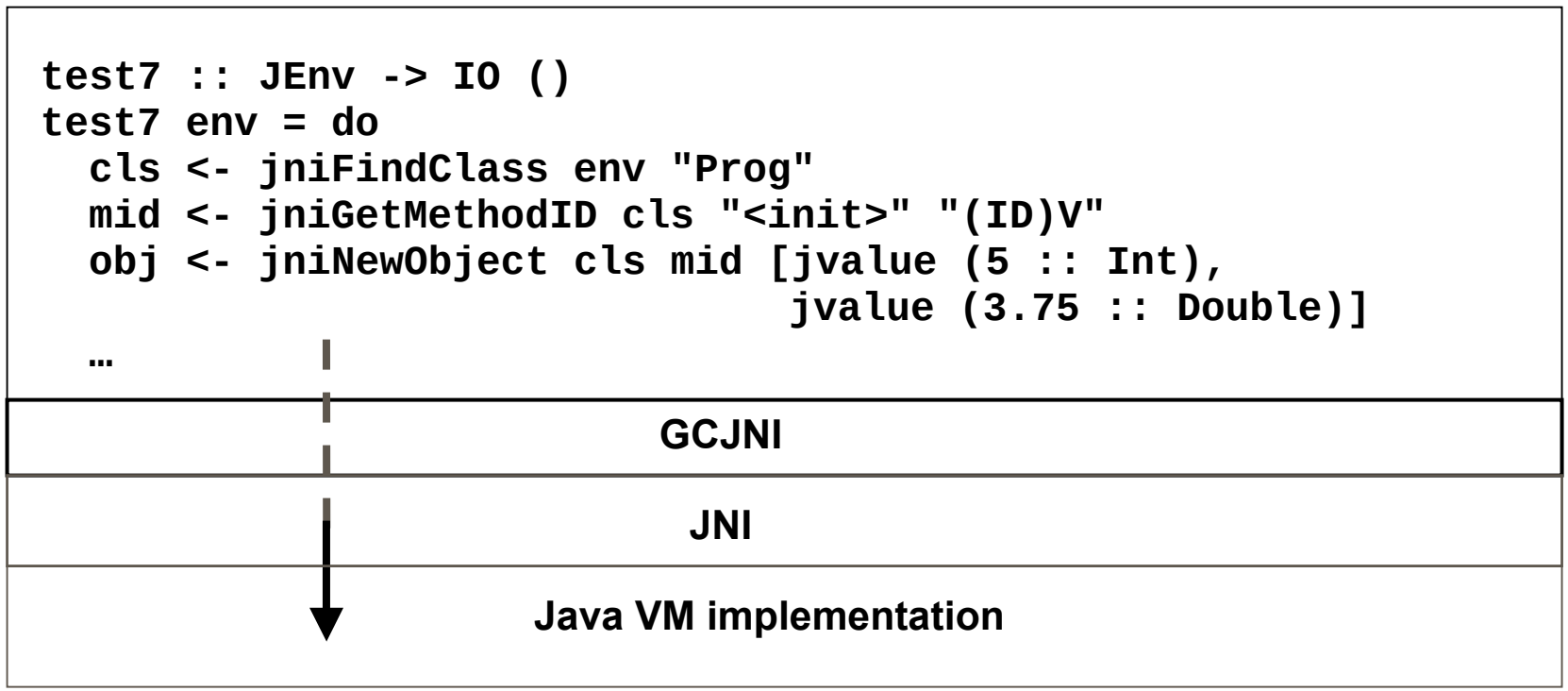
- **What does JNI code look like?**

```
JNIEnv env = … // initialization;
…
jclass c = (env)->FindClass("java/lang/String");
jmethod m = (env)->GetMethodID(env, c, "length","()I");
…
jobject sref; // an instance of java.lang.String
jint slen = (env)->CallIntMethod(env, s, m);
```

