Haskell Interim Test

Friday 10th November 2023

TWO HOURS TOTAL

The maximum mark is 20.

- Please make your swipe card visible on your desk.
- After the planning time log in using your username as **both** your username and password.
- Work in the file named `FunInt.hs` inside the `funint/src` sub-folder of your Home folder. **Do not move any files**.
- You are **not permitted** to edit the folder structure, or edit either the `hit.cabal` or `cabal.project` files without having been directed by the Examiners; any changes will be reverted.
- You may add additional tests, which will be reflected in the final script (*like in the PPTs*), but these will not be assessed. However, any changes made to the code that cause the **original given** tests to not compile will incur a compilation penalty.

It is important that you comment out any code that does not compile before you complete the test. There will be a TWO MARK PENALTY applied to any program that does not compile.

Credit will be awarded throughout for style, clarity and the appropriate use of Haskell's various language features including list comprehensions, higher-order functions etc.

1 Introduction

In this exercise you are required to write an interpreter and a simple analyser for a functional programming language, similar in style to a stripped-down version of Haskell. Rather than work with source code – characters in a text file, for example – the programs you will be manipulating will be represented by a Haskell data type. A program consists of an expression which can be one of the following:

1. A `Number`, represented as an `Int` (only integers are supported)
2. A `Boolean`
3. An identifier, represented as a `String`
4. A binary operator, which will be one of `==, >, +, *`. The operator name is represented as a `String`, e.g. `"+"` and each is designed to operate only on numbers – you can't compare two booleans, for example, as you can in Haskell.
5. A conditional, akin to Haskell's `if p then q else r`, where `p`, `q` and `r` are expressions.
6. An *anonymous* function akin to a Haskell lambda expression, e.g. `Fun ["x", "y"] e` is the representation of the equivalent Haskell expression `\x y -> e`, where `e` is an expression.
7. A function application, where the function is either a binary operator or an anonymous function
8. A "let" expression, akin to Haskell's `let v = e in e'`, where `v` is an identifier and both `e` and `e'` are expressions.

The following Haskell types can be used to represent programs in this simple language:

```
type Identifier = String

data Expr = Number Int
          | Boolean Bool
          | Id Identifier
          | Op Identifier
          | Let Identifier Expr Expr
          | If Expr Expr Expr
          | Fun [Identifier] Expr
          | App Expr [Expr]
          deriving Show

type Program = Expr
```

To illustrate this, the following shows the representation of the (non-recursive) function `minOf2`, which is equivalent to Haskell's `min` function defined on integers, e.g. `minOf2 a b = if a > b then b else a`:

```
minOf2 :: Expr
minOf2 = Fun ["a", "b"] (If (App (Op ">") [Id "a", Id "b"]) (Id "b") (Id "a"))
```

What about recursive functions? The language supports a restricted notion of recursion through `Let` expressions which allow you to name a function and then refer to it by its name. To illustrate this here's a representation of a program to compute the factorial of 6 (i.e. 720):

```
factOf6 :: Expr
factOf6 =
  Let "fact"
    (Fun ["x"] (If (App (Op "==") [Id "x", Number 0])
                   (Number 1)
                   (App (Op "*") [Id "x",
                                  App (Id "fact") [App (Op "+") [Id "x",
                                                                 Number (-1)]]])
               )
    )
    (App (Id "fact") [Number 6])
```

This is equivalent to the Haskell program/expression:

```
let fact = \x -> if x == 0 then 1 else x * fact (x - 1) in fact 6
```

Note the recursive reference to `"fact"` by virtue of the function being named by the `Let`[1]

These, and some additional examples, are provided in the skeleton that accompanies the exercise.

