INFOAFP Assignments

AFP Assignment 3

Deadline: Friday, Dec 14, 2012, 24:00

General remarks

- Mail your solution as a single file `doaitse@swierstra.net`, with in the subject "Assign-3: `name1` and `name2`" and as filename "afp2012-name1-name2.zip". Also give the unzipped directory this name.

- Team size: preferably 2, but 1 is possible.

- Programming style and quality of Haddock comments influences the grade.

- Put Haddock information in your code, so as not to make it a puzzel to mark it.

- Gathering information on the internet is okay, but copying entire solutions from the internet (or elsewhere) is not allowed.

The goal of this task is to create your own *monad*. In fact, you have to extend the well-known state monad (see `Control.Monad.State` in the hierarchical libraries), and add some extra features to it.

The first step is to introduce a type constructor for the new monad:

```
data StateMonadPlus s a = ...
```

The type variables `s` and `a` have the standard meaning: `s` is the type of the state to carry and `a` is the type of the return value. Make the new monad an instance of the `MonadState` type class. Hence, you have to include:

```
import Control.Monad.State
```

We now discuss the three additional features that your monad has to support.

Feature 1: Diagnostics

We want to gather information about the number of calls to all primitive functions that work on a `StateMonadPlus`. For this, you have to write a function `diagnostics` with the following type:

```
diagnostics :: StateMonadPlus s String
```

This function should count the number of binds ($\gg=$) and `returns` (and other *primitive* functions) that have been encountered, including the call to `diagnostics` at hand. Secondly, provide a function

```
annotate :: String -> StateMonadPlus s a -> StateMonadPlus s a
```

which allows a user to annotate a computation with a given label. The functions for Features 2 and 3, as well as `get` and `put`, should also be part of the diagnosis.

As an example, consider the input

```
do return 3 >> return 4
   return 5
   diagnostics
```

which should return the string

```
"[bind=3, diagnostics=1, return=3]"
```

Note that $\gg$ is implemented in terms of $\gg=$, and thus also counts as a bind.

Here is another example:

```
do annotate "A" (return 3 >> return 4)
   return 5
   diagnostics
```

This returns the string

```
"[A=1, bind=3, diagnostics=1, return=3]"
```

Feature 2: Failure

A second feature of your monad is that it can fail during a computation. The `Monad` type class offers the following member function:

```
fail :: (Monad m) => String -> m a
```

Calling this function should not result in an exception. To facilitate this, the function `runStateMonadPlus` (which will be explained later) returns an `Either` value: `Left` indicates that the computation failed, `Right` indicates success. You may have to change the `StateMonadPlus` data type to cope with this.

Feature 3: History of states

The last feature is to save the current state, and to restore a previous state as the current state. Include the following type class definition in your code, and make `StateMonadPlus` an instance of this type class.

```
class MonadState s m ⇒ StoreState s m | m -> s where
  saveState :: m ()
  loadState :: m ()
```

The part `m -> s` in the class declaration is a *functional dependency*, indicating that the type `s` is uniquely determined by the choice of `m`. Functional dependencies limit the instance declarations that are valid, and in turn allow the type checker to make use of the functional dependency while inferring type. Functional dependencies are a Haskell language extension. To enable it, you must put a language pragma at the top of your module:

```
{-# LANGUAGE MultiParamTypeClasses #-}
```

Saving and loading states should be implemented as a stack: saving a state means pushing the current state on the stack, while loading a state means popping a state from the stack to replace the current one. If `loadState` is called with an empty stack, then the computation in the monad should fail (as explained for Feature 2).

Here is an example expression:

```
do i1 <- get; saveState
   modify (*2)
   i2 <- get; saveState
   modify (*2)
   i3 <- get; loadState
   i4 <- get; loadState
   i5 <- get
   return (i1, i2, i3, i4, i5)
```

This program should return the value (1, 2, 4, 2, 1) if we start with the state consisting of the integer 1.

Running the monad

You have to write a function

```
runStateMonadPlus :: StateMonadPlus s a -> s -> Either String (a, s)
```

for running the monad. Given a computation in the `StateMonadPlus` and an initial state, `runStateMonadPlus` returns either an error message if the computation failed, or the result of the computation and the final state.

To turn your module into a proper library, you should also think about which functions should be exposed to outside this module, and which functions should be hidden (and only be visible inside the current module). You might want to consider re-exporting all functionality offered by the `Control.Monad.State` module.

Try also to define a number of unit tests or even QuickCheck properties.

Bonus questions

1. Do the monad laws hold for `StateMonadPlus`? Explain your answer.

2. What are the advantages of hiding (constructor) functions? How important is this for each of the three additional features supported by `StateMonadPlus`?

3. What are the modifications required to make a monad transformer for `StateMonadPlus`?

4. Suppose that we want to write a function

```
diagnosticsFuture :: StateMonadPlus s String
```

which provides information about the computations in `StateMonadPlus` that are still to come. Explain how this would affect your code. If you feel that such a facility cannot be implemented, then you should give some arguments for your opinion. If you believe it can be done, then try to do so.