ATS/Cairo Tutorial

Hongwei Xi

hwxi AT cs DOT bu DOT edu

ATS/Cairo Tutorial:
by Hongwei Xi

This tutorial focuses on employing types in ATS to facilitate safe and reliable programming with cairo[1], a comprehensive drawing package supporting 2D graphics, through the API for cairo in ATS. In particular, it is demonstrated concretely that linear types can be used effectively to prevent resources (such as contexts created for drawing) from being leaked due to programming errors. It is assumed that the reader have already acquired some rudimentary knowledge of ATS.

Table of Contents

Preface

ATS is a rich programming language equipped with a highly expressive type system for specifying and enforcing program invariants. In particular, both dependent and linear types are available in ATS to support practical programming. ATS/Cairo is the API in ATS for cairo[1], a comprehensive drawing package supporting 2D graphics. While there are already many on-line tutorials on using cairo (e.g., this one[2]), the current one focuses on employing types in ATS to facilitate safe and reliable programming with cairo. In particular, it is demonstrated concretely here that linear types can be used effectively to prevent resources (e.g., contexts and surfaces created for drawing) from being leaked due to programming errors. The reader of the tutorial is assumed to have already acquired some reasonable level of familiarity with ATS.

Notes

1. http://www.cairographics.org
2. http://www.cairographics.org/tutorial

Chapter 1. A Simple Example: Hello, world!

The first program we present in this tutorial is given as follows:

```
staload "contrib/cairo/SATS/cairo.sats"
extern fun cairo_show_text {l:agz}
  (cr: !cairo_ref l, utf8: string): void = "mac#atsctrb_cairo_show_text"
// end of [cairo_show_text]

implement main () = () where {
//
  val surface = // create a surface for drawing
    cairo_image_surface_create (CAIRO_FORMAT_ARGB32, 250, 80)
  val cr = cairo_create (surface)
//
  val () = cairo_select_font_face
    (cr, "Sans", CAIRO_FONT_SLANT_NORMAL, CAIRO_FONT_WEIGHT_BOLD)
  val () = cairo_set_font_size (cr, 32.0)
  // the call [cairo_set_source_rgb] sets the color to blue
  val () = cairo_set_source_rgb (cr, 0.0(*r*), 0.0(*g*), 1.0(*b*))
  val () = cairo_move_to (cr, 10.0, 50.0)
  val () = cairo_show_text (cr, "Hello, world!")
//
  val status = cairo_surface_write_to_png (surface, "tutprog_hw.png")
  val () = cairo_surface_destroy (surface) // a type error if omitted
  val () = cairo_destroy (cr) // a type error if omitted
//
  // in case of a failure ...
  val () = assert_errmsg (status = CAIRO_STATUS_SUCCESS, #LOCATION)
} // end of [main]
```

The functions in the cairo package are declared in the following file: contrib/cairo/SATS/cairo.sats[1] Note that in this tutorial, a file name, if relative, is always relative to the ATS home directory (stored in the environment variable ATSHOME) unless it is specified otherwise.

Suppose that the presented program is stored in a file named tutprog_hw.dats[2] the following command can be issued to compile the program to generate an executable tutprog_hw:

```
atscc -o tutprog_hw tutprog_hw.dats `pkg-config cairo --cflags --libs`
```

By executing tutprog_hw, we generate a PNG image file tutprog_hw.png[3], which is included as follows:

One can also use tools such **eog** and **gthumb** to view PNG files.