1.1 Program execution

Execution of a program is performed relative to an *environment*, which associates each identifier with a value (`Expr`) via a list of `(Identifier, Expr)` pairs:

```
type Environment = [(Identifier, Expr)]
```

A reference to an identifier invokes a look-up of the identifier in the environment. A function `lookUp :: (Eq a, Show a, Show b) => a -> [(a, b)] -> b` is defined in the skeleton. This attempts to look up the value associated with a key, in a table comprising a list of (key, value) pairs. If there is more than one value associated with the key the function returns the first (leftmost) one in the table. If there are no bindings the function aborts with a helpful error message, e.g.

```
*FunInt> lookUp "x" [("a",198), ("x",2), ("y",12), ("x",8)]
2
*FunInt> lookUp "a" []
*** Exception: error: failed to find "a" in []
```

1.2 Well-formedness

Various properties must hold in order for a program to be deemed "well-formed". One such property is that it should contain no *free variables*, i.e. variables for which there is no associated binding, as in the program `y - 1` in Haskell, where the `y` is free. The rules for computing the free variables in an expression are as follows:

- `Number`s, `Boolean`s and `Op`erators contain no free variables.
- An identifier contains a single free variable – the identifier itself.
- The free variables of `Let x e e'` are the union of those in `e` and `e'` *minus* the `x`. Hint: you can use `union` and `\\` from `Data.List` to implement union and minus respectively.
- The free variables of a function of the form `Fun as es` are the free variables of `es` minus the argument names listed in `as`.
- In all other cases the free variables are the union of those in all subexpressions.

[1]Note that the language does not support two or more mutually-recursive definitions in a `Let` expression. This can be fixed, but will require the interpreter to support a *circular* environment. Let's not go there!

1.3 What to do

There are two parts to the exercise. The bulk of the marks are for Part I, so you are advised to complete this before attempting Part II, which has few hints.

For Part I you may assume that all programs are well-typed, in the sense that the arguments to all functions/operators are of appropriate type, e.g. all operator arguments will be numbers and the predicate in each conditional will be a boolean.

Part I

1. Define a function `applyOp :: Identifier -> Int -> Int -> Expr` that will apply an operator, as specified by its `Identifier`, to its two integer arguments. For example:

   ```
   *FunInt> applyOp "+" 1 4
   Number 5
   *FunInt> applyOp ">" 2 2
   Boolean False
   ```

   [2 marks]

2. Define two mutually recursive functions:

   (a) `apply :: Expr -> [Expr] -> Environment -> Expr` which returns the result of applying a function (`Expr`) to a list of argument values (`[Expr]`) in a given `Environment`. A precondition is that the expression is well typed, so the function is guaranteed to be either an `Op` or a `Fun`. Furthermore, operators will always be applied to arguments of the correct type, i.e. `Numbers`. The precondition means that you do not have to check for badly-types applications, such as `apply (Op "+") (Number 7) (Boolean True)`.

   To apply a binary operator simply use `applyOp`; the environment in this case is not needed. To apply a user-defined function of the form `Fun as e` to a list of arguments `es` return the result of evaluating `e` using the given environment augmented by the bindings `(a1, e1), ... (an, en)` where `ai` and `ei` are the i^{th} elements of `as` and `es` respectively, $1 \leq i \leq n$. All function applications can be assumed to be *saturated* which means that the lengths of `as` and `es` will be equal. For example (these are defined as `app1`, `app2` and `app3` respectively in the skeleton, to save you typing):

   ```
   *FunInt> apply (Op "+") [Number 7, Number 4] emptyEnv
   Number 11
   *FunInt> apply (Op ">") [Number 8, Number 2] emptyEnv
   Boolean True
   *FunInt> apply minOf2 [Number 6, Number 0] emptyEnv
   Number 0
   *FunInt> [app1, app2, app3]
   [Number 11,Boolean True,Number 0]
   ```

   The empty environment, `emptyEnv`, is also defined in the skeleton. **[3 marks]**

   (b) `eval :: Expr -> Environment -> Expr` which evaluates a given expression using a given environment. The rules are as follows:

   - To evaluate an identifier (`Id`) use the `lookUp` function to look up the binding for the identifier in the environment.
   - To evaluate a conditional of the form `If p q r`, first evaluate `p` using `eval`, and return either the result of evaluating `q` or `r` similarly, depending on the value of `p`. All programs are assumed to be well-typed, so `p` can be assumed always to evaluate to a `Boolean`.

- To evaluate a let expression of the form `Let v e e'`, evaluate the result, `e'`, in an extended version of the environment in which `v` is associated with the result of evaluating `e` in the original environment.
- To evaluate a function application (`App`) use the `apply` function. Both the function and each argument expression should be evaluated using the current environment before calling `apply`[2].
- In all other cases the result is the given expression, as it cannot be simplified further.

