CS 589: Project Report

Ghadeer Alkubaish

June 11, 2019

1 Domain and modularity scenario

The system consists of three main modules: primitive terms, shapes and robotic moves. The primitive terms module is provided by the professor and I am are directly using it. The system is assumed to be used by multiple clients. The shape library is used in its own by people in geometry. The robotic moves library is used by people in robotics. The assumption is that different clients extended the system multiple times in all dimensions. Primitives are used in both shapes and robotic moves and this requires extending evaluation and pretty printing operations. The shape module is extended with an additional shape and a function that works with shapes. The robotic moves module is extended with an additional robot move. Robotic moves also use the shape module which requires extending some operations. Lastly, a new operation is made to map shapes into robotic moves.

2 System overview

The system involves several modules in 3 main categories: primitives, shapes, and moves. Two modules are directly used from the code Prof. Walkingshaw provides in $Prim.hs$, $Cond.hs$ which defines primitive operations and values.
The shape module extends pretty printing and implements a new evaluation operation which extends the old evaluations from primitives and introduces a new evaluation value for shapes. Shapes are then extended by a new operation ($Area$). After that, shapes are again extended by a new shape ($Rectangle$) and then a new operation ($Circumference$).
The robotic Moves module extends pretty printing and implements a new evaluation operation which extends the old evaluations from primitives and shapes and introduces a new evaluation value for moves. Moves are then extended by a new operation ($ShapeToMoves$) which takes a shape and returns a sequence of moves to draw that shape. Eventually, Robotic Moves are extended by a new move ($Step$). The system overview is showing in figure 1.

2.1 Module: Shape

2.1.1 Syntax

The shape interface holds the following data types: $Point$ and $Shape$. A point consists of two primitive terms. In the evaluation of the shape, we assume both terms evaluate to floats or fail. A shape is either a point, a vertical line or horizontal line, or a square. Both lines take three terms which are assumed to be floats; starting by the common coordinate. For example, for a horizontal line, this is what we expect ($Hline\ y\ x1\ x2$). For a vertical line, this is what we expect ($Vline\ x\ y1\ y2$). The square takes a point and a term that must evaluate to float representing a center and a side length respectively.

```
data Point t = P t t
data Shape t = Pt (Point t )
             | Hline t t t
             | Vline t t t
             | Square (Point t) t
```

2.1.2 Semantic

In this module we can directly extend pretty printing to represent shapes. However, the evaluation function can not be extended directly because the semantic is different with respect to shapes. For primitives, we evaluate every expression to either boolean or float, but in this module we want to evaluate every shape into a centre, width and height. Therefore, we extend our evaluations form primitives which

