Programmieren im Großen...

Zunächst...

- Programmierung mit Monaden
  - Zusammenhang Monaden und Ein-/Ausgabe in Haskell

Anschließend...

- Programmieren im Großen...
  - Das Modulkonzept von Haskell
  - Abstrakte Datentypen

Ein-/Ausgabe in Haskell und Monaden

Ein-/Ausgabe in Haskell...

- ...als Spezialfall eines allgemeineren Konzepts realisiert, dem Konzept der *Monade*.

Deshalb: Exkurs über das Monadenkonzept in Haskell

Monaden und Monadischer Programmierstil (1)

Charakteristisch...

- ...erlauben die Reihenfolge, in denen Operationen ausgeführt werden, explizit festzulegen.

Beispiel...

```
a-b    -- Keine Festlegung der Auswertungsreihenfolge;
       -- kritisch, falls z.B~Ein-/Ausgabe involviert ist.

do a <- getInt   -- Reihenfolge explizit festgelegt
   b <- getInt
   return (a-b)
```


Monaden und Monadischer Programmierstil (2)

Zentral...

- Die Funktionen (`>>=`) und `return` (und das abgeleitete (und in der Anwendung oft bequemere) `do`-Konstrukt)

Allgemein: Monaden sind...

...*Konstruktorklassen*, Familien von Typen `m a` über einem polymorphen Typkonstruktor `m` mit den Funktionen `(>>=)`, `return`, `(>>)` und `fail`.

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

  m >> k = m >>= \_ -> k     -- vordefiniert
  fail s = error s           -- vordefiniert
```

...wobei die Implementierungen der Funktionen gewissen Anforderungen genügen müssen.

Anforderungen (Gesetze)

```
return a >>= f               =  f a
c >>= return                 =  c

c >>= (\x -> (f x) >>= g)    =  (c >>= f) >>= g
```

Intuitiv:

- `return` gibt den Wert zurück, ohne einen weiteren Effekt.
- durch >>= gegebene Sequenzierungen sind unabhängig von der Klammerung (assoziativ)

Konstruktorklassen vs. Typklassen

Im Grundsatz ähnliche Konzepte, wobei...

- Konstruktorklassen
  - ...haben Typkonstruktoren als Elemente
- Typklassen (`Eq a`, `Ord a`, `Num a`,...)
  - ...haben Typen als Elemente
- Typkonstruktoren sind...
  - ...Funktionen, die aus gegebenen Typen neue Typen erzeugen
    (Bsp.: Tupelkonstruktor `( )`, Listenkonstruktor `[ ]`,
    aber auch: Ein-/Ausgabe `IO`,...)

Der abgeleitete Operator (>@>)...

...ist folgendermaßen definiert:

```
(>@>) :: Monad m => (a -> m b) ->
                    (b -> m c) ->
                    (a -> m c)

f >@> g = \x -> (f x) >>= g
```

Hinweis:

...`return` ist vom Typ `a -> m c`!

Damit...

```
(1) return >@> f = f
(2) f >@> return = f

(3) (f >@> g) >@> h = f >@> (g >@> h)
```

Intuitiv...

- (1)&(2): `return` ist Einselement von `(>@>)`
- (3): `(>@>)` ist assoziativ

Beachte:
Obige Eigenschaften gelten nicht a priori, sondern sind durch die Implementierung sicherzustellen!

Standardfunktionen über Monaden

Kombination von Monaden...

- ...zum Aufbau komplexerer Effekte

```
mapF :: Monad m => (a -> b) -> m a -> m b
mapF f m = do x <- m
              return (f x)

joinM :: Monad m => m (m a) -> m a
joinM m = do x <- m
             x
```

Bemerkung:

- Aus (1), (2) und (3) folgt:
  `(4) mapF (f.g) = mapF f . mapF g`

Ein-/Ausgabe und Monaden

Erinnnerung:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

wobei...

- `(>>=)`: Wenn `p` und `q` Kommandos sind, dann ist `p >>= q` das Kommando, das zunächst `p` ausführt, dabei den Rückgabewert `x` vom Typ `a` liefert, und daran anschließend `q x` ausführt und dabei den Rückgabewert `y` vom Typ `b` liefert.
- `return`: Rückgabewerterzeugung ohne Ein-/Ausgabe(aktion)

Somit...

- Ein-/Ausgabe in Haskell Monad über dem Typkonstruktor `IO`

Programmieren mit Monaden: Ein Beispiel (1)

Zum Einstieg und Vergleich: Die monadenlose Variante...