We now give a brief explanation on the program in tutprog_hw.dats[4]. We first create a cairo surface for drawing:

```
  val surface = // create a surface for drawing
    cairo_image_surface_create (CAIRO_FORMAT_ARGB32, 250, 80)
```

We then use the surface to create a cairo context:

```
val cr = cairo_create (surface)
```

We choose a font face and set the font size to 32.0:

```
val () = cairo_select_font_face
  (cr, "Sans", CAIRO_FONT_SLANT_NORMAL, CAIRO_FONT_WEIGHT_BOLD)
val () = cairo_set_font_size (cr, 32.0)
```

Imagine that we are holding a pen. We set the color of the pen to blue:

```
// the call [cairo_set_source_rgb] sets the color to blue
val () = cairo_set_source_rgb (cr, 0.0(*r*), 0.0(*g*), 1.0(*b*))
```

We now move the pen to the postion (10.0, 50.0):

```
val () = cairo_move_to (cr, 10.0, 50.0)
```

and use the pen to write down the text "Hello, world!":

```
val () = cairo_show_text (cr, "Hello, world!")
```

At this point, we have finished drawing. We store the image drawn on the surface into a PNG file:

```
val status = cairo_surface_write_to_png (surface, "tutprog_hw.png")
```

We now enter the cleanup phase, closing both the surface and the context:

```
val () = cairo_surface_destroy (surface) // a type error if omitted
val () = cairo_destroy (cr) // a type error if omitted
```

In case of a failure, we report it:

```
// in case of a failure ...
val () = assert_errmsg (status = CAIRO_STATUS_SUCCESS, #LOCATION)
```

On the surface, it seems that using cairo functions in ATS is nearly identical to using them in C (modulo syntatical difference). However, what happens at the level of typechecking in ATS is far more sophisticated than in C. In particular, linear types are assigned to cairo objects (e.g., contexts, surfaces, patterns, font faces) in ATS to allow them to be tracked statically, that is, at compile-time, preventing potential memory mismanagement. For instance, if the following line:

```
val () = cairo_surface_destroy (surface) // a type error if omitted
```

is removed from the program in tutprog_hw.dats[5], then a type-error message is issued at compile-time to indicate that the resource surface is not properly freed. A message as such can be of great value in practice for correcting potential memory leaks that may otherwise readily go unnoticed. ATS is a programming language that distinguishes itself in its practical and effective support for precise resource management.

Notes

1. https://ats-lang.svn.sourceforge.net/svnroot/ats-lang/trunk/contrib/cairo/SATS/cairo.sats
2. http://www.ats-lang.org/DOCUMENT/ATSCAIRO/CODE/tutprog_hw.dats
3. IMAGE/tutprog_hw.png

4. http://www.ats-lang.org/DOCUMENT/ATSCAIRO/CODE/tutprog_hw.dats
5. http://www.ats-lang.org/DOCUMENT/ATSCAIRO/CODE/tutprog_hw.dats

Chapter 2. Types for Some Objects in Cairo

The type for cairo drawing contexts in ATS is declared as follows:

```
absviewtype cairo_ref (l:addr) // = cairo_t*
```

The type cairo_ref(null) is just for a null pointer. Given an address *L* that is not null, the type cairo_ref(*L*) is for a reference to a cairo context located at *L*. We introduce an abbreviation cairo_ref1 as follows:

```
viewtypedef cairo_ref1 = [l:addr | l > null] cairo_ref l
```

Therefore, cairo_ref1 essentially represents a type cairo_ref(*L*) for some unknown *L* that is not null. Similarly, we have the following types in ATS for objects representing cairo surfaces, cairo patterns, and cairo font faces:

```
absviewtype cairo_surface_ref (l:addr) // = cairo_surface_t*

absviewtype cairo_pattern_ref (l:addr) // = cairo_pattern_t*
viewtypedef cairo_pattern_ref1 = [l:addr | l > null] cairo_pattern_ref l

absviewtype cairo_font_face_ref (l:addr) // = cairo_font_face_t*
viewtypedef cairo_font_face_ref1 = [l:addr | l > null] cairo_font_face_ref l
```

The above types for objects in cairo are all reference-counted. In other words, there is a reference count in each object that is assgined one of these types. When such an object is created, the initial count is 1. This count can increase or decrease depending on operations performed on the object, and the object is freed once the count drops to 0. In ATS, we can employ linear typs to track reference counts. Compared to various other APIs for cairo, the ability to track reference counts statically, that is, at compile-time, is arguably the greatest benefit one receives when programming with ATS/Cairo.

