A Special-Purpose Language for Picture-Drawing

Proc. USENIX Conf. on Domain-specific Languages, Santa Barbara, Oct. 1997, 297–310

Samuel N. Kamin* David Hyatt†
Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801
{*s-kamin,d-hyatt*}*@uiuc.edu*

Abstract

Special purpose languages are typically characterized by a type of primitive data and domain-specific operations on this data. One approach to special purpose language design is to embed the data and operations of the language within an existing functional language. The data can be defined using the type constructions provided by the functional language, and the special purpose language then inherits all of the features of the more general language. In this paper we outline a domain-specific language, FPIC, for the representation of two-dimensional pictures. The primitive data and operations are defined in ML. We outline the operations provided by the language, illustrate the power of the language with examples, and discuss the design process.

1 Introduction

FPIC is a special-purpose language for drawing simple pictures. It was built by defining types and functions in the functional language Standard ML [6]. This method of construction is easy and results in a language with many useful features. In addition to being concise for small examples, FPIC is powerful enough to allow the programming of large programs and program libraries, an area in which many special-purpose languages are weak.

Functional programming has been characterized in many ways. Our view is that it represents an approach to language design. This approach holds that some mathematical constructs—products, functions, disjoint unions, and others—are fundamental in computing and should be well supported in programming languages. This support means allowing the creation of "first-class" values of each type, that is, values not subject to arbitrary restrictions based on the type. It also means providing operations appropriate to those types in a concise, non-bureaucratic form.

In our view, this approach to language design is perfectly suited to the design of special-purpose languages. These languages are usually characterized by a type of primitive data specific to a problem domain, and operations on those data. These data can be incorporated into a language having the type constructions just mentioned. In fact, they can be incorporated into an *existing* functional language; the type constructions will apply to the new data, and the entire language will then become a special-purpose language, with its many other features included "for free."

*Partially supported by NSF Grant CCR 96–19655.
†Current address: Netscape Communications Corp., Mountain View, CA, hyatt@netscape.com.

The principal weakness of many special-purpose languages is that, beyond a concise and natural syntax, and efficient implementation, for the values and operations specific to the domain, overarching language structure is weak. This weakness would be very significantly mitigated if special-purpose languages were routinely designed—or at least prototyped—in the way we have outlined.

This paper is a case study of the design of FPIC according to this philosophy. We describe the process by which we decided what the primitive data were and how they should behave, then describe the language itself with numerous examples. Our emphasis throughout is on the advantages obtained by having the functional language superstructure of Standard ML as part of FPIC.

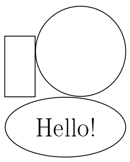
2 Simple FPIC Examples

PIC [4] is a language for drawing simple pictures, such as trees and block diagrams. It has primitives for drawing boxes, circles, and other shapes, with or without labels, and for drawing lines and arrows between them. It also has a facility for naming points on pictures, to be used, for example, as the endpoints of lines and arrows. These constructs are provided in a concise syntax, with a simple language structure (including loops) added on.

FPIC was inspired by PIC. Our goal was to demonstrate that we could benefit by following the language design philosophy outlined in the introduction. That is, by using essentially the same primitive data types and operations as in PIC, but embedding them in a functional language, we could obtain a far more powerful language than PIC and do so at a far lower cost than if we had built the language from scratch.

In this section, we present a few examples to show the principal *primitive* operations of FPIC, making only minimal use of the programming features of Standard ML. Section 5 gives many more examples, emphasizing the utility of the features of ML in combination with the FPIC primitives.

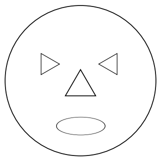
The most basic primitives are those for drawing simple shapes and placing them next to one another:[1]




```
box 1.0 2.0 hseq circle 1.5 vseq
     label "\\Huge Hello!" (oval 2.0 1.0);
```

`hseq` and `vseq` represent the operations of placing pictures next to one another, either horizontally or vertically. (Note that backslashes inside quotes must be doubled.)

Pictures can be moved and otherwise transformed in various ways. In this example, we use ML's name definition facility in the first line. `dtriangle` is a "default triangle;" similarly for `doval` (and `dcircle` and `dbox` later in the paper).



[1]Appendix A gives a concise overview of ML syntax. Appendix B lists all the FPIC primitives used in the examples in this paper, indicating the types of their arguments, whether or not they are infix, and what they return.

```
val nose = dtriangle;
circle 2.5
  seq (nose at (2.0,2.0))
  seq (nose rotate ~90.0 scale 0.7 at (1.2,2.7))
  seq (nose rotate 90.0 scale 0.7 at (3.1,2.7))
  seq doval scaleXY (0.5,0.3) at (1.7,0.7);
```

The `seq` operation simply places pictures on top of one another, without moving them either right or down. The expression *pic* `at` *point* draws the picture *pic* with its reference point (the lower-left corner) at *point*.

