

## Übungen zur „Deskriptiven Programmierung“ Blatt 5

Funktionale Sprachen eignen sich in besonderer Weise zur Modellierung von Schaltkreisen. Der einfachste Ansatz modelliert Schaltkreise als Schaltfunktionen.

```
data Bit = 0 | 1 deriving (Show, Eq, Ord)
neg      :: Bit → Bit
neg 0    = 1
neg 1    = 0
infix 3 ^
infix 2 ∨
(∧), (∨) :: Bit → Bit → Bit
0 ∧ b    = 0
1 ∧ b    = b
0 ∨ b    = b
1 ∨ b    = 1
```

Zur Gruppierung von Ein- und Ausgängen werden Tupel bzw. Listen verwendet.

**Aufgabe 12.** Definiere den Ripple-carry Addierer.

```
type Carry      = Bit
rippleCarryAdder :: Carry → [(Bit, Bit)] → (Carry, [Bit])
```

*Hinweis:* verwende—soweit möglich—vordefinierte listenverarbeitende Funktionen wie *map*, *zipWith* etc.

**Aufgabe 13.** Die Funktion *scan* wendet *fold* auf alle Suffixe einer Liste an.

```
scan      :: β → (α → β → β) → [α] → [β]
scan e f  = map (fold e f) · tails
tails    :: [α] → [[α]]
tails []  = [[]]
tails x@(a : x') = x : tails x'
```

Läßt sich *tails* mit Hilfe von *fold* definieren? Leite aus der obigen (ausführbaren) Spezifikation eine effiziente Implementierung her. Definiere *rippleCarryAdder* unter Verwendung von *scan*. *Hinweis:* berechne zunächst alle Überträge, dann die Summen.

**Aufgabe 14.** Die Laufzeit des Ripple-carry Addierers wird dominiert von der seriellen Berechnungen der Überträge. Ein *scan* läßt sich parallelisieren, wenn die Funktion *f* assoziativ ist. Das folgende Gesetz zeigt, wie sich eine beliebige Funktion auf eine assoziative Operation zurückführen läßt.

$$\text{scan } e \ f = \text{map } (\text{post } e) \cdot \text{scan } \text{id } (\cdot) \cdot \text{map } f$$

Zeige das Gesetz und verwende es, um den Ripple-carry Addierer zu verbessern.