

10. Übung zur Vorlesung Programmierung und Modellierung

Hinweise: 1) Entgegen früherer Ankündigungen findet am **Montag den 2.7.18** findet die Vorlesung regulär statt. 2) Die ProMo Tutoren und Korrektoren haben sich freundlicherweise bereit erklärt, am **Montag, 9.7.18, 12 Uhr, B101** eine freiwillige Fragestunde abzuhalten. 3) Lösen Sie zuerst **A9-4**, falls Sie dies noch nicht getan haben!

A10-1 DO-Notation Versuchen Sie, diese Aufgabe mit Papier und Bleistift zu lösen. Verwenden Sie GHC oder GHCI erst, wenn Sie nicht mehr weiter wissen. Was gibt das folgende Programm am Bildschirm aus? Wie oft wartet das Programm auf eine Benutzereingabe?

Hinweis: Sie dürfen sich selbst ausdenken, was der Benutzer bei jeder Eingabeaufforderung eingibt – alle Eingaben sollten jedoch verschieden sein. Die Aktion `hSetBuffering` (siehe Folie 9.32) können Sie hier ignorieren; diese sorgt nur dafür, dass das Programm auf allen Betriebssystemen gleich funktioniert.

```
import System.IO
main = do hSetBuffering stdout NoBuffering      -- 0 (Ignorieren)
         putStr "A: "                          -- 1
         a2 <- getLine                          -- 2
         b1 <- putStr "B: "                    -- 3
         let b2 = getLine                      -- 4
         let c1 = putStr "C: "                -- 5
         c2 <- getLine                         -- 6
         putStr "D: "                          -- 7
         b2 <- b2                              -- 8
         return $ show "Ergebnis: "         -- 9
         print $ "A="++a2++" B="++b2++" C="++c2 -- 10
```

A10-2 Fehler-Monade Machen Sie den folgenden Datentyp `Entweder` zur Monade:

```
import Control.Applicative
import Control.Monad

data Entweder a b = Eines b | Anderes a deriving (Show, Eq)
```

Die Grundidee dieser Monade ist wie bei `Maybe`: eine erfolgreiche Berechnung liefert einen mit `Eines` verpackten Wert, während ein Fehler durch die Rückgabe von `Anderes` signalisiert wird. Während die `Maybe`-Monade bei einem Fehler nur `Nothing` zurückliefert, könnte hier `Anderes` noch eine Fehlerbeschreibung zusätzlich liefern.

Beispiele:

```
> (*) <$> (Eines 3) <*> (Eines 4)
Eines 12
> let foo x y = if y>0 then Eines $ x `div` y else Anderes "Div-by-Zero"
> foldM foo 100 [2,5,3]
Eines 3
> foldM foo 100 [2,5,0,3]
Anderes "Div-by-Zero"
```

- Welchen Kind hat der Typkonstruktor **Entweder**? Welchen Kind benötigt die Instanzdeklaration für die Monade?
- Berücksichtigen Sie die Monaden-Gesetze!
- Der Wert **Anderes "foo"** des Typs **Entweder String Int** kann nicht einfach als Wert des Typs **Entweder String Double** aufgefasst werden! Hier muss umverpackt werden, d.h. den Konstruktor **Anderes** erst entfernen, danach wieder erneut davor setzen. Je nach Typ wird ja auch eine andere Menge an Speicherplatz reserviert. Fehlermeldungen wie **Couldn't match type 'a1' with 'b'...** oder **Could not deduce (b ~ a1)...** weisen auf dieses Problem hin.

A10-3 Hello Again

Ändern Sie Ihre Lösung

zu Aufgabe A9-4 wie folgt ab:

Beispiel:

- a) Falls beide Eingaben leer waren, soll als Antwort nur der String **"Spielverderber!"** ausgegeben werden, und danach soll das Programm wieder automatisch von vorne beginnen.

```
> ./helloTier3
Hi! Gib bitte zuerst Dein Lieblingstier und dann
in die nächste Zeile Deine Lieblingseigenschaft ein:
Spielverderber!
Hi! Gib bitte zuerst Dein Lieblingstier und dann
in die nächste Zeile Deine Lieblingseigenschaft ein:
```

- b) Falls nur die Eingabe für das Tier leer war, so beginnt das Programm ebenfalls von vorne, aber merkt sich heimlich die eingegebene Lieblingseigenschaft. Wenn danach mal Tier und Eigenschaft komplett eingegeben werden, wird die komplette Liste aller zuvor eingegeben Eigenschaften ausgegeben.

```
tolle
Tier eingeben!
Hi! Gib bitte zuerst Dein Lieblingstier und dann
in die nächste Zeile Deine Lieblingseigenschaft ein:
schnelle
Tier eingeben!
Hi! Gib bitte zuerst Dein Lieblingstier und dann
in die nächste Zeile Deine Lieblingseigenschaft ein:
Kröte
grüne
Psst, willst Du grüne schnelle tolle Kröte kaufen?
```

Hinweis: Für die erste Teilaufgabe könnte Ihnen Folie 9.22 die notwendige Inspiration liefern. Für die zweite Teilaufgabe muss man vielleicht etwas nachdenken. Wir verraten nur so viel: die Lösung benötigt keineswegs irgendwelche monadischen Tricks; es reicht ein gewöhnlicher funktionaler Akkumulator.