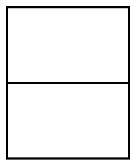
An important feature of PIC, which we have adopted in FPIC, is the ability to name points in a picture and subsequently refer to them. The compass points—`s` for south, `ne` for northeast, and so on, plus `c` for center—are automatically defined for every picture. This allows us to eliminate some of the guesswork in the previous example:

```
val face = circle 2.5;
val facecenter = face pt "c";
val lefteye = nose rotate ~90.0 scale 0.7;
val righteye = nose rotate 90.0 scale 0.7;
val mouth = doval scaleXY (0.5,0.3);
face seq (nose centeredAt facecenter)
     seq (lefteye centeredAt (facecenter -- (1.0,~0.7)))
     seq (righteye centeredAt (facecenter ++ (1.0,0.7)))
     seq (mouth centeredAt (facecenter -- (0.0,1.5)));
```

Named points can be added to a picture. A third way to produce the same "pumpkin face" is to draw the face and name the locations of the facial features:

```
val face =
    let val f = circle 2.5
        val facecenter = f pt "c"
    in namePts f
         [("nosepos", facecenter),
          ("lefteyepos", facecenter -- (0.9,~0.8)),
          ("righteyepos", facecenter ++ (0.9,0.8)),
          ("mouthpos", facecenter -- (0.0,1.5))]
    end;
face seq (nose centeredAt (face pt "nosepos"))
     seq (lefteye centeredAt (face pt "lefteyepos"))
     seq (righteye centeredAt (face pt "righteyepos"))
     seq (mouth centeredAt (face pt "mouthpos"));
```

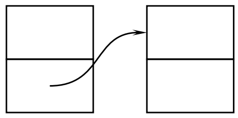
Pictures can be named as well as points. This is useful when a number of points on a picture may be of interest. For example, `cell` is a "cons cell" consisting of two boxes vertically stacked; for future reference, the individual boxes are named `car` and `cdr`, respectively:



```
val cell = namePic dbox "car"
           vseq namePic dbox "cdr";
```

In addition to optionally being named, subpictures are automatically numbered. p `nthpic` i is the ith subpicture in p.

Finally, another important feature of PIC and FPIC is the ability to draw lines and arrows. In this picture, two cells are drawn with a curved arrow from the cdr of the first to the car of the second:



```
let val cells = (namePic cell "left") hseq
                (hspace 1.0) hseq
                (namePic cell "right")
    val source = cells pic "left" pic "cdr" pt "c"
    val target = cells pic "right" pic "car" pt "w"
in cells seq (bezier source
                     (source ++ (1.0,0.0))
                     (target -- (1.0,0.0))
                     target
              withArrowStyle "->")
end;
```

3 How to design a special-purpose language

Much as we try not to, we often design a new language by thinking in terms of the syntax we would like it to have. For special-purpose languages, this generally means concentrating on the syntax of the domain-specific data and operations.

In our view, thinking about the desired concrete syntax of the new language is not at all a bad place to begin the design process, as long as that step is understood in its proper relation to the overall language design. In designing FPIC, we had an existing language, PIC, to start from. We used the syntax of PIC to help answer what we consider the most crucial question in the design process: what does each piece of syntax *mean*? In ordinary programming terms, syntactic phrases have some intuitive operational meaning; "`circle 2.0`" means "draw a circle with radius 2.0." A lesson from denotational semantics is that phrases can be given precise meanings as values of well-defined sets. It is this notion of "meaning" that we found useful in designing FPIC. We wish to give the reader some idea of the thought process we went through.

We will call the set in which the meaning of "`circle 2.0`" resides *Picture*. The question is, what exactly is in this set? What is a picture? The most obvious answer is that it is some concrete representation of a picture, for example, an array of pixels or a list of drawing commands. The precise representation does not matter to us, so we will just give the representation the generic name *BitMap*. So, our first idea is to say

$$Picture = BitMap$$

However, a closer look at PIC tells us that this can't be quite right. In PIC, one can write, for example,

```
box; box
```

meaning to draw one box next to another. (We would write this as `dbox hseq dbox`.) If each box is a *Picture*, and a *Picture* is just a bitmap, then these two boxes would represent the *same* bitmap, which would mean precisely the same pixels drawn at the same location!

We should instead say that a *Picture* is the *capability* to draw a bitmap, given a location at which to draw it; in other words, it is a *function* from locations to drawing commands:

$$Picture = Location \to BitMap$$

This is close, but we have also to deal with the picture that we write (from now on, we use FPIC syntax to avoid confusion):

```
dbox scale 5.0
```

Again, the bitmap cannot be determined from the value of `dbox`, even after knowing its location. We might try

$$Picture = Location \to ScaleFactor \to BitMap$$

which is basically correct, but we also need to deal with color, line width, and a host of other ways in which the simple bitmap might be altered.

From this point of view, the value of `dbox` is just the barest indication of what will actually show up on the printed page. We represent this by defining

$$Picture = GraphicsContext \to BitMap$$

where *GraphicsContext* contains information about any linear transformations that may have been applied to the picture, as well as color, fill style, line width, and so on.[2]

We're not quite done. We know that we can refer to the points on a picture, for example `dbox pt "nw"`. Evidently, `dbox` means something more than a bitmap. It means, in addition, a set of named points, which we call an *Environment*. This brings us to the definition we actually use in FPIC:

$$Picture = (GraphicsContext \to BitMap) \ \times \ Environment$$

This definition—and it is by no means the only one possible—is the most important in the design of FPIC. Just as the domains in the denotational semantics of a language are chosen to match the properties of that language, so here the definition of *Picture* determines, more than any other single thing, the properties of FPIC. Indeed, once this definition is made—along with the precise definitions of *GraphicsContext* and *Environment*—the rest of the language largely falls out.

When embedding a language in a functional language like Standard ML, this type definition can also guide the implementation. Indeed, we have used exactly the type above, written in ML as

```
type Picture =
    (GraphicsContext->BitMap) * Environment;
```

as the type of pictures. Many of the basic operations on pictures, and their implementations, are suggested directly by this type definition.