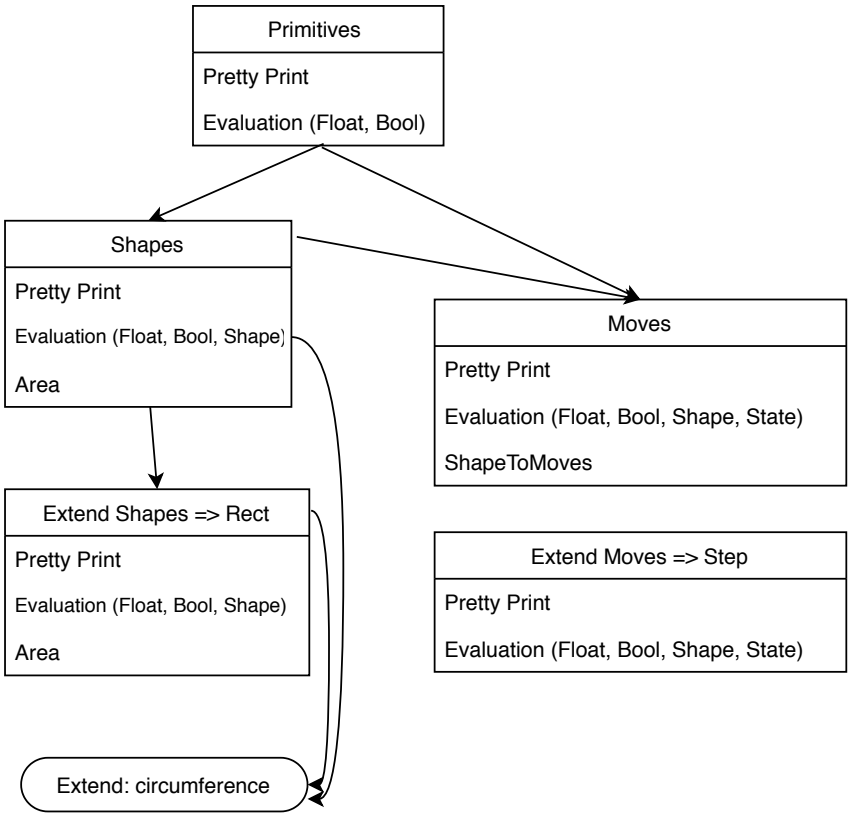

Figure 1: system overview

allow us to simplify terms used in shapes. For evaluating shapes, we build a new data type (*ShapeDom*) holding a center point and two terms for the height and width.

```
data ShapeDom t = S (Point t) t t
```

The new semantic value for evaluating terms in this module becomes the following.

```
type Value = Term (PVal :+: ShapeDom)

class Functor t => Eval t where
  evalAlg :: t Value -> Value
```

2.1.3 Operations

The two main operations in this module become pretty printing and evaluation. We show the pretty printing instance of *Square* below.

```
instance Pretty Shape where
  prettyAlg (Square p l) =
    concat ["sqr:  center ", prettyAlg p, " side len:  ", l]
```

We show the evaluation instance of *Square* below.

```
evalShapeSquare::(ShapeDom :<: t, PVal :<: t) => Term t -> Term t -> Term t -> Term t
evalShapeSquare x y l    =
   case (project x, project y,project l) of
           (Just (F x'), Just(F y'), Just(F l') ) ->
           shapeDom (P (float x') (float y')) (float l') (float l')
           _ -> error "Type error:  non-float values"

instance Eval Shape where
  evalAlg (Square (P x y) l) = evalShapeSquare x y l
  ..
```

2.1.4 Examples

Now we can build a square value which uses any primitive values from before as follows.

```
ex1 :: (Prim :<: t, PVal :<: t) => Term t
ex1 = op2 Add (op2 Mul (float 2) (float 3)) (op1 Neg (float 4))

ex5 :: (Cond :<: t, Prim :<: t, PVal :<: t, Shape :<: t, Point :<: t) => Term t
ex5 = (square (point (float 3) (float 4)) ex1)
```

The example above can be used after casting it to the appropriate type in both the evaluation and pretty printing operations as follows.

```
type SExpr = Term (Cond :+: Prim :+: PVal :+: Shape :+: Point )

testEx5 = pretty $ eval (ex5 :: SExpr)
testEx5' = pretty (ex5 :: SExpr)
```

Running the two examples above give the following results.

```
$ center: (3.0, 4.0) height: 2.0 width: 2.0
$ sqr: center (3.0, 4.0) side len: ((2.0 * 3.0) + -4.0)
```

2.2 Module: Extend Shape Operation - Area

2.2.1 Semantic

In this module, we extend the shape module to have a new operation called area which calculates the area of a shape. The return type of this operation is float. We need to simplify primitive expressions in shapes in order to calculate the height and width used in calculating the area. Therefore, we need to use the same semantic value of primitives ($PVal$) which is either a float or bool.

```
data PVal t = B Bool | F Float

class Functor t => Area t  where
  areaAlg :: t (PVal ()) -> PVal ()
```

2.2.2 Operation

We extend the area operation for shapes and primitives in order to calculate the area of shapes which include primitives in them. We show the $Area$ instance of $Square$ below.

```
instance Area Shape where
  areaAlg (Square (P x y) l) =
    case (l,x,y) of
      (F l', F _ , F _ ) -> F (l' * l')
      _ -> error "Type error:  non-float values"
```

2.2.3 Examples

Now we can calculate the area of the square we introduced in example 5 in section 2.1.4 as follows.

```
testEx5'' = area (ex5 :: SExpr)
```

The result of this example is as follows.

```
$ F 4.0
```

2.3 Module: Extend Shape - Rectangle

2.3.1 Syntax

In this module, we extend the shapes module to have an additional shape. The new shape is a *Rectangle* which has a center, width and height represented by terms (must evaluate to floats).

```
data Rect t = Rec (Point t) t t
```

2.3.2 Operations

This module would extend the 3 operations of shapes: pretty printing, evaluation, and area. The implementation of rectangle instance for these operations is very similar to the square instances in sections 2.1.3 and 2.2.2.

