A DSL for Simulation of Robot Motion
DSL Report Card

Xuhao Zhou
School of EECS
Oregon State University

June 8, 2016

1 Introduction

This is a language for robot motion simulation. We may use this language in solving path-finding problem. For example, finding a path in a maze. Also, this DSL can play a role as educational programming language and helping students learn the concepts of conditional branching, looping and procedural abstraction. The Karel language inspired us to implement this language. However, This DSL has its unique features compared to Karel. Embed in Haskell offers an advantage to implement a path-finding algorithm because functional programming is good at backtracking search.

2 Users

First, this language is designed for beginners who interested in computer programming. The visual interface attracts beginners to use this DSL and understand the basic concept of the computational logic because they get what they did immediately. Second, maze lovers can use this DSL to design a maze and let a robot to find out a path.

3 Outcomes

The outcomes of executing a DSL program are to change the state of the robot and the state of the world (environment). For example, we can use the built-in move function to change the position of a robot. Also, we can control a robot to pick up an item or put down an item. Last but not least, given a starting point and an ending point, the robot can find a path. Output result is represented by a visual interface.

4 Use Cases / Scenarios

(1) Trivial example: we use grid function to visualize the output result. And we use prog function to run a list of operations [move Up, move Rt] with an initial state exState.

The output result is returned as a grid and some information lines. In the grid, "R" represents the Robot; "@" represents a position of a path; "#" represents a wall; "*" represents items; "_" represents a open space that a robot can move into. In addition, "Robot Pos: (3,1)" shows that the position of the robot is (3,1);

"Robot Has: ["Blueberry"]" means that the robot has an item named Blueberry. It can have many items, which can be represented by a list. "(3,1) Has: ["Language"]" means that there is an item named Language at the position (3,1). Also, a position can have many items, which can be represented by a list. "Step: 0" means the robot did not move any step. The number represents that the number of the positions that the robot has moved. "ErrorInfo: " is followed by an error message.

Figure 1

(2) Example program 1: While the position of the robot has one item or more, the robot moves left; then trying to move up, if it fails then moves down 3 times.

Input:

```
grid $ prog [while hasItem (move Lt), try (move Up) (iterate 3 (move Dn))] exState
```

Output:

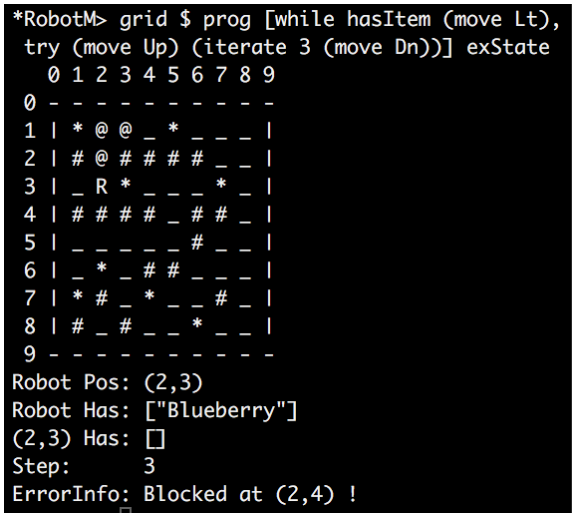

Figure 2

(3) Example program 2: If the bag of the robot is empty then the robot picking up an item else putting down an item; then the robot moves right 5 times; then while the position of the robot has at least one item, it tries to move down, if it fails then moves left.

Input:

```
grid $ prog [iff bagEmpty pick putItem, iterate 5 (move Rt),
while hasItem (try (move Dn) (move Lt))] exState
```

Output:

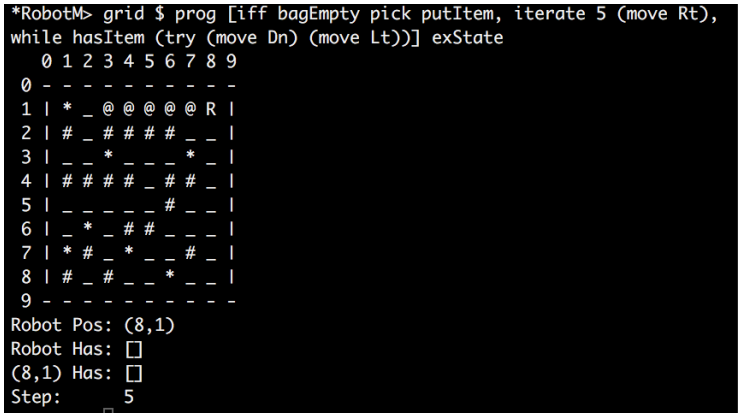

Figure 3

(4) Example program 3: The robot finds a path from (7,8) to (1,1); then moves down 2 times, then if the position of the robot has at least one item then picking up otherwise putting down an item from its bag.