For example (these examples are defined as `eval1`, ..., `eval5` in the skeleton):

```
*FunInt> env
[("x",Number 1),("b",Boolean True),("y",Number 5)]
*FunInt> eval (Number 8) env
Number 8
*FunInt> eval (Id "x") env
Number 1
*FunInt> eval (Op "+") env
Op "+"
*FunInt> eval (App (Op "+") [Id "x", Id "y"]) env
Number 6
*FunInt> eval factOf6 emptyEnv
Number 720
*FunInt> [eval1, eval2, eval3, eval4, eval5]
[Number 8,Number 1,Op "+",Number 6,Number 720]
```

A function `runProgram` is provided in the skeleton that evaluates a given program (`Expr`) in an empty environment, should you wish to use it for testing:

```
runProgram :: Program -> Expr
runProgram p = eval p []
```

For example,

```
*FunInt> runProgram factOf6
Number 720
```

[6 marks]

3. Define a function `isWellFormed :: Program -> Bool` which returns `True` if the given program (expression) contains no free variables. You should thus implement the rules listed in Section 1.2. For example:

```
*FunInt Data.List> isWellFormed factOf6
True
*FunInt Data.List> invalid1
Fun ["x"] (App (Op "+") [Id "x",Id "y"])
*FunInt Data.List> invalid2
If (App (Op "==") [Number 1,Number 0]) (Id "x") (Number 4)
```

[5 marks]

[2]For completeness, you can assume that the body of a function contains no free variables as this will break the interpreter. An example of a function that violates this (pre)condition is `let y = 5 in f x = x + y`, where the `y` is free in the body `x + y`; the `x` is "bound" by virtue of it being a named argument. It's easy to check this property statically, but in order to simplify the exercise we'll just assume that it holds throughout, without further mention.

Part II

Programs that are badly typed will fail during evaluation, typically because of a missing case in the interpreter. Rather than failing in such cases it would be preferable instead to return a special value indicating that a type error was encountered during evaluation. One way to do this is to use Haskell's `Maybe` type. Hence:

1. Define modified forms of your `apply` and `eval` functions that in each case will return a `Maybe Expr` rather than an `Expr`. The result should be `Nothing` if a type error was encountered and `Just v` if the evaluation succeeded with result value `v`. You should catch three specific cases:

   (a) An operator being given an argument that is not a `Number`.

   (b) An application being given something other than an operator (`Op`) or function (`Fun`).

   (c) A conditional in which the predicate is not a `Boolean`.

   To implement these functions, copy and paste your original versions (do not modify the originals!) and adapt them accordingly. The type signatures of the two functions you need are given in the skeleton:

   ```
   maybeApply :: Expr -> [Expr] -> Environment -> Maybe Expr
   maybeEval :: Expr -> Environment -> Maybe Expr
   ```

   For example:

   ```
   *FunInt Data.List> maybeEval factOf6 []
   Just (Number 720)
   *FunInt Data.List> typeError1
   App (Op ">") [Number 1,Op "+"]
   *FunInt Data.List> maybeEval typeError1 []
   Nothing
   *FunInt Data.List> typeError2
   App (Fun ["x","y"] (App (Op "*") [Id "x",Boolean True])) [Number 5,Number 6]
   *FunInt Data.List> maybeEval typeError2 []
   Nothing
   ```

   [4 marks]

2. **Note: This carries no marks, but feel free to implement it for personal satisfaction and eternal glory!** Define a function `maybeRunProgram :: Expr -> IO()` that will initially perform a static check of the program to determine whether it is well formed; use `isWellFormed` to do this. If not it should display a suitable error message. If it is well formed then it should display the result of evaluating the program in an empty environment (`[]`). If the evaluation is successful (`Just ...`) then you can use the `show` function to display the result; otherwise you should print a type-error message. The required error messages in each case are predefined in the skeleton, so you should use these rather than inventing your own. For example:

   ```
   *FunInt Data.List> maybeRunProgram factOf6
   Number 720
   *FunInt Data.List> maybeRunProgram invalid1
   Program not well-formed
   *FunInt Data.List> maybeRunProgram invalid2
   Program not well-formed
   *FunInt> maybeRunProgram typeError1
   Type error
   *FunInt Data.List> maybeRunProgram typeError2
   Type error
   ```

   [0 marks]