2.3.3 Examples

Similar to squares, we can now make rectangular shapes and use primitive values as arguments. Below is an example of a rectangle using primitives from example 1 in 2.1.4.

```
ex8::(Cond :<: t, Prim :<: t, PVal :<: t, Shape :<: t, Point :<: t, Rect :<: t) => Term t
ex8 = (rect (point (float 3) (float 4)) (float 7) ex1)
```

The example above can be used after casting it to the appropriate type in the evaluation, pretty printing and area operations as follows.

```
type RExpr = Term (Cond :+: Prim :+: PVal :+: Shape :+: Point :+: Rect )

testEx8 = pretty $ eval (ex8 :: RExpr)
testEx8' = pretty (ex8 :: RExpr)
testEx8'' = area (ex8 :: RExpr)
```

Running the two examples above give the following results.

```
$ center: (3.0, 4.0) height: 7.0 width: 2.0
$ rect: center (3.0, 4.0) height: 7.0 width: ((2.0 * 3.0) + -4.0)
$ F 14.0
```

2.4 Module: Extend Shape Operation - Circumference

2.4.1 Semantic

In this module we extend the shape module to have a new operation called *Circumference* which calculates the circumference of a shape. The return type of this operation is a float. We also need to simplify primitive expressions in shapes in order to calculate the height and width used in calculating the circumference. Therefore, we need to use the same semantic value of primitives and area.

```
class Functor t => Circumference t  where
  circumferenceAlg :: t (PVal ()) -> PVal ()
```

2.4.2 Operation

We extend the circumference operation for shapes and primitives in order to calculate the circumference of shapes which include primitives in them. We show the circumference instance of *Rectangle* below.

```
instance Circumference Rect where
  circumferenceAlg (Rec (P x y) h w) =
    case (h, w, x,y) of
        (F h',F w',F _,F _) -> F $ (2 * h') + (2 * w')
        _ -> error "Type error:  non-float values"
```

2.4.3 Examples

Now we can calculate the circumference of the rectangle we introduced in example 8 in section 2.3.3 as follows.

```
testEx8_'' = circumference (ex8 :: RExpr)
```

The result of this example is as follows.

```
F 18.0
```

2.5 Module 3: Robotic Moves

2.5.1 Syntax

In this module, we introduce a new language that controls robotic moves and uses our previous modules for primitives and shapes. We use the following data types to represent the moves enabled for a robot.

```
data Dir = Up | Down | LeftD | RightD

data ShapeCorner = TopLeft | TopRight |  BottomRight | BottomLeft

data Move t = Go
            | JumpTo (Point t)
            | JumpToCorner ShapeCorner t
            | Change Dir

data Moves t = L [t]
```

These commands control how the robot moves from one location to the other and in which direction. *Go* moves the robot one step in its direction. *JumpTo* moves the robot directly to a new location. *Change* changes the direction of the robot. *JumpToCorner* jumps to a the corner specified of the shape specified by the term passed.
Moves is another data type which holds a list of moves. You can see by now how everything we had so far can be integrated in this module.

2.5.2 Semantic

In this module we can directly extend pretty printing to represent moves. However, non of the previous evaluation functions can be extended directly because the semantic is different with respect to moves. For primitives, we evaluate every expression to either boolean or float. For shapes, we evaluate to *ShapeDom* of a centre, width and height. In this module, we need to evaluate every move to a state which holds the robot location and direction. This state has to be updated after every move.

```
data State t = St Dir (Point t)
```

Therefore, we extend our evaluations from shapes which allows us to simplify primitives and use shapes in our robotic moves. We build a new data type to represent the complete evaluation value in this module (*ValueM*). Our evaluation semantic (*EvalSem*) would have a function type which allows us to take the current state as input and return the updated state afterwards.

```
type ValueM = Term (PVal :+: ShapeDom  :+: State)

type EvalSem = ValueM -> ValueM

class Functor t => EvalM t where
  evalAlgM :: t EvalSem -> EvalSem
```

2.5.3 Operations

We will have the two main operations in this module: evaluation and pretty printing. The most interesting part in this module is the evaluation. Everything we evaluate in the shape module is exactly the same in this module, but it changes when working with robotic moves.