Chapter 3. Types for Some Functions in Cairo

We now present some functions in cairo and the types assigned to them in ATS. These types often reveal a lot more information about the functions to which they are assigned than their counterparts in C.

The following function cairo_destroy is for destroying a cairo context:

```
fun cairo_destroy (cr: cairo_ref1): void
```

What this funtion really does is to decrease by 1 the reference count of the object referred to by its argument. The object is freed, that is, truly destroyed only if the new count becomes 0. Because cairo_ref1 is a linear type (or viewtype in ATS), if cairo_destroy(cr)> is called, then cr can no longer be used as it is consumed: a linear value, that is, a value of a linear type, must be used once and only once. This point is made much clearer in the following example, where the function cairo_reference is presented:

```
fun cairo_reference {l:agz} (cr: !cairo_ref l): cairo_ref l
```

First, agz is a sort defined as follows:

```
sortdef agz = {l:addr | l > null}
```

Therefore, {l:agz} is simply a shorthand for {l:addr | l > null}. What cairo_reference does is to increase the reference count of its argument by 1. In the type assigned to cairo_reference, the symbol ! in front of cairo_ref indicates that the argument of the function cairo_reference is not consumed by a call to the function (and thus it can be used later). Clearly, the type also indicates that the value returned by cairo_reference(cr) is a reference pointing to the same location as cr does. If the symbol ! was omitted, the function would consume a cairo context and then return one, thus preserving reference count.

The following function cairo_create is for creating a cairo context:

```
fun cairo_create {l:agz} (sf: !cairo_surface_ref l): cairo_ref1
```

The type of this function indicates that it takes a reference to a cairo surface and returns a reference to a cairo context; the symbol ! indicates that the reference to the surface is preserved and thus is still available after the function being called; if the reference to the surface is no longer needed, it is necessary to call the function cairo_surface_destroy on the reference.

We can have another function cairo_create0 of the following type for creating a cairo context:

```
fun cairo_create0 {l:agz} (sf: cairo_surface_ref l): cairo_ref1
```

After calling cairo_create0 on a cairo surface, the surface is consumed, that is, it is no longer available for subsequent use, and therefore there is no need to destroy it by calling cairo_surface_destroy. If both cairo_create and cairo_create0 are provided to the programmer in a language like C, it can readily lead to memory leaks as one may mistakenly use cairo_create0 in place of cairo_create. This, however, is not an issue in ATS as such an error is surely caught during typechecking.

As various functions can modify the cario context they use, it is often necessary to save the state of a context so that the saved state can be restored at a later point. The functions for saving and restoring the state of a cairo context are given as follows:

```
fun cairo_save {l:agz} (cr: !cairo_ref l): (cairo_save_v l | void)
fun cairo_restore {l:agz} (pf: cairo_save_v l | cr: !cairo_ref l): void
```

The view constructor cairo_save_v is declared to be abstract:

```
absview cairo_save_v (l:addr) // abstract view generated by cairo_save
```

The simple idea behind cairo_save_v is this: Given a reference of the type cairo_ref(L) for some address L, a call to cairo_save on the reference returns a linear proof of the view cairo_save_v(L), and this proof must be consumed at some point by a call to cairo_restore on a reference of the type cairo_ref(L). In other words, calls to cairo_save and cairo_restore are guaranteed to be properly balanced in a well-typed ATS program. This is evidently a desirable feature given that balancing such calls can often be a onerous burden for the programmer programming in languages like C.

