SINH – SINH Is Not Haskell

A JavaScript-like programming language,
developed using Haskell (Happy & Alex)

Table of Contents

Outline

The first section of the report describes the basic installation and running procedure. The rest of the project report is divided into two parts. The first part contains the implementation of all the required features while the second part has details about the extra features implemented. The first part is further divided into sections in accordance with the project description.

Note: the code presented in this report is simplified version of the actual code, just to explain the implementation. Things like Monads and Errors may be omitted from the code, to improve the readability. The final version of the code is present in the source code, and in any case of discrepancy, please consider the source code as the final version.

Installation

This project is based on Stack and uses Haskell to develop a JavaScript-esque programming language, called SINH. The syntax for SINH is same as in the tutorials and project bundle, with extensions mentioned in the last part of the project. You can use the following commands to build and execute the program:

- `stack build` to build the project files
- `stack exec sinh` to run the shell
- `stack exec sinh <filename>` to run a program file, e.g. `stack exec sinh fact.txt`

Part 1: Question 1 - 13

First Class Functions

Question 1. *Complete CallFC and Fun cases in evaluate method in Interp.hs.*

The implementation of the First Class functions was very straight forward. On declaration, they were stored in the environment as ClosureV Value which stored the name, return type, body and the state of the environment at the time of declaration. Therefore, on the function call, the ClosureV value is restored and the body is evaluated using the given argument.

```
eval (Fun (x, t) body) env = ClosureV (x, t) body env

eval (CallFC fun arg) env = do
    (ClosureV (name, _) body env') <- eval fun env
     argV <- eval arg env
     eval body ((name, argV) : env')
```

Records

Question 2. *Implement Rcd and RcdProj cases in evaluate in file Interp.hs.*

The `Rcd` expression contained a list of (key, expression) pairs. Each expression was evaluated and the resulting value was stored in `RcdV` value as a list of (key, expression) tuples.

```
eval (Rcd xs) env =
     RcdV $ map (\(key, exp) -> (key, (eval exp env))) xs
```

Note: The implemented code is slightly different, but essentially does the same job.

For projection, the Value RcdV is restored, and the list is searched for the given key, and its corresponding value is returned.

```
eval (RcdProj exp str) env = do
     (RcdV record) <- eval exp env
     case lookup str record of
          Just v -> v
          Nothing -> Error "The given key does not exist"
```

Note: I used the lookup function here (and later on) as I saw the helper functions after I was done with the implementation. It functions same as `findrcdValue` in `Declare.hs`

Question 3. *Implement Rcd and RcdProj cases for pretty printer in file Declare.hs. Please note that you need to update pretty printer for expressions, values and types. Have a look at our example and figure out how to show records.*

To print the Record, the style given in the project description was adopted. The style is similar to how JavaScript prints the JSON Objects.

```
show (TRcd xs) = "{" ++
     intercalate ", " (map (\(key, t) -> key ++ ": " ++ show t) xs)
     ++ "}"

show (RcdV xs) = "{" ++
     intercalate ", " (map (\(key, v) -> key ++ ": " ++ show v) xs)
     ++ "}"
```

```
sinh>> {age: Int, ismale: Bool}
sinh>> {age: 30, ismale: true}
```

The projection is shown as a record followed by a key with dot in between, e.g. {}

```
showExp   (RcdProj e str) = show e ++ "." ++ str
```

```
sinh>> {age = 20, isMale = true}.isMale
```

Question 4. *Implement Rcd and RcdProj cases in techeck in file TypeCheck.hs.*

For Rcd, each expression in the list of (key, expression) pair was type checked and it's type was stored as a list of (key, type) tuple in TRcd.

```
tcheck (Rcd xs) env fenv =
     TRcd $ map (\(key, exp) -> (key, (tcheck exp env fenv))) xs
```

The RcdProj returned an expression that is associated with the given key, so type check for the projection returned the type of that expression. The type check for each expression is already done for us by the code displayed above.

```
tcheck (RcdProj exp str) env fenv = do
  TRcd record <- tcheck exp env fenv
  case lookup str record of
      Just t -> t
      Nothing -> Error $ "Record has no attribute " ++ show str
    t' -> Error $ "Type Error: Projection seems to be done on a
                                variable of type " ++ show t'
```

Type Declarations

Question 5. *Update pretty printer for the type declarations in file Declare.hs.*

In order to print type, we just printed the name of the type

```
show (TypDecl str) = str
```

Question 6. *Update execute and evaluate function for type declarations in file Interp.hs. Think about the possible changes!*

Since type declaration involves the type of the variable, not its value, this project made no changes to the logic of *Interp.hs* for this question.