The location is represented by a point (X,Y). Going down decrements the y coordinate, going Up increments the y coordinate, going to the left decrements the x coordinate and going to the right increments the x coordinate.
When evaluating the shape for the *JumpToCorner* command, we get back the value (*S center height width*). We use this result to calculate the corner. This shows a use case of information hiding because we do not access the details of the shape. However, we use this abstract representation to calculate the corner location.
We show the evaluation instance of *JumpToCorner* below. It first projects the state and then projects the shape. After checking that the shape has valid float values, it performs basic arithmetic's to calculate the corner location. The corner location is then returned in the updated state and the direction remains unchanged.

```
findCorner :: ShapeCorner -> Point (Term t) -> Term t ->  Term t -> Point (Term t)
findCorner c (P x y) h w =
    case (project x, project y, project h, project w) of
        (Just(F x'),Just(F y'),Just(F h'),Just(F w')) -> cornerLocation c x' y' h' w'
        _ -> error "Type error:  non-float values"

evaljumpToCorner :: ShapeCorner -> Term t ->  Term t -> Term t
evaljumpToCorner c s m =
    case evalState m of
        (St d p ) -> case (project s) of
                        (Just (S p h w)) -> state d (findCorner c p h w )
        _ -> error "Type error:  non valid shape"

instance EvalM Move where
  evalAlgM (JumpToCorner c s) m = evaljumpToCorner c (s m) m
```

We show the evaluation instance of *Moves* below which allows us to sequence moves in a list. The resulted state of every move is fed into the next.

```
instance EvalM Moves where
  evalAlgM (L [])     m = m
  evalAlgM (L (x:xs)) m =
    case project (x m) of
        Just (St d p ) -> evalAlgM (L xs) (state d p)
```

2.5.4 Examples

In the following examples, we build a sequence of moves. Example 13 represents a sequence of 3 moves. Example 8 is in section 2.3.3 which represents a rectangle shape.

```
ex13::(Cond :<: t,Prim :<: t,PVal :<: t,Shape :<: t,..,Move :<: t,Moves :<: t) => Term t
ex13 = movseq [jumpTo $ point (float 3)(float 5),jumpToCorner BottomLeft ex8,go]
```

The example above can be used after casting it to the appropriate type in both the evaluation and pretty printing methods as follows.

```
type MExpr = Term (Cond :+: Prim :+: PVal :+: Shape :+: .. :+: Move :+: Moves)

testEx13 = pretty $ evalM (ex13 :: MExpr)
testEx13' = pretty (ex13 :: MExpr)
```

Running the two examples above give the following results.

```
$ (Dir: RightD , Loc: (3.0, 0.5))
$ jump to (3.0, 5.0) ;
    jump to BottomLeft corner of
        rect: center (3.0, 4.0) height: 7.0 width: ((2.0 * 3.0) + -4.0) ;
    Go
```

2.6 Module: Extend Move Operation - ShapeToMoves

2.6.1 Semantic

In this module, we extend the robotic moves module to have a new operation called *shapeToMoves*. This operation takes a shape and gives back the sequence of moves needed to draw it. We want to return an actual value of type *Moves* and take as input an actual value of type *Shape*. We need to simplify primitive expressions in shapes in order to calculate the height, width and center point. The result (*Moves*) can then be passed to the evaluation and pretty print operations of the move module to evaluate the moves or print them. Therefore, the semantic value is as follows.

```
type MovVal = Term (PVal :+: Shape :+: Move :+: Moves :+: Rect )

class Functor t => ShapeToMoves t  where
  shapeToMovesAlg :: t (MovVal) -> MovVal
```

