TDA342
Report for Turtle Graphics - Part 2

QUFEI WANG
JACOB J. NILSSON

February 23, 2020

Task 1

The shallow embedding implementation of the functions in the program is included in file 'Turtle.hs'. The primitive operations exported are *forward, left, color, penup, pendown, die, limited,* $>*>$, $<|>$, the derived operations exported are *backward, right, idle, lifespan, times, forever* and the running functions exported are *runTextual, runGraphical*. Data type changes made different from *Part 1* of this lab are listed as follows.

1. Type **Time** was added just as a synonym of **Int**. In our application, time is defined as series of discrete incremental integers starting from zero. A turtle can only perform one operation in a single time unit. The explanation here is also an answer to the question in Task 1:

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

2. Type **Graph** was added as a conceptual representation of the 'line' the turtle draws in each operation. Fields *'from'* and *'to'* are the origin and end of the line, *'graphColor'* being the color.

```
-- | Abstraction of one line in graph
data Graph = Graph {
    from       :: Location,
    to         :: Location,
    graphColor :: Color
}
```

3. A new data type **Action** was added. Three fields are included, they are *'turtle', 'operation'* and *'time'*, which represent the state of the turtle after the operation in this action at the specific time.

```
-- | the action a turtle performs in every unit of
    time
data Action = Action {turtle :: Turtle, operation ::
    Operation, time :: Time}
```

4. The *'graphic'* field in type **Operation** was changed from type **HGL.Graphic** to **Maybe Graph** (also with a minor naming change from *'graphic'* to *'graph'* which is insignificant). This improves the modularity of the code and separates the representation of the concept of the 'line' from the labrary dependent implementation details. Also by using the type constructor **Maybe** here we are referring to the situation where no drawing operation is performed by the turtle, such as moving from one point to another with state *penup*, or just being *idle*.

```
-- | Operation taken
data Operation  = Op {
    -- | the line that might needs to be drawn
    graph    :: Maybe Graph,
    -- | the description of this operation, used for
        textual interface
    message :: String
}
```

5. The definition of **Program** was changed from a function of the form $Turtle \to Int \to [Operation]$ to $Turtle \to Time \to ([Action], [Turtle])$. The justification of such a change is little bit intricate. A list of **Operation** seems at first sufficient to provoide the graphs or textual contents needed by the running functions, but the requirement of the sequential and parallel execution of programs, which is enforced by operator $> * >$ and $< | >$, is beyond the functionality of such a design.

```
-- | a 'turtle program' is a function which returns a
     list of action and turtle, given a turtle and an
     initial step number
newtype Program = Prog {recipe :: (Turtle -> Time ->
    ([Action], [Turtle]))}
```

   - Consider a program of the form $p1 > * > p2$. we need the final state of the turtle after running program $p1$ to succeed in running program $p2$, that's why the **Turtle** object needs to be returned.

- Consider a program of the form $(p1 < | > p2) > * > p3$. By the semantics of these two operators, after the completion of $p1$ and $p2$(could be in different tiem), program $p3$ is supposed to continue at the precise positions where all turtles from $p1$ and $p2$ stop. So a list of turtle, that is the type $[Turtle]$, needs to be returned.
- Consider a program of the form $(limited\ n\ (forever\ p1)) > * > p2$. After running program $p1$ for n length of time, $p2$ is supposed to run precisely at the positions where all turtles from $p1$ halt. So a list of intermediate state of turtles is needed at this time, that's why a **Turtle** is attached to each **Action**

Task 2

The code used to implement the parallem composition of programs is excerpted here. The semantic of this operation is guaranteed by first obtaining the result of each program by applying the *turtle* and *time* parameters on them, then sort the total actions by order of time and concatenate the turtle states. Note that the function *sortActions* was implemented in an incremental way so that the laziness of haskell can be exploited to handle the infinite list.

```
-- | perform several programs in parallel
(<|>) :: Program -> Program -> Program
(Prog r1) <|> (Prog r2) = Prog $ \t n ->
    let (actlist1, turtles1) = r1 t n
        (actlist2, turtles2) = r2 t n
    in (sortActions [actlist1, actlist2], turtles1 ++
        turtles2)

-- | sort the actions by order of the time
sortActions :: [[Action]] -> [Action]
sortActions [] = []
sortActions x = case x' of
        [] -> []
        _ -> let n = time $ head $ head x'
                 pair = map (break ((> n).time)) x'
                 heads = concatMap fst pair
                 tails = map snd pair
             in heads ++ sortActions tails
    where x' = filter (not . null) x
```