GCJNI

- GCJNI is a simple *one-way* bridge from Haskell to Java:
  - Haskell can create Java objects and invoke Java methods.
  - Haskell can only pass simple types or Java objects to methods.
  - Java can **not** invoke Haskell functions or access Haskell values.

```
test7 :: JEnv -> IO ()
test7 env = do
  cls <- jniFindClass env "Prog"
  mid <- jniGetMethodID cls "<init>" "(ID)V"
  obj <- jniNewObject cls mid [jvalue (5 :: Int),
                               jvalue (3.75 :: Double)]
  …
```



GCJNI Implementation

- Mostly a (dull) port of JNI to Haskell via GreenCard.
- **`JValue`** type class for marshalling / unmarshalling
  - instances for Int, Double, Bool, String, etc…
- Integration of garbage collectors:
  - global **`jobject`** references treated as **`foreign`** pointers, so Java objects are GC'ed when no longer reachable from Haskell.
  - one-way $\Rightarrow$ no references to Haskell values in Java's heap.
    - limiting, but lets us avoid worrying about cyclic garbage! :-/
- GenBindings tool:
  - Generates a Haskell module with types and functions for a set of Java classes
  - Generated functions use GCJNI to invoke the appropriate Java method

Living with a One-Way System

- Java can't invoke Haskell functions or IO actions.
  - but we can return values to Haskell from Java methods.
- GUI Input Handling:

  In Java (AWT and Swing), usually done via listener classes:
  - application creates and registers a listener instance.
  - toolkit invokes method of listener when some event occurs.

  In GCJNI / Haven, we compensate with a little scaffolding:
  - scaffolding implements listeners for all events of interest.
  - Each listener places event on a queue and returns to toolkit.
  - Provide a synchronized method callable from Haskell to the first event from the queue.

GenBindings

- Generates Haskell module with types, type classes and functions for a set of Java classes
  - Uses Java reflection APIs to recover type information
- Based on encoding from Lambada (Meijer and Finne) (more recently: Shields and Peyton-Jones, Babel '01)
  - Key Idea: use phantom type parameters for inheritance, type classes for interfaces:

```
class C {
  public int
    foo(B arg1,
        double arg2,
        I arg3);
}
```

```
doFoo :: (I c) =>
         C a          -- this
         -> B b       -- arg1
         -> double    -- arg2
         -> c         -- arg3
         -> JEnv
         -> IO int
```

GenBindings Design Issues

Naming is boring but treacherous:

The "safe" approach results in Haskell function names like:

```
JavaAWTGeom.doQuadCurve2DSubdivide_LQuadCurve2D_LQuadCurve2
D_LQuadCurve2D__V :: …
```

- Namespaces:
  - Java: method names scoped by (static) type of receiver
  - Haskell: one flat namespace per module
- Name overloading:
  - Java: name clashes may occur as long as number or type of parameters resolves ambiguity
  - Haskell: one function type per name (+ type class overloading)
- Packages:
  - How should we map Java packages/classes to Haskell modules?
  - …raises a thorny set of deployment issues.

Deployment Issues

- When generating bindings for a Java library,
  - class C might mention class B defined in some other package.
  - …where / when / how / who generates bindings for these dependent packages?
  - **…should a generated library ship with dependent package bindings?**
- Real Issue here is not Java-specific:
  - Hierarchical module naming is not enough!
  - If we want to ship a Haskell library (module) Foo that depends on libraries Bar and Baz, do we:
    1. Include Bar and Baz in distribution of Foo? (redundancy issues)
    2. Force user to fetch Bar, Baz (and their dependencies etc.)? (installation nightmare!)
- Does Haskell need something like Perl's CPAN tools?
  - automated tool to fetch and install libraries (and their dependencies) from archive mirror sites

GenBindings Issues, cont.:

- I use a "worse-is-better" design for now:
  - Emits one big Haskell module for a set of Java classes given on command line.
  - Walks class hierarchy to find most-specific class for each method.
  - Generates simple names wherever possible; uses prefixing or mangling only if clashes occur.
  - Command-line argument to explicitly import dependent packages instead of generating them (HACK!).
- **I'd be interested in discussing better solutions.**

(.net people: how are you addressing these issues for C# / CLR?)