2.6.2 Operation

We extend the *shapeToMoves* operation for primitives to simplify all diminutions. For shapes, we have to build a value of type *Moves* representing the appropriate sequence of moves. The following is the *shapeToMoves* instance for the shape *Square*. The *goes* function returns a list of *Go* commands.

```
squareMoves :: (Move :<: t,..,Moves :<: t) => Term t -> Term t -> Term t -> Term t
squareMoves x y l    =
   case (project x, project y,project l) of
           (Just (F x'), Just(F y'), Just(F l') ) ->
             let lb = jumpToCorner BottomLeft $ square (P x y) l in
             let (L steps) = goes l' in
             let dir1 = change RightD in
             let dir2 = change Up in
             let dir3 = change LeftD in
             let dir4 = change Down in
             movseq $ [leftB, dir1 ] ++ steps ++ [dir2] ++ steps ++ [dir3] ++ ..
          _ -> error  "Type error:  non-float values"

instance ShapeToMoves Shape where
  shapeToMovesAlg (Square (P x y) l) = squareMoves x y l
```

ShapeToMoves has to fail in we pass a move because it has no meaning in this context.

```
instance ShapeToMoves Move where
  shapeToMovesAlg m = error  "Type error:  expecting a shape"

instance ShapeToMoves Moves where
  shapeToMovesAlg m = error  "Type error:  expecting a shape"
```

2.6.3 Examples

Now we can generate the sequence of moves for drawing the square we introduced in example 5 in section 2.1.4 and also evaluate its state after running that sequence of moves.

```
testEx8M = pretty $ shapeToMoves (ex8 :: MExpr)
testEx8ME = pretty $ evalM $ shapeToMoves (ex8 :: MExpr)
```

The results of theses examples are as follows.

```
$ jump to BottomLeft corner of sqr: center (3.0, 4.0) side len: 2.0 ;
    change dir to RightD ; Go ; Go ; change dir to Up ; Go ; Go ;
    change dir to LeftD ; Go ; Go ; change dir to Down ; Go ; Go
$(Dir: Down , Loc: (2.0, 3.0))
```

2.7 Module: Extend Move - Step

2.7.1 Syntax

In this module we extend the robotic move module to have an additional move. The new move (*Stept*) allows the robot to take multiple steps in its current direction. It takes a term as an argument which must evaluate to a float.

```
data Step t = Steps t
```

2.7.2 Operations

This module would extend the 2 operations of moves: pretty printing, and evaluation. The implementations of these instances for *Step* is very similar to other commands in section 2.5.3.
Note that we can not extend the operation *ShapeToMoves* unless we introduce a new semantic value as follows.

```
type MovVal' = Term (PVal :+: Shape :+: Move :+: Moves :+: Rect :+: Step )
```

Doing this can be useful and more efficient to make the sequence of moves shorter by short-cutting the number of *Go* commands. However, it requires duplicating all the previous work and dealing with all the overhead this AlaCarte process takes.

2.7.3 Examples

Similar to example 13 in section 2.5.4, we can now build a sequence of 4 moves in example 15 which uses the additional move *Step*. This successfully uses all the modules above in building an expression.

```
ex15 :: (Cond :<: t,Prim :<: t,.., Move :<: t,Moves :<: t,Step :<: t) => Term t
ex15 = movseq [jumpTo $ point (float 3)(float 5),..,steps $ float 5]
```

The example above can be used after casting it to the appropriate type in both the evaluation and pretty printing methods as follows.

```
type SMExpr = Term (Cond:+:Prim:+:PVal:+:Shape:+:Point:+:Rect:+:Move:+:Moves:+:Step)

testEx15 = pretty $ evalM (ex15 :: SMExpr)
testEx15' = pretty (ex15 :: SMExpr)
```

Running the two examples above give the following results.

```
$ (Dir: RightD , Loc: (8.0, 0.5))
$ jump to (3.0, 5.0) ;
    jump to BottomLeft corner of
        rect: center (3.0, 4.0) height: 7.0 width: ((2.0 * 3.0) + -4.0) ;
    Go ;
    Steps: 5.0
```

3 Techniques

We are using the techniques of data type à la Carte mainly and extensively in this project. This techniques solves the expression problem using smart constructors, fixed points and injections. In addition, using data types is a type of abstraction used to support the modularity of the system. A little information hiding has been used in evaluating moves by hiding the details of the shapes passed to them as in section 2.5.3. This project also intends to use a good modular design by making every module independent of the others (separation of concens).

