Praktische Informatik 3: Funktionale Programmierung

Vorlesung 7 (29.11.2022): Übungen

Christoph Lüth

Wintersemester 2022/23

Was zum Nachdenken

```
queens :: Int→ [Board]
queens n = qu n where
  qu :: Int→ [Board]
  qu i | i == 0     = [[]] — Nicht [] !
       | otherwise = [ p++ [(i, j)] | p ← qu (i-1), j ← [1.. n],
                                      safe p (i, j)]
```

Übung 7.1: Warum?

Wieso ist dort `[[]]` so wichtig? Was passiert, wenn wir `[]` zurückgeben?

Was zum Nachdenken

```
queens :: Int→ [Board]
queens n = qu n where
  qu :: Int→ [Board]
  qu i | i == 0     = [[]] — Nicht [] !
       | otherwise = [ p++ [(i, j)] | p ← qu (i-1), j ← [1.. n],
                                      safe p (i, j)]
```

Übung 7.1: Warum?

Wieso ist dort `[[]]` so wichtig? Was passiert, wenn wir `[]` zurückgeben?

Lösung:

- Mit `[]` gibt es **keine** Lösung, mit `[[]]` gibt es **eine, leere** Lösung für `i == 0`.
- Mit `[]` gäbe es **nie** eine Lösung für **alle** `i`.

Kurze Denkpause

Übung 7.2: Merkwürdige Zahlen

Wenn wir die natürlichen Zahlen mit einem Typ-Parameter versehen:

```
data FNat α = FZero | FSucc α (FNat α)
```

Was ist die kanonische Funktion `foldFNat`, und welcher Datentyp ist das?

Kurze Denkpause

Übung 7.2: Merkwürdige Zahlen

Wenn wir die natürlichen Zahlen mit einem Typ-Parameter versehen:

```
data FNat α = FZero | FSucc α (FNat α)
```

Was ist die kanonische Funktion `foldFNat`, und welcher Datentyp ist das?

Lösung:

```
foldFNat :: β→ (α→ β→ β)→ FNat α→ β
foldFNat e f FZero  = e
foldFNat e f (FSucc a n) = f a (foldFNat e f n)
```

Das sind natürlich Listen, mit `foldr`:

```
foldr :: (α → β → β) → β → [α] → β
```

`map` and `filter` via `fold`

Übung 7.3:

Implementiert `map` und `filter` unter Benutzung von `fold`. Die Lösung darf nicht rekursiv sein!

`map` and `filter` via `fold`

Übung 7.3:

Implementiert `map` und `filter` unter Benutzung von `fold`. Die Lösung darf nicht rekursiv sein!

Lösung:

```
map f = foldr ((:). f) []
```

```
filter p = foldr (λa as→ if p a then a:as else as) []
```

Kurzes Gehirnjogging

Übung 7.4:

Wie sieht die Version von `take` mit `fold` aus (`foldl` oder `foldr`)?

Kurzes Gehirnjogging

Übung 7.4:

Wie sieht die Version von `take` mit `fold` aus (`foldl` oder `foldr`)?

Lösung:

- Mit `foldl`:

```
takel i = snd ∘ foldl (λ(c, p) a→ (c+1, if c< i then (p++[a]) else p)) (0, [])
```

- Mit `foldr`:

```
taker i xs =
  snd $ foldr (λa (c, p)→ (c+1, if c ≥ length xs-i then a:p else [])) (0, []) xs
```

  - Beides terminiert nicht für zyklische Listen; das erste wegen `foldl`, das zweite wegen `length xs`.

- Folgendes terminert für zyklische Listen:

```
taker1 i = foldr (λ(j, x) xs→ if j≤ i then x:xs else xs) [] ∘ zip [1..i]
```

Geschummelt weil `zip` nicht mit `fold` implementiert werden kann