Pictures: A simple structured graphics model

Sigbjorn Finne and Simon Peyton Jones,
Department of Computing Science,
University of Glasgow.

Abstract

We present in this paper a simple, device-independent model for describing two-dimensional graphics using a functional language. Graphical scenes, or pictures, are represented as values that functions can manipulate and inspect to create new values. Complete pictures are constructed by repeatedly composing such picture values together using *picture combinators*. A novel aspect of the model presented is its use of *structured translation* to abstractly express the geometric composition of arbitrary pictures.

The structured graphics model presented has been implemented in Haskell, and we also give an overview of a general rendering framework for traversing a picture value. Applications of this renderer include both output to various graphical systems, testing for picking or selection of a picture and the computation of the bounding box of an arbitrary picture. The graphics model forms the basis for all graphical output in a user interface framework being developed in Haskell.

1 Introduction

Graphics is an under-used resource in applications. To avoid having to delve into inch-thick manuals to produce the simplest of outputs, textual output is often used, for better or worse. This is a somewhat unfortunate situation, as graphics capabilities aren't made use of by applications to better visualise their output. Simple graphics should be simple to express and make use of in a program[2].

We present in this paper a simple structured graphics model for expressing two-dimensional graphical scenes using a functional language. The declarative model is expressed using a data type called *Picture*, and the paper presents and explores the expressiveness of the model, comparing and contrasting it with existing techniques for expressing two-dimensional graphics.

The idea of using a functional language to describe graphics is by no means a novel one [2, 3, 12, 4, 6, 1]. Our design has been influenced by this previous work and builds on it by providing a more abstract account of picture composition. As an example, a basic operator like horisontal tiling of two pictures is not provided as a primitive in our model, instead a mechanism called *structured translation* is used to abstractly express the horisontal tiling combinator in terms of more primitive constructs. The advantage of providing such abstract glue for writing *picture combinators* is that the set of composition operators can readily be extended by the programmer if the set already provided doesn't fit the task at hand.

The graphics model presented is independent from a particular graphics system, and a generic renderer that traverses the representation of a picture to perhaps produce graphical output has been implemented. The renderer is parameterised on the low-level commands needed to draw a fixed set of primitives, so to map the picture type to a new graphics system, you just have to provide an instance for each of these drawing methods (Section 5.) The functional renderer will then perform the computations required to map the picture description into a sequence of calls to the drawing operations it is given.

The graphics model and accompanying renderer forms the basis for graphical output to both workstation and printer in Haggis[5], a user interface framework written in Concurrent Haskell[10]. The figures given in this paper are pictures produced using the system described here, mapping the picture to Encapsulated PostScript[9][1] after having first viewed it on a workstation display using Haggis.

2 Motivation

Before going any further, it is worth being a bit more precise about the term graphics and the use of it throughout this paper. We are concerned here with providing an appropriate programming abstraction for expressing two-dimensional graphical output in a functional language, providing a model with graphic capabilities similar to that provided by systems such as PostScript[9] and MetaPost[8]. The main goal is to provide an abstraction that is convenient and high-level enough for *the programmer* rather than creating a representation that could be used as a meta file picture format for drawing tools. We are not concerned with the description of 3D geometries here, and all the challenges that poses to both programmer and implementor.

What properties would we like such a two-dimensional, functional graphics model to possess?

- *Pictures should be concrete.* Representing a graphical object as a concrete value has its advantages. Apart from isolating and abstracting away the low-level details of a graphics system interface, having a value representing a picture allows a program to manipulate and inspect pictures to create new values. Representing a picture as a higher-order function or procedurally by a sequence of drawing actions is too opaque, as their structure cannot be unravelled by other functions, only composed together in a fixed number of ways.

  Instead we raise the level of abstraction and provide a declarative description of pictures, where a graphical scene is described concretely via a recursive data type that can be freely combined and

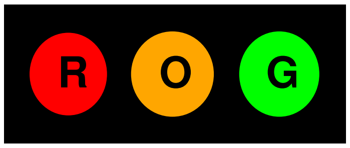
[1]PostScript is a trademark of Adobe Systems Incorporated.

manipulated by functions. The application operate in terms of the objects that they want to output and it is only at the rendering stage that the value representing the picture is actually mapped into a sequence of drawing actions.

- *Simple application interface.* Commonly used graphics programming interfaces have often painfully complex interfaces for accessing the graphical hardware[2]. The multitude of arguments required just to get simple graphics output from an application leads to either poorly structured programs or the avoidance of graphics completely. Simple graphics should be simple to express and integrate into an otherwise non-graphical application. Expressing graphics declaratively using a functional language focuses on the picture values that the application manipulates, shielding the programmer from having to deal with the complexities of a particular system.
- *Composition.* The ability to combine parts to make up a whole is desirable feature when describing two-dimensional graphics, as scenes can often be partitioned into a clear hierarchical structure. A compositional model provides a simple way of building these scenes by repeatedly combining existing pictures to construct even 'bigger' ones. The picture abstraction should also provide mechanisms for extending the range of *picture combinators*.
- *Device independence.* In the interest of portability, the graphics model should be as far as possible independent of any rendering model. The motivation for this independence is the wish to be able to provide a graphics description that could just as easily be used to provide output to a printer as to a workstation window.