4 The FPIC User's Manual

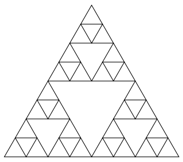
Since FPIC includes Standard ML, the manual is either very long or very short, depending upon how you look at it. In any case, FPIC consists of about 160 functions, amounting to about 1200 lines of ML code. In Appendix B, we list all the FPIC primitives that are used in the examples in this paper.

[2]PostScript [1] has a similar notion of "graphics context."

5 FPIC Examples

The examples of this section show how the features of Standard ML, when combined with the primitives of FPIC, create a powerful language for constructing pictures.

In a functional language, some fancy drawings are relatively easy to do. For example, here is the "Sierpinski gasket" of order 3:



```
fun sierpinski 0 = dtriangle
  | sierpinski n =
       let val s = sierpinski (n-1) scaleWithPoint
                       ((0.5,0.5),(0.0,0.0))
       in s hseq s seq (s at ((width s)/2.0, height s))
       end;
sierpinski 3;
```

(*pic* `scaleWithPoint` (s, *point*) scales picture *pic* by a factor s while keeping *point* fixed.)

However, of more interest in practice is the ability to create reusable pieces of pictures to ease the programming burden. Here is where functional programming shines.

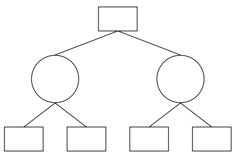
5.1 Defining lines

In FPIC, a line is simply a function from two points to a picture. Any drawing can be parameterized by a line to create a variety of effects.

Here is a simple function to draw trees. Its arguments are a picture representing the root of the tree and a list of pictures representing its children.

```
fun drawtree root subtrees =
   let val bottom = hseqtopsplist 1.0 subtrees
       val top = placePt root "s"
                  (bottom pt "n" ++ (0.0,1.0))
       val rootsouth = top pt "s"
   in group (top seq bottom seq
       (seqlist
          (map (fn p => line rootsouth (p pt "n"))
          (pics bottom))))
   end;
```

It draws a tree with its nodes connected by lines:

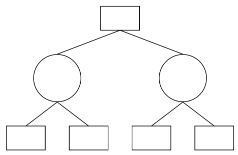


```
let val t = drawtree dcircle [dbox, dbox]
  in drawtree dbox [t, t] end;
```

We can define a function `drawTreeWithArrow` that would draw arrows instead of lines, simply by replacing "`line`" by "`arrow`." However, we can do better in ML, making the line-drawing function a parameter. `drawTree` becomes

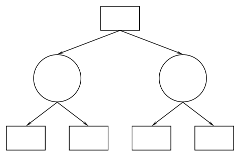
```
fun drawtree root subtrees linefun =
    ... exactly the same, until the end ...
    (seqlist
      (map (fn p => linefun rootsouth (p pt "n"))
      (pics bottom))))
end;
```

Then the tree above would be written as



```
let val t = drawtree dcircle [dbox, dbox] line
  in drawtree dbox [t, t] line end;
```

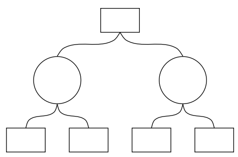
and we could also write



```
let val t = drawtree dcircle [dbox, dbox] arrow
  in drawtree dbox [t, t] arrow end;
```

Moreover, we can define our own line-drawing functions. We have seen earlier, in the cons-cell example, how to draw a curvy line. We use the same idea, except that here we want our curvy line to begin with a vertical leg rather than a horizontal one. Again, keep in mind that a line-drawing function is just a function from two points to a picture, nothing more or less:

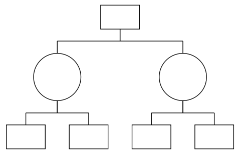
```
fun curvedvline pt1 pt2 =
   bezier pt1 (pt1--(0.0,1.0))
          (pt2++(0.0,1.0)) pt2;
```



```
let val t =
    drawtree dcircle [dbox, dbox] curvedvline
in drawtree dbox [t, t] curvedvline end;
```

Two more examples are a line-drawing function that uses a "Manhattan geometry":

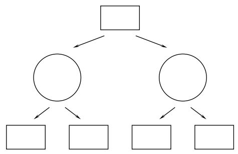
```
fun manline pt1 pt2 =
   let val ymid = (snd pt1 + snd pt2)/2.0
   in seqlist
        [line pt1 (fst pt1, ymid),
         line (fst pt1, ymid) (fst pt2, ymid),
         line (fst pt2, ymid) pt2]
   end;
```



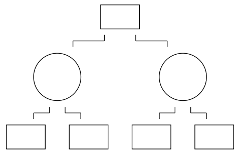
```
let val t = drawtree dcircle [dbox, dbox] manline
  in drawtree dbox [t, t] manline end;
```

and a function that draws short lines of whatever kind:

```
fun shortline linefun pt1 pt2 =
    let val diff = pt2 -- pt1;
        val pt1' = pt1 ++ diff**(0.25,0.25);
        val pt2' = pt2 -- diff**(0.25,0.25)
    in linefun pt1' pt2'
    end;
```



```
let val t = drawtree dcircle [dbox, dbox]
                             (shortline arrow)
  in drawtree dbox [t, t] (shortline arrow) end;
```



```
let val t = drawtree dcircle [dbox, dbox]
                             (shortline manline)
  in drawtree dbox [t, t] (shortline manline) end;
```

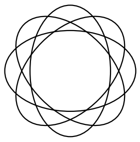
5.2 Defining sequencing operators

The functions that put pictures together into larger pictures are of key importance. In FPIC, these are generally binary operations with infix syntax. The basic sequencing operation is `seq`, which simply draws two pictures without prejudice; `hseq` and `vseq`, among others, combine `seq` with some translation of the second picture.