Question 7*. Make sure type declarations do type check. You may think of possible changes in TypeCheck.hs.*

Instead of making changes in the last question, all changes to implement type declarations were made in *TypeCheck.hs.* As recommended by the project description, all the type declarations in program were replaced with concrete types. This was done before the main type check was run and which means that the program was run 3 times:

1. To replace all the types
2. Type Check
3. Evaluate

This is highly inefficient, therefore a better strategy is also explained at the end of this chapter. The code used to replace the type goes through each expression and replace any `TypeDecl` type with the concrete type (recursively).

We do the type replacement in the `checkProgram` function. The `main` function of the program is replaced with `main'` and then executed.

```
checkProgram :: Program -> Either String Type
checkProgram (Program typeEnv fds main) = do
  fenv <- checkFunEnv typeEnv fds
  main' <- replaceTypeInExp typeEnv main
  tcheck typeEnv main' [] fenv
```

The `replaceTypeInExp` function recursively replaces TypeDecl in all of the expressions:

```
replaceTypeInExp :: TypeEnv -> Exp -> Either String Exp
replaceTypeInExp typeEnv (Lit v) = Right $ Lit v
replaceTypeInExp typeEnv (Unary op e) = do
  e1 <- replaceTypeInExp typeEnv e
  Right $ Unary op e1
replaceTypeInExp typeEnv (Decl v t e1 e2) = do
  t1 <- replaceType typeEnv t
...
...
-- Other cases ommitted
```

The types were replaced using replaceType function, which once again uses recursive substitution. This enabled us to use type within a type (More in second section)

```
replaceType :: TypeEnv -> Type -> Either String Type
replaceType typeEnv (TypDecl name) = do
  case lookup name typeEnv of
    Just t -> replaceType typeEnv t
    _ -> Left $ "Type " ++ name ++ " has not been declared"
replaceType typeEnv (TRcd xs) = do
  let xss = map (mapperFunc3 typeEnv) xs
  a <- extractRights xss
  Right $ TRcd a
replaceType typeEnv (TVarnt xs) = do
  let xss = map (mapperFunc3 typeEnv) xs
  a <- extractRights xss
  Right $ TVarnt a
replaceType typeEnv (TFun t1 t2) = do
  t3 <- replaceType typeEnv t1
  t4 <- replaceType typeEnv t2
  Right $ TFun t3 t4
replaceType _ a = Right a
```

A better Implementation

A better way of implementation would be to replace type as you encounter them during the type check. I partly implemented this strategy, but with this strategy, I didn't have enough time to implement recursive type replacement.

```
tcheck typeEnv (Decl v t e1 e2) tenv fenv =
  case t of
    (TypDecl str) -> - In case of custom type, replace it
      case lookup str typeEnv of
        Just t2 -> tcheck typeEnv (Decl v t2 e1 e2) tenv fenv
        Nothing -> Error
    _               -> ... Type Check as normal
```

Variants

Question 8. *Update pretty printer for variants.*

Each variant looks like an HTML tag without the closing tag, printed within the '<' and '>'.

```
show (TVarnt xs)  =
     "<" ++ intercalate ", "
           (map (\(key, t) -> key ++ ": " ++ show t) xs) ++ ">"

show (VarntV str v t) =
     "<" ++ str ++ "=" ++ (show v) ++ " : " ++ (show t) ++ ">"

showExp (Varnt str exp t) =
     "<" ++ str ++ "=" ++ (show exp) ++ " : " ++ (show t) ++ ">"
```

Similarly, to print the case, we print the case variant in between "case" and "of". A new line is added, and then each case printed separately in one line. The cases are present as a `(label, variant, exp) : [(String, String, Exp)` tuple and printed as:

```
    | <Label=variant> => exp
```

```
showCaseV :: Exp -> [(String, String, Exp)] -> String
showCaseV e xs = "case " ++ show e ++ " of" ++ showCases xs
  where
    showCases :: [(String, String, Exp)] -> String
    showCases [] = ""
    showCases ((label, variant, exp) : xxs) =
      "\n| <" ++ label ++ "=" ++ variant ++
                          "> => " ++ show exp ++ showCases xxs
```

Question 9. *Complete the Varnt and CaseV in the evaluate method in Interp.hs.*

Evaluating Varnt returned the value of type VarntV after evaluating the expression.

```
eval (Varnt str exp t) env = VarntV str (eval exp env) t
```

