Fakultät IV: Elektrotechnik und Informatik
Informatik 1 WS 2001/02,
Jähnichen / Flik / Lechtchinsky / Neunast / Tutoren

Aufgabenblatt 5

letzte Aktualisierung: 05. December, 14:00

Ausgabe:	*23.11.2001*		
Abgabe:	*3./4.12.2001*	*Prozent:*	*100*

Themen: *Funktionen höherer Ordnung*

1. Aufgabe (30 Prozent): Funktionen als Argumente und als Resultate von Funktionen

1.1. Funktionskomposition mit Auswertung (Tut) Implementiert eine Funktion `composeApply`, die zwei Funktionen und einen Wert als Argumente erhält und die Verkettung der Funktionen auf den Wert anwendet.

Simuliert die Funktionsaufrufe `composeApply(\\ x. x * x, \\ y. y, 4)` und `composeApply(\\ x. (2 * x) + 1, succ, 5)`

Lösung:

```
FUN composeApply: (nat -> nat) ** (nat -> nat) ** nat -> nat
DEF composeApply(f, g, x) == f(g(x))
```

`composeApply(\\ x. x * x, \\ y. y, 4)`
→ *(\\ x. x * x)((\\ y. y)(4))*
→ *(\\ x. x * x)(4)*
→ *4 * 4*
→ *16*

`composeApply(\\ x. (2 * x) + 1, succ, 5)`
→ *(\\ x. (2 * x) + 1)(succ(5))*
→ *(\\ x. (2 * x) + 1)(6)*
→ *2 * 6 + 1*
→ *13*

Ergebnisse der Simulationen: 16 *und* 13.

1.2. Die Funktion `flip` (5 Prozent) Schreibt die Funktion `flip`, die eine Funktion `f` mit zwei Parametern als Argument erhält und eine Funktion zurückliefert, die `f` mit vertauschten Parametern aufruft. Die Parameter und der Rückgabewert sollen vom Typ `nat` sein.

Lösung:

```
FUN flip: (nat ** nat -> nat) -> (nat ** nat -> nat)
DEF flip == \\ f. \\ x, y. f(y, x)
```

1.3. Die Funktion `twice` (10 Prozent) Schreibt die Funktion `twice`, die eine Funktion `f` mit einem Parameter als Argument erhält und eine Funktion zurückliefert, die `f` zweimal auf ihren Parameter anwendet. Der Parameter soll vom Typ `nat` sein.

Musterloesung

Lösung:

```
FUN twice: (nat -> nat) -> (nat -> nat)
DEF twice == \\ f. \\ x. (f(f(x)))
```

oder

```
DEF twice(f) == \\ x. (f(f(x)))
```

oder mittels Funktionskomposition

```
DEF twice(f) == f o f
```

1.4. Funktionsverschiebung (15 Prozent) Deklariert und definiert folgenden Funktionen in einer eigenen Struktur:

1. Deklariert und definiert eine Funktion `shiftX`, die eine Funktion `h` entlang der x-Achse um `deltaX` verschiebt.
2. Deklariert und definiert eine Funktion `mirrorY`, die eine Funktion `h` an der y-Achse spiegelt.
3. Deklariert und definiert eine Funktion `myCos` (Kosinus-Funktion) mit Hilfe bereits definierter Funktionen.
4. Deklariert und definiert eine Funktion `myCot` (Kotangens-Funktion) mit Hilfe bereits definierter Funktionen.

Hinweis: Benutzt nicht die Funktionen `cos` und `cot` aus der Struktur `Real`.

Lösung:

```
-- Die Funktion shiftX verschiebt die Funktion h entlang der x-Achse um deltaX.
FUN shiftX : real -> (real -> real) -> (real -> real)
DEF shiftX(deltaX)(f)(x) == f(x - deltaX)
-- Die Funktion mirrorY spiegelt die Funktion h an der y-Achse.
FUN mirrorY: (real -> real)-> (real -> real)
DEF mirrorY(f)(x) == f(-(x))
-- Die Funktion myCos soll die Kosinus-Funktion berechnen.
FUN myCos : real -> real
DEF myCos == shiftX(-(pi/("2"!)))(sin)
-- Die Funktion myCot soll die Kotangens-Funktion berechnen.
FUN myCot : real -> real
DEF myCot == mirrorY(shiftX(-(pi/("2"!)))(tan))
```

oder

```
DEF myCot == shiftX(-(pi/("2"!)))(mirrorY(tan))
```

2. Aufgabe (20 Prozent): Funktionalitäten

2.1. (Tut) Macht Euch mit den Klammerregeln von Typausdrücken in Opal vertraut.
Aus: P. Pepper: Funktionale Programmierung. 1999. S. 94:

- Die Tupelbildung '×' bindet stärker als die Funktionsbildung '→'.
- Die Tupelbildung '×' ist assoziativ.

- Die Funktionsbildung '→' bindet nach rechts, d.h., A → B → C ist das Gleiche wie A → (B → C).

