The University of Hong Kong

COMP3259: Principles of Programming Languages

Tutorial 8

Tutor
Wan Qianyong `qywan@cs.hku.hk`

Instructor
Bruno Oliveira `bruno@cs.hku.hk`

15 April 2025

Table of Contents

1 Introduction

This tutorial aims at providing students with experience in writing interpreters and type checkers with first-class functions and *recursion*. In addition, we will introduce writing interpreters in a *monadic* way.

2 A Language with Recursive Functions

We now continue to improve SNH by supporting recursive definitions. For example, you may want to write a Fibonacci function as follows:

```
var fib = function(x: Int) {
  if (x == 0) 0;
  else if (x == 1) 1;
  else fib(x-1) + fib(x-2)
};
fib(10)
```

Recall that the code for evaluating non-recursive variable declarations is:

```
evaluate (Decl x e body) env =
  let newEnv = (x, evaluate e env) : env
  in evaluate body newEnv
```

The declared variable `x` itself is not in the environment so it cannot be used in `e`. We can add support for recursion in SNH with the help of recursion in Haskell. By using a *circular program*, the change to variable declarations is minimal:

```
evaluate (Decl x e body) env =
  -- when evaluating `e`, use `newEnv` instead of `env`
  let newEnv = (x, evaluate e newEnv) : env
  in evaluate body newEnv
```

Similarly, when type checking a variable declaration, we don't know the type of `x` that may be recursively used in `e`:

```
tcheck (Decl x e body) env = do
  t <- tcheck e env
  tcheck body ((x, t) : env)
```

Unfortunately, the same trick doesn't apply here because it leads to an endless loop. To remedy this, one simple way is to give extra information to the type checker. Therefore, we slightly alter the syntax of variable declarations, that is, we require every variable to be explicitly typed. For example, `fib` now becomes:

```
var fib: Int -> Int = function(x: Int) {
  if (x == 0) 0;
  else if (x == 1) 1;
  else fib(x-1) + fib(x-2)
};
fib(10)
```

Here we specify the type of `fib` as `Int -> Int`. It may seem tedious to write types for non-recursive declarations, but it makes type checking recursive functions much easier.

How are variable declarations type checked then? The answer is that we assume the variable have the specified type and then check whether the definition is consistent with this assumption. In the case of `fib`, we assume it have type `Int -> Int` in the typing environment, and then call `tcheck e`. Since the result is indeed `Int -> Int`, it type checks.

Question 1. Fix type checking `Decl` in the file `TypeCheck.hs`.

Now you can recompile SNH and test if `demo.txt` can type check.

Question 2. In Tutorial 7, we built a stack machine called K. Does the K machine support recursive definitions? If so, show the evaluation derivation of `fib(2)`; if not, modify the implementation to support recursion.

3 Monads

Let's take a closer look at monads and understand why they can be useful. In functional programming (Haskell in particular), monads are used as a *design pattern*. This means that monads offer a reusable solution to a commonly occurring problem. The abstraction of monads reduces boilerplate code in Haskell programming. We are going to see several monads and how they can be used to model errors.

3.1 A Navigation Language

Suppose that we have a very small language which simulates movements in a plane. We may define the current location as a pair of coordinates:

```
type Loc = (Int, Int) -- (x, y)
```

We want to define three functions (`up`, `down`, and `left`), which represent safe movements within the plane. By "safe" we mean that only non-negative coordinates are allowed after movements. It should be clear that, in functional programming, we would like to have total functions. Thus, we want potential errors to be reflected in the type system, rather than take the risk of blowing up the program at run time.

Let's first consider the following (possibly unsafe) definitions:

```
up   :: Loc -> Loc  -- increases y by 1
down :: Loc -> Loc  -- decreases y by 1
left :: Loc -> Loc  -- decreases x by 1
```

Judged by their names and type signatures, the defintion of `up` should be correct: no matter what valid location is passed, it will always return another valid location. However, `down` and `left` are error-prone: for some valid locations taken as input, they may produce invalid locations. For instance, `down (3, 0)` will produce the invalid location `(3, -1)`. Therefore, we want `down` and `left` to return errors in those cases. Here are three possible options for `down`:

```
down1 :: Loc -> Maybe Loc
down2 :: Loc -> List Loc
down3 :: Loc -> Either String Loc
```

All the three implementations are trying to convey the same information. In case the movement is invalid, `down1` returns `Nothing`; `down2` returns an empty list; `down3` returns `Left err` (where `err` is the error message). Otherwise, `down1` returns `Just v`; `down2` returns `[v]` (a singleton list); `down3` returns `Right v`.

