Technische Universität München **WS 2006/07**
Institut für Informatik
Prof. Manfred Broy
A. Gruler, B. Spanfelner **Musterlösung zu Blatt 6**

Übungen zu „Formale Methoden“

Aufgabe 1 Funktionen höherer Stufe und Currying [LÖSUNG]

Funktion	Stufe	Beispiel
$f_1 : (Nat \to Nat) \to Nat$	2. Stufe	$f_1(f_2) = f_2(3)$
$f_2 : Nat \to Nat$	1. Stufe	$f_2(x) = x + 5 = x + f_3$
$f_3 : Nat$	0. Stufe (Konstante)	$f_3 = 5$
$f_4 : (Nat \to Nat), Nat \to Nat$	2. Stufe	$f_4(f, n) = f(n)$
$mult : Nat, Nat \to Nat$	1. Stufe	$mult(x, y) = x * y$

Sinnvolle Anwendung der Funktion curry auf die obigen Funktionen:

- $curry(f_4)(f_2) = (m : Nat)\ Nat : f_2(m)$
- $curry(mult)(n) = (m : Nat)\ Nat : n * m$

Aufgabe 2 Rekursive Funktionen [LÖSUNG]

(a) sumseq: Seq Tree Nat → Nat

Sei n die Länge der Sequenz.

$$\mathsf{sumseq}(s) = \sum_{1 \le i \le n} \Big(\sum_{k \in nodes(s[i])} k \Big)$$

wobei *nodes t* der Menge aller Knoten des Baumes t entspricht.

(b) Eine rekursive Rechenvorschrift für sumseq geben wir mit Hilfe der Funktion sum an, wobei folgender Zusammenhang gilt:

$\mathsf{sumseq}\ s = \mathsf{sum}(\mathsf{etree}, s)$

Es sei: sum : Tree Nat, Seq Tree Nat → Nat mit der Rechenvorschrift:

```
fct sum = (t: Tree Nat, s: Seq Tree Nat) Nat
   if t = etree    then if s = ⟨⟩ then 0
                             else sum(first(s), rest(s))
                        fi
                   else root(t) + sum(left(t), s ∘ ⟨right(t)⟩)
   fi
```

(c) Die Lösung dieser Aufgabe geschieht im Detail auf Blatt 7. Hier wurden nur Möglichkeiten diskutiert.

Aufgabe 3 Fehlermodellierung – Geldwechselautomat [LÖSUNG]

Es folgt nun ein Lösungsvorschlag in ML-Ähnlicher Notation

(a) Es wird angenommen, daß die Liste `coins` der zum Wechseln verfügbaren Münzwerte absteigend geordnet ist. Das Ergebnis des Umtauschvorgangs sind Paare: Wenn der Tauschvorgang möglich ist, enthält die erste Komponente das Ergebnis des Tauschvorgangs (= eine Liste von gewechselten Münzen mit den neuen Münzwerten) und die zweite Komponente ‘ok’. Ansonsten enthält die erste Komponente die leere Liste und die zweite ‘impossible’.

```
datatype dt = ok | impossible;



type coinsInt = Int;



change : (coinsInt List * Int) -> (Int List * dt)
```

```
change (coins, 0) = ([],ok)
  | change([],amount) = ([], impossible)
  | change(c::coins, amount) =
    if amount < 0 then ([], impossible)
                else
                let val
                (coinsSoFar, itWorked) = (change(c::coins, amount-c))
                in
                if itWorked = impossible then (change(coins, amount))
                                        else (c::coinsSoFar, ok)
                fi
    fi
```

Wenn wir keinen Typ `coinsInt` einführen wollen, können wir auch die folgende Signatur verwenden:

```
change : (Int List * Int) -> (Int List * dt)
```

(b) Kann ein Geldbetrag nicht gewechselt werden, so wird im Ergebnistupel der Typ *dt* mit "impossible" belegt. Dies entspricht einer expliziten Fehlerbehandlung. Im Gegensatz zur Verwendung des allgemeinen Fehlerelements $\perp$ vermeidet man hierdurch Probleme, die auftreten können, wenn $\perp$ das Nichtterminieren bedeutet. In solch einem Fall wäre ein Ausdruck

```
if itWorked = ⊥
```

nicht auswertbar.

Für den Fall, dass für die zur Verfügung stehenden Wechselbeträge $\langle a_n, \ldots, a_0 \rangle$ gilt

$$a_i > a_{i-1}, 1 \leq i \ \leq \ n \quad \wedge \quad \forall j \in \mathbb{N}, 1 \leq j \leq n: \ \sum_{i=0}^{j-1} a_i \ \leq \ a_j$$

(gilt z.B. für das Münzsystem $< 10,\ 5,\ 2,\ 1 >$) gibt der Algorithmus immer die kürzeste Liste an Wechselgeld. Gilt diese Einschränkung nicht, so müssen alle möglichen Kombinationen probiert werden.