Input:

```
grid $ prog [path (7,8) (1,1), iterate 2 (move Dn), iff hasItem pick putItem] exState
```

Output:

Figure 4

5 Basic Objects

(1) Pos represents a position

```
type Pos = (Int,Int)
```

(2) Item is a string

```
type Item = String
```

6 Operators and Combinators

Operators:

(1) World includes the information of each position

```
type World = [(Pos, Maybe [Item])]
```

(2) State includes position of the robot, the items that the robot has, the World environment, the position history that the robot has moved

```
type S = (Pos, [Item], World, [Pos])
```

(3) The type of a World in printing

```
type PWorld = [(Pos, String)]
```

(4) The type of a operation

```
type Step a = ExceptT Err (State S) a
```

(5) Prog is a list of operations

```
type Prog = [Step ()]
```

(6) a data type of directions

```
data Dir = Rt|Lt|Up|Dn deriving (Show,Eq)
```

(7) Define an Err date type for error messages

```
data Err = BlockedAt    Pos
         | CannotPickAt Pos
         | CannotPutAt  Pos
         | Unreachable  Pos
  deriving (Eq)
```

(8) an example state

```
exState :: S
exState = ((3,1), ["Blueberry"], exWorld, [])
```

(9) an example World

```
exWorld :: World
exWorld = [((1,1), Just ["Apple"]), ...]
```

(10) noop function return a ()

```
noop :: Step ()
noop = return ()
```

(11) Given direction and a position, return a new position

```
dir :: Dir -> Pos -> Pos
```

(12) Check whether a position is clear

```
isClear :: Pos -> S -> Bool
```

(13) Check whether the current position has item(s)

```
hasItem :: S -> Bool
```

(14) Check whether the bag of the robot is empty

```
bagEmpty :: S -> Bool
```

(15) Move right, left, up or down

```
move :: Dir -> Step ()
```

(16) The robot picks up an item at its position

```
pick :: Step ()
```

(17) The robot put down an item at its position

```
putItem :: Step ()
```

(18) Decrease an item at the specific position

```
decW :: World -> Pos -> World
```

(19) Increase an item at the specific position

```
incW :: World -> Pos -> String -> World
```

(20) Get a path from position A to position B

```
path :: Pos -> Pos -> Step ()
```

(21) A helper function for path finding

```
trylist :: [Pos] -> Pos -> Pos -> [Dir] -> Step ()
```

(22) A helper function for path finding

```
getmoves :: Pos -> Pos -> [Dir]
```

(23) A helper function for path finding

```
find :: [Pos] -> Pos -> Pos -> Step ()
```

(24) Helper functions for path finding

```
rtof :: Pos -> Pos -> Bool
ltof :: Pos -> Pos -> Bool
upof :: Pos -> Pos -> Bool
dnof :: Pos -> Pos -> Bool
neof :: Pos -> Pos -> Bool
swof :: Pos -> Pos -> Bool
nwof :: Pos -> Pos -> Bool
seof :: Pos -> Pos -> Bool
```

(25) Convert a World to a list of signs

```
pathworld :: S -> PWorld
```

(26) Print each line of the PWorld list

```
printLn :: (Int,Int) -> PWorld -> [String]
```

(27) Add number (top and left side) in the grid

```
addNum :: [String] -> [String] -> [String]
```

(28) Calculate the number of row and column

```
countW :: PWorld -> (Int,Int)
```

(29) Print a World

```
nprint :: PWorld -> IO()
```

(30) A helper function in printing a grid

```
pp ::(Either Err (), S) -> IO()
```

(31) Print a grid

```
grid :: (Either Err (), S) -> IO ()
```

(32) Only print Step and Error Information

```
nogrid :: (Either Err (), S) -> IO()
```

Combinators:

(1) While condition

```
while :: (S -> Bool) -> Step () -> Step ()
```

(2) If condition

```
iff :: (S -> Bool) -> Step () -> Step () -> Step ()
```

(3) Iterate n times operation

```
iterate :: Int -> Step () -> Step ()
```

(4) Try the first operation, if it fails then does the second operation.

```
try :: Step () -> Step () -> Step ()
```

(5) Run a list of operations

```
prog :: Prog -> S -> (Either Err (), S)
```

7 Implementation Strategy

This DSL uses shallow embedding to implement. We can directly manipulate the domain and do not need to write interpretation functions. In this DSL, we can apply an operation function on a state directly. For example, the move function directly changes the position of the robot without using an interpretation function. Therefore, using shallow embedding to implement a DSL is very convenient and fast. In addition, there is no obvious disadvantage of shallow embedding encountered in this DSL.