2.2. (Tut) Gegeben sind folgende Funktionalitäten. Mit Ausnahme von `f` stammen alle Funktionen aus der Struktur `Nat`. Welche Funktionalität paßt zu folgender Funktionsanwendung `f(<,=)(+)(3)(4)`?

```
1. FUN f: (nat -> bool) -> (nat -> bool) -> (nat ** nat -> nat) ** nat ** nat
          -> nat

2. FUN f: (nat ** nat -> bool) ** (nat ** nat -> bool) ->
          (nat ** nat -> nat) -> nat -> nat -> nat

3. FUN f: (nat ** nat -> bool) ** (nat ** nat -> bool) ->
          (nat ** nat -> nat) ** (nat ** nat -> nat)

4. FUN f: (nat ** nat -> bool) -> (nat ** nat  -> bool) ** (nat ** nat -> nat)
           -> nat -> nat -> nat

5. FUN f: (nat ** nat -> bool) -> ((nat ** nat -> nat) ** nat ** nat) -> nat

6. FUN f: (nat ** nat -> bool) ** (nat ** nat -> nat) ->
          (nat ** nat -> nat) ** nat ** nat -> nat
```

Lösung:

```
2. FUN f: (nat ** nat -> bool) ** (nat ** nat -> bool) ->
          (nat ** nat -> nat) -> nat -> nat -> nat
```

ist die richtige Funktionalität.

2.3. (20 Prozent) Ihr habt Euch in **2.2** für eine Funktionalität entschieden. Welche der folgenden Ausdrücke sind bzgl. dieser korrekt? Von welchem Typ sind diese korrekten Ausdrücke?

```
1. f                                6. f(<=,>)(*)(33)(pow(8,6))
2. f(|=,=)                          7. f(>=,div)(\\ x, y. y-x)(2+10)
3. f(even?)(odd?)                   8. f(=,>)(mod)(double(2))
4. f(>,mod)                         9. f(|=,=)(min)
5. f(<,*)(+)("2"!)(succ(0))        10. f(even?,double)(/)(half(8))
```

Lösung:
Die passenden Ausdrücke und ihre Funktionalitäten:

```
1. korrekt:
     f: (nat ** nat -> bool) ** (nat ** nat -> bool) ->
        (nat ** nat -> nat) -> nat -> nat -> nat

2. korrekt:
     f(|=,=): (nat ** nat -> nat) -> nat -> nat -> nat

3. falsch, weil
     even?: nat -> bool
     odd?: nat -> bool
```



```
4. falsch, weil
     mod: nat ** nat -> nat

5. falsch, weil
     *: nat ** nat -> nat

6. korrekt:
     f(<=,>)(*)(33)(pow(8,6)): nat

7. falsch, weil
     div: nat ** nat -> nat

8. korrekt:
     f(=,>)(mod)(double(2)): nat -> nat

9. korrekt:
     f(|=,=)(min): nat -> nat -> nat

10. falsch, weil:
      even?: nat -> bool
      double: nat -> nat
```

3. Aufgabe (20 Prozent): Currying/Uncurrying

3.1. Funktion `curry` (Tut) Definiert die Funktion `curry`, die eine Funktion in Tupelnotation in eine Funktion in Curry-Notation umwandelt.

Die umzuwandelnde Funktion hat zwei Parameter vom Typ `nat`, der Rückgabetyp ist ebenfalls `nat`.

Lösung:

```
FUN curry: (nat ** nat -> nat) -> nat -> nat -> nat
DEF curry == \\ f. \\ x. \\ y. f(x, y)
```

oder

```
DEF curry(f)(x)(y) == f(x, y)
```

3.2. Funktion `uncurry` (5 Prozent) Definiert die Funktion `uncurry`, die das Gegenteil von `curry` tut. Sie wandelt Funktionen von Curry-Schreibweise in Tupelschreibweise um.

Lösung:

```
FUN uncurry: (nat -> nat -> nat) -> nat ** nat -> nat
DEF uncurry == \\ f. \\ x, y. f(x)(y)
```

oder

```
DEF uncurry(f)(x, y) == f(x)(y)
```

3.3. Currying anwenden (15 Prozent) Implementiert die Funktion `FUN myAdd: nat ** nat -> nat`, die zwei Zahlen addiert, in Lambda- und in Gleichungsnotation.

Definiert des weiteren die Funktion `myAddCurry` unter Verwendung der Funktion `curry`. Sie soll ebenfalls zwei Zahlen addieren, aber Currying verwenden.

Definiert die Funktion `plusFive`, die unter Verwendung der Funktion `myAddCurry` 5 zu ihrem Parameter addiert.

