Advanced Programming 2024

Assignment Two

Monadic Programming

Due: 2024-9-22

Synopsis: Implementing a monadic interpreter and type checker for a small language of arithmetic expressions.

1 Introduction

In this assignment you will be adding various new features to the language implemented during the exercise classes. You will also be implementing a very simple type checker. If you wish, you can base the assignment on your own solutions to the exercises, rather than ours, but only do so if you are quite sure that they are correct. Specifically, you will be implementing the following:

- Extending the interpreter with a notion of state.
- Support for printing values.
- Support for key-value stores.
- Implementing a type checker.

For all tasks you are expected to add appropriate tests to the file `Eval_Tests.hs`. Do not rename any of the definitions already present in the handout. Do not change the types of any functions.

2 Task: Printing

In this task you will extend the `EvalM` monad with a simple notion of state. Specifically, we will add the ability to print values in an APL program.

Change the type of `runEval` to the following:

```
runEval :: EvalM a -> ([String], Either Error a)
```

The `[String]` that is returned will be the list of strings that were printed during execution of the program. We will not actually perform true IO during execution, so "printing" just means that the string ends up in this list. The list must be ordered such that the first string to be printed comes first in the list. You will also need to change the type of the `eval'` function in `APL.Eval_Tests` accordingly.

Add the following constructor to `Exp`:

```
data Exp
  = ...
  | Print String Exp
```

`Print s e` evaluates `e` to a value v, then prints the string "`s:` $p(v)$", where $p(v)$ is the textual representation of v (see below). The expression then returns v.

- If v is an integer, then its textual representation is the integer in decimal notation. (Use `show`.)
- If v is a Boolean, then its textual representation is `True` and `False`. (Use `show`.)
- If v is a function, then its textual representation is `#<fun>`.

2.1 Examples

```
> runEval $ eval $ Print "foo" $ CstInt 2
(["foo: 2"],Right (ValInt 2))
> runEval $ eval $ Let "x" (Print "foo" $ CstInt 2)
                           (Print "bar" $ CstInt 3)
(["foo: 2","bar: 3"],Right (ValInt 3))
> runEval $ eval $ Let "x" (Print "foo" $ CstInt 2)
                           (Var "bar")
```

```
(["foo: 2"],Left "Unknown variable: bar")
> runEval $ eval $ Print "foo" (Lambda "x" (Var "x"))
(["foo: #<fun>"],Right (ValFun [] "x" (Var "x")))
```

2.2 Suggested implementation

Define a type `State` that can store a list of the strings printed. Modify `EvalM` to maintain such a `State`. **Hint:** Consider how the course notes discussed a combined reader-and-state monad. `EvalM` is a little more complicated as it also has to handle the notion of failure.

Define the following helper function:

```
evalPrint :: String -> EvalM ()
```

The function `evalPrint` adds a string to the list of printed strings. Use this to implement the `eval` case for `Print`. You will also need to modify `runEval`.

3 Task: Key-value store

In this task we will add a slightly more useful notion of effects, by extending APL to support an implicitly available key-value store, where a key (which can be any value) can be associated with any value, and later referenced again.

Add the following constructors to `Exp`:

```
data Exp
  = ...
  | KvPut Exp Exp
  | KvGet Exp
```

- `KvPut k_exp v_exp` evaluates `k_exp` and `v_exp` to values k and v respectively. It then records the association $k \mapsto v$ in the store and returns v. If there is already an association for the value k, it is replaced by the new association.
- `KvGet k_exp` evaluates `k_exp` to a value k, then retrieves the value previously associated with k from the store. If there is no value associated with k, `KvGet` fails.