To evaluate Case, we first evaluate the case and then for each case, match the label. If there is a match, we evaluate and return the corresponding expression.

```
eval (CaseV exp xs) env = do
  (VarntV label v1 _) <- eval exp env
  matchCase xs
  where
    matchCase :: [(String, String, Exp)] -> Value
    matchCase [] = Error "No case match"
    matchCase ((labelC, varName, e) : xxs) =
      if labelC == label then
        eval e ((varName, v1) : env)
      else
        matchCase name value xxs
```

Question 10. *Make sure Variants type check. Think about the possible changes in TypeCheck.hs!*

Type checking with variant required some out-of-the-box thinking to simplify the code. In tcheck function, we simply check the expression and return the associated type after matching.

```
tcheck (Varnt str exp t) env fenv = do
  t1 <- tcheck typeEnv exp env fenv
  if t == t1 then
     TVarnt [(str, t)]
  else
     Error "Type Error"
```

For the case type check, we made sure that the return type was same for all the given cases. Therefore, you can't have Case A with a Bool expression and Case B with an Int expression.

```
tcheck typeEnv (CaseV exp caseList) env fenv = do
      (TVarnt variantTypeList) <- tcheck typeEnv exp env fenv
      matchCaseType caseList Nothing
      where
        matchCaseType :: [(String, String, Exp)] -> Maybe Type -> Type
        matchCaseType [] (Just t) = Right t
        matchCaseType [] Nothing = Error
        matchCaseType ((label, varName, e) : xxs) t = do
            case lookup label variantTypeList of
              Just t -> (do
                t2 <- tcheck typeEnv e ((varName, t) : env) fenv
                if t == Nothing || t == (Just t2) then
                  matchCaseType xxs (Just t2)
                else
                  Error
              Nothing -> matchCaseType xxs t
```

In order for this code to work properly, we had to add a comparator operator for the TVariant. Two types of TVariants were equal if both of them had at least one common pair of (variable, type) in the list.

```
-- Declare.hs
instance Eq Type where
TVarnt a == TVarnt b = not $ null $ intersect a b
```

Exceptions

Question 11. *Add Raise and Try in the pretty printer.*

To print Try and Raise, we add key word “try” followed by the try expression and then “with” followed by the raise expression.

```
showExp (Try exp1 exp2) = "try " ++ show exp1 ++ " with " ++ show exp2
```

To print Raise, we printed the message saying “Runtime Error” followed by the error corresponding the error code raised.

```
showExp (Raise exp) = show exp

show (RaiseV v) = "Runtime Error: " ++
    case v of
      (IntV 0) -> "Division with zero!"
      (IntV 1) -> ...
      .....
      (StringV msg) -> msg -- Custom message string
```

Note: StringV is explained later in the second part.

Question 12. *Implement the Raise and Try in evaluate method in Interp.hs.*

To evaluate Try and Raise, we evaluated the try expression to see if it raises any exceptions. In case of exceptions, raise expression was then evaluated. Otherwise, the evaluated value for the first expression was returned.

```
eval (Try exp1 exp2) env = do
  v1 <- eval exp1 env
  case v1 of
    (RaiseV _) -> eval exp2 env
    v -> v

eval (Raise exp) env = RaiseV $ eval exp env
```

Question 13. *Make sure Raise and Try type check. Think about the possible changes in TypeCheck.hs!*

Just like in TypeScript, we type check both expressions but return the value of the first (Try) expression. The two expressions can have different types as in the given example.

```
tcheck (Try exp1 exp2) env fenv = do
  tcheck exp2 env fenv
  tcheck exp1 env fenv
tcheck (Raise exp) env fenv = tcheck typeEnv exp env fenv
```

Part 2: Free Extensions

For the second part, this project added the following features:

1. Improved error messages
2. Unified top-level and first-class function calls
3. A command line interpreter
4. Mutable Values
5. String Value and Type
6. Type Definition within a custom type
7. Type Check for declarations
8. Colour coded output

Improved Error Messages

To improve the error propagation, we used `Either String Type` as the return type for the tcheck. If type check passed, we returned `Right type`, otherwise `Left "Error message..."`.

For Runtime Errors that occur during evaluation (for e.g. Division with 0), we return the value `RaiseV` with an error code or an error message. To facilitate custom error messages, we introduced the string type (described later on in this section).