Then we can write a sample program that composes `left`, `down`, and an extra `update`. Note that we have to constantly check errors:

```
update :: (Int -> Int) -> (Int -> Int) -> Loc -> Loc
update f g (x, y) | f x >= 0 && g y >= 0 = (f x, g y)
                  | otherwise            = (x, y)

ex1 :: Loc -> Maybe Loc
ex1 loc = case left1 loc of
            Nothing   -> Nothing
            Just loc' -> case down1 loc' of
                           Nothing    -> Nothing
                           Just loc'' -> Just $ update (+1) (+1) loc''
ex2 :: Loc -> List Loc
ex2 loc = case left2 loc of
            Nil         -> Nil
            Cons loc' _ -> case down2 loc' of
                             Nil          -> Nil
                             Cons loc'' _ -> singleton $ update (+1) (+1) loc''

ex3 :: Loc -> Either String Loc
ex3 loc = case left3 loc of
            Left err   -> Left err
            Right loc' -> case down3 loc' of
                            Left err'   -> Left err'
                            Right loc'' -> Right $ update (+1) (+1) loc''
```

As you can witness, this kind of code is extremely tedious to write. Furthermore, no matter which option we choose, the code looks remarkably similar. This motivates the need for a new design pattern. The most common solution amongst functional programmers is probably monads.

3.2 Introducing Monads

Monads are data structures that support two primitive operators: `return` (`pure`) and `bind`. Haskell models them as a type class, which can be thought as an interface that requires every member of it to implement the two operators:

```haskell
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The latter (`bind` is denoted as `>>=` in Haskell) serves the purpose of sequencing; while the former injects a value into a monad. The benefit of using a monad is that constant checking for corner cases is no longer visible to the programmer.

We can declare all three types (i.e. `Maybe`, `List`, and `Either String`) as monads by define the two primitive operators. You should have tried to implement `bind` for `Either` in Assignment 1. It is important for you to understand what is happening under the hood for the other two cases. For example, the `Maybe` monad is implemented as:

```haskell
instance Monad Maybe where
  return a    = Just a
  ma >>= a2mb = case ma of Nothing -> Nothing
                           Just a  -> a2mb a
```

We define two functions: `return a` and `ma >>= a2mb`. The first function, given a value of any type, injects it into the monad by applying the `Just` constructor to it. The second function, given a computation which is already inside the monad (i.e. a term of type `Maybe a`) and a continuation in case the evaluation is successful, performs pattern matching and applies the continuation only when the first computation succeeds. Therefore, error checking is hidden behind the `>>=` operator. In fact, our previous example `ex1` was no more than a sequence of these checks, followed by an injection of an application of the `update` function. It means that we can write less verbose code to express equivalent logic:

```haskell
ex1monad :: Loc -> Maybe Loc
ex1monad loc = left1 loc  >>= \loc'  ->
               down1 loc' >>= \loc'' ->
               return $ update (+1) (+1) loc''
```

This pattern is so common in Haskell that there is a special syntax for it. The same program can be written with the *`do`-notation*:

```haskell
ex1do :: Loc -> Maybe Loc
ex1do loc = do loc'  <- left1 loc
               loc'' <- down1 loc'
               return $ update (+1) (+1) loc''
```

Here, `do` is a keyword indicating that the inner block is running inside a monad. Each occurrence of `<-` is translated to a use of `>>=`. Finally, `return` remains unchanged. This notation is reminiscent of some form of imperative programming.

Question 3. Complete the definitions of `ex2do` and `ex3do` using the `do`-notation.

3.3 Parametrization of Monads

Though we have already written neat monadic code, we can still add a final degree of abstraction. Since we went through the trouble of defining safe operators for all the three monads, we can now easily offer programmers the opportunity to choose their preferred ones!

The idea is that we define a new function that runs our language inside any monad, as long as it provides the basic constructs. Thus, the example can be parametrized by the type of monads:

```
exM :: Monad m => (Loc -> m Loc) -> (Loc -> m Loc) -> (Loc -> m Loc)
               -> Loc -> m Loc
exM up down left loc = undefined
```

The first three parameters are the base constructs: `up`, `down`, and `left`. The forth parameter is the input location. Notice that even though we are free to choose a `Monad` and its implementation for the basic operators, we can not mix different monads together. We can choose to call `exM` with `up1`, `down1`, and `left1`, but not with nonsensical combinations like `up1`, `down2`, and `left3`. For example, the following program instantiates `exM` with the `Either String` monad:

```
exEither :: Loc -> Either String Loc
exEither = exM up3 down3 left3
```

Question 4. Complete the definition of `exM` using the `do`-notation, whose logic is equivalent to `ex1do`, `ex2do`, or `ex3do`.

The evaluators we wrote in previous lectures and tutorials can also be simplified by monads and `do`-notation. Try to find some of them and refactor by yourself!