Computer Science at Kent

Turtle Graphics: Exercises in Haskell

Eerke Boiten

Technical Report No. 11- 04
June 2004

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

1 Introduction

This document contains a collection of programming exercises in the functional programming language Haskell. The exercises are all concerned with Turtle Graphics – interpreting and generating programs for "turtles".

Full solutions to the exercises have not been included (these are available on request). However, the document contains both hints for students and comments for teachers (*the latter in small italics*). The difficulty level of exercises is marked with stars. The next section gives the teaching context in which this material was used – in particular, it lists assumptions made about the students' knowledge.

2 History

The module CO312 Case Studies was introduced into the first year of the BSc Computer Science programme at the University of Kent in 1996. Its purpose was to reinforce teaching in other first year modules through the use of larger case studies. I was responsible for the "functional programming" component of the module from the start, to supplement the first year module on Functional Programming and Logic. Initially this was taught in Miranda; in recent years, Haskell was used. Textbooks by Simon Thompson were used throughout – since 1999, the second edition of "Haskell: The Craft of Functional Programming" [3]. This edition differs from the previous one and the Miranda textbook by starting with a combinator library for "pictures" before the traditional build up through topics such as recursion and list algorithms. A notation from this book used here is that `~>` means "evaluates to". For visualising turtle graphics, the exercises below use the `Picture` module from Thompson's book, but little knowledge of the internals of this is required here.

By the time the students started on their CO312 Haskell case studies, they would typically have covered:

- basic types `Bool`, `Int` and `Char` with operations;
- tuples; lists, list comprehensions, list library functions;
- list programs through pattern matching and recursion;
- very simple algebraic types (enumerated types).

They would have seen library functions on lists, but often not the higher order ones, and without much awareness of parametric polymorphism; they would *not* have seen complicated algebraic types or type classes.

The module was assessed through 50% coursework and 50% exam. A series of seven lectures would build up to the coursework assessment, by offering preparatory exercises, to be solved individually or in a class. Correct and incorrect solutions to these exercises would be discussed in the following lecture. Students would then have a few weeks to do the final assessment. The exam would consist of a model solution to this assessment (also handed out well in advance) with a number of questions related to modifications or extensions of the code, e.g. based on changes to the problem or the suggested solution.

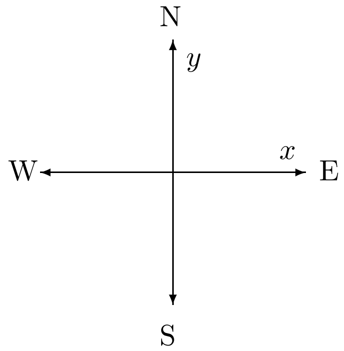
3 Turtle Graphics

This is a famous example in introductory programming, invented by Seymour Papert [2]. The treatment of it here is inspired by the one in the textbook by Bird and Wadler [1, Section 4.4]; an important difference is that turtle commands are defined in an algebraic type here rather than directly as functions; also, we add the generation of turtle programs.

A turtle lives in a (potentially infinite) two-dimensional grid.

```
type Location = (Int,Int)
```

(Note: all locations will be integer pairs.) Besides a location, it has a *pen* which is `Up` or `Down`, and it faces a *direction* which is `North`, `South`, (which means that when it next moves, it will increase or decrease its y coordinate), `East`, or `West` (increasing or decreasing x).




```
data Direction = North | East | South | West
```

```
                 deriving (Show,Eq)

data Pen = Up | Down
           deriving (Show,Eq)

type Turtle = (Location, Direction, Pen)
```

I would explain `deriving (Show,Eq)` *as a useful idiom by itself for enumerated types – type classes had not yet been covered.*

3.1 Exercises (*)

Define functions which get the components of a turtle state:

1. `myX :: Turtle -> Int`
2. `myY :: Turtle -> Int`
3. `face :: Turtle -> Direction`
4. `pen :: Turtle -> Pen`

Although the main point of these exercises is to use the right patterns for the `Turtle` *arguments (including possibly* `_`*), the existence of these functions may actually dissuade students from using the right patterns in later exercises ...*

4 Turtle Commands

A turtle knows the following commands:

```
data Cmd = Step | PenUp | PenDown | RotR | RotL
           deriving (Show,Eq)
```

It can do one step in the direction it is facing; it can put the pen up or down; and rotate 90 degrees to the left or to the right.