Again, the ability to define new sequencing operations is of the greatest interest. A sequencing operation is a function from a pair of pictures to a picture. The constructed picture should include the two pictures.

A simple example is `cseq`, which aligns the centers of two pictures:

```
infix 6 cseq;
fun (p1 cseq p2) = (p1, center p1) align
                   (p2, center p2);
```

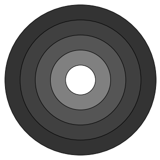


```
doval cseq (doval rotate 45.0)
      cseq (doval rotate 90.0)
      cseq (doval rotate 135.0);
```

Given a sequence operation *seq*, we often use the related operation *seq*`list`, which applies to lists of pictures. Specifically:

$$seq\texttt{list}\ [p_1, p_2, \ldots, p_n] \equiv p_1\ seq\ p_2\ seq \ldots seq\ p_n$$

The function `mkseqlist` creates the list version of a sequencing operation from the ordinary binary version.



```
val cseqlist = mkseqlist (op cseq);

val bullseye = cseqlist (map
       (fn rad => circle rad withFillColor
                     (1.0/rad, 1.0/rad, 1.0/rad))
      [5.0, 4.0, 3.0, 2.0, 1.0]);
```

The function `mkseqfun` is provided to facilitate the creation of new sequencing functions. Given two functions f and g from pictures to points, it creates the sequencing operation that, given two pictures p and q, draws the two so that fp and gq coincide. For example, in the tree-drawing function presented earlier, we used the sequencing operation `hseqtopsplist`, the list version of the sequencing operation `hseqtopsp`. The latter sequences two pictures horizontally with their tops aligned, adding some space between them. It is not built-in, but is defined as follows:

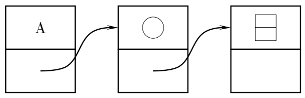
```
fun hseqtopsp gap =
   mkseqfun (fn p => northeast (p right 1.0))
            northwest;
```

The cons-cell example suggests another kind of sequencing: sequencing with an arrow. `cellseq` has as its arguments two cons cells—that is, two pictures that are presumed to have subpictures called `cdr` and `car`, respectively—and draws them a bit separated, with a curvy arrow. To allow more than one cell to be sequenced in this way, the combination of cells is defined to have `car` and `cdr` subpictures itself.

```
infix 7 cellseq;
fun cell1 cellseq cell2 =
   let val c = (group cell1)
               hseq (hspace 1.0)
```

```
               hseq (group cell2)
      val cells = c seq curvedharrow
                     (c nthpic 1 pic "cdr" pt "c")
                     (c nthpic 3 pic "car" pt "w")
  in addNamedPics cells
             [("car", cells nthpic 1 pic "car"),
              ("cdr", cells nthpic 3 pic "cdr")]
  end;
```

For this example, we have defined `labelledCell`, which draws a cons cell with a label in its car:

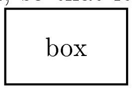


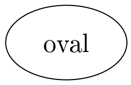

```
fun labelledCell L =
   cell seq (L centeredAt (cell pic "car" pt "c"));

labelledCell (text "A")
  cellseq
labelledCell (dcircle scaleTo (0.5, 0.5))
  cellseq
labelledCell (cell scale 0.3);
```

6 Latex integration

FPIC pictures are included in LaTeX documents by using the `fpic` command:

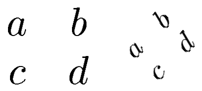
`\fpic{`*picture-name*`}{`
    ... *FPIC specification* ...
`}`

The specified picture becomes an ordinary box in LaTeX, so that it can be included anywhere within the document like any other piece of text. For example, this 

and this 

are placed in-line. Notice that LaTeX knows their sizes and creates the right amount of horizontal and vertical space for them.

The other side of this coin is the inclusion of TeX text within FPIC pictures. The `text` function turns a string—interpreted as LaTeX input—into an FPIC picture. Any LaTeX input can be used here, and once the `text` function is applied it becomes subject to the same transformations as any other piece of text:

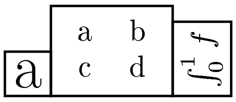



```
let val A = text
 "$\\begin{array}{cc} a & b \\\\ c & d \\end{array}$"
in hseqsplist 0.5 [A scale 1.5, A rotate 45.0]
end;
```

The only difficulty here is that FPIC does not know how large the text will turn out to be. We adopt a solution to this problem similar, in essence, to that used by Chailloux and Suarez in *mlPicTeX* [2]. When FPIC is first run, it produces LaTeX code that causes LaTeX to write the size of the text to its intermediate (`aux`) file; on subsequent runs, FPIC reads this information from that file. As is common with LaTeX utilities, this requires that LaTeX be run twice when a new picture with text is added, or when the text of an existing picture is changed, to be sure that FPIC knows the size of the text.