GCJNI - Status

- Reasonably portable and robust (so far):
  - Binaries available for all of (Windows,Linux)*(hugs,ghc).
  - Stress-tested by Haven and Fruit (graphics, interactive GUIs).
- A few missing features:
  - arrays, field access
- Documentation: Release Notes
- Feedback welcome! (especially installation issues)
- Available from:

  `http://www.haskell.org/gcjni`

Part II: Haven

Functional Vector Graphics for Haskell

Vector Graphics

- **What is Vector Graphics?**
  - 2D Geometry Model based on line and (Bezier) curve segments.
  - Familiar Examples: PostScript, Illustrator (Adobe)
- **Why should you care?**
  - Spatially scalable, resolution independent representation
  - High-quality outline fonts
  - Generalized stroke and fill operations
    - pens of varying size, shape and join properties
  - Geometric operations on shapes / paths:
    - Bounds calculations
    - Affine Transforms (scale, rotate, shear)
    - Intersection Tests (hit detection on arbitrary shapes)
    - Constructive Area Geometry (CAG): union, intersection, etc…
  - Alpha-blending / Transparency

(All of which is missing from HGL, Fran, Xlib, GDK, etc.)

Java2D

- Java2D

  …includes all of the features just mentioned, plus things like anti-aliasing.

- Java2D API a combination of:
  - **`java.awt.geom:`**
    - classes for (mutable) Points, Rectangles, Ellipses, Paths, etc.
  - **`java.awt.Graphics2D:`**
    - (mutable) context for rendering state (transform, color, font, etc.)
    - handle to a (mutable) drawing surface

```
Graphics2D g2;
Ellipse2D.Double circle = new Ellipse2D.Double(…);
…
g2.setTransform(tx);
g2.setFont(f);
g2.setColor(Color.red);
g2.fill(circle);
```

Haven API Design

What Should the Haven API look like?

- ***We Want:*** to use Java2D's functionality in Haskell programs.
- ***We Have:*** an imperative, object-oriented library (Java2D).
- ***We Could:*** just use the IO monadic bindings from GenBindings.

...But that has some serious deficiencies:

1. Can't invoke IO actions in pure Haskell functions
   - e.g.: bounds calculations, intersection tests, etc.
2. Poor documentation / reasoning
   - In preceding code snippet, does **`g2.setFont()`** affect subsequent **`g2.fill()`**?
3. Portability
   - There are other 2D rendering libraries (e.g. MacOS X's Quartz)

An Alternative Approach

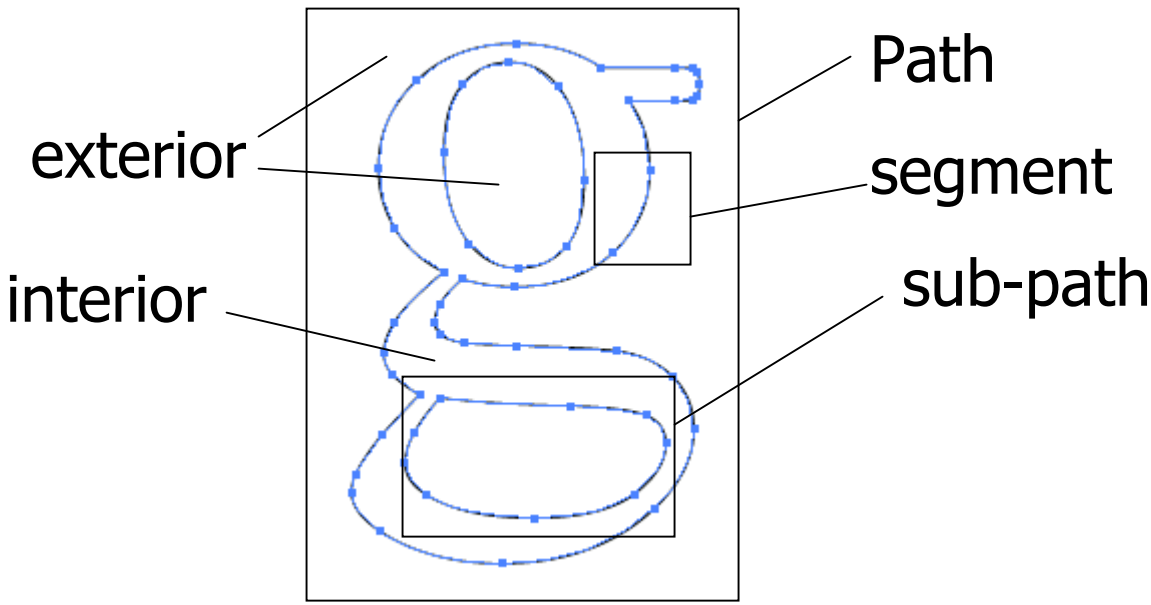
- Forget about Java2D's Java API (for now).
- Instead focus on the denotational semantics:

  What, exactly, are…
  - Images ?
  - Shapes ?
  - Paths ?
  - Pens ?
  - Fonts ?
  - Colors ?
- Answers determine a functional API specification, which also gives us:
  - simple, precise user documentation.
  - simple, effective practical reasoning about programs.
- There are many possible models:
  - …so let's strive for simplicity, generality, orthogonality.

Paths

The most fundamental type is *Path:*

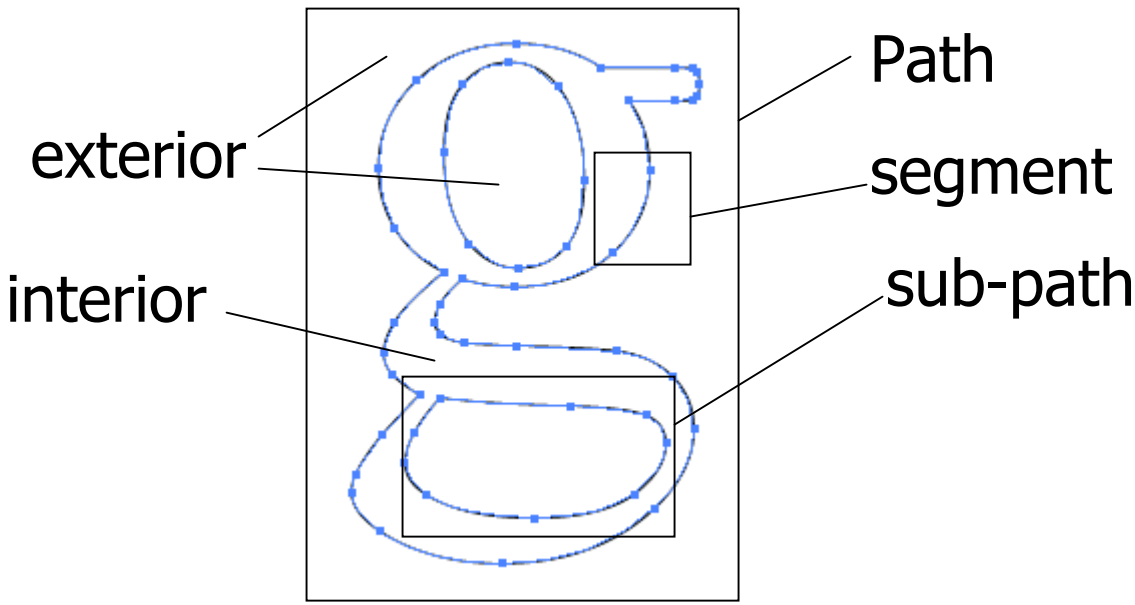
- A *Path* is a sequence of sub-paths.
- A *sub-path* is a connected sequence of *segments.*
- A Path determines an *interior* and *exterior*.