Chapter 4. Drawing Lines

In cairo, drawing often starts with the construction of a path consisting of a sequence of points. For example, the function draw_triangle for drawing a path connecting three points is given as follows:

```
fun draw_triangle {l:agz} (
    cr: !cairo_ref l
  , x0: double, y0: double, x1: double, y1: double, x2: double, y2: double
  ) : void = () where {
  val () = cairo_move_to (cr, x0, y0)
  val () = cairo_line_to (cr, x1, y1)
  val () = cairo_line_to (cr, x2, y2)
  val () = cairo_close_path (cr)
} // end of [draw_triangle]
```

The functions involved in the body of draw_triangle are assigned the following types in ATS:

```
fun cairo_move_to {l:agz} (cr: !cairo_ref l, x: double, y: double): void

fun cairo_line_to {l:agz} (cr: !cairo_ref l, x: double, y: double): void

fun cairo_close_path {l:agz} (cr: !cairo_ref l): void
```

When called, cairo_move_to starts a new (sub)path whose initial point is (x, y) and cairo_line_to connects the current point on the current path to (x, y) and then set (x, y) to be the current point. The function cairo_close_path simply adds a segment connecting the current point to the initial point of the current (sub)path.

There is also a function cairo_rel_line_to of the following type:

```
fun cairo_rel_line_to {l:agz} (cr: !cairo_ref l, x: double, y: double): void
```

This function is similar to cairo_line_to except for (x, y) being relative to the current point on the current (sub)path.

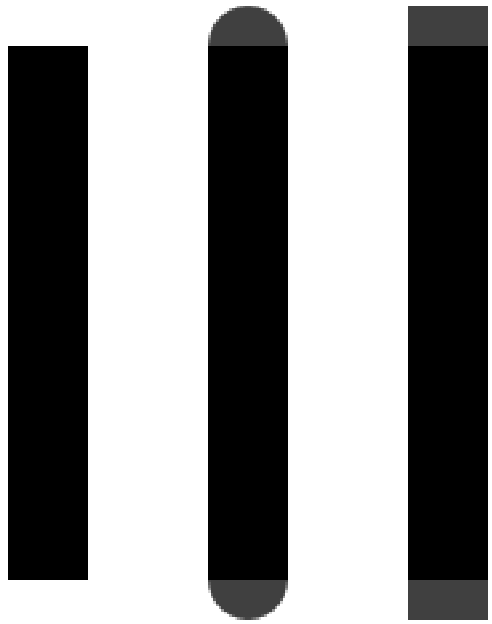
Once a path is constructed, cairo_stroke can be called to draw line segments along the path. There are a few line attributes that can be set in cairo. For instance, the styles of line cap and line join as well as the width of line can be set by the following functions:

```
fun cairo_set_line_cap {l:agz} (cr: !cairo_ref l, line_cap: cairo_line_cap_t): void

fun cairo_set_line_join {l:agz} (cr: !cairo_ref l, line_join: cairo_line_join_t): void

fun cairo_set_line_width {l:agz} (cr: !cairo_ref l, width: double): void
```

The following styles of line cap are supported:

```
CAIRO_LINE_CAP_BUTT
CAIRO_LINE_CAP_ROUND
CAIRO_LINE_CAP_SQUARE
```

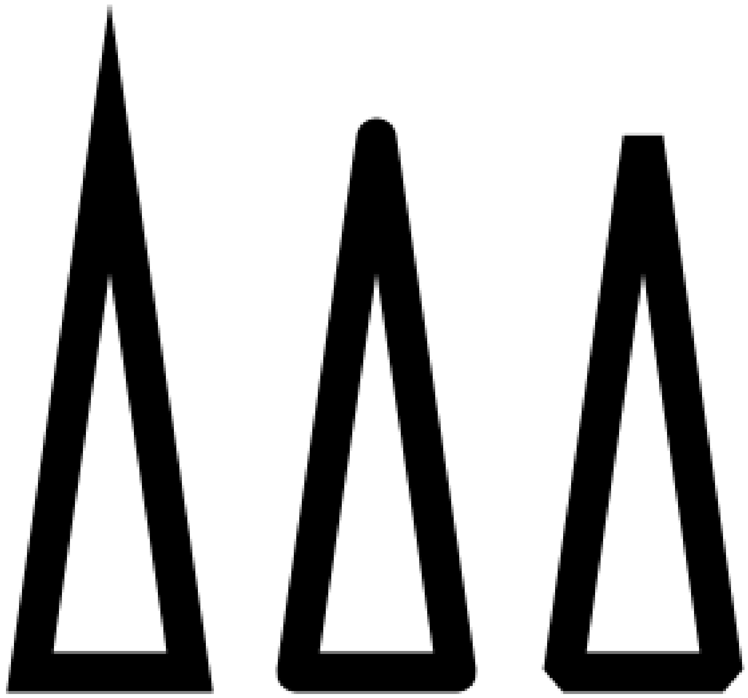
and the following lines, from left to right, are drawn according to these styles, respectively:



