Turtle Graphics

Adam Olsson

February 2020

Questions Asked

Question: Can you define the limited version in terms of the unlimited one?

Yes, it is possible. We simply run the unlimited version run for a limited amount of time.

```
-- | A spiral that will continue forever
spiralForever :: Double -> Double -> Program
spiralForever size angle =  (>*>) (forward size) $
                            (>*>) (right angle)
                            (spiralForever (size+2) angle)


spiral' :: Double -> Double -> Program
spiral' size angle = limited 100 $ spiralForever size angle
```

Question: What definition of time do you use (what can a turtle achieve in a single time unit)?

We assume that a single action takes 1 time unit to complete.

Questions: What happens after a parallel composition finishes? Is your parallel composition commutative, is it associative? (To answer this question you must first define what it means for programs to be equal.) What happens if a turtle runs forever only turning left in parallel with another turtle running the spiral example? Does your textual interface handle this situation correctly, if not — how would you fix it?

Once a parallel composition finishes the program ends. Initially I thought of making the program continuing but because running programs in parallel creates multiple states a question rises of which state to return. Therefor I decided to simply end the program after a parallel composition.

If we would define two programs to be equal when they produce the same end result, or graphic, the parallel composition is associative because p1 PARALLEL (p2 PARALLEL p3) will produce the same result as (p1 PARALLEL p2) PARALLEL p3. Just executed in a different order. However, it is not commutative. Consider the case of (forward Seq right) and (right Seq forward). Both programs produces a line on the canvas but the angle of the line is different. Therefor, the parallel composition is not commutative.

If a program running forever is composed with another program (p2) the forever program will only be executed. However, once the forever program has run once, it allows for p2 to take one step in its execution. This process is repeated until p2 finishes and then the forever program runs freely. According to my interpretation the textual interface handles this correctly.

Question: How does parallel composition interact with lifespan and limited? (lifespan does not need to correspond realistically to actual life spans, just specify how it works.)

The lifespan counts down a timer and once this timer reaches below 1 it sequences a die program before any potential following program. Once the parallel composition reaches a die program in simply removes this program from the parallel composition and continues to execute the rest of the programs. The limited program works very similar to lifespan except that is does not append a die program, it simply stops running the program that is limited by returning. For the parallel composition, this is the same as an end to one out of several programs running. It simply removes the ended program and continues to execute the rest of them.

Did you use a shallow or a deep embedding, or a combination? Why? Discuss in a detailed manner (giving code) how you would have implemented the Program type if you had chosen the other approach. What would have been easier/more difficult?

A deep embedding has been used where the functions available only returns constructors as the following code:

```
-- | Move a distance backward
backward :: Double -> Program
backward = Backward
```

```
-- | Turn degrees right
right :: Double -> Program
right = Right
```

This leaves the run function to take care of the program logic, as the following example:

```
runProgram (Forward d) (w, st, t, l) = do
  runTextual (Forward d) st st'
  drawNoTick (w, st, t, l) st'
  return (w, st', t, l)
  where
    (x, y)  = getPos st
    x'      = x + sin(getAngle st)*d
    y'      = y + cos(getAngle st)*d
    st'     = updatePos st (x',y')
```

Left and Backward can easy be implemented using Right and Forward respectively as:

```
left :: Double -> Program
left d = Right (-d)

backward :: Double -> Program
backward d = Forward (-d)
```

Additionally, the penup and pendown functions return the same constructor that would toggle a boolean saying if the turtle should draw or not.

```
penup :: Program
penup = TogglePen

pendown :: Program
pendown = TogglePen
```

A Counter constructor was also created that would work as a stop watch for the programs that are running for a limited time. It keeps track of if a program is out of time and should be stopped.

```
  Counter n p -> do
    case n < 1 of
      True -> return (w, (t:ts))
      _    -> do
        runTextual (Counter n p) st st
        let t1 = updateTurtleWithProgram t p
        runProgram (w, t1:ts)
        where
          st = getTurtleState t
```

A deep embedding was chosen because it would be simpler to expand the EDSL by adding a new function that returns a constructor. After that we simply need to write the interpretation of this constructor. Additionally, a deep embedding is more intuitive in my experience. A shallow implementation would have moved the logic to the functions available via the API but no other obvious changes to the logic would have changed.

Had I done this again I would probably have looked more into a monad that would keep track of the state and also provide IO. Currently I have a lot of very simple functions that basically expands a type and provides the requested argument. I believe that many of these functions could have been more efficiently using monads (i.e with less code).

Compare the usability of your embedding against a custom-made implementation of a turtle language with dedicated syntax and interpreters. How easy is it to write programs in your embedded language compared to a dedicated language? What are the advantages and disadvantages of your embedding?

My EDSL should in theory be as simple to use as a custom language, that is the point of an EDSL. The advantage is that we can bridge the gap between people who knows the domain and people who knows how to code.

Compare the ease of implementation of your embedding against a custom-made implementation. How easy was it to implement the language and extensions in your embedded language compared to a dedicated language? What are the advantages/disadvantages of your embedding?

The advantage is that we can use an already implemented language with dedicated syntax and interpreters to create a new embedded language. This saves a lot of time because we don't need to build everything from scratch. The only disadvantage I can think of is that we are somehow at mercy of the host language. By that I mean that we are bound by whatever features the host language has which could possibly make the implementation difficult. By writing a new language from the ground up allows us to develop whatever functionality we wish for but at the cost of time.

In what way have you used the following programming language features: higher-order functions, laziness, polymorphism?

We have used higher-order functions in terms of the functions the API provides. These functions are composed and then sent to the run function in the EDSL.

We have used polymorphism in defining our programs. A single interface can represent multiple types.

The programs themselves are a series of functions composed with each other. We use the lazy evaluation of haskell to run the programs. As an example, the parallel composition would first run the first program and then the second program in sequence had it not been for the lazy evaluation.