3 Representing Pictures

Graphical scenes are represented as values of type `Picture`, a recursive data type that contains both the geometric primitives supported, and the core set of operators that transform and combine `Picture`s together. Rather than go through the whole `Picture` (Appendix A has the complete definition), type from top to bottom, let's instead focus on a simple example of a picture and the issues and features that highlight for the `Picture` type. The picture is that of a traffic light (shown here horizontally to save space):



3.1 Drawing primitives

The `Picture` type has a set of basic drawing primitives (complete list can be found in the definition of the type in Appendix A.) The circles used for the traffic light are straightforwardly expressed:

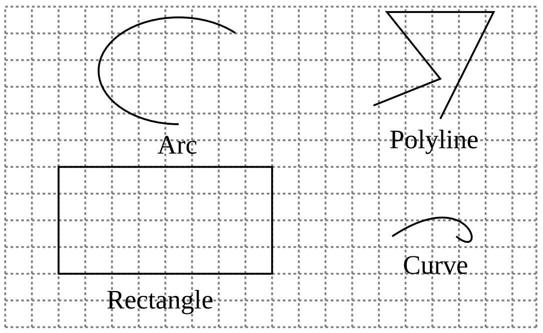


Figure 1: Drawing primitives supported by `Picture` data type.

```
circle :: Unit -> Picture
circle radius = Arc (radius,radius) (0,2*pi)
```

The `Picture` type uses printers' points (i.e., 72 points $\approx$ 1 inch) for all the lengths and sizes (which we represent with the type `Unit`), so `circle 36` returns a circle with radius of 1/2 an inch. The circle is not primitive, but a special case of the circular arc. The `Arc` primitive is parameterised on the length of the major and minor axes plus the start angle (in radians), and the number of radians to sweep through to reach its final angle. Each picture value is expressed in terms of its local coordinate system, and the origin of the `Arc` coincides with the origin of its coordinate system.

Rectangles and text have their own `Picture` primitives, `Rectangle (40,40)` is a square 40 points wide, with the lower left hand corner of the rectangle as origin. Similarly, `Text "hello"` is a picture of the `"hello"` string, its origin equal to the origin of the first character in the string.

Other drawing primitives supported are poly-lines, Bézier curves and rasters (i.e., two-dimensional array of coloured points) etc., some of them are shown in Figure 3.1, the full list can be found in Appendix A. The inclusion of both `Rectangles` and poly-lines as drawing primitives highlights a design tradeoff, what should be primitive in a declarative representation of pictures? The one extreme of just providing points or lines as primitives and operations to support the combination of them (as done by [1, 6]), is low-level and not a descriptive enough representation of the pictures we want to express and render. At the other end, making every possible known two-dimensional geometric shape a primitive results in a language that is very precise, but complex. The reason for making rectangles into primitives rather than express them in terms of poly-lines, is that they are commonly occurring forms, so functions that want to inspect `Picture` values should readily be able to identify rectangles. Similarly, circles or ellipses were not made primitives, since testing whether an `Arc` value is a circle is computationally less expensive than interpreting the geometric meaning of a poly-line `Picture` value as a rectangle.

3.2 Geometric Transformations

To geometrically transform a `Picture` value, the `Transform` constructor can be applied to produce a new `Picture`,

```
data Picture =
 ...
 | Transform Transformation Picture
 ...
```

`Transform` returns a new picture by applying a transformation to an existing picture. `Transformation` is an abstract data type for 2D transformations, allowing both uniform (scaling, rotating) and non-uniform transformations (shearing, reflection) to be expressed. Some of the most commonly used modelling transformation functions are:

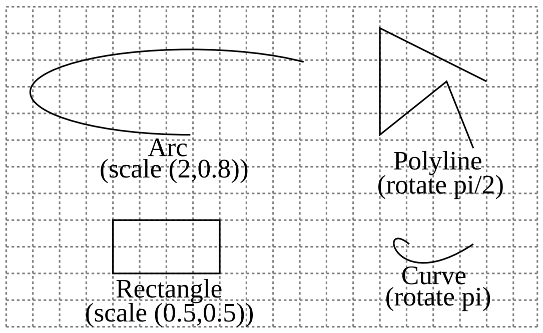
```
type Scaling = (Unit,Unit)
type Radians = Unit
type Translation = (Unit,Unit)

identity :: Transformation
scale    :: Scaling -> Transformation
rotate   :: Radians -> Transformation
xlt      :: Translation -> Transformation
combine  :: Transformation -> Transformation -> Transformation
```

