

# Algebraic Data Types Case Study: A Graphics Package

**COMP1100 — Introduction to Programming and Algorithms**

Clem Baker-Finch

Australian National University

Semester 1, 2006

## Algebraic Data Types Case Study

This section studies the development a **graphics package** written in Haskell. The focus is on the use of algebraic data types in the program.

### A Graphics Package

There are two important initial design questions:

- How to **represent** pictures in Haskell
- How to **display** those pictures

Our focus will be on the first question.

The Graphics package is available at

</dept/dcs/comp1100/public/ANUPlot/>. It provides simple 2-D graphics and animation.

It is built on top of the Haskell [OpenGL](#) binding. OpenGL is widely used in CAD, computer games, visualisation etc.

The week 7 practical exercises give basic directions on using ANUPlot.

There are also a number of examples in

</dept/dcs/comp1100/public/ANUPlot/Examples/>

## Representing Pictures in Haskell

The system will be based on **plane co-ordinate geometry**.

Each point will be represented as a pair of `Float` values:

```
type Point = (Float, Float)
```

Each unit will correspond to a single pixel, so we could have chosen to represent points as `(Int, Int)`. There are advantages and disadvantages but such a choice would be less extensible.

## Picture Components

The package will allow for the drawing of lines, text and polygons.

A picture will consist of a collection of such things, each of type `Component`.

A picture will be a **list** of `Component`s. The order of the list indicate the layering of components, if they overlap.

```
type Picture = [Component]
```

## Components

*Lines* are specified by a sequence of points, representing the beginning and end of each segment.

*Polygons* are similarly represented by a sequence of points being the vertices of the polygon.

We call a sequence of points a *Path*:

```
type Path = [Point]
```

*Text* components consist of a string.

By default, text components always appear at the origin, which is the central point of the window opened by the drawing function.

Since pictures consist of **different kinds of objects**, an **algebraic type** seems appropriate.

```
data Component = Line      Path
               | Polygon  Path
               | Text     String
```

An example of a `Line` component:

```
square :: Component
square = Line [(72,72), (144,72), (144,144),
              (72,144), (72,72)]
```

An example of a `Text` component:

```
title :: Component
title = Text "A square"
```

Combining `square` and `title` into a picture:

```
picture :: Picture
picture = [square, title]
```

(SimpleSquare.hs)

## Transformations

Notice that the text component is at the origin, horizontal, and a pre-determined size.

Obviously we need to be able to move things around in the window when composing pictures.

We could transform *individual Components*, but it's more convenient to transform *collections* of components.

But a collection of components is a *Picture*, so we may as well transform *Pictures*.

Often we want to apply several transformations at the same time (in sequence).

## Which Transformations?

The most general notion of a transformation of the plane is an (*onto*) function of type `Point -> Point` but that's not really workable.

### Design Influence

Since our graphics package will use OpenGL, it might be convenient for our transformations to correspond to those in OpenGL:

- **Translate**
- **Rotate**
- **Scale**

## Data Type Transform

```
data Transform = Translate Float Float
                | Rotate   Float
                | Scale    Float Float
```

Details:

- Translate x-distance y-distance
- Rotate degrees *clockwise*
- Scale x-scale y-scale

(Why clockwise? Because the OpenGL default is that the  $z$ -axis is *into* the screen.)

## Applying Transformations

We want to apply sequences transformations to `Picture`.

The result of the transformation is a `Component` of a `Picture`, so we add it to the `Component` Algebraic Data Type.

```
data Component = Line      Path
               | Polygon  Path
               | Text      String
               | Transform [Transform] Picture
```

*Note:* in OpenGL transformations are represented by matrices, and there is a notion of the Current Transformation Matrix.

Consequently `Transform [  $t_1, t_2, \dots, t_n$  ]` applies  $t_n$  **first** and  $t_1$  **last**.

Finally, we also want to **colour** picture components, so:

```
data Component = Line      Path
               | Polygon  Path
               | Text      String
               | Transform [Transform] Picture
               | Color     Color      Picture
```

where `Color` is another algebraic data type:

```
data Color = RGBA8 Int Int Int Int
           | RGBA  Float Float Float Float
```

(See `Plot/Picture.hs` and `Plot/Color.hs` for details.)

Notice we are using the names `Transform` and `Color` to mean two different things:

- the names of types
- the names of constructor functions

Haskell won't get confused, but you might . . .

`(Square.hs)`

`(Parabola.hs)`

*(The library also has simple animation facilities.)*