All the algebraic rules listed in task 2 are met by the implementation except for the last one, $forever\ p1 > * > p2 = forever\ p1$, where no output can be generated from our program. By examining the code of our implementation of

the operator $>*>$, we noticed that the evaluation of the expression $p1 >*> p2$ depends on the evaluation of $p1$, when $p1$ has the form $forever\ p$, the code fails. So far I've not been able to find a way to completely circumvent this limitation. Although this awkwardness can be alleviated in some degree by using deep embedding like

```
data Program where
    Forever :: Program -> Program
    (:>*>)  :: Program -> Program -> Program
    ...

forever :: Program -> Program
forever = Forever
runProgram :: Turtle -> Time -> Program -> ([Action], [
   Turtle])
runProgram t n ((:>*>) p1 p2) =
    case p1 of
        Forever p' -> runProgram p1
        otherwise -> ...
```

But it would not handle expression like $(p1 >*> (foreverp2)) >*> p3$. We also tried to implement in an incremental way, that is, we keep all the sub programs of a program in a list and only evaluate the program at the head of list in one unit of time. But this requres to keep also the context information necessary for operations like $limited, forever, >*>$ *and* $<|>$ in a hierarchy way, which is very much like the process of implementing an interperter. We consider this might not be the right approach for this lab. Hope to get some feedback on this point.

Answers to the question:

> **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?**

When a parallel composition finishes, the turtles of all the programs stop at their last position, either dead or ready to execute the next program, also in parallel.

We say two programs are equal in our language when running these

two programs with the same parameters generate the same sequence of actions after filtering out the actions where the turtle being *idle* and the same sequence of turtle states, within finite time of execution. Under this definition of equality, our parallel composition is commutative and associative.

Our textual interface can handle the situation where a turtle runs forever turning left in parallel with another turtle running the spiral example. This can be shown by loading *RunExample.hs* in *ghci* and running the program *runTextual parallel_composition_1*

Answers to question:

> **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.)**

Our parallel composition interact well with $lifespan$ and *limited*, this can be shown by running program *parallel_composition_2* in *RunExample.hs*.

Task 3

The *TurtleExtras.hs* file can be found in the package, with derived operators annotated as the file describes.

Task 4

A few examples has been included in the file *RunExample.hs*, which you can find under the directory *executable*. The function *reportText* is a program that does not terminate, and can be used to show the functionality of the textual interface.

Task 5

Q.1: Start by answering all the questions in the assignment description above.

The answers to the questions can be found above.

Q.2: 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?

The implementation we used is shallow embedding, because it is more straightforward by clearly defining the semaintics of a *Program* by incorporating the type information directly into its data definition.We have only one interpretation of the data type *Program* and all the other primitive or derived constructors and combinators are just implemented by using the defined semantics.

The code in file *TurtleDeep.hs* shows one possible implementation of deep embeeding. We didn't provide a full implementation but only a few functions as a means of illustration. The definition of basic data types remains the same, except for *Program*:

```
data Program where
    Forward   :: Double -> Program
    Backward  :: Double -> Program
    Right     :: Double -> Program
    Left      :: Double -> Program
    Color     :: Color -> Program
    Penup     :: Program
    Pendown   :: Program
    Die       :: Program
    Idle      :: Program
    Limited   :: Int -> Program -> Program
    Lifespan  :: Int -> Program -> Program
    Times     :: Int -> Program -> Program
    Wait      :: Int -> Program -> Program
    Forever   :: Program -> Program
    (:>*>)    :: Program -> Program -> Program
    (:<|>)    :: Program -> Program -> Program

forward :: Double -> Program
forward = Forward

(>*>) :: Program -> Program -> Program
(>*>) = (:>*>)

(<|>) :: Program -> Program -> Program
(<|>) = (:<|>)
```

As you can see from the code, both the constructors and combinators for the type *Program* are defined as value constructors. Functions need to be exported just refer to the corresponding constructors.

Next, we show the code about how a program is interpreted in the running function:

```
runProgram :: Turtle -> Time -> Program -> ([Action], [
    Turtle])
runProgram t n (Forward d) =
    let pos = location t
        nextPos = move d pos (direction t)
        t' = t {location = nextPos }
        op = constructOperation n (pen t) pos nextPos
    in ([Action t' op n], [t'])
runProgram t n ((:>*>) p1 p2) =
    case runProgram t n p1 of
        (as, []) -> (as, [])
        (as, ts) -> (as ++ sortActions as', ts')
            where
                lastTime = time . last $ as
                followingStates = map (\t -> runProgram t
                    (lastTime + 1) p2) ts
                as' = map fst followingStates
                ts' = concatMap snd followingStates
runProgram t n ((:<|>) p1 p2) =
    let (as1, ts1) = runProgram t n p1
        (as2, ts2) = runProgram t n p2
    in (sortActions [as1, as2], ts1 ++ ts2)
```

We see that the function *runProgram* is defined as taking a *Turtle*, *time* and *Program*, and returning a tuple with a list of *Action* and *Turtle*, with the same semantics as that in the shallow embeeding. The different semantics between different types of programs are expressed by pattern matching on the constructors.