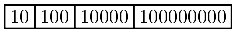
With this feature, we can define a picture-framing function that will correctly frame text:

```
fun frame p = box (width p + 0.2) (height p + 0.2)
              cseq p;
```




```
frame (text "\\huge a")
hseq frame (text ("\\begin{tabular}{cc} a & b"
                 ^ "\\\\ c & d \\end{tabular}")
hseq frame (text "$\\int_{0}^{1} f$" rotate 90.0);
```

Note that the argument to `text` can be any string, not just a string literal. The size of the text will still be calculated correctly:




```
hseqlist
  (map (fn i => frame (text (Int.toString (pow 10 i))))
           [1, 2, 4, 8]);
```

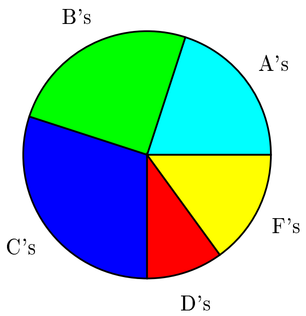
7 Packages

The real point, of course, is that FPIC can *do more*. That is, it is not merely a special purpose language for one type of picture, but is infinitely extensible to a variety of picture-drawing domains.

Our last example is a collection of functions to draw pie charts. This package contains the function `pieChart`, which takes a list of pairs, each consisting of a percentage and a slice-drawing function, and draws the slices. A slice-drawing function is a function from an angle (in degrees) to a picture; that picture will normally be a wedge of a circle centered at $(0, 0)$ and starting at the given angle. The package contains a variety of functions for creating slice-drawing functions. They are shown in Figure 1.

The function `slice` takes a collection of arguments and returns a slice-drawing function. The arguments are: an external label to be drawn outside the slice; the percentage of the pie that this slice should occupy; and a color with which to fill the slice.

Here is an example:

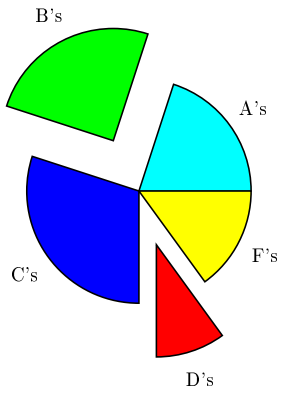


```
fun pieChart radius pieList =
   let fun pieBuilder n [(a,pfun)] = pfun radius n
         | pieBuilder n ((a,pfun)::slices) =
             let val newangle = n+((a/100.0) * 360.0)
             in (pfun radius n) seq (pieBuilder newangle slices)
             end
   in pieBuilder 0.0 pieList
   end;

fun slice lab percent color =
   let fun makeslice radius startAngle =
         let val endAngle = startAngle + ((percent/100.0) * 360.0)
             val pieSlice = (wedge radius startAngle endAngle)
             val filledPie = pieSlice withFillColor color
             val midAngle = (startAngle + endAngle)/2.0
             val labelDist = radius/4.0
             val xPt = (radius+labelDist)*(dcos midAngle)
             val yPt = (radius+labelDist)*(dsin midAngle)
             val extLabel = (text lab) centeredAt (xPt, yPt)
         in filledPie seq extLabel
         end
   in (percent, makeslice)
   end;

fun explodeSlice (percent, picfun) =
   (percent, fn rad => (fn startAngle =>
               let val angleDelta = ((percent/100.0) * 360.0)/2.0
                   val centerAngle = startAngle + angleDelta
                   val centerUnitVec = (dcos centerAngle, dsin centerAngle)
               in (picfun rad startAngle)
                   offsetBy (scaleVec 1.0 centerUnitVec)
               end));

fun triangleSlice lab percent color =
   let fun makeslice radius startAngle =
         let val endAngle = startAngle + ((percent/100.0) * 360.0)
             val unitVec1 = (dcos startAngle, dsin startAngle)
             val unitVec2 = (dcos endAngle, dsin endAngle)
             val pieSlice = (triangle (0.0,0.0)
                                      (scaleVec radius unitVec1)
                                      (scaleVec radius unitVec2))
             val filledPie = pieSlice withFillColor color
             val midAngle = (startAngle + endAngle)/2.0
             val unitVec3 = (dcos midAngle, dsin midAngle)
             val bisect = midpoint (scaleVec radius unitVec1)
                                   (scaleVec radius unitVec2)
             val labelLoc = bisect ++ (scaleVec (radius/4.0) unitVec3)
             val extLabel = (text lab) centeredAt labelLoc
         in filledPie seq extLabel
         end
   in (percent, makeslice)
   end;
```

Figure 1: Functions in the pie chart package

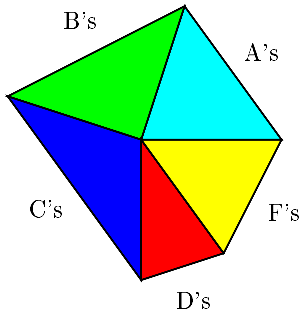
```
pieChart 2.0
  [(slice "A's" 20.0 cyan),
   (slice "B's" 25.0 green),
   (slice "C's" 30.0 blue),
   (slice "D's" 10.0 red),
   (slice "F's" 15.0 yellow)];
```

The definition of a slice-drawing function leaves a good deal of flexibility. The function `explodeSlice` takes a slice and moves it a certain distance away from the center of the pie:



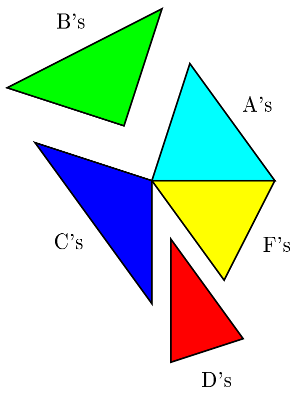

```
pieChart 2.0
  [(slice "A's" 20.0 cyan),
   explodeSlice (slice "B's" 25.0 green),
   (slice "C's" 30.0 blue),
   explodeSlice (slice "D's" 10.0 red),
   (slice "F's" 15.0 yellow)];
```

