User Guide for our Funky Interpreter

Zheng Long Yang
Dina Benkirane

June 2019

1 Introduction

1.1 Abstract

The idea of this project is to build a mini interpreter which allows us to essentially create a pseudo language inside an environment which is built upon Haskell. Since Haskell language allows various scope for defining let's say a variable, this allows us to essentially modify every single element of our functions to adapt itself to new types of scopes.

The concept of building an interpreter resides in the fact that there are reserved spaces inside the compiler or interpreter which allows for a certain call to action or change to occur inside the machine's memory. For instance if we were to create a specific Type then we would allocate a certain amount of memory inside our memory modules to allow the definition of such Type. Haskell isn't any different, it allows us to create custom definition or keyword which operates as language reserved items (DataTypes, Definitions, etc.) which we will store inside an Environment

2 Helper functions and Other Useful Functions

Here is the section for special functions which will be used throughout the environment

2.1 specialForm2Exp (Symbol, Variable, Value)

Notice that there are many variants of Symbol that can be inserted inside the specialForm2Exp() function. Earlier we have described a few Types (Defined by Exp) which defines these Types depending on the Symbol added as argument. In this case this function allows us to extract the raw information contained inside an array and set them in a proper pattern for us to match it later which the environment will understand.

This will convert the expression we read regularly in a pattern of

$$Type\ variable\ value$$

For instance :

- define x 4 becomes EDefine variable Expression $\Rightarrow$ *Environment*

2.2 recursiveInsertion

Custom Function to extract tuples into a new Environment which exist outside of our current environment Note that we do this in a recursive pattern and we will call it for the "Let" keyword.

2.3 caseMatchedEval

Helper class to help match the actual information to the input-ed parameter.

2.4 caseEval

Helper class to help evaluate the datatypes.

2.5 test

Helper function to check if the the dataType and its comparedType is the same.

3 DataTypes, Primitives and Other Reserved Keywords

In this section we will take a look into a few of our environmental specific keywords that we made.

3.1 Define and EDefine

The define keyword serve to create variable our first goal is to convert the expression into something that the environment will understand we may use the specialForm2Exp defined for our "define" keywored. We will return it under the form of $Edefine\ Variable\ Expression$
Once this has been converted into the proper form we will store this inside our environement with our evalGlobal env (EData t cs). At this point we will want to store it inside our environment. We do so by calling our eval env expression and by the tuple inside our environment and the value itself

- $((Symbol, Value), Value) \Rightarrow Environment$

3.2 Lambda expressions and Elam

Lambda expressions are just lines of expressions which will be later compiled, thus we convert the expression into two parts the first one as a list of parameters the second the expression to evaluate.
Since the env store a data Value we have to send into the environement as a VLam (as defined earlier in the monad of data Value).

3.3 Let and Elet

Let is a bit different than the rest of the codes. In fact the Let keyword allows a locally declare information to be stored temporarily in a sub environment which we then have to delete or remove once we finished evaluation its body.

We first have to convert its form using specialForm2Exp so we can evaluate it. Once this is done we can now call in recursiveInsertion (Mentionned in the helper section) to recursively add in elements in a new environment.

3.4 EData and Data

Edata also comes into a special area we decided to utilise the EApp evaluator in order to evaluate it. Once again we have to convert the Data under a form that the environment understands. We can use specialForm2Exp defined for "data". We re-use the def for EApp since it takes in 2 types back to back and we can use this to convert the data into VData.
We'll have to do special operation on the data Types since they contain a Type and Data Constructors we'll have to extract it and put it into two seperate arrays one containing the types and the other with the lists of constructors. Keep in mind that inside Edata there are VPrim and VUnit which VPrim defines primitive types and VUnit which can indicate the end of our constructor. (Note that if we reached the end of the constructor then we'll find VUnit) So we can do this recursively but in our case we decided to filter both table and create a singular array containing inside the constructor which then we can send it with evalGlobal and insert the variables.

Issue with overloading argument. It would seem there is an error with one of the tests of datatype. We run into an error when we overload the datatype with too many arguments. To fix it, we believe we need to check the quantity of DataConstructors provided in the argument during the Evaluation phase. If we see that there are too many argument provided inside the environment we should stop and return an error. In other words, inside our environment we want to match uniquely with one DataConstructor if we match with others then we know we're comparing different DataTypes.

3.5 Cases and Ecase

Ecase and Case works the same as the other before once again we convert using specialForm2Exp so we can pass that into our evaluator. Once this is done, we

can call in test and caseEval function to help us establish the patterns between the cases and the body of their function. We can now detect if the case given as param are the same

4 Lexical and Dynamic Environment

4.1 How to implement Dynamic

The idea of dynamic environment is to be able to mutate variable and affect all the way through our code. In order to do so we should implement a local singular environment that works to keep in the value and retain it in its memory. For instance if we create a variable x and assign it to 1 (x=1), and we later wishes to affect it to (x=2), we would wish to change the variable in our singular environment. Keep in mind there are design patterns which allows this to be done notably the Singleton pattern however it is out of the scope of this class. Instead we should emulate a Heap and affect variable directly, in this scenario our heap is represented by our environment and so we would need to lookup the variable x and affect it

4.2 How to implement Lexical Environment

Contrary to the dynamic environment, lexical environment is immutable and thus values can possess the same name. Based on the principle of the dynamic environment, this time we want to do the contrary, we want to create a local environment for each information we wishes to store. Inside the main Environment, this one will contain a series of Sub-Environment (kind of like the ELet and Let Keyword). Each sub environment may contain a variable or information with the same name but this will not matter since they exist only inside their scope, thus we have created a lexical environement.

4.3 Set

Which brings us finally to the last part. How to deal with mutability. This scenario we will have to create 2 types of global environment the first one is the Dynamic Environment and the second on a Static Environment. Set will allow us to choose which information to set. Note by default values are immutable.

- If we want mutate a variable then we have to take the variable from the static environment and store it inside the Dynamic one and then delete it from its former Environement (Static).