Ein einfaches „interaktives Spiel“

Towers of Hanoi:

- es gibt drei Türme/Plätze: A, B und C
- zu Beginn liegen n Scheiben unterschiedlicher Größe auf A; B und C sind leer
- Ziel ist es, die Scheiben von A nach B zu transportieren
- zu keinem Zeitpunkt darf eine Scheibe auf einer kleineren anderen liegen

Beispiel mit drei Scheiben:

- zu Beginn „Konfiguration“ (A: 1,2,3; B leer; C leer)
- Zug A $\mapsto$ B, führt zu (A: 2,3; B: 1; C leer)
- Zug A $\mapsto$ C, führt zu (A: 3; B: 1; C: 2)
- Zug B $\mapsto$ C, führt zu (A: 3; B leer; C: 1,2)
- Zug A $\mapsto$ B, führt zu (A leer; B: 3; C: 1,2)
- Zug C $\mapsto$ A, führt zu (A: 1; B: 3; C: 2)
- Zug C $\mapsto$ B, führt zu (A: 1; B: 2,3; C leer)
- Zug A $\mapsto$ B, führt zu (A leer; B: 1,2,3; C leer)

Ein einfaches „interaktives Spiel“

Towers of Hanoi, allgemeine Strategie (Divide and Conquer):

- um n Scheiben von A nach B unter Nutzung von C zu bringen:
  - zunächst n – 1 Scheiben von A nach C unter Nutzung von B
  - dann eine Scheibe von A nach B
  - schließlich n – 1 Scheiben von C nach B unter Nutzung von A
- um n – 1 Scheiben von A nach C unter Nutzung von B zu bringen:
  - zunächst n – 2 Scheiben von A nach B unter Nutzung von C
  - dann eine Scheibe von A nach C
  - schließlich n – 2 Scheiben von B nach C unter Nutzung von A
- …

```
data Place = A | B | C  deriving Show

towers 0 i j k = [ ]
towers n i j k = towers (n – 1) i k j ++ [ (i, j) ] ++ towers (n – 1) k j i
```

```
> towers 3 A B C
[ (A, B), (A, C), (B, C), (A, B), (C, A), (C, B), (A, B) ]
```

Kommentiertes Ergebnis des „Live-Programmierens“ (1)

in der Version des Vorjahrs

```
module Main where

data Place = A | B | C  deriving (Show, Read, Eq)

type Move = (Place, Place)
type Conf = ([Int], [Int], [Int])

run :: [Move] → Conf → [Conf]
run [ ]          c = [ c ]
run (m : ms) c = c : run ms (step m c)

step :: Move → Conf → Conf
step (A, B) (a : as, bs, cs) = (as, a : bs, cs)
step (A, C) (a : as, bs, cs) = (as, bs, a : cs)
step (B, A) (as, b : bs, cs) = (b : as, bs, cs)
step (B, C) (as, b : bs, cs) = (as, bs, b : cs)
step (C, A) (as, bs, c : cs) = (c : as, bs, cs)
step (C, B) (as, bs, c : cs) = (as, c : bs, cs)
```

Typsynonyme zur Abkürzung

„Simulierter Lauf“ einer Zugfolge auf einer gegebenen Konfiguration

Kommentiertes Ergebnis des „Live-Programmierens“ (2)

in der Version des Vorjahrs

```
animate n [ ]        =  return ()
animate n (c : cs)   =  do output' c n
                           getLine
                           animate n cs

main  =  do n ← readLn
            animate n (run (towers n A B C) ([1 .. n], [ ], [ ]))

output' (as, bs, cs) n = output n
                         (replicate (n – length as) 0 ++ as,
                          replicate (n – length bs) 0 ++ bs,
                          replicate (n – length cs) 0 ++ cs)

output n ([ ], [ ], [ ])              =  return ()
output n (a : as, b : bs, c : cs)     =  do putStr (disc a n)
                                            putStr (disc b n)
                                            putStrLn (disc c n)
                                            output n (as, bs, cs)
```