The following styles of line join are supported:

```
CAIRO_LINE_JOIN_MITER
CAIRO_LINE_JOIN_ROUND
CAIRO_LINE_JOIN_BEVEL
```

and the following triangles, from left to right, are drawn according to these styles, respectively:



There is also a function cairo_set_dash for setting up line dash pattern.

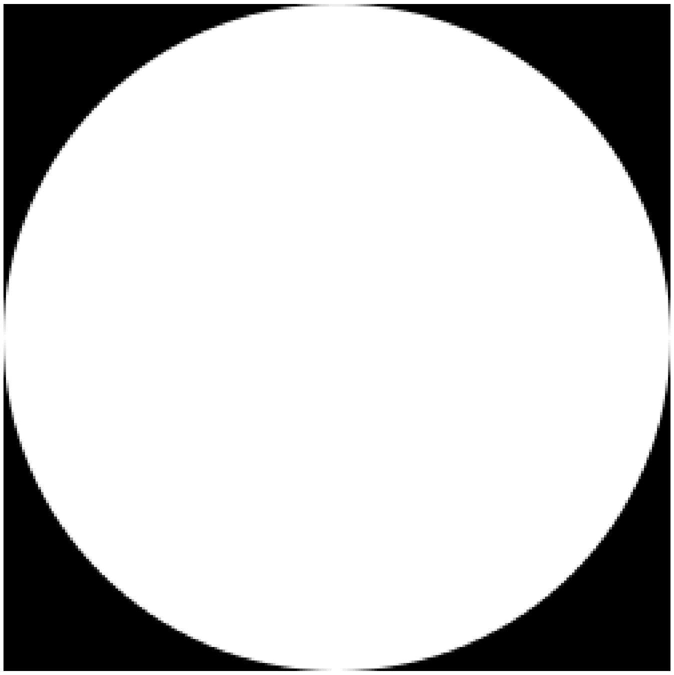
Please find in tutprog_triangle.dats[1] a program with a GUI interface that employes the function draw_triangle to draw randomly generated triangles.

Notes

1. http://www.ats-lang.org/DOCUMENT/ATSCAIRO/CODE/tutprog_triangle.dats

Chapter 5. Drawing Rectangles and Circles

We are to generate an image showing a white circle inside a black square:



The main function for drawing this image is given as follows:

```
fun draw_sqrcirc {l:agz}
  (cr: !cairo_ref l): void = let
  val () = cairo_rectangle (cr, ~0.5, ~0.5, 1.0, 1.0)
  val () = cairo_set_source_rgb (cr, 0.0, 0.0, 0.0) // black color
  val () = cairo_fill (cr)
  val () = cairo_arc (cr, 0.0, 0.0, 0.5, 0.0, 2*PI)
  val () = cairo_set_source_rgb (cr, 1.0, 1.0, 1.0) // white color
  val () = cairo_fill (cr)
in
  // nothing
end // end of [draw_sqrcirc]
```

At this moment, let us assume that the square is centered at the position (0, 0) and the length of each of its sides is 1. Therefore, the upper left corner of the square is at (-0.5, -0.5) as x-axis and y-axis increase from left to right and from top to bottom, respectively. We first draw as follows a rectangle which happens to be a square:

```
  val () = cairo_rectangle (cr, ~0.5, ~0.5, 1.0, 1.0)
```

The function *cairo_rectangle* is given the following type in ATS:

```
fun cairo_rectangle {l:agz} (
  cr: !cairo_ref l, x: double, y: double, width: double, height: double
) : void // end of [cairo_rectangle]
```

When called, this function draws a rectangle whose width and height are *width* and *height*, respectively, and whose upper left corner is located at (x, y).

