Tentamen TDA555/DIT440
Introduction to Functional Programming

2020-08-18 8.30-12.30

Responsible teacher: Jonas Duregård, contacted by Zoom (information on Canvas page)

In case of urgent technical issues, contact by phone: 031 772 1028

Submit your solutions in Canvas.

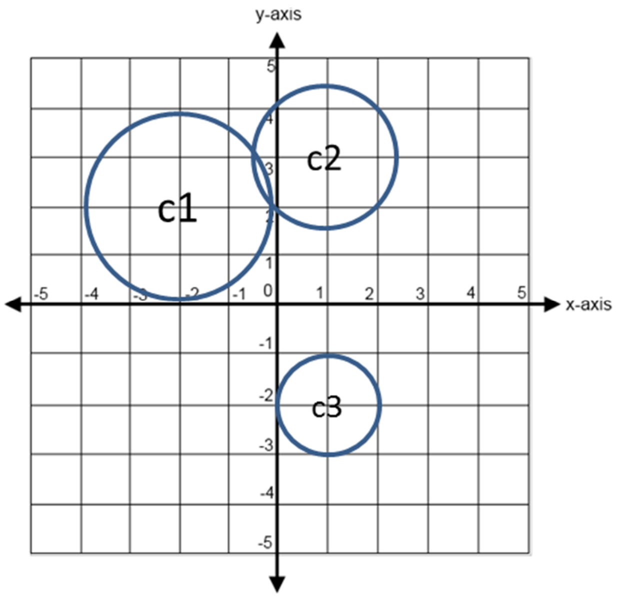
If technical issues prevent submitting through canvas, email your solution to jonas.duregard@chalmers.se before the end of the exam time.

Parts of assignments marked ”For Excellent” are only required to achieve higher grades than 3 (Chalmers) or G (GU).

Problem 1: Circles

Your task is to create a data type Circle, for representing circles in a plane, and a few related functions.

Each circle has a radius and a position. Here is an illustration of three circles:
c1 with a radius of 2.0 and position (-2.0, 2.0)
c2 with a radius of 1.5 and position (1.0, 3.0)
c3 with a radius of 1.0 and position (1.0, -2.0)



Note that the task does not involve drawing any graphics, you will only treat circles as abstract representation of geometrical shapes and do some computations.

These are all the functions you should implement, with type signatures and brief descriptions:
```haskell
-- Gives the radius of a circle
radius :: Circle -> Double

-- Gives the (x,y)-position of the center of a circle
position :: Circle -> (Double, Double)

-- move c (dx,dy) moves a circle dx steps in x- and dy steps in y-direction
move :: Circle -> (Double, Double) -> Circle

-- scales the radius of a circle up or down.
-- Examples: scale c 0.5 halves the radius of c, scale c 2 doubles it.
scale :: Circle -> Double -> Circle

-- Computes the (Euclidean) distance between the centers of two circles
distance :: Circle -> Circle -> Double

-- Checks if two circles touch or overlap
collides :: Circle -> Circle -> Bool
```

Specific tasks:

Create circles: write a piece of code that creates the three circles c1, c2 and c3 using your Circle data type.

radius and **position**: should be self-explanatory. For c1 they would give 2.0 and (-2.0, 2.0) respectively.

move and **scale**: builds new circles by modifying position and scale respectively. Note that the second parameter of move is not an absolute position but movement relative to the current position, and similarly the second parameter of scale is not a radius value but a radius multiplier. **`position (move c1 (1,1))`** would result in (-1, 3) and **`radius (scale c1 3)`** would result in 6.

distance: computes the length of a straight line between the centers of two circles. Use Pythagoras' theorem. In the example above **`distance c1 c3 = distance c3 c1 = 5`**.

collides: answers the question "are these two circles colliding", meaning they are close enough for the circles to touch. In the example c1 and c2 are colliding, whereas c3 is not colliding with neither c1 or c2. Hint: How does the distance of two touching circles relate to their radii?

Explanatory text: Write a brief explaining the choices you have made in your solution. You may write this as one or a few comments in a .hs file or in a .pdf file if you prefer.

For excellent: Add a QuickCheck test data generator and an Arbitrary instance for Circle. Enforce the invariant that the radius of a circle is never negative.

Problem 2: Number sequence function

Your task is to write a function **`intSequence`** for creating infinite number sequences.

Different applications of the function result in different (but similar) sequences, by varying several parameters. For instance one application may result in the Fibonacci-sequence, another in a linearly increasing sequence and a third in an exponentially increasing one.