```
data Tree a = Nil | Node a (Tree a) (Tree a)

-- Summieren der Marken in einem "Integer"-Baum
sTree :: Tree Int -> Int
sTree Nil            = 0
sTree (Node n t1 t2) = n + sTree t1 + sTree t2
```

Beachte:

- Reihenfolge der Berechnung weitgehend nicht festgelegt (Freiheitsgrade!)

Programmieren mit Monaden: Ein Beispiel (2)

Zum Vergleich: Eine monadenbehaftete Variante

```
sumTree :: Tree Int -> Id Int
sumTree Nil = return 0
sumTree (Node n t1 t2) = do num <- return n
                            s1  <- sumTree t1
                            s2  <- sumTree t2
                            return (num + s1 + s2)
```

Beachte:

- Reihenfolge der Berechnung explizit festgelegt (keine Freiheitsgrade!)

Programmieren mit Monaden: Ein Beispiel (3)

Dabei gilt...

```
data Id a = Id a

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

Beachte:

- Anders als `sTree` hat `sumTree` einen "imperativen Anstrich". Vergleiche:

```
num  := n;
sum1 := sumTree t1;
sum2 := sumTree t2;
return (num + sum1 + sum2);
```


Programmieren mit Monaden: Ein Resümee (1)

Die Programmierung mit Monaden erlaubt es...

- Berechnungsabläufe zu strukturieren.

Spezielle Eigenschaften von Monaden machen ihre Wahl hierfür besonders vorteilhaft:

- *Wohldefiniert*: ...Strategie sequentielle Programmteile systematisch zu spezifizieren.
- *Angemessen*: ...höhere Abstraktion durch Entkopplung der zugrundeliegenden Monade von der Struktur der Berechnung.
- *Fundiert*: ...Eigenschaften wie etwa (4) werden impliziert von den Monadforderungen (1), (2) und (3).

Programmieren mit Monaden: Ein Resümee (2)

Monaden...

- ...ein in der Kategorientheorie geprägter Begriff
  $\leadsto$ ...zur formalen Beschreibung der Semantik von Programmiersprachen (Eugenio Moggi, 1989)
- ...populär in der Welt funktionaler Programmierung (ohne obigen Hintergrund) insbesondere weil (Philip Wadler, 1992)
  - ...erlauben gewisse Aspekte imperativer Programmierung in die funktionale Programmierung zu übertragen
  - ...eignen sich insbesondere zur Integration von Ein-/Ausgabe, aber auch für weitergehende Anwendungsszenarien
  - ...geeignete Schnittstelle zwischen funktionaler und effektbehafteter, z.B. imperativer und objektorientierter Programmierung.

Programmieren im Großen

Das Modulkonzept von Haskell...

Modularisierung im allgemeinen... (1)

Intuitiv:

- Zerlegung großer Programm(systeme) in kleinere Einheiten, genannt *Module*

Ziel:

- Sinnvolle, über- und durchschaubare Organisation des Gesamtsystems

Modularisierung im allgemeinen... (2)

Vorteile:

- *Arbeitsphysiologisch* ...Unterstützung arbeitsteiliger Programmierung
- *Softwaretechnisch* ...Unterstützung der Wiederbenutzung von Programmen und Programmteilen
- *Implementierungstechnisch* ...Unterstützung von "separate compilation"

Insgesamt:

- Höhere Effizienz der Softwareerstellung bei gleichzeitiger Steigerung der Qualität (Verlässlichkeit) zu reduzierten Kosten

Modularisierung im allgemeinen... (3)

Anforderungen an das Modulkonzept zur Erreichung vorgenannter Ziele:

- Unterstützung des Geheimnisprinzips
  ...durch Trennung von
  - Schnittstelle (Import/Export)
    ...wie interagiert das Modul mit seiner Umgebung? Welche Funktionalität stellt es zur Verfügung (*Export*), *welche Funktionalität erwartet es* (*Import*)*?*
  - Implementierung (Daten/Funktionen)
    ...wie ist die Funktionalität des Moduls realisiert?

  in einem Modul.

Module in Haskell – Allgemeiner Aufbau

```
module MyModule where

-- Daten- und Typdefinitionen
data D1 ... = ...
...
data Dn ... = ...

type T1 = ...
...
type Tn = ...

-- Funktionsdefinitionen
f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = ...
```

Das Modulkonzept von Haskell...

...unterstützt:

- Export
  - Ungezielt/Gezielt
- Import
  - Ungezielt/Gezielt
  - Qualifiziert
  - Mit Umbenennung

...unterstützt nicht:

- automatischer Reexport!