H10-1 Fizz buzz (2 Punkte; Datei H10-1.hs als Lösung abgeben)

Im Kinderspiel “Fizz buzz” sitzen alle Teilnehmer in einem Kreis; ein Spieler beginnt und sagt “1”, der nächste Spieler sagt dann schnell die nächsthöhere Zahl. Falls die Zahl jedoch durch 3 teilbar ist, so muss der Spieler “fizz” sagen. Falls die Zahl durch 5 teilbar ist, so muss der Spieler “buzz” sagen. Ist die Zahl sowohl durch 3 als auch durch 5 teilbar, so muss “fizz buzz” gesagt werden. Wer einen Fehler macht, scheidet aus!

Schreiben Sie fix ein Haskell Programm, welches dieses Spiel für die Zahlen 1 bis 111 ausführt. Dabei wird in jede Antwort in einer eignen Zeile wiedergegeben:

```
1
2
fizz
4
buzz
fizz
7
```

Versuchen Sie eine Version dieses Programmes zu erstellen, welche möglichst kurz und ohne direkte rekursive Aufrufe auskommt! Verwenden Sie also die in der Vorlesung behandelten Funktionen aus Modul `Control.Monad`

H10-2 Aktionskette (2 Punkte; Datei H10-2.hs als Lösung abgeben)

Vervollständigen Sie in der beiliegenden Dateivorlage die Funktionen `chainAction1`, `chainAction2` und `chainAction3`, welche alle drei den Typ `Monad m => a -> [(a -> m a)] -> m a` haben und auch das gleiche tun sollen, so dass folgendes Beispiel in GHCI wie gezeigt abläuft:

```
> chainAction1 1 test1
1 -> 3
3 -> 4
4 -> 4
4 -> 9
9 -> 18
18
```

- Implementieren Sie `chainAction1` nur unter Verwendung von Rekursion und der DO-Notation, aber ohne Verwendung von Funktionen der Standardbibliothek! Lediglich `return` und `fail` sind erlaubt!
- Implementieren Sie `chainAction2` wie in der vorangegangenen Teilaufgabe, aber jetzt ohne Verwendung der DO-Notation. Sie dürfen stattdessen alle Funktionen der Klasse `Monad` einsetzen, also `(>>)`, `(>>=)`, `return` und `fail`.
- Implementieren Sie `chainAction3` noch ein drittes mal, dieses Mal jedoch mit umgekehrter Bedingung im Vergleich zu ersten Teilaufgabe: Sie dürfen weder direkte Rekursion, noch DO-Notation und auch keine Funktionen der Klasse `Monad` verwenden. Stattdessen dürfen Sie alle anderen Funktionen aus den Modulen `Prelude` und `Control.Monad` einsetzen!

H10-3 Zustandsmonade (2 Punkte; Datei H10-3.hs als Lösung abgeben)

In der Vorlesung am 20.06.2018 wurde eine Zustandsmonade “zu Fuss” implementiert. In dieser Aufgabe möchten wir nun lernen, wie wir stattdessen die fertige Zustandsmonade aus Modul `Control.Monad.Trans.State` der Standardbibliothek verwenden.

Dieser Aufgabe sollte eine Vorlage beiliegen, in der zwei Stellen mit `-- TODO: Ihre Aufgabe !!!` markiert sind. Wenn Sie diese Stellen korrekt bearbeitet haben, sollte das Programm wie folgt ablaufen:

```
> :load H10-3.hs
[1 of 1] Compiling Main           ( H10-3.hs, interpreted )
Ok, one module loaded.
> main
Zustand Welt bei Start ist:
Welt {zeit = 0, wetter = "Regen"}
Zustand Welt bei Ende ist:
Welt {zeit = 7, wetter = "Sturm"}
Ergebnis der Aktion ist:
[("Regen",4),("Sonne",6)]
```

Sie müssen dazu implementieren:

- a) `tick :: State Welt ()`
Eine monadische Aktion, welche die Zeit der Welt um eins erhöht und kein Ergebnis liefert (in der Vorlesung lieferte Tick die aktuelle Zeit als Ergebnis, kein Ergebnis ist also einfacher).
- b) `swapWetter :: Wetter -> State Welt Wetter`
Eine monadische Aktion, welche das alte Wetter der Welt zurückgibt und der Welt ein neues Wetter setzt.

Eine monadische Aktion des Typ `State s a` liefert als Ergebnis einen Wert des Typs `a` und kann dabei einen Wert des Typs `s` lesen und verändern; der Zustand hat also Typ `s`.

In der Vorlesung war der monadische Typ `Zustand` bekannt und wir haben direkt damit gearbeitet (`Zu/noZu`). Dies entfällt hier, da der monadische Typ `State s` hier ein abstrakter Datentyp ist, d.h. wir können diesen nur mit den bereitgestellten monadischen Aktionen bearbeiten. Der aktuellen Zustand kann mit der monadischen Aktion `get :: State s s` ausgelesen werden und mit `put :: s -> State s ()` gesetzt werden. Eventuell geht es auch etwas bequemer unter Verwendung von `gets :: (s->a) -> State s a` (Zustand nur lesen und ein funktional verarbeitetes Ergebnis zurückliefern) oder `modify :: (s->s) -> State s ()` (Zustand funktional verändern, aber kein Ergebnis zurückliefern).

Abgabe: Lösungen zu den Hausaufgaben können bis Samstag, den 7.7.18, mit UniWorX nur als `.zip` abgegeben werden. Abschreiben bei den Hausaufgaben gilt als Betrug und kann zum Ausschluss von der Klausur zur Vorlesung führen. Bis zu 4 Studierende können gemeinsam als Gruppe abgeben. Bitte beachten Sie auch die Hinweise zum Übungsbetrieb auf der Vorlesungshomepage (www.tcs.ifi.lmu.de/lehre/ss-2018/promo/).