Formale Grundlagen der Informatik 3

Prof. Dr. Christoph Walther / Visar Januzaj, Nathan Wasser
Technische Universität Darmstadt — Wintersemester 2011/12

Lösungsvorschlag zu Übung 1

Version 1 vom 4.11.2011

Aufgabe 1.1 (Arithmetik)

Aus der Vorlesung kennen Sie die folgende Datenstruktur für natürliche Zahlen:

`structure` $\mathbb{N}$ `<=`
0,
$^{+}(^{-} : \mathbb{N})$

Implementieren Sie die folgenden arithmetischen Operationen:

(a) Addition: `function [infixr,10]` $+(a, b : \mathbb{N}) : \mathbb{N}$

Hinweis (vgl. L 1.0 Primer): `[infixr]` bedeutet, dass a `+` b anstelle von $+(a, b)$ geschrieben werden muss und dass a `+` b `+` c als a `+` $(b$ `+` $c)$ interpretiert wird.

Lösungsvorschlag

```
function [infixr,10] +(a, b : ℕ) : ℕ <=
if ?0(a)
  then b
  else +(-(a) + b)
end_if
```

(b) Multiplikation: `function [infixr,20]` $\times(a, b : \mathbb{N}) : \mathbb{N}$

Hinweis (vgl. L 1.0 Primer): Da $\times$ mit Priorität 20 stärker bindet als `+` mit Priorität 10, können Sie $a \times b$ `+` c statt $(a \times b)$ `+` c schreiben.

Lösungsvorschlag

```
function [infixr,20] ×(a, b : ℕ) : ℕ <=
if ?0(a)
  then 0
  else -(a) × b + b
end_if
```

(c) Fakultät: `function [postfix] !`$(a : \mathbb{N}) : \mathbb{N}$

Hinweis (vgl. L 1.0 Primer): `[postfix]` bedeutet, dass $(a)!$ anstelle von $!(a)$ geschrieben werden muss.

Lösungsvorschlag

```
function [postfix] !(a : ℕ) : ℕ <=
if ?0(a)
  then 1
  else a × (⁻(a))!
end_if
```

Aufgabe 1.2 (Listen)

Wir betrachten die aus der Vorlesung bekannte Datenstruktur für Listen:

```
structure list[@I] <=
ø,
[infixr,100] ::(hd : @I, tl : list[@I])
```

Implementieren Sie die folgenden Listen-Operationen:

(a) Das Aneinanderhängen von Listen: `function append`$(l$: `list[@I]`, k : `list[@I]`$)$: `list[@I]`

Lösungsvorschlag

```
function append(l : list[@I], k : list[@I]) : list[@I] <=
if ?ø(l)
  then k
  else hd(l) :: append(tl(l), k)
end_if
```

(b) Die Berechnung der Länge einer Liste: `function length`$(k$: `list[@I]`$)$: $\mathbb{N}$

Lösungsvorschlag

```
function length(k : list[@I]) : ℕ <=
if ?ø(k)
  then 0
  else ⁺(length(tl(k)))
end_if
```

(c) Das Spiegeln einer Liste: `function mirror`$(k$: `list[@I]`$)$: `list[@I]`.
Beispiel: `mirror`$(a :: a :: b :: c :: ø) = a :: a :: b :: c :: c :: b :: a :: a :: ø$.

Lösungsvorschlag

```
function mirror(k : list[@I]) : list[@I] <=
if ?ø(k)
  then ø
  else append(hd(k) :: mirror(tl(k)), hd(k) :: ø)
end_if
```

Aufgabe 1.3 (Terme und Typen)

(a) Geben Sie die Typen der folgenden Terme an.

a) `1::ø`

b) `length(ø)`

c) `(1::ø)::ø`

d) `(1::ø)::ø::ø`

e) `((1::ø)::ø)::ø`

f) `length(1::ø)::ø`

Lösungsvorschlag

a) `list[`$\mathbb{N}$`]`

b) $\mathbb{N}$

c) `list[list[`$\mathbb{N}$`]]`

d) `list[list[`$\mathbb{N}$`]]`

e) `list[list[list[`$\mathbb{N}$`]]]`

f) `list[`$\mathbb{N}$`]`

(b) Geben Sie die Längen der folgenden Listen an.