Import: Ungezielt/Nicht selektiv

```
module M1 where
...

module M2 where
import M1    -- Ungezielter/Nicht selektiver Import:
             -- Alle im Modul M1 (sichtbaren) Bezeichner/
             -- Definitionen werden importiert und koennen
             -- in M2 benutzt werden.
```


Import: Gezielt/Selektiv

```
module M1 where
...                              -- wie auf voriger Folie

module M2 where
import M1 (D1, D2 (), T1, f5)    -- Gezielter/Selektiver Import:
                                 -- Lediglich D1 (ohne Konstruktoren),
                                 -- D2 (einschliesslich Konstruktoren,
                                 -- T1 und f5 werden importiert und
                                 -- koennen in M2 benutzt werden.

module M3 where
import M1 hiding (D1, T2, f1)    -- Gezielter/Selektiver Import:
                                 -- Alle (sichtbaren) Bezeichner/
                                 -- Definitionen mit Ausnahme der
                                 -- explizit genannten werden importiert
                                 -- und koennen in M3 benutzt werden.
```

Export: Ungezielt/Nicht selektiv

```
module M1 where     -- Ungezielter/Nicht selektiver Export:
data D1 ... = ...   -- Alle im Modul M1 (sichtbaren)
...                 -- Bezeichner/Definitionen werden
data Dn ... = ...   -- exportiert und koennen von anderen
                    -- Modulen importiert werden.
type T1 = ...
...
type Tn = ...

f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = ......
```


Export: Gezielt

```
module M1 (D1, D2 (), T1, f2, f5) where
data D1 ... = ...
...                     -- Gezielter/Selektiver Export:
data Dn ... = ...       -- Nur die explizit genannten Bezeichner/
                        -- Definitionen werden exportiert und
type T1 = ...           -- koennen von anderen Modulen importiert
...                     -- werden.
type Tn = ...           -- Unterstuetzt Geheimnisprinzip!


f1 :: ...
f1 ... = ...
...
fn :: ...
fn ... = ......
```


Export: Kein automatischer Reexport (1)

```
module M1 where ...

module M2 where
import M1     -- Ungezielter/Nicht selektiver Import:
              -- Alle im Modul M1 (sichtbaren) Bezeichner/
fM2...        -- Definitionen werden importiert und koennen
              -- in M2 benutzt werden.

module M3 where
import M2     -- Alle im Modul M2 (sichtbaren) Bezeichner/
              -- Definitionen werden importiert und koennen
              -- in M2  benutzt werden, nicht jedoch von
              -- M2 aus M1 importierte Namen. Mithin: Kein
              -- automatischer Reexport!
```


Export: Kein automatischer Reexport (2)

Abhilfe: Expliziter Reexport!

```
module M4 (D1 (), f1, f2) where   -- Gezielter/Selektiver
import M1                         -- Reexport

module M2 (M1,fM2) where   -- Ungezielter/Nicht selektiver
import M1                  -- Reexport
```

Sonderfälle: Namenskollisionen, Lokale Namen

- Namenskollisionen
  - Abhilfe/Auflösen: Qualifizierter Import
    `import qualified M1`
- Umbenennen importierter Module
  - Lokale Namen importierter
    - Module
      `import MyM1 as M1`
    - Bezeichner
      `import M1 (f1,f2) renaming (f1 to g1, f2 to g2)`

Konventionen und gute Praxis

- Konventionen
  - Pro Datei ein Modul
  - Modul- und Dateiname stimmen überein (abgesehen von der Endung `.hs` bzw. `.lhs` im Dateinamen).
  - Alle Deklarationen beginnen in derselben Spalte wie `module`.
- Gute Praxis
  - Module unterstützen *eine* (*!*) klar abgegrenzte Aufgabenstellung (vollständig) und sind in diesem Sinne in sich abgeschlossen; ansonsten Teilen (Teilungskriterium)
  - Module sind "kurz" (ideal: 2 bis 3 Druckseiten; prinzipiell: "so lang wie nötig, so kurz wie möglich")

Zu guter Letzt...

Module `main`...

- ...sollte in jedem Modulsystem als "top-level" Modul vorkommen und eine Definition namens `main` festlegen.
  $\leadsto$ ...ist der in einem übersetzten System bei Ausführung des übersetzten Codes zur Auswertung kommende Ausdruck.

Organisatorisches...

Abgabe- und Prüfungsgespräche...

- Wann, wo, Anmeldung...
  - *Wann*: Im gesamten Februar (Schwerpunkt: Mo, 02.02.04 - Fr, 06.02.04); einige (wenige) Termine in der ersten Märzwoche)
  - *Wo*: siehe Webseite (kurzfristige Festlegung/Änderung wg. Übersiedlung)
  - *Anmeldung*: Mo, 26.01.04 - Fr, 30.01.04 (durch Eintragen in Listen in den Laborräumen im 2. Stock, Argentinierstr. 8)
- Inhalt
  - Fragen zu Aufgaben und Vorlesung