Lösung:


```
FUN myAdd: nat ** nat -> nat
DEF myAdd == \\ x, y. x + y
DEF myAdd(x, y) == x + y
```

oder in Curry-Notation:

```
FUN myAddCurry: nat -> nat -> nat
DEF myAddCurry == \\ x. \\ y. x + y
DEF myAddCurry(x)(y) == x + y
DEF myAddCurry(x) == \\ y. x + y
```

Unter Verwendung der Funktion `curry`*:*

```
DEF myAddCurry == curry(myAdd)

FUN plusFive: nat -> nat
DEF plusFive == myAddCurry(5)
```

4. Aufgabe (30 Prozent): Funktionen höherer Ordnung auf Sequenzen

4.1. `map` und `filter` (Tut) Diskutiert die zwei Funktionale `map` und `filter`.

Implementiert die Funktionen `map` und `filter`.

Implementiert die Funktion `quicksort` unter Verwendung dieser Funktionale.

Implementiert eine Funktion `toUpper`, die eine Sequenz von `char` in Großbuchstaben umwandelt.

Lösung:

```
FUN myMap: (nat -> nat) -> seq[nat] -> seq[nat]
DEF myMap(f)(s) ==
  IF <>?(s) THEN <>
  ELSE f(ft(s)) :: myMap(f)(rt(s))
  FI

FUN myFilter: (nat -> bool) -> seq[nat] -> seq[nat]
DEF myFilter(p)(s) ==
  IF <>?(s) THEN <>
  OTHERWISE
  IF p(ft(s)) THEN ft(s) :: myFilter(p)(rt(s))
  ELSE myFilter(p)(rt(s))
  FI

FUN quicksort: seq[nat] -> seq[nat]
DEF quicksort(<>) == <>
DEF quicksort(a :: R) ==
  LET
     smaller == filter(\\ x. x < a)(R)
     greatereq == filter(\\ x. x >= a)(R)
  IN
```



```
    quicksort(smaller) ++ %(a) ++ quicksort(greatereq)

FUN toUpper: seq[char] -> seq[char]
DEF toUpper(s) == map(upper)(s)
```

Musterloesung

4.2. Anwendung von Funktionalen (15 Prozent) Verwendet für die Lösung dieser Aufgabe die Funktionale `map` und `filter`.

1. Definiert eine Funktion `addFive`, die auf alle Zahlen einer Sequenz 5 addiert. Benutzt hierfür die Funktion `plusFive`.
2. Definiert eine Funktion `toEven`, die alle Zahlen einer Sequenz gerade macht. Überlegt euch ein geeignetes Verfahren.
3. Definiert eine Funktion `splitOddEven`, die alle geraden und ungeraden Zahlen einer Sequenz trennt.
4. Benutzt die Funktionen `splitOddEven` und `myAddCurry`, um in einer Funktion `raiseEvenBy10` alle geraden Zahlen einer Sequenz um 10 erhöht zurückzuliefern.
5. Definiert die Funktion `countEven`, die die geraden Zahlen einer Sequenz zählt.

Lösung:

```
FUN addFive: seq[nat] -> seq[nat]
DEF addFive(s) == map(plusFive)(s)

FUN toEven: seq[nat] -> seq[nat]
DEF toEven(s) ==
  map(\\ x. IF odd?(x) THEN x + 1 ELSE x FI)(s)

FUN splitOddEven: seq[nat] -> seq[nat] ** seq[nat]
DEF splitOddEven(s) ==
  (filter(odd?)(s), filter(even?)(s))

FUN raiseEvenBy10: seq[nat] -> seq[nat]
DEF raiseEvenBy10(s) ==
  LET (_, evens) == splitOddEven(s) IN
    map(myAddCurry(10))(evens)

FUN countEven: seq[nat] -> nat
DEF countEven(s) ==
  #(filter(even?)(s))
```

4.3. Die Funktion `filteredMap` (15 Prozent) Definiert die Funktion `filteredMap`, die der vordefinierten Funktion `map` entspricht. Die übergebene Funktion wird aber nur auf die Listenelemente angewendet, für die ein gegebenes Prädikat `true` liefert.

Benutzt die Funktion `filteredMap`, um eine Funktion `raise` zu definieren, die alle Zahlen einer Sequenz, für die ein gegebenes Prädikat gilt, um einen bestimmten Betrag erhöht. Benutzt dafür `myAddCurry`.

Lösung:

```
FUN filteredMap: (nat -> bool) ** (nat -> nat) -> seq[nat] -> seq[nat]
DEF filteredMap(p, f)(s) ==
  IF <>?(s) THEN <>
```

```
  OTHERWISE
  IF p(ft(s)) THEN f(ft(s)) :: filteredMap(p, f)(rt(s))
              ELSE ft(s) :: filteredMap(p, f)(rt(s))
  FI
```

oder

```
DEF filteredMap(p, f)(s) ==
  map(\\x. IF p(x) THEN f(x) ELSE x FI)(s)
```

und

```
FUN raise: (nat -> bool) ** nat ** seq[nat] -> seq[nat]
DEF raise(p, n, s) == filteredMap(p, myAddCurry(n))(s)
```