The errors have also been colour coded and shown in red colour.

```
sinh>> 5/0
Runtime Error: Division with zero!
sinh>>
```

We also changed the Parser.y file to introduce better Error Projection using a custom data type and monad functionalities:

```
data E a = Ok a | Failed String

thenE :: E a -> (a -> E b) -> E b
m `thenE` k =
   case m of
      (Ok a) -> k a
      (Failed e) -> Failed e

returnE :: a -> E a
returnE a = Ok a

failE :: String -> E a
failE err = Failed err

catchE :: E a -> (String -> E a) -> E a
catchE m k =
   case m of
      (Ok a) -> Ok a
      (Failed e) -> k e

parseError :: [Token] -> E a
parseError _ = failE $ "Parse error"
```

This implementation technique is taken from the official documentation of happy.

```
sinh>> var a = 2
Parse error: Unexpected syntax used in the program
sinh>>
```

Unified top-level and first-class function calls

Initially, the syntax for calling top-level functions and first-class functions was different, which is not ideal. Therefore, as suggested in the project description, the function calls were unified removing the need to use @ when calling top-level functions. We can still use @ to explicitly call a top-level function, but there is no need for it.

Whenever a function is called, we assume that it is a top-level function call. We do that due to two reasons:

1. First variable is a string instead of an expression
2. Multiple arguments are parsed

```
App : ...
    | id '(' Exps ')'                    { Call $1 $3 }
```

When evaluating `Call`, we lookup the Function Environment and then if Function is not found, then consider it as a First Class Function and evaluate the expression using `CallFC`.

```
eval (Call fun args) env =
  case findFunction fun fenv of
    Just (Function xs body) -> -- Evaluate TL function

    -- Otherwise evaluate FC function
    _ -> eval (CallFC (Var fun) (args !! 0)) env
```

Sinh shell interpreter

The idea for implementation was taken from Tutorial 7's solution provided. The implementation is done in `app/Main.hs`. You can run the shell using the following command:

```
➜ PROJECT_3035393157 git:(master) X stack exec sinh
Welcome to SINH! Current version: 1.0.0
Type :help to get full list of utility functions
sinh>> 1+1
2
sinh>>
```

The main function now initiates the shell on the `stack exec sinh` command. If additional argument of filename is provided, then the program in the file is executed (and shell is not initiated, in order to facilitate execution of `test.sh` script). The main function looks something like this:

```
main :: IO ()
main = do
  args <- getArgs
  case args of
    [] -> shell (ini)
    [fname] -> load2 fname
    _ -> do
      putStrLn $ yellow ++ "Invalid arguments! Proper use:"
      putStrLn "\t\t<stack build> sinh <filename>"
```

Welcome Message

When the shell starts, a welcome message with current version and help command is displayed.

```
ini :: Repl ()
ini = do
  liftIO $ putStrLn $ "Welcome to SINH! Curent version: "  ++ version
  liftIO $ putStrLn $ "Type :help to get full list of utility functions"
```

Additional commands

There are utility commands in the shell which can be accessed using `:` followed by the command. The shell supports 3 types of commands (with shortcut commands also provided).

```
➜ PROJECT_3035393157 git:(master) X stack exec sinh
Welcome to SINH! Current version: 1.0.0
Type :help to get full list of utility functions
sinh>> :help
SINH language has following utility commands:
     1. :load <filename>       To load a program file
        :l <filename>
     2. :quit         To quit the shell command
        :q
     3. :help         Get full list of utility functions
```

With `:load filename.txt` command, you can load any pre-scripted program. However, unlike the direct method of loading file (`stack exec sinh filename.txt`), this method returns to the shell after the execution.

Autocomplete commands and shortcuts

Once again, inspired by the design in Tutorial 7, an option to autocomplete commands was added. This means if you type `:qu` and then pressed TAB, it would autocomplete to `:quit`. Additionally, to facilitate quick execution, shortcuts were introduced, like `:l` for `:load`.

Since `:load` prompted a second argument which is a filename, autocompletion option for files was also introduced.

Mutable Values

Mutable values are values that can be changed after they are defined. The design provided in Tutorial was adopted with some changes. Mutable variables have to be explicitly declared and they can be accessed only with a prefix $.

```
data Stateful t = ST (Memory -> (t, Memory))

instance Monad Stateful where
  return val = ST (\m -> (val, m))
  (ST c) >>= f =
    ST
      (\m -> let (val, m') = c m
                 ST f' = f val
             in f' m')

type Memory = [Value]
```

Instead of using `Seq Exp Exp` to define a sequence of instructions, we used an extra `Exp` in `Assign` to define the next expression, similar to the `Decl` expression.

```
data Value
  = ...
  | AddressV Int                -- new
  deriving Eq

data Type
  = ...
  | TMutable Type               -- new

data Exp = ...
         | Mutable Exp              -- new
         | Access Exp               -- new
         | Assign Exp Exp Exp       -- new
         deriving Eq
```

The Mutable Variables can be of type Int, Bool or String (mentioned below). The syntax used for the can be broken down into 3 main parts:

- Keyword ‘mutable’ is used while defining these variables
- ‘$’ prefix is used to access their value
- ‘<-‘ is used while assigning a value to the mutable instead of an ‘=’.
    - This currently has no design value, but I used a different symbol (Token) as I planned to introduce variable declaration without the keyword ‘var’ like in JavaScript. However, this idea caused too many problems, so it wasn’t implemented.