A picture combinator for doubling the size of an arbitrary picture is now simply an application of `Transform`:

```
doubleSize :: Picture -> Picture
doubleSize = Transform (scale (2,2))
```

when `doubleSize (circle 20)` is rendered, a circle with a radius of 40 (points) will be displayed. As a further example of geometric transformation, here are the drawing elements of Figure 3.1, transformed in various ways:



When rendering a `Picture` value, the modelling transformation that `Transform` applies to a picture will be combined with the accumulated transformation matrix the renderer function keeps track of, so `quadSize`

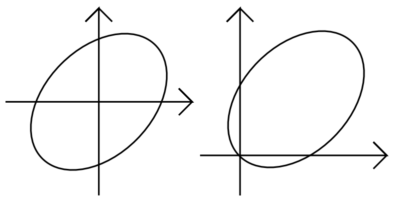
```
quadSize :: Picture -> Picture
quadSize pic =
 Transform tr (Transform tr pic)
 where
  tr = scale (2,2)
```

scales a picture by a factor of four by applying a scaling transformation twice.

3.3 Structured translation

Every picture value is expressed within its own local coordinate system. The geometric `Transform` constructor applied to a `Picture` returns a new picture with a transformed local coordinate system, so, for instance, `doubleSize` doubles the scaling in both directions. Since scaling and rotation are about the origin of the local coordinate system, we often need to translate the picture prior to performing a scaling or rotation:

```
ellipseA =
 Transform (rotate (pi/4)) $
 ellipse (30,20)
ellipseB =
 Transform (rotate (pi/4)) $
 Transform (xlt (30,0)) $
 ellipse (30,20)
```



To rotate around the leftmost point on an ellipse (rightmost picture), we first have to translate the ellipse along the X-axis before rotating, as seen in the definition for `ellipseB`. For ellipses, rotation around the centre is straigtforward, as the origin of the ellipse picture is the same as the origin of its local coordinate system.

However, to correctly translate the ellipse in `ellipseB` depended on knowing its actual size, which makes it hard to write a general picture combinator for rotating a picture around the leftmost or western point of its bounding box, without some extra support. Rather than providing a function that computes the bounding box (i.e., the smallest rectangle that encapsulates the picture shape), we provide a mechanism called *structured translation*:

```
data Picture =
  ...
  | Move Offset Picture
  ...                                  data CompassDirection
data Offset =                           = West  | NorthWest
    OffDir CompassDirection              | North |
  | OffPropX Double -- [0.0..1.0]          ...
  | OffPropY Double -- [0.0..1.0]        | South | Centre
```

Structured translation allows you to abstractly translate a picture with respect to its bounding box, leaving it up to the renderer to compute the actual translation amount. Generalising the rotation performed by `ellipseB` becomes then

```
westRot :: Radians -> Picture -> Picture
westRot rad pic =
 Transform (rotate rad) $
 Move (OffDir West) pic
```

`westRot` translates `pic` such that its bounding box is shifted to the right of the vertical axis and centred around the horisontal axis. The structured translation constructor `Move` is parameterised on `Offset` which is either a translation to one of eight points on the bounding box perimeter (or the centre), or a proportional translation in either X or the Y direction.

Nested applications of the `Move` constructor are handled as follows:

```
Move dir1 (Move dir2 pic) = Move dir1 pic
```

i.e., since the `Move` constructor does not alter the size of a picture's bounding box, the inner application of `Move` can safely be ignored since the outer `Move` will potentially undo whatever translation the inner `Move` did. This useful rule is made use of by the rendering function (see Section 5) to 'simplify' a `Picture` value before rendering.

3.4 Graphical transformations

Another type of transformation is graphical, where you want to change or set the graphical attributes that a particular picture is to be drawn with. For example, to create a filled, green circle for our traffic light:

```
greenCircle :: Unit -> Picture
greenCircle rad =
 Pen [PenForeground green,
      PenFill True] $
 circle rad
```

we apply the `Pen` constructor to a picture describing a simple circle. The constructor returns a new picture by adding a set of `PenModifier` to a `Picture`:

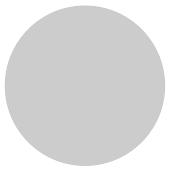
```
data Picture =
 ...
 | Pen PenModifier Picture
 ...
```

such that when the circle is rendered, the graphical attributes specified in the `PenModifier` value will be used, i.e., that the foreground colour should be green and that the closed area the circle describes should be filled in. The `PenModifier` value in the `Pen` constructor consist simply of a list of graphical attributes, see Appendix A for complete list.

The elements of the `PenModifier` list gives you a fine-grained control over how you want to draw a picture, but sensible defaults are defined for all values, so the `Pen` constructor is only required if you want to override these default values.