We can also change the shape of a slice. The function *triangleSlice* draws triangular slices.




```
pieChart 2.0
  [(triangleSlice "A's" 20.0 cyan),
   (triangleSlice "B's" 25.0 green),
   (triangleSlice "C's" 30.0 blue),
   (triangleSlice "D's" 10.0 red),
   (triangleSlice "F's" 15.0 yellow)];
```

`explodeSlice` works for any slice-drawing function:




```
pieChart 2.0
  [(triangleSlice "A's" 20.0 cyan),
   explodeSlice (triangleSlice "B's" 25.0 green),
   (triangleSlice "C's" 30.0 blue),
   explodeSlice (triangleSlice "D's" 10.0 red),
   (triangleSlice "F's" 15.0 yellow)];
```

8 What a picture is (slight return)

The process whereby we arrived at the definition of *Picture* was not as smooth as we described it in section 3. Let us continue the analysis we began there, and now consider the PIC (*not* FPIC) picture

```
box at last box.ne
```

This places a new box at the northeast corner of the most recently drawn box. The phrase "`last box`" suggests that a picture may depend upon the entire set of previously-drawn pictures. Assuming these are all collected into an environment, our definition of a picture would become

$$\mathit{Picture} = \mathit{Environment} \rightarrow ((\mathit{GraphicsContext} \rightarrow \mathit{BitMap}) \;\times\; \mathit{Environment})$$

(Our current definition does not allow the definition of `last box`, precisely because pictures do not see an incoming environment. The result is that we must assign a picture to an ML variable before we can access one of its named points.)

This is the "obvious" definition of *Picture*, and it was the first one we used. We worked with it for quite a while before deciding it was untenable. We still believe it is the correct definition, in principle, but it makes a clean integration into Standard ML impossible.

There are two problems with defining pictures in this way. The first is that it requires the redefinition of much of Standard ML. Consider a picture of the form (here we revert to FPIC-style syntax, though `lastbox` is *not* an FPIC primitive)

```
dbox at (lastbox pt "ne")
```

`dbox` is a picture, so it has the type given above. `lastbox` is a function from environments to pictures. Thus, `pt` must have type

$$(\mathit{Environment} \rightarrow \mathit{Picture}) \rightarrow \mathit{Name} \rightarrow \mathit{Point}$$

so that the expression `lastbox pt "ne"` has type *Environment* $\rightarrow$ *Point*. Thus, `at` has type

$$\mathit{Picture} \times (\mathit{Environment} \rightarrow \mathit{Point}) \rightarrow \mathit{Picture}$$

So far, so good. But now consider

```
dbox at (1.0,2.0)
```

According to the type of `at`, which we just agreed upon, the expression `(1.0,2.0)` must be of type *Environment* $\rightarrow$ *Point*, not *Point*! We might define a function

```
fun constantPoint p = (fn env => p);
```

and then we could write the expression above as

```
dbox at (constantPoint (1.0,2.0))
```

This is annoying enough, but now consider

```
dbox scale ((width lastbox) + 0.5)
```

`width lastbox` is a function from environments to real numbers, but then what is the type of "`+`"? It cannot have type *real*$\times$*real*$\rightarrow$*real*, so it is not the built-in multiplication of ML. Instead, it is a new multiplication operator of type

$$(\mathit{Environment} \rightarrow \mathit{real}) \times \mathit{real} \rightarrow (\mathit{Environment} \rightarrow \mathit{real}).$$

Clearly, we are on a slippery slope: all the constants and built-in operators need to be "lifted" to the type *Environment* $\rightarrow$ *whatever.*

To see the other problem with this definition of *Picture*, consider this example:

```
let val b = dbox
    val c = dcircle at (b pt "ne")
in b seq dtriangle seq c
end
```

The obvious intention is that the circle should be drawn at the northeast corner of the box. However, this is not what will happen. The way we have defined *Picture*, a picture is drawn only after all the previous pictures have been drawn. Thus, `c` is not drawn after `b`, but instead after the `dtriangle`. *At that time,* it will look in the environment, then calculate where the northeast point of a box *would be* if drawn at that time, and then draw the circle there. In short, there is no actual connection between box `b` and circle `c`.

Instead, something more like this would be needed:

```
let val b = namePic dbox "b"
    val c = dcircle at (lastpic "b" pt "ne")
in b seq dtriangle seq c
end
```

At the time `c` is drawn, `lastpic` finds the most recent picture named `b` and draws the circle there. Even this is not a direct connection between `b` and `c`; if `dtriangle` were instead a picture containing a picture named `b`, the circle would be drawn there.

In any case, we finally abandoned this approach as being too confusing.

So, our language design approach does not always work as well as we would hope. We would like to make two observations, however, before ending this discussion. One is that the two problems we've described are problems with integrating the new primitives into the existing language. In particular, the problems arise from the inescapable distinction between ML's ordinary variable environment and the picture environments created by FPIC primitives. Neither problem would exist, as far as we can see, if FPIC were designed as a new language. (In fact, the first problem is already partially solved in the functional language Haskell [3], in that literals and built-in operations can be "lifted" in the way that we require).

Our second observation is that the technical problem described in this section should not be considered to imply that the language design is a failure. We still consider that our original thesis has been substantially borne out.

9 Related Work

We have acknowledged our debt to Kernighan's PIC [4], and hopefully made clear how FPIC differs. There are quite a few languages for specifying pictures. We should particularly mention Timothy Van Zandt's PSTricks [9], a collection of TeX macros, because FPIC is implemented using them (a *BitMap* is actually just a sequence of PSTricks macro calls). Another is Kristoffer Rose's *xyPic* [8] package.

