# RobotLang: A Domain-Specific Language for Land-Roving Robots

Final Project Report for the conference on Robotic Function Programming (RFP), Spring 2016

Shane McKee
Oregon State University
2500 NW Monroe Avenue
Corvallis, Oregon
mckeesh@oregonstate.edu

Minfeng Wen
Oregon State University
2500 NW Monroe Avenue
Corvallis, Oregon
wenm@oregonstate.edu

## 1. INTRODUCTION

### 1.1 What is the domain?

The goal of this project was to implement a simple Domain-Specific Language for a land-roving robot.

### 1.2 Who are the users?

This language allows programmers to control the robot by sending it missions to complete.

### 1.3 What kinds of things can those users do with your project?

Our DSL allows developers to move the robot, keep track of energy and energy usage, and collect samples such as stones and sand. It also provides simple interfaces to access sensor data such as temperature, humidity, atmospheric pressure, and images of surroundings. The DSL is also designed in such a way that adding a new sensor is easy to do. As this project is not implemented with a particular set of hardware in mind, we did not implement any actual data collection code. Users must implement that part of the code themselves and use our code to set values.

### 1.4 Are there similar tools/languages that exist already, and if so, how are they different?

To our knowledge, there are not any languages that solve this narrow set of requirements. We do know that the Mars Rover and other similar robots must have certain APIs or DSLs to control them, but none seem to be publicly accessible.

## 2. BACKGROUND



*RFP '16 June 9, 2016, Corvallis, OR, USA*



DOI: 10.475/123_4

### 2.1 Important types

- **Robot** Robot is a state monad that allows us to access state information about the robot.
- **RobotE** The RobotE monad allows us to combine the Robot state monad with the IO and Maybe monads in order to better handle IO and errors.
- **Energy** This type is an integer that represents the energy level of the robot.
- **Pos** This type is a pair of integers that represents the current position of the robot.
- **Load** This type is an Object that represents a physical object that the robot is carrying.
- **Schedule** This type is a list of Actions that represents the list of commands that robot must complete. This could also be called the robot's "Mission".
- **Object** Object is a type that represents either Sand, Stone, or and Empty value.
- **Action** Action is a type that is represented by the MoveBy function, PickUp function, Drop function, GetData function, and DoNothing.
- **World** This data type is used to hold sensory data. This data is not kept over time, as it is expected that the data will be sent back to a server for storage.
- **RobotState** This contains information about the state of the robot.

### 2.2 Important functions

- run :: Show a ⇒ Energy → Pos → RobotE a → IO () Takes three parameters (Energy, Position, and the RobotE monad), runs the actions in the schedule, and prints the final state.
- run' :: Energy → Pos → Schedule → IO () Takes three parameters (Energy, Position, and Schedule), runs the actions in the schedule, and prints the final state.
- moveBy :: (Int, Int) → RobotE () Alters the (x, y) position in the current state and deducts the energy needed to perform the movement. If energy is insufficient, the user is asked if they want to add more energy in order to complete that action.

- pickUp :: Object → RobotE () Picks up an object if there is nothing in its current load.
- drop :: RobotE () Sets the current load to empty.
- Other various setter and getter functions

# 3. DESIGN

## 3.1 RobotE

The design for RobotE came out of several different iterations of improvements and refactoring. From the beginning, we had the state passed into each function and returned from each function. We soon realized that passing the state around was not ideal, so we implemented the Robot monad, which allowed us to refactor functions without passing the state into and out of each one. After a while, we realized that we needed a way to handle errors and IO, so we combined these monads using Robot as a monad transformer applied to MaybeT IO. The resulting monad was RobotE, which is an improvement over the previous version because we can use state and IO while also being able to handle errors.

## 3.2 Sensors

We originally wanted to use a type class because it is easy to extend with new sensors. However, we encountered a problem in trying to do this. We could not understand how to add a type constraint to the type definition for Robot. In the end, our solution to this problem was implementing the World type with record syntax. This allows us to easily add a new sensor by adding a new field name. Since easy extensibility was our reason for using type classes in the first place, we did not see a need to implement Sensor as a type class.

## 3.3 IO

Initially, we had designed the code to separate get functionality and print functionality, but we later decided that it would be convenient to combine the getting and printing functions. This required that we combine the Robot and IO monads. The example of getting the Robot energy level is shown below:

```
getEnergy = do
  e <- liftM energy get
  lift . lift . putStrLn . show $ e
  return e
```

In this case, we can see that we are using both *get* from the Robot monad and *show* from the IO monad.

Another case where this is useful is in the *moveBy* function. We use the IO monad for output, to communicate that the robot is out of energy, and for input, to allow the user to choose if more energy should be added (charging the robot) before executing more actions. If the user chooses not to add more energy, it will simply stop executing actions.

# 4. USABILITY

This DSL provides many functions and types with which a user can run commands and select robot functions. However, we also have many methods that provide a nice boilerplate in case the user wants to extend our work even further. For instance, we have types with attributes that may be accessed very easily. If users desire to build more functionality into a robot with an even more specific purpose, they can use those attributes and use our setter and getter functions as a template for how access them. We even provide ways to add, remove, peek, and pop actions from the schedule to provide easy schedule manipulation.

# 5. FUTURE WORK

## 5.1 Recharging

Currently, the robot recharges a fixed amount only if it runs out of energy while moving. We also have movement being the only action that costs energy. Future work should include adding costs to other actions as well as allowing users to recharge dynamic amounts as its own separate action.

## 5.2 Extend Robot Functionality

In future, the DSL should be extended to cover more cases. For instance, adding more Object types, adding more sensors, providing analysis on all of the data collected, and providing a way for the robot to communicate. In addition, we should provide an interface for hardware devices such as thermometers, barometers, etc.