The example program and syntax is defined in demo9.txt (below)

```
-- demo9.txt
var x : Bool = mutable true;
var y : Bool = mutable false;
y <- true;
$y && $x
-- Returns true
```

String Values

A natural step after introducing Int and Bool types of was to introduce the String type. This implementation is essential to propagate better error messages using `RaiseV`.

`StringV` is defined similar to the `IntV` and `BoolV`. A binary operation ‘+’ joining the two strings as well as a string with an integer is also introduced.

```
data Type = ...
  | TString             -- new

data Value = ...
  | StringV String      -- new
  deriving Eq
```

The example program and syntax is defined in demo10.txt (below)

```
-- demo10.txt
var x : String = "Hello ";
var y : String = "World!";
x + y
-- Returns “Hello World!”
```

Parsing a string

In order to parse the string, I first tried to copy the parsing style of function, using:

```
-- Parser.y
App: ...
    | '"' id '"'  { Lit (StringV $2) }
```

However, this id only matched with words and not numbers or symbols or spaces. Next, I tried to create a separate token to get string using the following expression:

```
-- Token.x
tokens :- ...
  [$alpha $digit \_ \' ...]*      { \s -> TokenString s }

-- Parser.y
&token
  str              { TokenString $$ }
App: ...
    | '"' str '"'  { Lit (StringV $2) }
```

With this, everything was considered as a String so this definitely failed.

Finally, I tried pattern matching similar to Int, which worked. This pattern reads everything inside the double inverted commas “ ”, except the inverted commas.

```
-- Token.x
tokens :- ...
  \"[^\"]*\"       { \s -> TokenString (read s) }

-- Parser.y
&token
  str              { TokenString $$ }
App: ...
    | str          { Lit (StringV $1) }
```

Type Definition within a custom type

This was an added benefit of our implementation of recursive type substitution. As a result, we can define a type as another custom type. For example:

```
-- demo11.txt
type Person = {age : Int, name : String}
type Car = {owner: Person, speed: Int}
var john : Person = {age = 30, name = "John"};
var honda : Car = {owner = john, speed = 1000}; honda.owner
```

Here type Car is a record, where one of the field is of type Person. Our programming language allows such declarations, and correctly evaluates this program.

```
➜ PROJECT_3035393157 git:(master) X stack exec sinh demo11.txt
```

Type Check for Declarations (and Function return type bug)

This is a very minor addition to the program. The code given in the project bundle did not check the type of the variable against the expression. I could, for example, declare a variable of type Bool and equate it to Int.

```
var x : Bool = 5 -- This would pass the type check
```

Therefore, I modified the `tcheck` function for `Decl` expressions and enabled type checking. However, this caused problems with First Class Functions, as their return type didn't match.

Function return type bug (& how to solve it)

Looking deeper into the matter, I realized that the program in demo1.txt had wrong return type: Int -> Int. Instead it should be (Int -> Int) -> Int. Adding brackets to the type caused parsing error. I could not confirm this with the professor whether this really is a bug or misunderstanding on my part, so I haven't implemented it yet. However, if it really is a bug, it can be fixed by introducing recursive return types with brackets in the function declarations.

```
typ : ...
    | typ          '->' typ              { TFun $1 $3 }
    | '(' typ ')' '->' typ               { TFun $2 $5 }
    | typ          '->' '(' typ ')'      { TFun $1 $4 }
    | '(' typ ')' '->' '(' typ ')'       { TFun $2 $6 }
```

Workaround: A workaround this was to remove the type checker during variable declaration, as was done in the project bundle. However, I decided to type check all other types except function. I was able to achieve it by modifying the equality check of Types.

```
instance Eq Type where
  TFun _ _ == TFun _ _ = True        -- Bug
  _        == TFun _ _ = True        -- Bug
  TFun _ _ == _        = True        -- Bug
```

Colour coded output

Finally, a very minor improvement that I personally really prefer. The output is colour coded, for example the errors are shown in red, the debugging information in magenta, and the banner in blue. This makes it easier for some people to read, especially when there's plethora of information on screen, and therefore I included it.