5. (**) Define the function

   ```
   execute :: Cmd -> Turtle -> Turtle
   ```

   for the given set of commands and the given interpretations. Auxiliary functions you might use are

```
step :: Turtle -> Turtle
rotr :: Direction -> Direction
```

for executing a single step, and rotating right.

Pitfalls: capitalisation of constructors matters; confusion between a command and its interpretation; use of `_` *on the right hand side to denote unchanged components.*

6. (**) Define a function

```
executes :: [Cmd] -> Turtle -> Turtle
```

which uses `execute` to execute a sequence of commands.

The cynical view of introductory functional programming: as soon as you have fixed the type errors in a typical exercise, the answer is probably correct. Using this principle and the idea that parameters and computed results need to be used gets you far here. The main source of error is that the recursive call to `executes` *and the call to* `execute` *both produce a* `Turtle`*, possibly leading to reverse execution of the sequence.*

5 Programs for Turtles

We will not just be interpreting programs for turtles, but also generating them. A program is just a sequence of commands. ("Real" turtle graphics languages such as Logo [2] offer more control structures and other constructs.)

```
type Tprog = [Cmd]
```

The *specifications* of these programs use `executes`.

7. (**) Give a program for any turtle that moves the turtle to the x-axis (i.e., where $y = 0$).

```
moveX :: Turtle -> Tprog
```

The property that will need to hold (for any `t`) is

```
myY ( executes (moveX t) t) ~> 0
```

You need to ensure the turtle faces in the right direction, and then use `replicate` and the `Step` command.

8. (**) Define

```
moveToX :: Turtle -> Tprog
```

which satisfies, in addition, that

```
face (executes (moveToX t) t) == face t ~> True
```

(i.e., it needs to turn back the way it was facing originally).

9. (**) Analogously, define

```
moveY :: Turtle -> Tprog
```

and as a challenge

```
moveToY :: Turtle -> Tprog
```

10. (*) Assuming that you have available `moveToX` and `moveToY` as defined above, define

```
moveTo0:: Turtle -> Tprog
```

such that

```
myY ( executes (moveTo0 t) t) ~> 0
myX ( executes (moveTo0 t) t) ~> 0
face ( executes (moveTo0 t) t) == face t ~> True
pen ( executes (moveTo0 t) t) == pen t ~> True
```

11. (**) Define

```
mirror :: Turtle -> Tprog
```

such that

```
myY ( executes (mirror t) t) == - (myY t) ~> True
myX ( executes (mirror t) t) == - (myX t) ~> True
```

You may of course use the functions defined above.

6 Turtle Trails

So far, we have not actually recorded the paths taken by a turtle with pen down, there have been turtles but no "graphics".

The strategy is to replace `executes` by a function which also records the points touched, define windows, display the points which are in a window (as a `Picture = [[Char]]`!)

12. (*) Define a function

    ```
    locs :: [Turtle] -> [Location]
    ```

    which lists all locations in the list of turtle states.

13. (*) Define a (slightly different) function

    ```
    dots :: [Turtle] -> [Location]
    ```

    which lists all locations in the list of turtle states where the turtle's pen was `Down`.

    E.g.

    ```
    t1 =[((0,0),North,Up),((0,0),North,Down),((0,1),North,Down)]
    locs t1 ~> [ (0,0), (0,0), (0,1) ]
    dots t1 ~> [ (0,0), (0,1) ]
    ```

14. (**) Define a function

    ```
    trail :: Tprog -> Turtle -> [Turtle]
    ```

    such that `trail cs t` records all turtle states obtained by a turtle executing `cs` starting from state `t`.

    E.g.

    ```
    trail [PenDown,Step] ((0,0),North,Up) ~> t1  (see above)
    trail [Step] ((0,0),North,Up)
    ~> [((0,0),North,Up), ((0,1),North,Up)]
    ```

    The structure of this function is similar to that of `executes`.

    Pitfall: forgetting the first or last position.

15. (*) Define a function

    ```
    alldots :: Tprog -> Turtle -> [Location]
    ```

    such that `alldots cs t` records all locations visited with the pen down by a turtle executing `cs` starting from state `t`.

