Heutiges Thema

- Anmerkungen zu Aufgabenblatt 2 und 3
- Rückblick und Ergänzungen zur Typklasse `Monad`, zu monadischer Programmierung und zu den...
  - Instanzen
    - Zustandsmonade
    - `Id`
    - `IO`
- Typklassenüberblick

Literaturhinweise zur heutigen Vorlesung

- Kapitel 18
  Paul Hudak. *The Haskell School of Expression – Learning Functional Programming through Multimedia*, Cambridge University Press, 2000.
- Kapitel 18
  Simon Thompson. *Haskell – The Craft of Functional Programming*, Addison-Wesley, 2nd edition, 1999.
- Kapitel 21 und 22
  Peter Pepper. *Funktionale Programmierung in Opal, ML, Haskell und Gofer*. Springer-Verlag, 2. Auflage, 2003.
- Kapitel 10
  Fethi Rabhi, Guy Lapalme. *Algorithms - A Functional Approach*. Addison-Wesley, 1999.
- Kapitel 7
  Manuel Chakravarty, Gabriele Keller. *Einführung in die Programmierung mit Haskell*. Pearson, 2004.

Monaden: Ergänzungen und Rückblick

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

  (>>)   :: m a -> m b -> m b
  m >> k = m >>= \_ -> k        -- vordefiniert

  fail   :: String -> m a
  fail s = error s              -- vordefiniert
```

Intuitiv...

- Wenn `m` Instanz der Klasse `Monad` ist, repräsentieren Objekte vom Typ `m a` Berechnungen, die ein Resultat vom Typ `a` liefern.
- `(>>)` für Situationen, in denen die zweite Berechnung vom Resultat der ersten unabhängig ist, deren Wert für die Weiterberechnung also nicht benötigt wird.
- Vgl. `m >>= k` mit `f(g(x))` und `g'(x,f)` (Stichwort: *Programming with Continuations*).

Monadische Programmierung: "Killer"-Applikationen

Lösungen mittels *monadischer* Programmierung bieten sich in besonderer Weise an für *Aufgabenstellungen* mit...

- *globalem Zustand*
  - Aktualisierung im Zuge der Berechnung oft einfacher als Explizieren aller Datenabhängigkeiten (*Zustandsmonade* (engl. *State Monad*)).
  - *Aber*: Durchreichen des Zustandes kann eine ansonsten klare Algorithmusstruktur verdecken und unkenntlich machen.
- *großen Datenstrukturen*
  - Notwendigkeit zur Replikation einer ansonsten nicht mehr benötigten Datenstruktur entfällt.
- *Seiteneffekten und expliziten Auswertungsfolgen*
  - Klassische Situation: Ein- und Ausgabeaktionen (*Ein-/Ausgabemonade* (engl. *IO Monad*)).
- *Ausnahme- und Fehlerbehandlung* (vgl. *Maybe-Monade*)

Zustandsmonade

Modellierung von Programmen mit *globalem* (*internem*) Zustand und *Seiteneffekten*

- ...mittels Funktionen, die angesetzt auf einen Anfangszustand einen Endzustand als Teil des Gesamtergebnisses liefern.

```
newtype State s a = St (s -> (s,a))

instance Monad (State s) where
  return x     = St (\s -> (s,x))    -- Auf Zustaenden die Identitaet!
  (St m) >>= f = St (\s -> let (s1,x) = m s    -- m angesetzt auf s
                               St f'  = f x    -- liefert s1 und x, wo-
                           in f' s1)           -- auf f angesetzt wird
```

Intuitiv...

- *Zustandstransformationen* (engl. *state transformer*) modellieren und transformieren *globale* (*interne*) Programmzustände.
- *Zustandstransformationen*
  - ...sind (hier) Abbildungen vom Typ `s -> (s,a)`
  - ...bilden einen Anfangszustand auf ein Paar aus einem (möglicherweise modifizierten) Endzustand und einem (weiteren Teil-) Ergebnis vom Typ `a` ab.

Die Identitätsmonade

...die konzeptuell einfachste Monade:

```
newtype Id a = Id a

instance Monad Id where
  (Id x) >>= f  =  f x
  return        =  Id
```

Intuitiv:

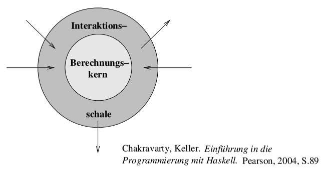
- ...bildet einen Typ auf sich selbst ab.
- ...erlaubt Berechnungsabfolgen auf diesem Typ festzulegen
  (vgl. Beispiel auf Folien 71 bis 77, Vorlesungsteil 5).

Ein-/Ausgabe konzeptuell

Intuitiv...

Programme bestehen konzeptuell aus

- einem *Berechnungskern* und
- einer *Interaktionskomponente*



Chakravarty, Keller. *Einführung in die Programmierung mit Haskell*. Pearson, 2004, S.89

Ein-/Ausgabe ...eine wesentliche Quelle für Seiteneffekte in Programmen!

Ein-/Ausgabe monadisch realisiert

Mittels Monadenkonzepts in Haskell...

- Unterscheidung (und konzeptuelle Trennung) von Funktionen...
  - des *Berechnungskerns* (*reine* Funktionen) und
  - der *Interaktionskomponente* (*seiteneffektbehaftete* Funktionen)

  durch unterschiedliche *Typen*

  ⇝ `Int`, `Real`, `String`,... vs.
  `IO Int`, `IO Real`, `IO String`,... mit `IO` Instanz von `Monad`
- Spezifikation der Auswertungsreihenfolge von Funktionen der Interaktionskomponente (also von in Haskell bereitgestellten *Ein-/Ausgabeprimitiven*)
  - explizit mittels monadischer Programmierung

Typische Interaktionsbeispiele 1(3)

Einfache Frage/Antwort-Interaktion mit dem Benutzer...

```
ask          :: String -> IO String
ask question = do
                 putStrLn question
                 getLine

interAct :: IO ()
interAct = do
             name <- ask "May I ask your name?"
             putStrLine ("Welcome " ++ name ++ "!")
```


Typische Interaktionsbeispiele 2(3)

Ein-/Ausgabe von/auf Dateien...

```
type FilePath = String  -- Dateinamen gemaess
                        -- Betriebssystemkonvention

writeFile  :: FilePath -> String -> IO ()
appendFile :: FilePath -> String -> IO ()
readFile   :: FilePath -> IO String
isEOF      :: FilePath -> IO Bool

interAct :: IO ()
interAct = do
             putStr "Please input a file name: "
             fname <- getLine
             contents <- readFile fname
             putStr contents
```


Typische Interaktionsbeispiele 3(3)

Und noch einmal zum Zusammenhang von do-Notation

```
do writeFile "testFile.txt" "Hello File System!"
   putStr "Hello World!"
```

...und Monadenoperatoren:

```
writeFile "testFile.txt" "Hello File System!" >>
putStr "Hello World!"
```

Beachte folgenden (subtilen) Unterschied:

```
Prelude> putStr ('a':('b':('c':[])))     Prelude> putChar (head ['x','y','z'])
             => abc :: IO ()                          => x :: IO ()
```

wohingegen

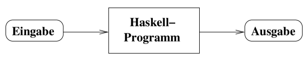
```
Prelude> ('a':('b':('c':[])))            Prelude> head ['x','y','z']
             => "abc" :: [Char]                       => 'x' :: Char

Prelude> print "abc"                     Prelude> print 'x'
             => "abc" :: IO ()                        => 'a' :: IO ()
```


Fazit über Ein-/Ausgabe in Haskell 1(2)

Insgesamt ist in Haskell durch die monadische Behandlung von Ein-/Ausgabe erreicht:

...der Übergang von einer

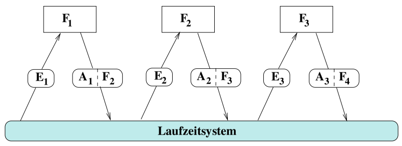
- *"stapelartigen"* (engl. *batch*) Ein-/Ausgabeverarbeitung ausschließlich durch reine Funktionen (des Berechnungskerns):



Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, S.245

Fazit über Ein-/Ausgabe in Haskell 2(2)

...hin zu einer interaktiven, dialogorientierten Ein-/Ausgabeverarbeitung:

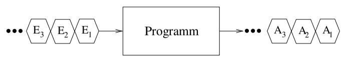


Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, S.253

...ohne Bruch des funktionalen Programmierparadigmas (Stichwort: *referentielle Transparenz*)!

Seitenblicke: Alternative Formen der Ein-/Ausgaberealisierung 1(3)

- *Strommodell*
  ⇝ Programme als Funktionen über Strömen
  `IOprog :: String -> String`



Peter Pepper. *Funktionale Programmierung.* Springer–Verlag, 2003, S.271

Ein-/Ausgabeströme auf Konsole, Dateisystem, Drucker,...

- *ML-Stil*
  ⇝ Unix-ähnliche Behandlung von Bildschirm, Tastatur, etc. als Dateien (`std_in, std_out, open_in, open_out, close_in,...`)

In der Tat...

- In frühen Versionen von Haskell war Ein-/Ausgabe strombasiert realisiert!

Seitenblicke: Alternative Formen der Ein-/Ausgaberealisierung 2(3)

Vor- und Nachteile...

- *Strommodell*
  - In Sprachen mit *eager*-Semantik:
    - kein echtes Strommodell (Eingabe muss vollständig vorweg konsumiert und damit insbesondere endlich sein); somit beschränkt auf stapelartige Ein-/Ausgabeverarbeitung.
  - In Sprachen mit *lazy*-Semantik:
    - Auch Interaktion möglich; dank *lazy evaluation* Ein- und Ausgaben stets in "richtiger" Reihenfolge.
    - Aber: kausaler und zeitlicher Zusammenhang zwischen Ein- und Ausgabe oft "undurchsichtig"; Abhilfe ggf. mit spezieller Synchronisation möglich.
    - Insgesamt: strombasierte Ein-/Ausgaberealisierung findet ihre Grenzen mit graphischen Benutzerschnittstellen und wahlfreiem Zugriff auf Dateien.
- *ML-Stil*
  - Einfach, aber unter Inkaufnahme von Anomalien wie im Zusammenhang mit Ein-/Ausgabe im letzten Semester diskutiert; Verlust referentieller Transparenz.

Seitenblicke: Alternative Formen der Ein-/Ausgaberealisierung 3(3)

Mehr über Ein-/Ausgaberealisierung in funktionalen Programmiersprachen in:

- A. J. Gordon. *Functional Programming and Input/Output*. British Computer Society Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

Übung: “Rechnen mit Monaden” 1(2)

Wir hatten behauptet (siehe Folie 50, Vorlesungsteil 5):