We then fill the rectangle with black color:

```
  val () = cairo_set_source_rgb (cr, 0.0, 0.0, 0.0) // black color
```

```
  val () = cairo_fill (cr)
```

We next draw a circle of radius 0.5 whose center is at (0.0, 0.0):

```
  val () = cairo_arc (cr, 0.0, 0.0, 0.5, 0.0, 2*PI)
```

The function *cairo_arc* is given the following type in ATS:

```
fun cairo_arc {l:agz} (
  cr: !cairo_ref l
, xc: double, yc: double, rad: double, angle1: double, angle2: double
) : void // end of [cairo_arc]
```

When called, this function draws an arc that is part of the circle whose radius equals *radius* and whose center is at (xc, yc). The arc begins at the angle *angle1* and ends at the angle *angle2*, where clockwise rotation is assumed. If counterclockwise rotation is needed, the following function can be used instead:

```
fun cairo_arc_negative {l:agz} (
  cr: !cairo_ref l
, xc: double, yc: double, rad: double, angle1: double, angle2: double
) : void // end of [cairo_arc_negative]
```

Lastly, we fill the circle with white color:

```
  val () = cairo_set_source_rgb (cr, 1.0, 1.0, 1.0) // white color
  val () = cairo_fill (cr)
```

We can now make a call to the function *draw_sqrcirc* to generate a PNG file:

```
implement main () = () where {
//
  val W = 250 and H = 250
//
  val surface = // create a surface for drawing
    cairo_image_surface_create (CAIRO_FORMAT_ARGB32, W, H)
  val cr = cairo_create (surface)
//
  val WH = min (W, H)
  val WH = double_of (WH)
  val (pf0 | ()) = cairo_save (cr)
  val () = cairo_translate (cr, WH/2, WH/2)
  val () = cairo_scale (cr, WH, WH)
  val () = draw_sqrcirc (cr)
  val () = cairo_restore (pf0 | cr)
//
  val status = cairo_surface_write_to_png (surface, "tutprog_sqrcirc.png")
  val () = cairo_surface_destroy (surface) // a type error if omitted
  val () = cairo_destroy (cr) // a type error if omitted
//
  // in case of a failure ...
  val () = assert_errmsg (status = CAIRO_STATUS_SUCCESS, #LOCATION)
} // end of [main]
```

The functions *cairo_translate* and *cairo_scale* are given the following types in ATS:

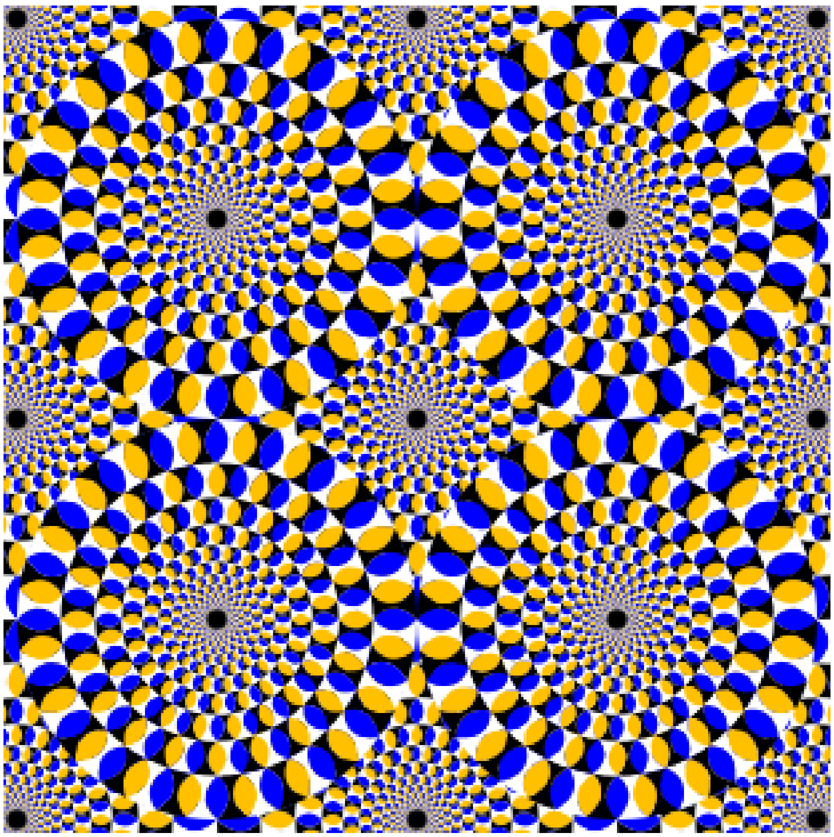
```
fun cairo_translate
  {l:agz} (cr: !cairo_ref l, x: double, y: double): void
// end of [cairo_translate]

fun cairo_scale
```

```
  {l:agz} (cr: !cairo_ref l, sx: double, sy: double): void
// end of [cairo_scale]
```

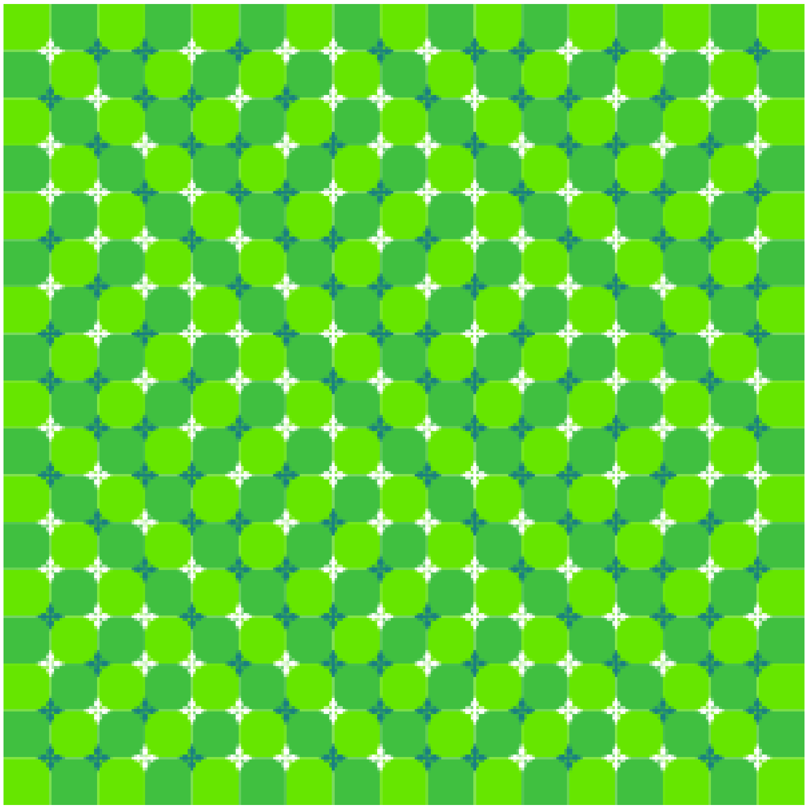
When called, *cairo_translate* creates a new coordinate system by shifting the origin of the current coordinate system to the point (x, y) and *cairo_scale* creates a new coordinate system whose x-unit and y-unit are sx and sy times the x-unit and y-unit of the current system, respectively.