The closest relatives of this work are Chailloux and Suarez's *mlPicTeX* [2] and Simon Peyton Jones and Sigbjorn Finne's "simple structured graphics model" [7]. Both are embeddings of picture-drawing primitives in a functional language (ML and Haskell, respectively). In Peyton Jones and Finne's work, the type *Picture* contains abstract syntax trees of picture primitives; a program produces such a tree, and then a renderer traverses this tree and produces the picture.

We have emphasized in this paper the search for an appropriate definition of *Picture*, and we consider this an important step in the language design, but this is a philosophical issue. (Peyton Jones and Finne may also have considered this issue and then implemented the language as they did; they do not mention it. Chailloux and Suarez say nothing about what the type *Picture* is in their system.)

The substantive difference between FPIC and these other two systems is that FPIC has a naming facility for points and pictures that they lack. This comes directly from PIC. We think this is a significant difference, both because the facility is in fact used heavily in our examples (as it was in PIC) and because it represents the most interesting challenge in the language design.

10 Conclusions

We have outlined an approach to special-purpose language design andn implementation using the well-established technology of functional programming languages. Our recommendation is to consider carefully the type of primitive values peculiar to the domain, and embed this type in an existing functional language, such as Standard ML or Haskell. We illustrated this process with respect to FPIC, a language for picture-drawing inspired by the language PIC, and illustrated some of its benefits. FPIC is not perfect, but we would argue that the quality-to-cost-of-development ratio is very high.

11 Acknowledgments

We would like to thank the anonymous referees for their very helpful comments.

12 Availability

FPIC can be obtained from

```
http://www-sal.cs.uiuc.edu/~kamin/fpic
```

To run it, you will need to obtain Standard ML and the PSTricks macros; the FPIC web page has links to sources for both.

References

[1] Adobe Systems Inc. *PostScript Language Reference Manual.* Addison Wesley, second edition, 1990.

[2] Emmanuel Chailloux and Ascander Suarez. *mlPicTEX, a picture environment for LaTEX.*

[3] P. Hudak, S. Peyton Jones, and P. Wadler (eds.), *Report on the Programming Language Haskell (Version 1.2)*, ACM SIGPLAN Notices, 27(5), May 1992.

[4] B.W. Kernighan. *PIC: A crude graphics language for typesetting.* Bell Laboratory, 1981.

[5] Donald E. Knuth. *The TEXbook.* Addison-Wesley Co., Inc., Reading, MA, 1984

[6] Robin Milner, Mads Tofte, and Robert Harpert, *The Definition of Standard ML*, The MIT Press, Cambridge, MA, 1990.

[7] Simon Peyton Jones and Sigbjorn Finne. *Pictures: A Simple Structured Graphics Model.*

[8] Kristoffer H. Rose. *XyPic User's Guide.* 1995.

[9] Timothy Van Zandt. *PSTricks: PostScript macros for Generic TEX.* 1993.

Appendix A

We briefly review some aspects of ML syntax, enough to allow the examples to be read by someone not familiar with ML.

Here is the first example of FPIC in the paper:

```
box 1.0 2.0 hseq circle 1.5 vseq
     label "\\Huge Hello!" (oval 2.0 1.0);
```

Function application in ML is indicated by juxtaposition. Here, `box` is a two-argument function applied to arguments `1.0` and `2.0`, `circle` is a one-argument function, `label` and `oval` are two-argument functions. `hseq` and `vseq` are infix operators. As in many languages, backslash is an escape character so it must be doubled within quotes.

Function application (juxtaposition) has a high precedence, so that the above expression is equivalent to

```
(box 1.0 2.0) hseq (circle 1.5) vseq
     (label "\\Huge Hello!" (oval 2.0 1.0));
```

Note that that subexpression `oval 2.0 1.0` *must* be parenthesized, however, because otherwise the last part of the expression would be parenthesized as

```
(label "\\Huge Hello!" oval) 2.0 1.0;
```

This produces a type error, because the second argument to `label` must be a picture, and `oval` is not a picture (rather, it is a function from two reals to a picture).

Top level definitions of variables are signalled by the word `val`, as in

```
val nose = dtriangle;
```

Functions are usually introduced by the keyword `fun`, as in

```
fun drawtree root subtrees = ...
```

The `let` expression is used to introduce a temporary name.

$$\texttt{let val } v \texttt{ = } e_1 \texttt{ in } e_2 \texttt{ end}$$

binds the value of e_1 to v and then evaluates e_2, returning its value.

Structures are of two kinds: tuples and lists. Tuples are written with parentheses, as in `(1.0, 1.5)`. Lists are written with square brackets, as in `[dbox, dcircle, dbox]`.

We occasionally use the syntax `fn` *var* `=>` *expr* to define functions anonymously, usually when applying `map`. Thus, `map` (`fn` *var* `=>` *expr*) L applies the function that takes *var* to *expr* to each element of the list L.

ML allows for user-defined infix operators, as in

```
infix 6 cseq;
```

The "6" gives the precedence of the operator (in the range 0 to 9).

Lastly, two details about built-in operators: The tilde character (`~`) is the unary negation operator. Carat (`^`) is the string concatenation operator.

Appendix B

The following are the FPIC primitives used in this paper. This represents about one third of the total number of FPIC primitives. We have also included those user-defined operations that are defined in one part of the paper and used in a different part.