4 Challenges and Design Decisions

4.1 Extending Primitives with Floats

In the beginning, I thought of extending the primitive values to include floats, but this turns out to be more tedious. For example, I was trying to do something like the following to test if the value type is float or of other primitive type.

```
evalP1' :: Op1 -> ValueE  -> ValueE
evalP1' o@Neg e =
    case project e of
      Just (F f) -> inject(F (negate f))
      Just (B b) -> inject (evalP1 o (B b)  )
      _ -> error  "Type error:  non-valid values"
```

This did not work because I am assumed to do another project call on the same term e to get the other type.

```
evalP1' :: Op1 -> ValueE  -> ValueE
evalP1' o@Neg e =
    case project e of
      Just (F f) -> float (negate f)
      _ -> case project e of
            Just pv -> inject (evalP1 o pv )
            _ -> error  "Type error:  unary operator applied to ..."
```

This makes working with primitive values very very tedious because I have to build nested case-of statements to do a simple evaluation. Therefore, I decided to build a new primitive value library which uses floats instead of ints because it makes more sense in my domain.
A solution that was suggested by the professor is to introduce a view type and write another project function that allows us to pattern match directly on different types of an either type. I would leave this as a future work.

4.2 Using More specific Terms

Another challenge I have been through was working with more specific Terms. In the following code, I was confused about using a term t as the first argument to $Square$ or use $(Point\ t)$ which makes the argument more specific. I then thought that using $(Point\ t)$ makes type checking easier later, otherwise, I can expect anything to be passed to this first argument and not necessarily a point.

```
data Point t = P t t
data Shape t =  Square (Point t) t | ...
```

As in the following code, if $Square$ had a general term instead of $Point$, I have to check that it is actually a point and then check both its x and y arguments, but now, it can all be done in one line.

```
  evalAlg (Square (P x y) l)  =
    case (project x, project y,project l) of
        (Just (F x'), Just(F y'), Just(F l') ) ->
            shapeDom (P (float x') (float y')) (float l') (float l')
        _ -> error  "Type error:  non-float values"
```

4.3 Re-factoring Decisions

For building the evaluation function of $Moves$, we had to reuse all the evaluation code from the previous modules which caused a lot of code duplication. At the beginning, I thought I would do the following for evaluating Shapes in a separate function.

```
evalShape :: (ShapeDom :<: t,PVal :<: t) => Shape(Term t) -> Term t
evalShape(Square (P x y) l) =
  case (project x, project y,project l) of
        (Just (F x'), Just(F y'), Just(F l') ) ->
            shapeDom (P (float x')(float y')) (float l') (float l')
        _ -> error  "Type error:  non-float values"
evalShape (Vline x y1 y2) = ...
```

This works well at the beginning for all data types in the $Shap$ module and I was able to directly call the function above from the class instance $Eval$ of a $Shape$ and assumed to be able to call it from the $Moves$ module.

```
instance Eval Shape where
  evalAlg s   = evalShape s
```

However, this failed when I tried to call the same function from the *Moves* Module because I am expecting an state environment (m) to be passed to every project call.

```
instance EvalM Shape where
  evalAlgM (Square (P x y) l) m =
    case (project (x m), project (y m),project (l m)) of
    ...
```

Therefore, I had to go back and make every *Shape* case its own function and instead of passing the entire value of type shape, I pass the arguments of every case. The following is the re-factored function for *Square*.

```
evalShapeSquare :: (ShapeDom :<: t,PVal :<: t) => Term t -> Term t -> Term t -> Term t
evalShapeSquare x y l    =
   case (project x, project y,project l) of
        (Just (F x'), Just(F y'), Just(F l') ) ->
          shapeDom ( P (float x') (float y')) (float l') (float l')
        _ -> error  "Type error:  non-float values"
```

Therefore, in the *Eval* instance of *Shape*, I pattern match on the shapes and call the function associated with each case.

```
instance Eval Shape where
    evalAlg (Square (P x y) l) = evalShapeSquare x y l
    evalAlg (Pt (P x y))    = ...
```

Now this code can easily be used for in the *moves* module because we apply the environment to every argument before passing it to the evaluation function.

```
instance EvalM Shape where
    evalAlgM (Square (P x y) l) m = evalShapeSquare (x m) (y m) (l m)
    evalAlgM (Pt (P x y))       m = ...
```

5 Link to the project

This is the link to the project: `https://github.com/alkubaig/ShapeAndRobot`.