For the entirety of the code used in this section, please see tutprog_sqrcirc.dats[1]

For a more elaborate example involving circles, please see illucircmot.dats[2], which generates the following interesting image:



For a more elabortate example involving squares and circles, please see illuwavy.dats[3], which generates the following interesting image:



Notes

1. http://www.ats-lang.org/DOCUMENT/ATSCAIRO/CODE/tutprog_sqrcirc.dats
2. http://www.ats-lang.org/DOCUMENT/ATSCAIRO/CODE/illucircmot.dats
3. http://www.ats-lang.org/DOCUMENT/ATSCAIRO/CODE/illuwavy.dats

Chapter 6. Drawing Text

We present a function showtext as follows that draws the text represented by a given string in a manner that puts the center of the drawing at the position (0, 0).

```
fun showtext {l:agz}
  (cr: !cairo_ref l, utf8: string): void = () where {
  var te : cairo_text_extents_t
  val () = cairo_text_extents (cr, utf8, te)
//
  val width = te.width
  and height = te.height
  val x_base = te.width / 2 + te.x_bearing
  and y_base = ~te.y_bearing / 2
//
  val (pf0 | ()) = cairo_save (cr)
//
  val () = cairo_rectangle (cr, ~width / 2, ~height/ 2, width, height)
  val () = cairo_set_source_rgb (cr, 0.5, 0.5, 1.0)
  val () = cairo_fill (cr)
//
  #define RAD 2.0
  val () = cairo_arc (cr, ~x_base, y_base, RAD, 0.0, 2*PI)
  val () = cairo_set_source_rgb (cr, 1.0, 0.0, 0.0) // red
  val () = cairo_fill (cr)
//
  val () = cairo_arc (cr, ~x_base+te.x_advance, y_base+te.y_advance, RAD, 0.0, 2*PI)
  val () = cairo_set_source_rgb (cr, 1.0, 0.0, 0.0) // red
  val () = cairo_fill (cr)
//
  val () = cairo_move_to (cr, ~x_base, y_base)
  val () = cairo_text_path (cr, utf8)
  val () = cairo_set_source_rgb (cr, 0.25, 0.25, 0.25) // dark gray
  val () = cairo_fill (cr)
//
  val () = cairo_restore (pf0 | cr)
//
} // end of [showtext]
```

For instance, the following image is produced by calling showtext (see tutprog_showtext.dats[1]) for the entire code):

Given a string utf8, we can find out some properties about the path that draws the text represented by the string as follows:

```
  var te : cairo_text_extents_t
  val () = cairo_text_extents (cr, utf8, te)
```

The type cairo_text_extents_t is defined as an external struct type in ATS:

```
//
// This external struct type is originally defined in [cairo.h]:
//
typedef cairo_text_extents_t =
  $extype_struct "cairo_text_extents_t" of {
  x_bearing= double
, y_bearing= double
```

```
, width= double
, height= double
, x_advance= double
, y_advance= double
} // end of [cairo_text_extents_t]
```

and the function cairo_text_extents is given the following type:

```
fun cairo_text_extents {l:agz} (
  cr: !cairo_ref l, utf8: string
, extents: &amp;cairo_text_extents_t? >> cairo_text_extents_t
) : void
```

In the above image depicting the text Top Secret, the center of the left red dot is often referred to as the base point of the text, which initiates the path that draws the text. The width and height of the rectangle forming the background of the text are stored in the fields of width and height of the struct in te, respectively. The vector is (x_bearing, y_bearing) from the base point to the upper left corner of the rectangle, and the vector is (x_advance, y_advance) from the base point to the center of the right red dot, which is the suggested base point for the text that follows.

The function call cairo_text_path(cr, utf8) generates a path that draws the text represented by utf8, where the function cairo_text_path is given the following type in ATS:

```
fun cairo_text_path {l:agz} (cr: !cairo_ref l, text: string): void
```

Note that a call to cairo_text_path followed by a call to cairo_fill is essentially equivalent to a call to cairo_show_text, which is given the following type in ATS:

```
fun cairo_show_text {l:agz} (cr: !cairo_ref l, utf8: string): void
```

Notes

1. http://www.ats-lang.org/DOCUMENT/ATSCAIRO/CODE/tutprog_showtext.dats