Paths - Denotation

- Denotationally (modulo some subtle details):

```
newtype Path = Path [SubPath]
newtype SubPath = SubPath (Point, [Segment])
data Segment = SegLineTo Point           -- straight line
             | SegQuadTo Point Point     -- quadratic curve
             | SegCurveTo Point Point Point  -- cubic
```



Shapes

- Paths can represent any 2D shape (or set of shapes).
- But many shapes have useful projection functions:
  - e.g.: **`rectWidth :: Rectangle -> Double`**
  - awkward to recover such information from a Path.
- For convenience, we define a *Shape* type class:

```
-- a Shape is anything that has an outline:
class Shape a where
  outline :: a -> Path
```

- Path is (trivially) an instance of Shape.
- Most API functions take Shapes rather than Paths.

Paths and Shapes, cont.

- What can we do with paths and shapes?
  - Paths form a monoid:

```
pathEmpty = Path []

<++> :: Path -> Path -> Path
(Path p1) <++> (Path p2) = Path (p1 ++ p2)
```

  - Some simple geometry primitives:

```
-- compute bounding rectangle for any shape:
shapeBounds :: (Shape a) => a -> Rectangle

-- interior test:
contains :: (Shape a) => a -> Point -> Bool

-- apply an affine transform to a Path:
pathTransform :: Transform -> Path -> Path
```

Rendering Shapes

- What we have are Paths:
  - Idealized outline of a shape – no thickness or fill color
- What we display are *Images:*

denotation:
```
Image = { imgD :: Point → Color,
          imgBoundsD :: Maybe Rectangle}
setImage :: Window -> Image -> IO ()
```

- Given a Shape, how do we produce an image?

  Most graphic libraries provide functions like:

```
fillShape :: (Shape a) => Color -> a -> Image
drawShape :: (Shape a) => Color -> Pen -> a -> Image
textImg :: Color -> Font -> String -> Image
…
```

  …adequate, but not compositional!

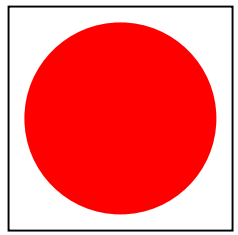
Rendering: Crop

- Instead of:

```
fillShape :: (Shape a) => Color -> a -> Image
```

e.g.:

```
fillShape red (circle 20) =
```

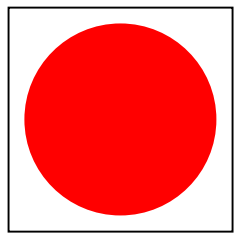


- Haven provides:

```
monochrome :: Color -> Image
imgCrop :: (Shape a) => a -> Image -> Image
```

```
monochrome red =
```

```
imgCrop (circle 20) (monochrome red) =
```



fillShape vs. imgCrop

`imgCrop` is far more versatile than **`fillShape`**:

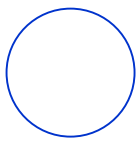
- Use **`imgCrop`** on *any* image:
  - color gradients (implemented):

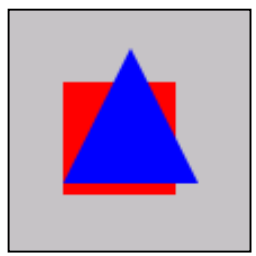
```
gradient :: Point -> Color -> Point -> Color -> Image
```

  - bitmaps (not implemented, but easy)
- Compose crop operations:

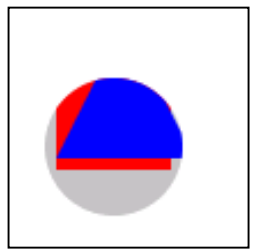
```
imgCrop s1 ((imgCrop s2 … ) <++>
            (imgCrop s3 (imgCrop …)))
```