a) `1::ø`

b) `ø`

c) `ø::ø`

d) `(1::ø)::ø`

e) `(1::ø)::ø::ø`

f) `((1::ø)::ø)::ø`

g) `length(1::ø)::ø`

Lösungsvorschlag

a) 1

b) 0

c) 1

d) 1

e) 2

f) 1

g) 1

Aufgabe 1.4 (Spezifikationen)

Wir wollen nun die Spiegelung einer Liste aus der vorherigen Aufgabe formal spezifizieren. Offenbar ist das Ergebnis der Spiegelung ein Palindrom, d. h. für jede Liste k gilt:

$$\texttt{reverse}(\texttt{mirror}(k)) = \texttt{mirror}(k) \tag{1}$$

Die Prozedur `reverse` dreht dabei die Liste um und kann wie in der ersten praktischen Übung implementiert werden.

Außerdem verdoppelt sich durch die Spiegelung die Länge der Liste, also gilt für jede Liste k:

$$\texttt{length}(\texttt{mirror}(k)) = 2 \times \texttt{length}(k)) \tag{2}$$

Schließlich sollte die ursprüngliche Liste ein Präfix der Ergebnisliste sein. Um dies zu formulieren benutzen wir eine Prozedur `isPrefix`, die testet, ob eine Liste k Präfix einer Liste l ist, d. h. $\texttt{isPrefix}(k, l)$ liefert `true`, falls k ein Präfix von l ist und `false` sonst. Für jede Liste k gilt damit:

$$\texttt{isPrefix}(k, \texttt{mirror}(k)) \tag{3}$$

Untersuchen Sie, welche Kombinationen der Eigenschaften (1), (2) und (3) die Spiegelung einer Liste korrekt spezifizieren. Geben Sie für die anderen Kombinationen jeweils ein Beispiel einer Prozedur, die zwar die Spezifikation erfüllt, aber nicht die Spiegelung einer Liste berechnet.

Lösungsvorschlag

Keine der drei Eigenschaften ist für sich ausreichend:

(1) Definieren wir `function mirror(k : list[@I]) : list[@I] <= ø`, dann gilt (1), obwohl dies offensichtlich keine Spiegelung berechnet.

(2) Die folgende Prozedur erfüllt zwar (2), berechnet aber im Allgemeinen auch keine Spiegelung:

```
function mirror(k : list[@I]) : list[@I] <=
if ?ø(k) then ø
  else hd(k) :: hd(k) :: mirror(tl(k))
end_if
```

(3) Die Eigenschaft (3) wird von der Identitätsfunktion (`function id(x : @I) : @I<= x`) erfüllt, die offensichtlicht nicht die Liste spiegelt.

Als nächstes untersuchen wir Kombinationen von jeweils zwei Eigenschaften:

(1) und (2) Diese Kombination ist nicht ausreichend, denn die folgende Prozedur `mirror` erfüllt beide Eigenschaften, spiegelt aber nicht die Liste

```
function mirror(k : list[@I]) : list[@I] <=
if ?ø(k) then ø
  else do_not_mirror(k, hd(k))
end_if

function do_not_mirror(k : list[@I], i : @I) : list[@I] <=
if ?ø(k) then ø
  else i :: i :: do_not_mirror(tl(k), i)
end_if
```

(2) und (3) Auch diese Kombination reicht nicht aus, denn die folgende Definition von `mirror` erfüllt beide Eigenschaften, liefert aber nicht die gewünschte Spiegelung:

```
function mirror(k : list[@I]) : list[@I] <= append(k, k)
```

(1) und (3) Die folgende Prozedur `mirror` erfüllt beide Eigenschaften, spiegelt aber nicht die Liste, also reichen auch diese beiden Eigenschaften nicht aus.

```
function mirror(k : list[@I]) : list[@I] <=
if ?ø(k)
  then ø
  else if ?ø(tl(k))
         then hd(k) :: hd(k) :: hd(k) :: ø
         else append(hd(k) :: mirror(tl(k)), hd(k) :: ø)
       end_if
end_if
```

Anscheinend ist die Kombination aller drei Eigenschaften hinreichend, um die Spiegelung einer Liste korrekt zu spezifizieren. Für diese Spezifikation brauchen wir also die Prozeduren `reverse`, `isPrefix`, `length` und $\times$, so dass es fast einfacher erscheint, folgender Implementierung zu vertrauen:

```
function mirror(k : list[@I]) : list[@I] <= append(k, reverse(k))
```