16. (**) Give a recursive definition of `alldots` which does not use `trail`.

7 Pictures in Windows

A window is denoted by a pair of locations: the *bottom-left* (bl) and the *top-right* (tr) of the window.

```
type Window = (Location,Location)
```

A window is OK if its `bl` corner is not to the right of, and not above its `tr` corner. *All further windows are assumed to be OK.*

```
windowOK :: Window -> Bool
windowOK (bl,tr) = leftBelow bl tr
```

17. (*) Define the function

    ```
    leftBelow :: Location -> Location -> Bool
    ```

18. (*) Points can be either inside or outside a window. The "edges" of the window count as *inside*.

    Define the function

    ```
    isInWin :: Location -> Window -> Bool
    ```

    such that `isInWin l w` evaluates to `True` whenever location `l` is in the window `w`.

19. (*) Define the function

    ```
    locsInWin :: [Location] -> Window -> [Location]
    ```

    which picks, from a list of locations, all those within the window.

Given a window, we can create a `Picture` [3, Chapter 1] from it. Remember that

```
type Picture = [[Char]]
```

with the character `'#'` representing black, and `'.'` representing white.

20. (**) To generate a completely "white" picture from a given window `((blx,bly),(trx,try))`, we would have to take, for each `y` (going down from `try` to `bly`), a line of `'.'`, one for each `x` (going up from `blx` to `trx`).

    Complete the definition below.

    ```
    winToPic :: Window -> Picture
    winToPic ((blx,bly), (trx,try))
        = [ ??? | y <- downto ???? ?? ]
        where downto m n = reverse [n..m]
    ```

    `???` is itself a list comprehension.

    Note that `winToPic` could also be defined using `replicate` (twice) but that solution would not be useful for the next question.

21. (***) Instead of a completely white picture, we could also produce a picture with dots taken from a list of locations.

    Modify the definition of `winToPic` to get a function

    ```
    locsToPic:: [Location] -> Window -> Picture
    ```

    such that the size of the resulting picture is the size of its input window, and such that it is 'black' for all given locations and 'white' for all others.

    You may assume that the locations are all within the window. You will need to check for certain `(x,y)` whether they are in the input list of locations – use the function `elem`, do not attempt to optimise this (yet).

22. (****) The function `locsToPic` can be optimised by sorting the locations into the order in which they occur in the picture (i.e. first by decreasing y, then by increasing x). You can then always check for just the first of the remaining locations rather than use `elem`. The

downside of this is: the list comprehensions will have to be replaced by recursion.

Assuming the list of locations is sorted in this way, define a more efficient version of `locsToPic`.

8 Turtle Objects and Pictures

We will develop a function that, from an abstract description of a "picture" (as a collection of geometric shapes inside a window), generates a turtle program which will let a turtle "draw" that picture.

A turtle picture consists of a window (the "frame" of the picture) and a list of turtle picture objects.

```
type TPic = (Window, [TObj])
```

In the definition of `TObj`, we will use a type `Nat`, defined by

```
type Nat = Int
```

to indicate which values can *always* be assumed to be ≥ 0.

Turtle picture objects may be:

```
data TObj = Dot Location |
```

a dot in a given location, or

```
            Rect Location Nat Nat Bool |
```

a rectangle (bottom left corner, horizontal size, vertical size, filled in?), or

```
            Square Location Nat |
```

a square (bottom left, size, not filled in). The size of a square or rectangle is the *difference* between its coordinates, so a rectangle of width two and height one will occupy six locations. Finally, it could be

```
            Line Location (Int,Int) Nat
```

a line starting from a location with given slope and extent (see later).

8.1 Drawing Lines

When we have `Line f s x` we are thinking of a line that starts in location `f`, has slope `s` and *horizontal extent* `x`. E.g., `Line (1,3) (2,-1) 8` runs from `(1,3)` to `(9,-1)`. (Draw a picture!) The *exception* is when the line is vertical, i.e., the slope is `(0,y)`, then `x` gives the *vertical* extent. So: the extent is usually the absolute difference between x coordinates of the endpoints of the line, except for vertical lines, where it is the difference between y coordinates.