`imgCrop`





=



Rendering II: Stroking a Path

- Instead of:

```
drawShape :: (Shape a) => Color -> Pen -> a -> Image
```

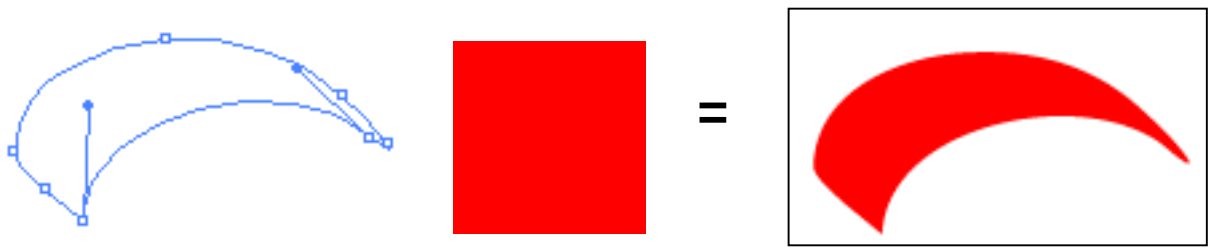
- Haven provides:

```
stroke :: (Shape a) => Pen -> a -> Path
```

- `(stroke p s)` is a Path whose interior is the result of stroking shape s with pen p:

`stroke` 

- Can easily compose `stroke` with `imgCrop`:

`imgCrop` 

Rendering III: Text

- Instead of:

```
textImg :: Color -> Font -> String -> Image
```

- Haven provides:

```
text :: Font -> String -> Path
```

```
e.g.:
  f=font "Times" italic 64
```

```
  text f "hello" =
```

- result path can be used in **`stroke`**, **`imgCrop`**, bounds calculations, intersection tests, CAG, etc…

Implementation – Simple Values

Observation:

Objects are just records + identity + mutation.

∴ if, for any class, we hide:

- mutator methods
- object identity

the class behaves like a pure (strict) Haskell record type:

class constructors ≅ record constructor functions

field accessor methods ≅ projection functions

object constructors / accessors have no side effects!

Objects as Values

(Trivial) Example:

```
class Point2D {
   double x, y;
   public Point2D(double x, double y);
   public double getX();
   public double getY();
 }
```

- From HavenJavaBindings.hs (generated by **`GenBindings –unsafe …`**):

```
runUnsafe :: (JEnv -> IO a) -> a
runUnsafe cmd = unsafePerformIO $ (jniGetEnv >>= cmd)

doPoint :: Double -> Double -> JEnv -> IO Point2D
doPoint = … -- jni object invocation stuff

point :: Double -> Double -> Point2D
point x y = runUnsafe $ doPoint x y

doPointGetX :: Point2D -> JEnv -> IO Double

pointGetX :: Point2D -> Double
pointGetX pt = runUnsafe $ doPointGetX pt
```

What about Images?

- Recall:

  denotation: `Image = { imgD :: Point → Color,`
  `imgBoundsD :: Maybe Rectangle}`

- We provide:

```
setPicture :: Window -> Image -> IO ()
imgBounds :: Image -> Rectangle
imgCrop :: (Shape a) => Image -> a -> Image
imgTransform :: Transform -> Image -> Image
…
```

- We don't provide direct access to **`imgD:`**
  - No projection / sampling functions for imgD itself.
- So…

  …how can we *implement* Image as an immutable class?

  i.e. what is an *operational* account of an Image?