For each operation, we give a generic call, with the arguments in italics. The form of the call shows whether or not the operation is infix. The names of the arguments indicate their types: *pic* for pictures, *pt* for points, *i* for integers, *r* for reals, *f* for functions, and *str* for quoted strings.

pt1 `++` *pt2*	Addition of points
pt1 `--` *pt2*	Subtraction of points
pt1 `**` *pt2*	Multiplication of points
`addNamedPics` *pic* [(*str1*, *pic1*), ...]	Add the list of named pictures to *pic*'s environment
(*pic1*, *pt1*) `align` (*pic2*, *pt2*)	Place *pic1* and *pic2*, moving *pic2* so that its point *pt2* coincides with *pt1* on *pic1*
`arrow` *pt1* *pt2*	Arrow from *pt1* to *pt2*
pic `at` *pt*	*pic* translated so that its reference (southwest) point is at *pt*
`bezier` *pt1* *pt2* *pt3* *pt4*	Bezier curve from *pt1* to *pt4* with control points *pt2* and *pt3*
`box` *r1* *r2*	Box of width *r1* and height *r2*
`center` *pic*	The center of *pic*, calculated from its bounding box
pic `centeredAt` *pt*	*pic* translated so that its center is at *pt*
`circle` *r*	Circle of radius *r*
pic1 `cseq` *pic2*	*pic1* on top of *pic2*, their centers aligned
`curvedharrow` *pt1* *pt2*	Arrow from *pt1* to *pt2*, starting and ending with horizontal segments
`dbox`	Box of default size (1.618034×1.0)
`dcircle`	Circle of default radius (1.0)
`dcos` *r*	Cosine of *r* (in degrees)
`doval`	Oval inscribed in `dbox`
`dsin` *r*	Sine of *r* (in degrees)
`dtriangle`	An equilateral triangle
`frame` *pic*	*pic* with a box drawn around it (*user-defined*)
`group` *pic*	Same as *pic*, but considered as a single picture without subpictures, for purposes of calculating the subpictures of a containing picture. For example, `dbox hseq dbox hseq dbox` has three subpictures, but `group (dbox hseq dbox) hseq dbox` has two (the first of which also has two).
`height` *pic*	The height of *pic*
pic1 `hseq` *pic2*	*pic1* next to *pic2*
`hseqlist` [*pic*, ...]	A list of pictures sequenced horizontally
`hseqsplist` *gap* [*pic*, ...]	A list of pictures sequenced horizontally, with space between them
`hseqtopsp` *gap* *pic1* *pic2*	*pic1* and *pic2* are drawn next to each other, their tops aligned, with a gap between them (*user-defined*)
`hseqtopsplist` *gap* [*pic*, ...]	A list of pictures sequenced horizontally, their tops aligned, with space between them (*user-defined*)
`hspace` *r*	Empty space of lenght *r*
`label` *str* *pic*	Place *str* in the middle of *pic*
`line` *pt1* *pt2*	Line from *pt1* to *pt2*
`midpoint` *pt1* *pt2*	The mid-point of *pt1* and *pt2*

`mkseqlist` *f*	Given sequencing function *f*, return the function that applies *f* to a list of pictures
`namePic` *pic str*	Give *pic* the name *str*
`namePts` *pic* [(*str1*, *pt1*), ...]	Add the named points to *pic*'s environment
`north` *pic*	North point of *pic*, calculated from its bounding box
`northeast` *pic*	*similarly*
`northwest` *pic*	*similarly*
pic `nthpic` *i*	The *i*th subpicture of *pic*
pic `offsetBy` *pt*	*pic* translated by *pt*
`oval` *r1 r2*	Oval inscribed in `box` *r1 r2*
pic `pic` *str*	The subpicture named *str* contained in *pic*
`pics` *pic*	All the (top-level) subpictures of *pic*
`place` *pic f pt*	Move *pic* so that the point *f pic* is at *pt*
`placePt` *pic str pt*	Move *pic* so that the point named *str* is at *pt*
pic `pt` *str*	The point named *pt* in *pic*
pic `right` *r*	*pic* moved right by *r*
pic `rotate` *r*	*pic* rotated counter-clockwise by *r* (in degrees)
pic `scale` *r*	*pic* scaled by *r* in both dimensions
pic `scaleTo` *pt*	*pic* scaled to fit within *pt*
`scaleVec` *s pt*	*pt* multiplied component-wise by *s*
pic `scaleWithPoint` (*pt1*, *pt2*)	*pic* scaled by *pt1* (see `scaleXY`), but without moving its point *pt2*
pic `scaleXY` *pt*	*pic* scaled by x in the x direction and y in the y direction, where $pt = (x, y)$
pic1 `seq` *pic2*	*pic1* superimposed on *pic2*
`seqlist` [*pic*, ...]	The pictures *pic*, ..., superimposed on one another
`text` *str*	A picture consisting of the text *str*
`triangle` *pt1 pt2 pt3*	Triangle connecting *pt1*, *pt2*, and *pt3*
pic1 `vseq` *pic2*	*pic1* on top of *pic2*
`wedge` *r1 r2 r3*	Create a wedge of a circle with radius *r1* extending counter-clockwise from angle *r2* to angle *r3*
`width` *pic*	The width of *pic*
pic `withArrowStyle` *str*	*pic* drawn with a given arrow style, if it is a line; the arrow styles are as in PSTricks [9]
pic `withFillColor` (*r1*, *r2*, *r3*)	*pic* drawn in the color given by RGB values (*r1*, *r2*, *r3*) (all in the range 0-1)