```
do x <- xs; return (f x)
   =>
xs >>= \x -> return (f x)

   => ... ( Definitionen von >>=, map, concat, return, ... )

[ f x | x <- xs ]
```


Übung: “Rechnen mit Monaden” 2(2)

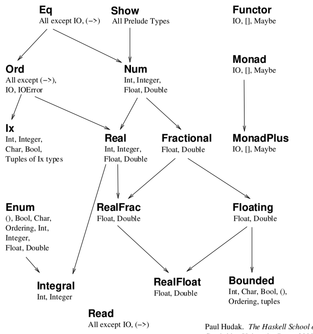
...wovon wir uns durch Nachrechnen überzeugen:

```
xs >>= \x -> return (f x)
   => ( Festlegung m >>= k = concat (map k m) )
concat (map (\x -> return (f x)) xs)
   => ( Definition von map )
concat [ (\x -> return (f x)) y | y <- xs ]
   => ( Funktionsapplikation )
concat [ return (f y) | y <- xs ]
   => ( Definition von return )
concat [ [ f y ] | y <- xs ]
   => ( Definition von concat )
[ f y  | y <- xs ]
```

...there is no magic. It is just functional programming!

Überblick über vordefinierte Typklassen

...und ihre Mitgliedstypen:



Paul Hudak. *The Haskell School of Expression.* Cambridge University Press, 2000, p.156

Überblick über vordefinierte Typen

...und ihre Klassenmitgliedschaften:

Type	Instance of	Derivation
()	Read	Eq Ord Enum Bounded
[a]	Read Functor Monad	Eq Ord
(a,b)	Read	Eq Ord Bounded
(−>)		
Array	Functor Eq Ord Read	
Bool		Eq Ord Enum Read Bounded
Char	Eq Ord Enum Read	
Complex	Floating Read	
Double	RealFloat Read	
Either		Eq Ord Read
Float	RealFloat Read	
Int	Integral Bounded Ix Read	
Integer	Integral Ix Read	
IO	Functor Monad	
IOError	Eq	
Maybe	Functor Monad	Eq Ord Read
Ordering		Eq Ord Enum Read Bounded
Ratio	RealFrac Read	

Fethi Rabhi, Guy Lapalme. *Algorithms.* Addison−Wesley, 1999, Table 2.4, p. 47

Literaturhinweise zu Monaden 1(2)

- Eugenio Moggi. *Computational Lambda Calculus and Monads*. In Proceedings IEEE Symposium on Logic in Computer Science, June 1989.
  ⇝ ...*zur Herkunft des Terminus Monade in programmiersprachlichem Kontext* (lange vorher schon von Leibniz als Gegenstück zum Begriff “Atom” verwendet).
- M. P. Jones. *Functional Programming with Overloading and Higher-Order Polymorphism*. Advanced Functional Programming, LNCS 925, Springer-Verlag, 1995, 97 - 136.
- Simon Peyton Jones, John Launchbury. *State in Haskell*. Lisp and Symbolic Computation 8(4), 1995, 293 - 341.

Literaturhinweise zu Monaden 2(2)

- Philip Wadler. *Comprehending Monads*. Mathematical Structures in Computer Science, vol. 2, 1992, 461 - 493.
- Philip Wadler. *Monads for Functional Programming*. Advanced Functional Programming, LNCS 925, Springer-Verlag, 1995, 24 - 52.
- Philip Wadler. *How to Declare an Imperative*. ACM Computing Surveys 29(3), 1997, 240 - 263.

Vorschau auf die nächsten Vorlesungstermine...

- Di, 07.06.2005, 14 - 16 Uhr, CompLang-Bibliothek
- Di, 21.06.2005, 14 - 16 Uhr, CompLang-Bibliothek