The shallow embeeding is easier than the deep embeeding because the type *Program* in our language has only one interpretation, that is, given a turtle and a starting moment in time, a program returns a list of actions to be preformed and a list of turtles in the final state. If some other functionalities need to be added to our language and a program has more than one ways of interpretation, then the deep embeeding would be more preferable because it is more flexiable in expressing the semantics of running a program.

Q.3: 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?

A custom-made implementation of turtle language we choose to compare with is an online program called *Logo Interpreter*(see the website **here**). It is a DSL with a rather complete syntax support for *variables, lists, IO, arithmetic and logical operation, basic graphic operations, function definition and control statements*.

For simple operations like *forward, backward, left, right, color, etc*, there's almost no difference between the dedicated language and our embedded one. However, for complicated grahps which consist of repetative simple operations, it is easier to write in our language by some higher order functions we exported in *TurtleExtra* module. The advantages in the usage of our embedding lies exactly in this kind of simplicty to express sophisticated operations(such as *infinite iteration, graph translation*) in fewer code than that of an imperitive style language. Another advantage is the ability to run programs in parallel, which is a feature we dind't find in *Logo Interpreter*. The disadvantage is the limitation on the semantics of our combinators. For example, currently we can only draw one program at a time, we can not run another program on the same window after the previous one finishes, without considering all its final states.

Q.4: 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?

Regarding the ease of implementation, it is easier to implement the language and extensions in our embedded language. First, because our language is an EDSL on haskell, we don't need to write the compilers and interpreters to handle new features and extensions of the language. Second, because we use haskell as our host language, the language features such as support of higher-order functions, laziness and polymorphism can be exploited to acheieve a cleaner and more compact implementation of most of the functions of the turtle language.

The advantages of our embeeding, when compared with an imperative programming language implementation, lie exactly in the extra expressiveness of complex functions and possible combinations of various functions through higher-order functions and polymorphism granted by the language of haskell. The disadvantage is that, since the functionality of the language is implemented in a highly modular way, and a complex function usually consist of layers of other functions, it is less convenient to modify the behavior of one existing function withour worrying about its global impact, which increases the cost of the maintenance of the language.

Q.5: In what way have you used the following programming language features: higher-order functions, laziness, polymorphism?

The data type *Program* was defined as a function which accepts parameters of type *Turtle* and *Time*, and return a list of *Actions* to be performed and a list of *Turtle* in the final state. The function *runTurtle* is thus a higer-order function which accepts a *Program* and feeds it initial parameters to get the results. Extensions like function *loop* from file *TurtleExtras.hs* is also defined as a higher-order function.

```
-- Turtle.hs
newtype Program = Prog {recipe :: (Turtle -> Time -> ([
    Action], [Turtle]))}
runTurtle :: Program -> [Graph]
-- TurtleExtras.hs
loop :: (a -> Program) -> (a -> a) -> a -> Program
```

Laziness was used when the result of a *Program* is evaluated. Since the list of *Action* was consumed in an incremental way, laziness here enables us to handle infinite lists from running inifite *Programs*.

```
-- Turtle.hs
runTextual :: Program -> IO ()
runTextual (Prog p) =
  let (as, _) = p iniTurtle 0
  in printMsg as

printMsg :: [Action] -> IO ()
printMsg [] = return ()
printMsg (h:r) = do
    putStrLn . message . operation $ h
    printMsg r
```

Polymorphism was used in the definition of function *loop*. Although we only used *Double* as a concrete type in the examples of this function, virtually all concrete types with a function $a- > Program$ and another function $a- > a$ can be applied on this function.

```
-- TurtleExtras.hs
loop :: (a -> Program) -> (a -> a) -> a -> Program
```

Algebraic Laws: The algebraic laws of the operators can be characterized as follows:

1. $(Program, < | >)$ is a commutative monoid with identity element *die*:
   (a) $(p1 < | > p2) < | > p3 = p1 < | > (p2 < | > p3)$

(b) $die <|> p = p <|> die = p$

(c) $p1 <|> p2 = p2 <|> p1$

2. $(Program, >*>)$ is a monoid with identity element $idle$:

   (a) $(p1 >*> p2) >*> p3 = p1 >*> (p2 >*> p3)$

   (b) $idle >*> p = p >*> idle = p$

3. $>*>$ left and right distributes over $<|>$:

   (a) $p1 >*> (p2 <|> p3) = (p1 >*> p2) <|> (p1 >*> p3)$

   (b) $(p1 <|> p2) >*> p3 = (p1 >*> p3) <|> (p2 >*> p3)$

Question: Is your program data type a monoid? Under which operations? There may be several possible monoid instances. Would it be a monoid if some small change was made to your operators?

Yes, our program data type is a monoid. Under operations $<|>$ and $>*>$. It will always be a monoid as long as semantics of $idle$, die, $<|>$, $>*>$ hold.