In the case of nested application of `Pen` constructor, the `PenModifier` attributes are 'lexically' scoped, i.e., *the (attribute,value) pair set in an application of* `Pen` *overrides any previous value set for that attribute.* To illustrate, when drawing the following picture value

```
 picture =
  Pen [PenFill False,
       PenForeground black] $
  Pen [PenFill True,
       PenForeground grey80] $
  circle 30
```



should give you the picture on the right. When the circle is rendered, the foreground colour will be `grey50` and the circle should be filled, since the innermost application of `Pen` override the attribute values of the outer application.

Note that the graphical attribution done by `Pen` creates a new `Picture` value, and avoids having to use some shared, mutable graphics state. The graphical transformer, `Pen`, simply associates a set of graphical attribute values with a picture.

Representing the three lights in the traffic light becomes now just:

```
filledCircle :: Colour -> Unit -> Picture
filledCircle col rad =
 Pen
   [PenForeground col,
    PenFill True] (circle rad)

redLight, orangeLight, greenLight :: Unit -> Picture
redLight    = filledCircle (red::Colour)
orangeLight = filledCircle (orange::Colour)
greenLight  = filledCircle (green::Colour)
```

3.5 Composing pictures

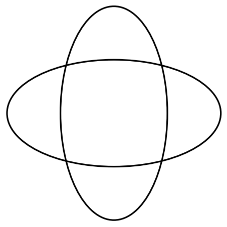
To build the traffic light picture we presented at the start of this Section, the different `Picture` values for the lights will have to be combined together. The `Picture` type provides three basic composition operators:

- *Overlays* take the sum of two pictures, combining two `Picture` values by aligning their origins and drawing one on top of the other:

  ```
  data Picture =
   ...
   | Overlay Picture Picture
   ...
  ```

  i.e., `Overlay picA picB` is a picture formed by putting `picA` on top of `picB`, so that their origins match up:

```
picture =
 Overlay
   (ellipse (40,20))
   (ellipse (20,40))
```



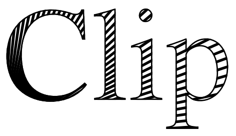
The bounding box of the combined picture is the union of the bounding boxes of the two pictures.

- *Clipping* combines two pictures by aligning their origins like `Overlay`, but interprets one picture as defining the clip mask to be used when drawing the second:

  ```
  ...
  | Clip Picture Picture
  ..
  ```

  `Clip clip clipped` is a new picture that clips the second picture by the clip mask defined by the first:

```
picture
 = Clip
    (Pen largeFont (text "Clip"))
    lines
```



  The bounding box of the constructed picture is equal to the bounding box of the picture describing the clip mask.

- *Constrained overlay.* The picture union operator, `Overlay`, allows you to combine two `Picture` values, but the composition does not impose any constraints between the sizes of the two `Picture`s combined. Having an overlay operator that imposes such constraints between the two pictures turns out to be quite useful in a number of cases, e.g.,

  ```
  inBox :: Picture -> Picture
  ```

  is a picture combinator that puts a bounding rectangle around an arbitrary picture. This could combinator could of course be expressed if we had a function for computing the bounding box of a picture, but in the same way as in Section 3.3, we introduce a higher-level mechanism for expressing size constraints between two pictures being combined:

  ```
  ...
  | ConstrainOverlay RelSize RelSize
                     Picture Picture
  ```

```
 data RelSize =
    RelNone
  | RelFixed Bool Int
  | RelProp  Bool Double
```

`ConstrainOverlay RelNone (RelProp True 2.0) picA picB` is a picture that when rendered, will align the origins of `picA` and `picB`, drawing `picA` on top of `picB`. The operator will also scale `picB` in the Y direction such that the size of its bounding box will double that of `picA` along this axis. (The boolean used in `RelProp` indicates whether it is `picA` that should be related to `picB` or vice versa.) The `RelSize` data type contains the different types of size constraints we can place between the two dimensions, `RelNone` indicates that no size constraints should be imposed.

The `ConstrainOverlay` constructor provides a superset of the functionality of `Overlay`,

```
Overlay = ConstrainOverlay
                RelNone RelNone
```

but we choose to provide the `Overlay` constructor separately.

Combining the `Overlay` operator with the structured translation operator in Section 3.3, picture combinators that tile two pictures together can be expressed:

```
beside :: Picture -> Picture -> Picture
beside picA picB =
 Overlay
  (Move (OffDir East) picA)
  (Move (OffDir West) picB)

above :: Picture -> Picture -> Picture
above picA picB =
 Overlay
  (Move (OffDir South) picA)
  (Move (OffDir North) picB)
```

The `beside` combinator overlays two pictures, but translate their local origins such that `picA` will be shifted to the left of the vertical axis and `picB` wholly to the right. `above` uses the same trick, but this time the translation is with respect to the horisontal axis.