jetzt „Hauptschleife“

„hübsche Ausgabe“

Kommentiertes Ergebnis des „Live-Programmierens“ (3)

in der Version des Vorjahrs

```
disc :: Int → Int → String
disc 0 n  =  replicate (2 * n – 1) ' '
disc i  n  =  replicate (n – i) ' ' ++ replicate (2 * i – 1) '*'
              ++ replicate (n – i) ' '

towers 0 i j k = [ ]
towers n i j k = towers (n – 1) i k j ++ [ (i, j) ] ++ towers (n – 1) k j i
```

reine Berechnung einer String-darstellung

unveränderte „Ursprungslogik“

Anmerkungen:

- inkrementelle Entwicklung:
  - zunächst nur run und step; statt animate nur print
  - dann animate, mit print statt output'
  - schließlich output, output', disc
- zwischendurch jeweils Testen von Teilfunktionalität
- nicht überall Funktionstypen „gepflegt“, stattdessen inferieren lassen

Anregungen zum Ausprobieren

- Wir könnten das „interaktive Programm mit Visualisierung“ zu Towers of Hanoi dahingehend ändern, dass der Nutzer die Züge vorgibt.
  - … dabei zunächst annehmen, dass stets nur korrekte Züge eingegeben werden,
  - … oder explizite Checks mit (Fehler-)Meldung an den Nutzer einfügen,
  - … vielleicht sogar die Möglichkeit schaffen, dass der Nutzer zu jedem Zeitpunkt um „Hilfe“ bitten kann, woraufhin das Puzzle vom aktuellen Stand aus wieder automatisch gelöst wird.

Funktionale Programmierung in Haskell: Zwischenbilanz (1)

- Prinzip der FP:
  - Spezifikation = Folge von Funktionsdefinitionen
  - Funktionsdefinition = Folge von definierenden Gleichungen
  - Operationalisierung = stufenweise Reduktion von Ausdrücken auf Werte

- Ausdrücke:
  - Konstanten, Variablen, strukturierte Ausdrücke: Listen, Tupel
  - Applikationen
  - list comprehensions

- Systeme definierender Gleichungen:
  - Kopf, Rumpf (mit div. Beschränkungen)
  - (ggfs.) mehrelementige Parameterlisten
  - Wächter

- syntaktische Besonderheiten von Haskell:
  - von mathematischer Notation abweichende Funktionssyntax
  - lokale Definitionen (let, where)
  - Layoutregel

Funktionale Programmierung in Haskell: Zwischenbilanz (2)

- Reduktion/Auswertung:
  - pattern matching, eindeutige Fallauswahl, Rekursion
  - lazy evaluation
  - besondere Rolle von IO, do-Blöcke

- Listen:
  - Klammerdarstellung vs. Baumdarstellung (:), pattern matching
  - spez. Listenfunktionen (z.B. length, ++, !!)
  - arithmetische Sequenzen, unendliche Listen, list comprehensions

- Typen (starke Typisierung, Typprüfung, Typinferenz):
  - Datentypen
    - Basisdatentypen (Integer etc.)
    - strukturierte Typen (Listen, Tupel)
    - algebraische Datentypdeklarationen, Konstruktoren
  - polymorphe Typen, Typvariablen
  - Funktionstypen
    - Funktionstypdeklarationen, Curryfizierung
  - Typklassen, Deklarationen, Instanzdefinitionen

Funktionale Programmierung in Haskell: Zwischenbilanz (3)

- Funktionen höherer Ordnung:
  - Funktionen als Parameter und/oder als Resultate
  - partielle Applikation, Sections
  - Lambda-Ausdrücke
  - Funktionen höherer Ordnung auf Listen: map, filter, foldr, …

- Verwendung expliziter Rekursionsschemata