Every slope can be assumed to satisfy `okslope`.

```
okslope :: (Int,Int) -> Bool
okslope (x,y) = (gcd(abs x)(abs y) ==1)
```

I.e., `x` and `y` have no common divisor > 1.

Since we are only interested in integer points, the line specified by `Line (1,3) (2,-1) 8` runs from `(1,3)` to `(9,-1)`, and it visits `(3,2)`, `(5,1)`, `(7,0)` on the way. But what does it do between those points? We don't draw diagonals, so . . . we have to approximate them. In this case, that would be by taking `(1,3), (1,2), (2,2), (3,2), (3,1), (4,1), (5,1)` etc (primitive), or by `(1,3), (2,3), (3,3), (3,2), (4,2), (5,2), (5,1)` etc or even (smooth) `(1,3), (2,3), (2,2), (3,2), (4,2), (4,1), (5,1)` etc. (Again, draw pictures to understand this.) Using `moveToFrom` between the endpoints of the line only leads to the least smooth line of all.

8.2 Exercises

Where you are asked to write functions that generate turtle programs, do *not* optimise these turtle programs; this will be done in Exercise 33.

20. (*) Define functions

    ```
    base, limit :: TObj -> Location
    ```

    such that the `base` of any `TObj` is the `Location` given in its definition, and the `limit` is the "opposite end" – this will be the same location for a dot, the top right corner for a rectangle or square, and the other endpoint for a line.

    For lines you will need to distinguish between slopes of $(0, y)$ and others, and take into account that slopes may have negative coordinates.

21. (**) Define a function

```
rectangle :: Int -> Int -> Bool -> Tprog
```

such that executing `rectangle x y b` "draws" (with the pen down!) a rectangle, whose first and third side are length `x`, and second and fourth sides are length `y`. The first side should be in the direction the turtle is facing; then at every corner the turtle should turn left for `b == True` or right for `b == False`.

Note that the current state of the turtle is irrelevant and thus not a parameter of the function.

22. (**) It is hard to put together generated turtle programs without losing track of where the turtle is.

    Define a function

    ```
    andThen :: (Turtle -> Tprog)
            -> (Turtle -> Tprog)
            -> Turtle
            -> Tprog.
    ```

    such that `andThen p1 p2 t` (which can also be written as: ``p1 `andThen` p2``) gives a turtle program which starts with `p1 t` and then `p2` applied to the turtle state you get from executing `p1 t`.

    Reformulate `moveTo0` using `andThen`.

23. (**) Define a function

    ```
    doAll :: [ (Turtle->Tprog)] -> Turtle -> Tprog
    ```

    such that

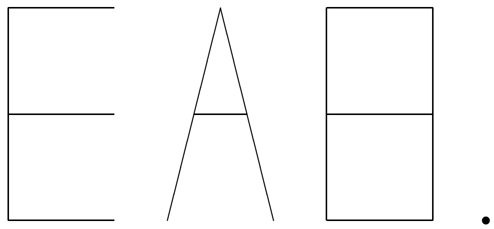
    ```
    doAll [p1,p2,p3,p4] t
       = (andThen p1 (andThen p2 (andThen p3 p4))) t
    ```

    etcetera.

24. (*) Define a constant `house :: TPic` as a picture of a square "house" with a pointed roof, a rectangular door in the middle of the house, with a door knob.

25. (*) Define a value of type `TPic` which represents your name, your login, or some other originality. It should contain a list of at least 8 `TObj`. E.g., mine could be

```
( ((0,0),(18,8)) ,
  [ Line (0,0) (0,1) 8, Line (0,0) (1,0) 4,
    Line (4,4) (-1,0) 4, Line (0,8) (1,0) 4,
    Line (6,0) (1,4) 2, Line (10,0) (-1,4) 2,
    Line (7,4) (1,0) 2, Rect (12,0) 4 8 False,
    Square (12,4) 4, Rect (18,0) 0 0 True ] )
```



26. (*) Define a function

```
fits :: TObj -> Window -> Bool
```

which checks whether a given picture object is within a `Window`. (Use previous functions.)

27. (*) Define a function

```
checkObj :: TObj -> Window -> Bool
```

which checks whether a given picture object is within a `Window`, and if it is a (non-vertical) `Line`, *also* whether its extent is divisible by the `x` coordinate of its slope. (I.e.,the slope fits into the extent an *entire* number of times.)