As an example of these various composition operators, we can finally present the construction of the traffic light example presented at the beginning of this Section, starting with a combinator for placing an arbitrary text string within a coloured oval:

```
light :: Colour -> String -> Picture
light col lab =
 ConstrainOverlay
     (RelFixed True 20)
```

```
    (RelFixed True 20)
    (withColour black $ centre $ Text lab)
    (filledCircle col 2)
```

`light` will centre the text string `lab` within an ellipse that will have a horisontal and vertical extent 20 units bigger than that of the extent of the picture representing the string. Using this combinator we can construct the pictures for the individual lights:

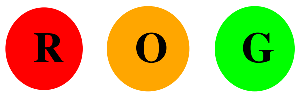
```
redTLight    = light red    "R"
orangeTLight = light orange "O"
greenTLight  = light green  "G"
```

To align the the lights horisontally, we want to use the horisontal tiling operator `beside`, but have to add some 'air' between the lights first:

```
besideSpace :: Unit -> Picture -> Picture -> Picture
besideSpace spc picA picB =
 beside
   picA
   (Transform (xlt (spc,0)) $
    moveWest picB)
```

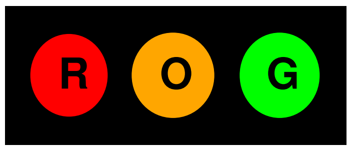
`besideSpace` using a `Transform` constructor to enlarge the bounding box of `picB` before invoking `beside`. The lights then become just:

```
lights =
 foldr1
   (besideSpace 10)
   [redTLight, orangeTLight,
    greenTLight]
```



The final step is then adding a black background for the casing for the traffic lights:

```
trafficLight =
 ConstrainOverlay
   (RelFixed True 20)
   (RelFixed True 20)
   (Move (OffDir Centre)
         lights)
   (Move (OffDir Centre)
         (Rectangle (2,2)))
```



The example, while small, demonstrate the compositional programming style that follows naturally, where complete `Picture`s are formed by repeatedly applying *picture combinators* to existing `Picture`s.

4 Example

To further demonstrate and bring together the various features that the `Picture` data type provides, let's consider the problem of plotting 2D

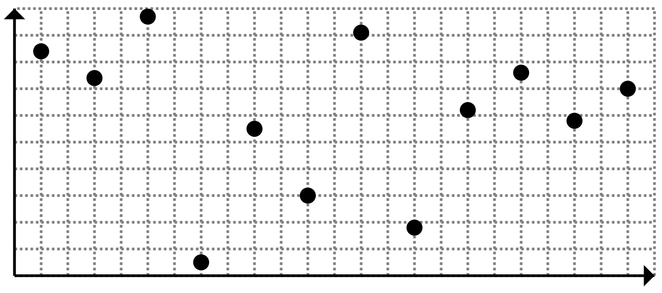
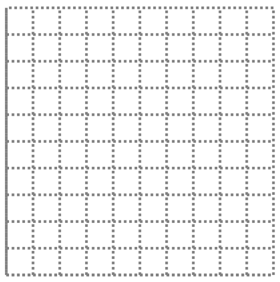


Figure 2: `graph (scatter) dataPts` - scatter plot of annual data

graphs. A common situation is to have a set of data generated by a program that we want to visualise quickly using a graph. For the purpose of this example, let us assume that the data measure the annual distribution of some value, producing output like Figure 2. The X axis represents the months and the Y axis the values we've measured each month in. The `Picture` representing this graph consists of several smaller pictures joined together, starting with the gridded background:

```
grid :: Size -> Size -> Picture
grid (w,h) (stepx,stepy) =
 let
  pen =
   [PenForeground grey50,
    PenLineStyle (LineOnOffDash 1 1)]
  no_lines_x = h `div` stepx
  no_lines_y = w `div` stepy
 in
 Pen
     pen $
 Overlay
   (Move (OffDir Centre) $
    Rectangle (w,h))
   (overlay
     (Move (OffDir Centre) $
      hlines stepx no_lines_x w)
     (Move (OffDir Centre) $
      Transform (rotate (pi/2)) $
      hlines stepy no_lines_y h)
```



The `grid` function, given a size and spacing between the grid lines in both directions, returns a `Picture` of the grid, built by overlaying horisontal and vertical lines. To make the grid-lines appear discretely in the background, we apply a pen modifier that dashes the lines and renders them in grey (see Appendix for definition of the graphical attributes . The picture of the horisontal lines `hlines` is also a combined picture:

```
hlines :: Unit -> Unit -> Unit -> Picture
hlines spc no x =
 nabove
    (map (Transform (xlt (0,spc)))
         (replicate no $ hline x))

nabove :: [Picture] -> Picture
nabove = foldr (above) NullPic -- empty picture
```

The horisontal lines are composed out of a collection of lines arranged vertically using `above`. To achieve the necessary spacing between the lines, each line is translated so as to enlarge the bounding box the `above` uses to compute the geometric arrangement between two pictures.