Each sequence is an infinite Haskell-list of Intergers. In the text below, X_n refers to the nth number in the sequence, starting with X_0.

Parameters: The function takes X_0 and X_1, i.e. the first two numbers of the sequence as parameters. Additionally, it takes three integer parameters: m_1, m_2 and a. For $n > 1$, X_n in the resulting sequence should be given by this formula:

$X_n = X_{n-1}*m_1 + X_{n-2}*m_2 + a$

So the parameter *a* is a constant that is added in each step, m_1 and m_2 are multipliers for the two previous numbers in the sequence.

Example: For the parameters $X_0=1$, $X_1=2$, $m_1=2$, $m_2=0$, $a=0$ we get the formula:
$X_n = X_{n-1}*2 + X_{n-2}*0 + 0 = X_{n-1}*2$
Thus the value is doubled in each step, giving the sequence [1,2,4,8,16,...]

Using $X_0=0$, $X_1=1$, $m_1=1$, $m_2=1$, $a=0$ yields:
$X_n = X_{n-1}*1 + X_{n-2}*1 + 0 = X_{n-1}+ X_{n-2}$
This is the Fibonacci-sequence where every number is the sum of the two previous ones (0,1,1,2,3,5,8,...).

A call to intSequence should look like this: **`intSequence X₀ X₁ m₁ m₂ a`**.

Explanatory text: Explain how you solved this problem. If you use any helper functions or variables (e.g. in a where-clause) explain what they are.

For Excellent: Write a QuickCheck property that tests that after removing an arbitrary number of values from the start of any sequence, the resulting list should still have at least three values and the third of those values can be computed from the first two using the formula above. The property should create a random sequence for each test, and drop a random number of values from the start of it.

Problem 3: Computing valid chess moves

Your task is to implement a small part of a chess game. You are given the following interface to work with (note: you do not need to implement any of these functions or types!, you only have to write code that uses them.

```
-- A position on a chessboard
--  For valid positions both numbers should be between 1 and 8 inclusive
type Position = (Int, Int)
-- Represents an ongoing chess round, including all positions of pieces
data Game = ...
-- All types of chess pieces
data Piece = King | Bishop | Pawn | ... deriving (Show, Eq)
-- The two players of a chess game, White and Black.
data Player = White | Black deriving (Show, Eq)

-- getPiece g p will give the chess piece occupying position p on the
--   gameboard g, along with the player that owns it, or Nothing if
--   the position is not occupied.
--   Will crash for invalid positions.
getPiece :: Game -> Position -> Maybe (Piece, Player)
```

Use this interface to write a function that computes allowed moves for a bishop chess piece:

```
-- bishopMoves g pos p computes the valid moves for a bishop
--   standing on position pos on the chessboard g owned by player p.
bishopMoves :: Game -> Position -> Player -> [Position]
```

The result is a list of positions a bishop can move to if starting on the given position and owned by the given player. Which moves are valid depend on the positions of other chess pieces in the Game g. Valid moves must respect these rules:

- A bishop can only move any number of steps in any of the four diagonal directions.
- The move must end either in an empty position on the board or on a position occupied by a piece owned by the opposing player (which is then captured).
- The move can never cross any piece owned by any player. There must only be empty positions between the destination position and the origin position.

You do not have to consider any additional restrictions present in actual chess rules.

You do not have to check that there is a bishop on the given position, and your function should work even if there isn't one.

There is an example of usage on the next page.

Explanatory text: Write a brief description of the algorithm you used and some techniques you used to overcome some problem you ran into. Write short comments explaining any functions you create other than bishopMoves.

For Excellent: Sketch a function **`bishopCanCheck`** that determines if a bishop can capture the opponents king in two consecutive moves. In other words: Is it possible to move the bishop to put the opponent in check? This function does not need to be complete, but you should demonstrate a good idea for how to do it and explain your reasoning in text.

Example:

Here is a chess board along with position values for some cells. The dashed red lines indicate valid moves for the bishop at (6,4). Any cell along those lines is a valid move.

If g is the representation of this gamestate, the following describes how getPiece works:

- getPiece g (4,2) == Just (Pawn, Black)
- getPiece g (4,6) == Just (Pawn, White)
- getPiece g (6,4) == Just (Bishop, Black)
- For all other valid postions, getPiece would give Nothing.

If bishopMoves works as intended, the result of **`bishopMoves g (6,4) Black`** should be some permutation of the list [(5,3),(5,5),(4,6),(7,3),(8,2),(7,5),(8,6)].