28. (*) Define a function

```
checkTPic :: TPic -> Bool
```

which checks whether all the turtle picture objects in the `Tpic` pass the `checkObj` test, using the `Tpic`'s window.

29. (**) Define a function

```
moveToFrom :: Location -> Turtle -> Tprog
```

which gives the `Tprog` to move a turtle (its current state is given in the `Turtle` argument) to a specified `Location`. The pen position should remain unchanged throughout this program, and the turtle orientation should be the same after this program as it was before. I.e., one should have that

```
executes (moveToFrom (xn,yn) ((x,y),d,p)) ((x,y),d,p)
~> ((xn,yn),d,p)
```

30. (**) Define a function

```
drawDot :: Location -> Turtle -> Tprog
```

which produces a turtle program that, from a given initial turtle position, will move to the given location and "put a `#` there".

8.3 How to Fill a Rectangle

The function `rectangle` does a lot of work for "open" rectangles already, but it doesn't cover "filled in" rectangles.

However, drawing

```
Rect (3,5) 7 4 True
```

should have the same result as drawing all of

```
Rect (3,5) 7 4 False
Rect (4,6) 5 2 False
Rect (5,7) 3 0 False
```

(Draw a picture. Note that the last of these *is* essential, and this shows why we do have `Rect` with sides possibly `0`.)

8.4 Exercises, Continued

31. (***) Define a function

```
drawObj :: TObj -> Turtle -> Tprog
```

such that `drawObj tob t` gives the turtle program to draw picture object `tob` for a turtle which has to start from state `t`. (So this function should deal with dots, squares, open rectangles, filled in rectangles, and lines.)

Could award bonus marks for smooth lines. Pitfalls: trying to draw rectangles from the "nearest" corner and getting that wrong; the booleans in `rectangle` *and* `Rect` *are unrelated; lines: forgetting (downward) vertical lines, division by 0 errors from vertical lines, incorrect endpoints when negative slope components.*

32. (**) Define a function

    ```
    drawTPic :: TPic -> Turtle -> Tprog
    ```

    such that `drawTPic tp t` gives the turtle program to draw picture `tp` for a turtle which has to start from state `t`.

    The program losing track of where the turtle is is normally found out by using `drawTPic pic ((-4,4),West,Up)`.

33. (**) Define a function

    ```
    optimise :: Tprog -> Tprog
    ```

    to optimise the turtle programs you have generated.

    It should optimise every subsequence of unnecessary rotations (e.g. 7 `RotR` should go to a single `RotL`, and `RotR` should never be followed immediately by `RotL` and vice versa). It also should optimise *adjacent* pen movement, e.g. 2 `PenUp` followed by a `PenDown` can be replaced by a single `PenDown`. (You need to be careful with pen optimisations – the pattern `[PenDown,PenUp]` cannot be optimised.)

    Other pitfalls: indexing rather than recursion, `take i1 xs++drop i2 xs` *is inefficient and usually wrong. Optimising* `[PenUp,RotR,PenUp,RotR,PenUp, RotR,PenDown,RotR]` *to* `[PenDown]` *requires a non-obvious approach.*

34. (***) **Nested pictures.**

    Extend the declaration of `TObj` with

    ```
            Nest Location TPic
    ```

    which has the meaning of including a picture as an object in another picture, such that the inner picture's `(0,0)` coincides with the location given.

There are essentially two ways of dealing with a nested picture: either flatten it (i.e. replace it by all the objects in it, with their bases possibly changed), or deal with it directly. The former seems easier.

Extend the definitions given earlier (and `base` etc) to also cover `Nest`. There are two possible interpretations for a nested picture "fitting": either all the objects in it fit, or it fulfills `checkTPic` and its entire window fits. Either of these is fine.

"Wallpaper" from an existing 10x10 picture `mypic`*:*

```
wallp :: TPic
wallp = ( ((0,0),(40,20)), [Nest (0,0) mypic,
          Nest (0,10) mypic, Nest (10,0) mypic,
          Nest (10,10) mypic, Nest (20,0) mypic,
          Nest (20,10) mypic, Nest (30,0) mypic,
          Nest (30,10) mypic])
```

9 Exam Questions

Exam papers for CO312 consisted of a "case study" (a model solution to the year's major assessment) which had been handed out well in advance, with questions such as the following.