The axes of the coordinate system is also created by combining smaller pictures together, this time two arrowed lines:

```
axes :: Size -> Picture
axes (w,h) =
 overlay
   (leftArrowLine w)
   (upArrowLine   h)
```

The arrowed lines can also be subdivided into a picture element for the arrow line and head that has been combined together, but for lack of space we will leave out their definition here.

To get the picture of a gridded coordinate system, we simply overlay the picture returned by `axes` with that for the grids, making sure of moving the reference point for the grid to its lower left corner, so that the gridding starts at the origin of the axes:

```
cartesian :: Size -> Size -> Picture
cartesian sz steps =
 overlay
   (axes sz)
   (Move (OffDir SouthWest) $
    grid sz steps)
```

To plot data points within the coordinate system, the picture(s) representing the points just have to be placed on top. Here's how a scatter plot of a set of coordinates could be done:

```
scatter :: [Coord] -> Picture
scatter = noverlay $ map (plotAt)
 where
  plotAt pos =
   Transform
    (xlt pos)
    (filledCircle 2)

noverlay :: [Picture] -> Picture
noverlay = foldr (overlay) NullPic
```

The different points are plotted by translating a circle to each data point and then overlaying all the resulting pictures. Since overlaying is performed by matching up the origins of two pictures, and the points to be plotted are all expressed within the same coordinate system, the pictures will also have the same origin. The resulting plot can then be superimposed on a coordinate system to produce the plot in Figure 2:

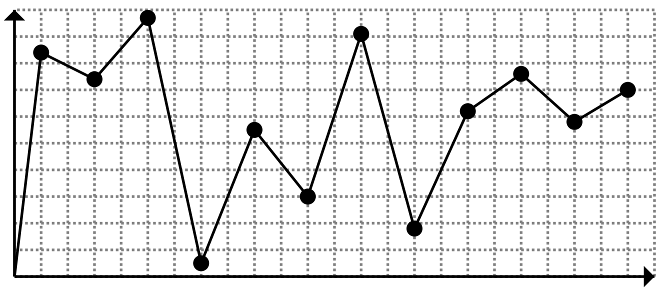
```
graph :: ([Coord] -> Picture)
      -> [Int]
      -> Size
      -> Size
      -> Picture
graph plot pts size steps@(dx,dy) =
 let
  coords = zip pts [dx `div` 2,dx..]
 in
 overlay
   (plot coords)
   (cartesian size steps)
```

The `graph` takes a function for producing the plot of the supplied data together with the data points themselves and a size plus grid steps. For the purpose of this example, we assume that the size and data points are in the same range, adding the code that checks and appropriately scales the data to fit has been omitted for reasons of space.

Now let's change the plot by having the points connected up via a solid line instead:

```
solid :: [Coord] -> Picture
solid ls =
 overlay
    (polyline ls)
    (scatter ls)
```

The scatter plot as produced with `scatter` is overlaid with a poly-line connecting all the data points up. Using `solid` in a call to `graph` will produce output like this:



4.1 Histogram

Instead of plotting data points, we could instead plot the data using a histogram and to make the resulting graph a bit more understandable, add month labels to the X-axis. The month labels can be added by overlaying the X axis with the appropriate labels:

```
xAxis :: [String] -> Int -> Int -> Picture
xAxis labels sz spc =
 overlay
  (leftArrow sz)
  (Move (OffDir NorthWest) $
   noverlay
     (zipWith (\ p pic -> Transform (xlt (p,-15)) pic)
              [spc',(spc+spc')..]
              (map (label) labels)))
 where
  spc' = spc `div` 2
  label str =
    Transform (rotate pi/2) $
    Move (OffDir East) $
    text str
```

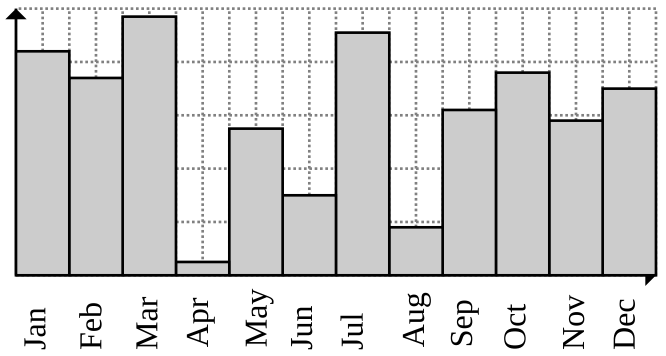
The labels in the X direction are placed on top of the axis by rotating each label 90 degrees clockwise beforehand. The rotated labels are then placed along the X axis. To incorporate the labelled axis, the functions `cartesian` and `axes` have to be altered to thread the labels through to `xAxis`, but we will leave out the details here.