Images - Implementation

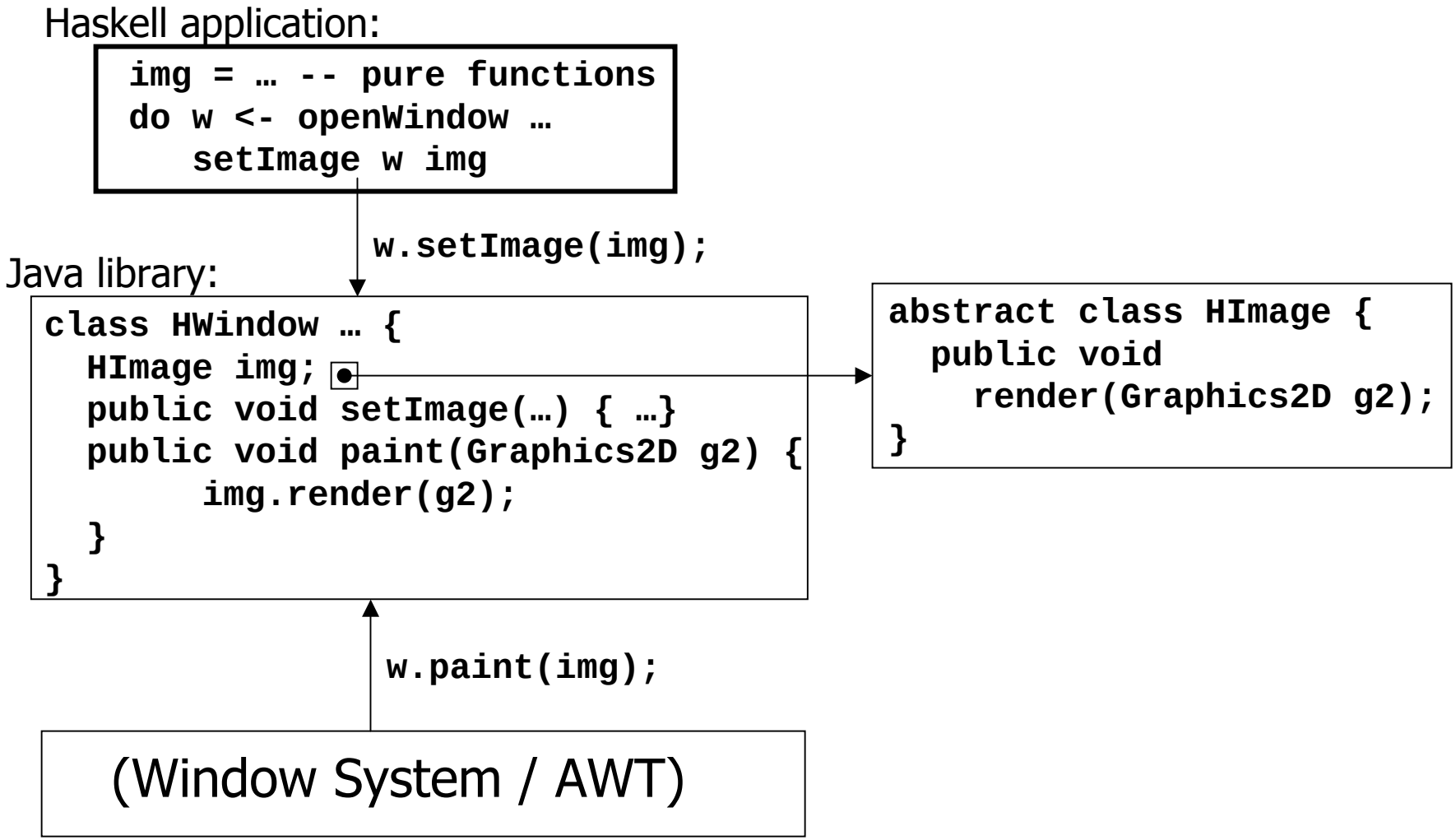
- An immutable object is one whose fields are never updated.
- But an immutable object's methods can mutate *other* objects (passed as arguments).

…such as a **`Graphics2D`** (imperative graphics context):

```
public abstract class HImage {
   public abstract void render(Graphics2D g2);
   public abstract Rectangle getBounds();
}
```

- This is exactly like treating an IO action as a first class value in Haskell.