1. Select a part of the code that, in your view, has been done in a clumsy or inefficient way, and present an alternative solution for it. Explain in which respect your solution improves the solution given.

2. Imagine we are modeling a new generation of turtles (or are they now frogs?), which have the same repertoire of commands, plus a new command called `Jump`. In terms of the type definitions, that would only change the type `Cmd`. The interpretation of `Jump` is similar to that of `Step`, except that the turtle moves *two* steps in whichever direction it is facing. For example, a turtle starting from state `((0,2), East, Down)` would end up at state `((2,2), East, Down)` after executing a `Jump`. For drawing purposes, the location "jumped over" does not count – so in the example, location `(1,2)` would *not* be drawn. As a consequence, `Jump` is only useful for "speeding up" turtle movement when the turtle is *not* drawing.

   Go through the code systematically, and indicate which changes would have to be made in order to accommodate this new command. For functions that generate and optimise turtle programs, at which point do you think `Jump` commands were best introduced? Why, and how?

3. Define a function

```
union :: Window -> Window -> Window
```

such that `union w1 w2` is the *smallest* window which contains all points in `w1` and `w2`. This may also include points in neither, for example `(6,4)` is in neither of `((0,-1),(6,3))` and `((-1,0),(5,5))` but in their union, as

```
union ((0,-1),(6,3)) ((-1,0),(5,5)) ~> ((-1,-1),(6,5))
```

4. Using the function `union`, define a function

```
merge :: TPic -> TPic -> TPic
```

which combines two turtle pictures into one, using the union of their windows and including all objects of both pictures.

5. Define a function

```
createLine :: Location -> Location -> TObj
```

which, given two locations, returns a line connecting them. The result of this function should satisfy the properties of `Line` objects: positive extent, `okslope`. Ensure that the function returns an error (using `error::String -> a`) in the one circumstance where it is not possible to return such a line.

6. Using the function `createLine` define a function

```
connectAll :: [Location] -> [TObj]
```

which, given a list of $n \geq 1$ locations, returns the list of $n - 1$ lines connecting every pair of locations adjacent in the list. For example, `connectAll [a,b,c]` should return a list containing two lines, one between `a` and `b` and one between `b` and `c`.

7. Imagine we are modeling a new generation of smarter turtles, which have eight instead of four possible orientations: they can face `NE`, `NW`, `SE`, or `SW` (northeast, northwest, etc) as well. In terms of types, that would only change the type `Direction`. The set of commands would not change, but the interpretation of rotations left/right would be 45 degrees (used to be 90) and a `Step` could also lead to a diagonal movement. For example, a turtle facing `SE` at location `(3,4)` would end up in `(4,3)` if it did a `Step`.

   Go through the code systematically, and indicate which changes would have to be made in order to accommodate this new version of turtles and changed interpretation of some command.

   The most complicated change is the one possible in `moveTo0` (which is likely to change `moveToFrom` and line drawing functions) that makes profitable use of the availability of diagonal turtle movement. For this, first sketch how you would let the turtle move as quickly as possible to `(0,0)` when it can also move diagonally, and only then write the code which produces such a turtle program.

8. Imagine we are modeling a new generation of smarter turtles, which have the same repertoire of commands, plus a new command: `Flip`. In terms of types, that would only change the type `Cmd`. The interpretation of `Flip` is like two `RotR` or `RotL` commands, i.e., a 180 degree rotation.

   Go through the code systematically, and indicate which changes would have to be made in order to accommodate this new command.

   Use of the `Flip` command could also be a way of generating shorter turtle programs. There are essentially two ways (sections of the code) for doing this. Which are they, how would it be done, and which of the two ways would you prefer, and for what reason?

Acknowledgements

I would like to thank the CO312 students from 1997/98 to 1999/2000 for their feedback and ideas.

References

[1] Richard Bird and Philip Wadler, "Introduction to Functional Programming", Prentice Hall, 1988.

[2] Seymour Papert, “Mindstorms; Children, Computers and Powerful Ideas”, Basic Books, 1980.

[3] Simon Thompson, “Haskell: The Craft of Functional Programming”, 2nd edition, Addison-Wesley, 1999.