Plotting a histogram instead of a scatter-plot is straightforward, just substitute `scatter` with `histo` in a call to `graph`:

```
histo :: [Int] -> Int -> Picture
histo pts spc =
 foldl
   (besideB)
   NullPic
   (map (bar) ls)
 where
  bar sz =
   Move (OffDir South) $
   Overlay
     (Rectangle (spc,sz))
     (fillColour grey80 $
      Rectangle (spc,sz)))

besideB :: Picture -> Picture -> Picture
besideB picA picB =
 Overlay
   (Move (OffDir SouthEast) picA)
   (Move (OffDir SouthWest) picB)
```

The bars are created by going through the data points left to right. Note that instead of using `beside` to combine the bars together, we use the combinator `besideB` to align the bars by their bottoms instead. Visualising the data using `histo` will then produce output like the following:



The graphing example shows that using the `Picture` data type, it is relatively easy to write application-specific combining forms for generating drawings. While this is a toy example, an interesting experiment would be to try to build a complete graph drawing library using `Picture`s and a functional language, and see how well the structured graphics model scales to larger examples.

5 Rendering Pictures

To convert a `Picture` value into actual output, a generic rendering framework fo the type have been implemented. The rendering function is parameterised on both `Picture` and the `Painter` to invoke to handle the drawing of the drawing primitives:

```
render :: Painter -> Picture -> IO Rectangle
```

Before traversing the `Picture`, the renderer tries to simplify the `Picture` value by removing constructs that does not contribute (see Section 3.3 for how nested applications of the `Move` constructor can be removed.) After having walked over the `Picture` structure and performed I/O actions to draw the primitives, the function returns the bounding box of the picture just rendered. The `Painter` is a dictionary of methods for, amongst other things, drawing the primitives:

```
data Painter
 = Painter
     ...
      -- drawText str ctm
     (String -> Transformation -> IO ())
      -- drawRectangle sz ctm
     (Size -> Transformation -> IO ())
      -- drawEllipse (w,h) (a1,da) ctm
```

```
    (Size -> Angles -> Transformation -> IO ())
    ...
```

When the renderer encounters one of the primitives mentioned in Section 3.1, it looks up and invokes the corresponding method in the `Painter`. Currently, two graphical `Painter`s exist for producing output in PostScript and to Haggis [5], but the `Painter` interface has also been used to implement picking, i.e., testing whether a point intersects the picture, and to incrementally update parts of a `Picture` structure.

6 Related work

As stated in the introduction, the work reported here build on previous approaches to graphics using a functional language. One of the earliest attempts at using a functional language to express graphics was Henderson's functional geometry[6]. Using Escher's square limit as an example, a set of basic tiles were repeatedly combined together using a small set of tiling picture combinators. The repertoire of primitive drawing elements were restricted to lines (a simplification which Arya's functional animation also uses, [1]), each of which had to be explicitly placed within some tile coordinate system. Only combinators for horisontal and vertical tiling were provided. The `Picture` data type extends this early work by providing a fuller set of drawing primitives and picture transformers, and through the use of structured translation, the set of composition mechanisms can easily be extended, c.f., `above` and `beside`.

Several 'functional' systems have made use of PostScript[9] as the basic drawing model, layering functional abstractions on top it[4, 12]. These approaches make good use PostScript's page description model, but forces the programmer to use PostScript's model of stencil and paint for describing the basic picture elements. While powerful, the inherent statefulness of the stencil and paint model can lead to unexpected results when used from within a lazy functional language.

Although the `Picture` graphics model differs significantly from the PostScript model, a module for describing PostScript stencil paths in terms of `Picture`s have been defined:

```
module Path
   (
    Path,
    ...
    currentPoint,  --:: Path  -> Coord
    moveTo,        --:: Coord -> Path -> Path
    rline,         --:: Size  -> Path -> Path
    ...
   )
```

Graphical output is described by incrementally building larger and larger `Path`s. The `Path` module does not provide the full set of features that a PostScript interpreter has, but it shows that the `Picture` type could be used as a basis for creating other graphics abstractions. One interesting point to note is that the Path module elevates the path to a first-class value, something that is not the case for PostScript interpreters.

Another area of related work is the declarative description of graphics using constraint-based systems [11, 14, 8, 7]. Through the use of constraints, relationships between components of a picture can be expressed declaratively. The drawing of a picture is preceded by a pass where the constraint expressions are satisfied. Whether the generality and flexibility that these constraint-based systems offer compared to the Picture data type is worth the additional overhead of solving and maintaining these relationships, is an open issue.

7 Conclusions and Future Work

We have in this paper presented a simple model for expressing structured graphics in a functional language. The Picture type was introduced, providing to the functional programmer a concrete representation of two-dimensional graphical scenes. As an example of the Picture model in action, a set of basic graph drawing combinators were developed on top of the model.

The Picture type offers yet another demonstration of how straightforward it is to define and use 'little languages' in a functional language. By defining a data type Picture containing the core primitives and operators for pictures, full use could be made of the first-class property that values enjoy in a functional language. Using standard combining forms such foldr and map, the repertoire of Picture combinators could then be readily extended. This ability to create such new abstractions via a little data type is not news to a functional programmer, but the graphics model presented hopefully provides a simple abstraction that will make it easier to use graphics from within a functional program.