8 Related DSLs

(1) The Karel [1] language is topically similar to our DSL. However, This DSL has its unique features compared to Karel, such as the path-finding feature and representing items as a string list. In addition, although Karel is originally based on Pascal, there are different versions based on different programming language, such as C / C++, Java and Python. This robot DSL is implemented in Haskell. Furthermore, there is a special feature in Karel. It can define a code block as a function. And, users can call a code block when they need it. For example, we can define TurnRight() using 3 TurnLeft() and call TurnRight() in the main function.

```
void TurnRight()
{
        TurnLeft();
        TurnLeft();
        TurnLeft();
}
int main()
{
        TurnOn();
        TurnRight();
        Move();
        PickBeeper();
        TurnOff();
}
```

(2) The second related DSL is named robot-karel [2], which is a clone of Robot Karol in JavaScript. But, the author timjb defines a Karel library and core functions in Haskell. The robot-karel language is technically similar to our DSL.

For example,

(a) In robot-karel, the author defines a Karel type that is similar to our Step type. The idea is to use State monad to manipulate one state and use ErrorT or ExceptT to return error information when one error happens.

```
newtype Karel a = K { runK :: ErrorT KarelError (State World) a }
                  deriving (Monad, MonadError KarelError, MonadState World)

type Step a = ExceptT Err (State S) a
```

(b) The definition of the World type is different in which between robot-karel and our DSL. In robot-karel, the author defines the World type and related types as below.

```
newtype World = W { getW :: Row (Row Field) }
data Row a = Row {before  :: [a]
                 , current :: a
                 , after   :: [a]
                  } deriving (Show)
data Field = Field {marker :: Bool
                   , bricks :: Int
                    }
```

In our DSL, we define the World type and related types as below:

```
type World = [(Pos,Maybe [Item])]
type Item = String
```

In short, the World type in robot-karel seems like handy, but it is a little hard to read and manipulate.

9 Design Evolution

(1) We have changed the definition of the Step type some times. At the beginning, the type of the Step was `State -> Maybe State`, because we wanted to use Maybe type to represent error situation. Soon, we wanted to know the error information and changed the type Step as `State -> Either String State`. Shortly afterwards, we needed to store an old state even though encountering error, because we wanted to print a state rather than only error info. We changed the Step to `State -> Either (String,State) State`. Finally, we wanted to use monad transformer and defined the Step as `type Step a = ExceptT Err (State S) a`. We believe that this definition is more convenient and concise.
(2) We need to consider the performance of code when designing a DSL. Originally, we wrote the try function as below:

```
try :: Step () -> Step () -> Step ()
try f g = do s <- lift get
             case runState (runExceptT f) s of
               (Right (),s') -> f
               (Left err,s') -> g
```

But, the implementation of this function would cause a performance problem. We rewrote the implementation of this function as below and successfully solved the performance problem.

```
try :: Step () -> Step () -> Step ()
try f g = do s <- lift get
             catchError f (\_ -> lift (put s) >> g)
```

10 Future Work

1. We can try to add the binding idea into this DSL. For example, using a name to refer a code block in a program.
2. We can try to write functions to randomly compose a path finding strategy.
3. We can try to define a World randomly.
4. We need to refine the functions of pretty printing or use OpenGL to visualize the output result.

5. The concrete benefit to extend a shallow DSL into a deep DSL is that we can add many interpretations into a DSL to enable analyses and transformations. For example, pretty printing. Also, we can define our type system and override other features of the host language.
6. A GUI interface plays an important role in a DSL, because GUI offers a convenient way for users to interact with a DSL. Also, users are most likely to click buttons instead of inputting text-based commands.

References

[1] Richard E. Pattis. Karel The Robot: A Gentle Introduction to the Art of Programming. John Wiley & Sons, 1981. ISBN 0-471-59725-2.
[2] https://github.com/timjb/robot-karel