Advanced Programming

Assignment 1: Subs Interpreter

Per Steffen Czolbe, Konrad Gnoinski

Handed in: September 21, 2017

Contents

Description of the Assignment

This assignment is about implementing an interpreter for a conservative subset of Mozilla's JavaScript implementation, including list comprehensions, which we will call SubScript. The interpreter reads an abstract syntax tree and returns the interpreted values.

Solution

This section gives a brief overview about the structure of the solution and how to execute the code.

Implementation

This solution of the assignment is implemented in Haskell.

Source-code Organisation

The code is organised in multiple files:

- `src/Primitives.hs`: a file containing the primitive functions of the SubScript language.
- `src/Subs.hs`: the file containing the main function.

- `src/SubsAst.hs`: the file containing the abstract syntax tree, represented by the data type `Expr`
- `src/SubsIntepreter.hs`: the heart of the application: the interpreter.
- `src/Valu.hs`: the file containing the data type `Value`.
- `src/Test/MonadLawsTest.hs`: a file containing tests to assure adherence to the monadic laws
- `src/Test/SimpleTests.hs`: a file containing tests of the interpreter functionality.

Test Cases

The `src/Test` directory contains multiple tests. The test framework used for running multiple tests is `HSpec`. For propery based testing, `QuickCheck` is used.

How to Execute the Program

The interpreter can be compiled using the `src/Subs.hs` file as the primary component. The compiled program can be executed by passing a file containing the abstract syntax tree as an argument:

```
$ gci Subs.hs
$ ./Subs <filename>
```

Alternatively, the `SubsInterpreter` module can be loaded into the `ghci`. This way the function `runExpr` can be called with an expression. The interpreted value is returned.

The tests can be run similar to the mentioned ways, just using the files or modules from the `src/Test` directory instead.

Implementation

A description of the implementation.

Assumptions

The implementation of the interpreter is based on the assumption that its behaviour is similar to firefox's javascript intepreter. However, the presented interpreter does not attempt to mirror the functionality of the javascript interpreter exactly. Instead of focusing on the details of edge-cases in javascript, this solution has been written with the goal of learning haskell and not javascript. Conseuently, it focuses on getting the major features of js right, while omitting the weird details.

Choice of Algorithms and Data Structures

This solution is based on the handed out code skeleton, which dictates the data structure and general organization of the code.

Support of Array Comprehensions

This solution supports the full functionality of list comprehensions, as specified by the assignment.

The Monad SubsM

This is the implementation of the SubsM monad used in this solution:

```
newtype SubsM a = SubsM {runSubsM :: Context -> Either Error (a, Env)}

instance Monad SubsM where
return x = SubsM ( \(env, _) -> Right (x, env) )
m >>= f  = SubsM ( \c -> case runSubsM m c of
Left e          -> Left e
Right (a, env') -> let (_, penv) = c
c' = (env', penv)
in runSubsM (f a) c'
)
fail s = SubsM ( \_ -> Left s )
```

The `SubsM` monad is used by the interpreter for two main features:

- Providing a simple and easy to use way to handle errors during the interpretation.
- Storing the state of the interpreter. A state is necessary to store and retrieve the value of variables.

To account for both these requirements, the monad `SubsM` has both a state, indicated by its signature as a function, and can return either an error (`Either Left`) or a correct result (`Right`).

The `return` operation follows the signature of the `return` operator. It takes a value of type a, indicated by the parameter x, and returns a monad `SubsM` with the value x wrapped into it. The Right side is returned, indicating that this is not an error case. The environment is propagated without being changed.

The >>= operator binds a function of type `f :: a -> m b` to the monad. It "runs" the given monad `m` with the context, and checks for the type of the result. If the result is a `Left`, an error happened. In this case no further action is performed, the error is propagated. If the monad does not return in a state of error, the value `a` and the updated variable environment `env'` is retrieved. A new context `c'` is created and the monad is executed again with the new context `c'` and the result of `f a`, which applies the function `f` to `a`.

The `fail` operator is similar to the return operator, but it is used in the event of an error. Instead of passing a value to wrap into the monad, a string describing the error is given. As the monad can handle errors thanks to the `Either Error (a, Env)` signature, the implementation of `fail` just wraps a `Left Error` into the monad.

Satisfaction of the Monad Laws

Any monad should adhere to the three monad laws of left identity, right identity and associativity.
To test the accordance of the `SubsM` monad to these laws, a test case for each law is implemented in the `src/Test/MonadLawsTest.hs` file. While these tests are not a formal proof, they do provide some evidence of compliance with the laws.

Assessment

This section contains the assessment of the presented solution.

Correctness of Solution

Based on the extensive tests, including property based tests performed with QuickCheck, we assume the solution is likely to be correct.
Additionally, the solutions passed the Online TA test.

Summary of Code Quality

Based on the tests, we do believe that the submited program functions like specified in the assignment. However, the quality of the code and test cases could be significantly improved, for example by using QuickCheck more extensively to generate random expressions, or by using more functions from the haskell standard library instead of implementing our own functions. Due to time constraint of the assignment, we were not able to improve the code quality further.

$$\begin{aligned} jhsd &= sf \\ &\mid kfssdsadasdas \end{aligned}$$