An interesting area of future work is how to make the Pictures come alive. In Haggis[5], layout combinators exist for interactive widgets, that perform operations similar to the tiling Picture combinators used in this paper, and, ideally, we would like to be able to provide a common set of such combinators, covering both static pictures and interactive objects. We are currently experimenting with a basic mechanism for tagging parts of a picture, and through a Painter (see Section 5) that instead of generating drawing output, tests and records the picking of tagged parts of a Picture, we're able to reuse the generic renderer to perform picking as well as drawing.

Another area for future work would be to try and apply the techniques used here for two dimensional graphics to three dimensions. Promising results have already been achieved by the TBAG system [13], which uses a functional model for building three-dimensional interactive worlds.

References

[1] Kavi Arya. Processes in a functional animation system. In *Proceedings of the 4th ACM Conference on Functional Programming and Computer Architecture*, pages 382–395, London, September 1989.

[2] Joel F. Bartlett. Don't Fidget with Widgets, Draw! Technical Report 6, DEC Western Digital Laboratory, 250 University Avenue, Palo Alto, California 94301, US, May 1991.

[3] Brian Beckman. A scheme for little languages in interactive graphics. *Software-Practice and Experience*, 21(2):187–207, February 1991.

[4] Emmanuel Chailloux and Guy Cousineau. Programming Images in ML. In *Proccedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[5] Sigbjorn Finne and Simon Peyton Jones. Composing Haggis. In *Proceedings of the Fifth Eurographics Workshop on Programming Paradigms in Computer Graphics*, Maastrict, Netherlands, September 1995.

[6] Peter Henderson. Functional geometry. In *ACM Symposium on LISP and Functional Programming*, pages 179–187, 1982.

[7] Allan Heydon and Greg Nelson. The Juno-2 Constraint-Based Drawing Editor. Technical Report 131a, DEC Systems Research Center, Palo Alto,CA, December 1994.

[8] John Hobby. A User's Manual for MetaPost. Technical report, Bell Labs, 1994.

[9] Adobe Systems Inc. *PostScript language reference manual.* Addison Wesley, second edition, 1990.

[10] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent Haskell. In *ACM Symposium on the Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996.

[11] Donald E. Knuth. *TEX and METAFONT, New Directions in Typesetting.* Digital Press and the American Mathematical Society, Bedford, MA, 1979.

[12] Peter Lucas and Stephen N. Zilles. Graphics in an Applicative Context. Technical report, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120-6099, July 8 1987.

[13] Greg Schechter, Conal Elliott, Ricky Yeung, and Salim Abi-Ezzi. Functional 3D graphics in C++ - with an object-oriented, multiple dispatching implementation. In *Proceedings of the 1994 Eurographics Object-Oriented Graphics Workshop.* Eurographics, Springer Verlag, 1994.

[14] Christopher J. van Wyk. A High-Level Language for Specifying Pictures. *ACM Transactions on Graphics*, 1(2):163–182, April 1982.

A Complete Picture type

The complete definition of the `Picture` is as follows:[2]

[2]Note that the `Picture` type is an instance of the `Text` class, which means that it can directly be used as a metafile format for graphical output as well.

```
type Unit  -- repr. of a pr. points.
type Size  = (Unit,Unit)
type Coord = (Unit,Unit)
type Angles = (Unit,Unit)
type Translation = (Unit,Unit)

data Picture
 = NullPic | Point | Text String
 | PolyLine [Translation] | Rect Size
 | Arc    Size Angles | Curve Coord Coord Coord
 | Raster Raster      | Pen PenModifier Picture
 | Move Offset Picture
 | Transform Transform Picture
 | Tag Tag Picture
 | Overlay Picture Picture
 | ConstrainOverlay RelSize RelSize Picture Picture
 | Clip Picture Picture
   deriving (Eq,Text)

data RelSize
 = None | Fixed Bool Int | Prop  Bool Double
   deriving (Eq, Text)

type Tag = Int

data Offset
 = OffDir CompassDirection
 | OffPropX Double
 | OffPropY Double
   deriving (Eq,Text)
```

A.1 Graphical attributes

The Pen constructor associates a set of graphical (attribute,value) pairs with a picture. The attributes currently supported are (the definition of the types used by some of the attributes have been elided for lack of space):

```
type PenModifier = [PenAttr]
data PenAttr =
 | LineWidth Ixont
 | Foreground Colour
 | LineStyle LineStyle  -- dashed lines or not?
 | JoinStyle JoinStyle  -- for polyline joints
 | CapStyle  CapStyle   -- end point caps.
 | Fill Bool            -- fill picture or not?
 | Invisible Bool       -- should the picture be drawn?
 | Font Font            -- what font to use.
 | Function PenFunction -- blit op to eventually apply
```