3.1 Examples

```
> runEval $ eval $ Let "x" (KvPut (CstInt 0) (CstBool True))
                           (KvGet (CstInt 0))
([],Right (ValBool True))
> runEval $ eval $ Let "x" (KvPut (CstInt 0) (CstBool True))
                           (KvGet (CstInt 1))
([],Left "Invalid key: ValInt 1")
> runEval $ eval $ Let "x" (KvPut (CstInt 0) (CstBool True))
                     (Let "y" (KvPut (CstInt 0) (CstBool False))
                      (KvGet (CstInt 0)))
([],Right (ValBool False))
```

3.2 Suggested implementation

Extend the `State` type to also contain a key-value mapping.

Define the following helper functions:

```
evalKvGet :: Val -> EvalM Val
evalKvPut :: Val -> Val -> EvalM ()
```

Use this to implement the `eval` cases. You will also need to modify the definition `runEval`, but *not* its type.

4 Task: Type checking

In this task you will develop a type checker for APL. Since APL is dynamically typed, all we can check for is that the program does not contain references to variables that are not in scope. The purpose of this task is for you to demonstrate that you can construct a monad definition from scratch. Therefore, the code handout is somewhat spartan.

The module `APL.Check` exports a function

```
checkExp :: Exp -> Maybe Error
```

where `Error` is a synonym for `String`. This function must return `Just` (along with an error message) if the provided expression at any point references a variable that is not bound by an enclosing `Let` or `Lambda`. You must implement this function by making use of a monad, similar to the evaluator. In particular, you must take a monadic approach to

error handling, and a monadic approach to tracking which variables are in scope.

Also add appropriate tests to `APL.Check_Tests`.

4.1 Examples

Note that the specific error message is up to you.

```
> checkExp $ CstInt 2
Nothing
> checkExp $ Var "x"
Just "Variable not in scope: x"
> checkExp $ Lambda "x" (Var "x")
Nothing
```

5 Code handout

The code handout consists of the following nontrivial files.

- `a2.cabal`: Cabal build file. **Do not modify this file.**
- `runtests.hs`: Test execution program. **Do not modify this file.**
- `src/APL/AST.hs`: The APL AST definition. **Do not modify this file.**
- `src/APL/Check.hs`: Where you will implement the type checker.
- `src/APL/Check_Tests.hs`: Tests for the type checker.
- `src/APL/Eval.hs`: The APL evaluation function.
- `src/APL/Eval_Tests.hs`: The evaluation tests. Some tests are already included. You are expected to add more.

6 Your Report

You are expected to comment on the *interesting* details of your implementation. You are *not* expected to give a line-by-line walkthrough of your code. Most importantly, you are expected to reflect on the *quality* of your code:

- Do you think it is functionally correct? Why or why not?
- Is there some improvement you'd have liked to make, but didn't have the time?

It is more important to be aware of the strengths or shortcomings of your solution, than it is to have a complete solution.

6.1 The structure of your report

Your report must be structured exactly as follows:

Introduction: Briefly mention very general concerns, your own estimation of the quality of your solution, and possibly how to run your tests.

A section for each task: Mention whether your solution is functional, which cases it fails for, and what you think might be wrong.

A section answering the following numbered questions:

1. What is the asymptotic complexity of your implementation of `Print`? Can this be improved?
2. In your implementation, if a program keeps updating the same key with `KvPut`, does the memory usage of the program remain constant or does it increase?
3. In `TryCatch e1 e2`, if `e1` fails after performing some effects, are those effects visible in `e2`, and which of your tests demonstrate this? If you wanted different behaviour (either making the effects visible or invisible), what would you need to change in your implementation? Show exactly what the new code would look like.

All else being equal, **a short report is a good report**.

7 Deliverables for This Assignment

You must submit the following items:

- A single PDF file, A4 size, no more than 5 pages, describing each item from report section above.
- A single zip/tar.gz file with all code relevant to the implementation, including at least all the files from the handout. For this assignment it is not necessary to add additional files.

Remember to follow the general assignment rules listed on the course homepage.

8 Assessment

You will get written qualitative feedback, and points from zero to four. There are no resubmissions, so please hand in what you managed to develop, even if you have not solved the assignment completely.