Runtime View



Implementing HImage

- We provide one concrete implementation class for each HImage-returning function in Haven API:

```
-- monochrome :: Color -> HImage
public class MonochromeHImage extends HImage {
   Color c;
   public MonochromeHImage(Color c) { this.c = c; }
   public void render(Graphics2D g2) { g2.fill(c); }
   …
}
-- imgCrop :: (Shape a) => a -> Image -> Image
public class CroppedHImage extends HImage {
  Shape s;  HImage child;
  public CroppedHImage(Shape s,HImage child) { … }
  public void render(Graphic2D g2) {
    g2.setClip(s);
    child.render(g2);
  }
}
-- compositeHImage, …
```

Other Goodies

- So Far: HavenCore (some Java req'd)
  - clear, precise signatures for combinators.
- HavenUtils (pure Haskell):
  - Utilities like: `fillShape s c = imgCrop s (monochrome c)`
- Pictures (pure Haskell):
  - Precision of HavenCore can be a bit cumbersome.
  - **`Picture`** is an Image parameterized by RenderAttributes (RA):

```
type Picture = EnvM RA Image
withFont :: Font -> Picture -> Picture
picText :: String -> Picture
…
```

  - TODO: try implicit parameters as an alternative.
- Layout (pure Haskell):
  - **`Placeable`** type class for simple spatial composition, using bounding rectangles. Instances for **`Path, Image, Picture`**.

Some Demos

Sierpinski Gasket

```
sierpinskiTri :: Point -> Double -> Path
sierpinskiTri pt size =
  if size <= minSize
  then rightTri pt size
  else let size2 = size / 2
           (x,y) = pointXY pt
           t1 = sierpinskiTri pt size2
           t2 = sierpinskiTri (point x (y-size2)) size2
           t3 = sierpinskiTri (point (x+size2) y) size2
       in t1 <++> t2 <++> t3

sierpinski :: Picture
sierpinski =
  let spath = sierpinskiTri (point 250 250) 500
  in place origin $ withColor blue $ picFill spath
```

Zhanyong's FRP Logo

```
frpLogo :: Picture
frpLogo =
  let f_SS_b = font "SansSerif" bold 96
      …
      txt1_1 = withColor red $ withFont f_SS_b (picText "F")
      txt1_2 = withColor cyan $ withFont f_SS (picText "unctional")
      txt1  = place origin $ txt1_1 `hcomp` txt1_2
      …
      txt  = place origin $ rtxt1 `vcomp` txt2 `vcomp` txt3

      curve1 = picFill (cubicCurve (point 170 20) (point 350 90)
                   (point 400 150) (point 100 270))
      transpCurve1 = withAlpha 0.3 $ withColor blue curve1 …

      colorBG = withAlpha 0.05 $ withColor green picMonochrome
      whiteBG = withAlpha 1.0 $ withColor white picMonochrome

      pic = (transpCurve1 <++> transpCurve2) <++> colorBG <++> txt <++>
   whiteBG
  in pic
```

My Haven Logo

```
-- A rectangle with a horizontal gradient
hgRect :: Color -> Color -> Double -> Double -> Picture
hgRect lc rc w h =
  let p = paintGradient (point 0 (h/2)) lc (point w (h/2)) rc False
  in withPaint p $ picFill (rectangle origin w h)

box1 = yPlace 50 $ dgRect white red 80 60

boxes = box1 <++> box3 <++> box2

-- A thick pen to use for drawing the lines:
lpen = pen 3.0 CapButt JoinRound miterDefault

vline :: Picture
vline = withPen lpen $ picOutline (line origin (point 0 100))

slide = xyPlace 160 20 ttext
   <++> decor
   <++> xyPlace 300 180 rectPic
   <++> xyPlace 220 150 cpic
   <++> xyPlace 160 100 btext
   <++> xyPlace 225 125 bodyPic
   <++> bg
```

...Performance?

- Haven't done any serious measurement.
- Good enough for interactive use.

Haven Status / Conclusions

- portable, thanks to GCJNI / Java.
- Presents a nice, functional API to the programmer.
- TODO: API Documentation
  - but there are Release Notes, examples and type signatures.
- Very interested in Users / Feedback!
- Available From:

  `http://www.haskell.org/haven/`