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

Aufgabenblatt 6

letzte Aktualisierung: 12. December, 1322

Ausgabe: *30.11.2001*
Abgabe: *9.12. / 10.12.2001* *Prozent:* *100*

Thema: *Funktionen höherer Ordnung; algebraische Datentypen*

1. Aufgabe (30 Prozent): Funktionale auf Listen

1.1. Die Listenfunktionale `reduce` und `zip` (Tut) Diskutiert die Listenfunktionale `reduce` und `zip`. Implementiert die Funktionen `reduceSum` und `skalarProd` (für natürliche Zahlen) aus Aufgabenblatt 3 mit Hilfe von `reduce` und `zip`.

Lösung:

Die Funktion `reduce` *verknüpft die Elemente einer Sequenz sowie einen zusätzlichen Wert miteinander. Der zusätzliche Wert ist notwendig, um bei einer leeren Sequenz ein Ergebnis liefern zu können; er stellt üblicherweise das neutrale Element bzgl. der Verknüpfung dar. Die folgende Gleichung verdeutlicht die Funktionsweise von* `reduce`*:*

$$\texttt{reduce}(\oplus, e)(\langle x_1, x_2, \ldots, x_n\rangle) = x_1 \oplus (x_2 \oplus (\cdots \oplus (x_n \oplus e)\ldots))$$

Die Funktion `zip` *verknüpft zwei Sequenzen elementweise miteinander. Die Sequenzen müssen dabei gleichlang sein.*

```
FUN reduceSum : seq[nat] -> nat
DEF reduceSum == \\s. reduce(+, 0)(s)
-- oder
DEF reduceSum == reduce(+, 0)

FUN skalarProd : seq[nat] ** seq[nat] -> nat
DEF skalarProd == \\s1, s2. reduce(+, 0) (zip(*)(s1,s2))

FUN myReduce : (nat ** nat -> nat) ** nat -> seq[nat] -> nat
DEF myReduce == \\f, e. \\s. IF s <>? THEN e
                                     ELSE f(ft(s), myReduce(f,e)(rt(s)))
                             FI

FUN myZip : (nat ** nat -> nat) -> seq[nat] ** seq[nat] -> seq[nat]
DEF myZip == \\f. \\s1, s2. IF s1 <>? and s2 <>?
                              THEN <>
                              ELSE f(ft(s1), ft(s2))
                                     :: myZip(f)(rt(s1),rt(s2))
                            FI
```


1.2. Linksreduktion (20 Prozent) Das Funktional `reduce` verknüpft die Listenelemente von rechts nach links. Implementiert das Funktional `fold`, das die Elemente von links nach rechts verknüpft, also

$$\texttt{fold}(\oplus, e)(\langle x_1, x_2, \ldots, x_n\rangle) = (\ldots((e \oplus x_1) \oplus x_2) \oplus \ldots) \oplus x_n$$

Implementiert mit Hilfe dieses Funktionals die Funktion `foldDiff` aus Aufgabenblatt 3.

Lösung:

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

FUN foldDiff : seq[nat] -> nat
DEF foldDiff == \\s . IF s <>? THEN 0
                                 ELSE fold(-,ft(s))(rt(s))
                      FI
```

1.3. Verknüpfung mehrerer Sequenzen (10 Prozent) Implementiert das Funktional `zip3`, das analog zu `zip` Elemente aus drei Sequenzen durch eine gegebene Funktion verknüpft.

Lösung:

```
FUN zip3 : (nat ** nat ** nat -> nat) -> seq[nat] ** seq[nat] ** seq[nat] -> seq[nat]
DEF zip3(f)(<>, <>, <>) == <>
DEF zip3(f)(x :: xs, y :: ys, z :: zs) == f(x,y,z) :: zip3(f)(xs, ys, zs)
```

2. Aufgabe (20 Prozent): Algebraische Datentypen

2.1. Aufbau von Sequenzen (Tut) Diskutiert den Aufbau der Typs `seq`. Welche Signatur induziert der Typ? Welche Konstruktoren, Selektoren und Diskriminatoren gibt es?

Lösung:

Der Sequenzentyp `seq` *ist ein generischer, rekursiver Datentyp. Er ist wie folgt definiert:*

```
TYPE seq == <>
            ::(ft : data, rt : seq)
```

`data` *ist hier der Typparameter. Diese Deklaration induziert die folgende Signatur (viele dieser Funktionen sind bereits aus früheren Aufgabenblättern bekannt):*

```
SORT seq

-- Konstruktoren

FUN <> : seq
FUN :: : data ** seq -> seq

-- Diskriminatoren

FUN <>? : seq -> bool
```

```
FUN ::? : seq -> bool

-- Selektoren

FUN ft : seq -> data
FUN rt : seq -> seq
```

2.2. Der Typ `date` (Tut) Welche Signatur induziert die folgende Datentypdeklaration?

```
TYPE date == date( day   : nat,
                   month : nat,
                   year  : nat)
```

Handelt es sich hierbei um einen Produkt- oder Summentyp?

Lösung:

```
SORT date

-- Konstruktoren

FUN date : nat ** nat ** nat -> date

-- Diskriminatoren

FUN date? : date -> bool

-- Selektoren

FUN day   : date -> nat
FUN month : date -> nat
FUN year  : date -> nat
```

Hierbei handelt es sich um einen Produkttyp.

2.3. Der Typ `person` (10 Prozent) Gegeben ist der folgende Datentyp:

```
TYPE person == person(name      : denotation,
                     surname   : denotation,
                     birthdate : date)
```

Welche Signatur induziert er?

Lösung:

```
SORT person

-- Konstruktoren

FUN person : denotation ** denotation ** date -> person

-- Diskriminatoren

FUN person? : person -> bool
```



```
-- Selektoren

FUN name      : person -> denotation
FUN surname   : person -> denotation
FUN birthdate : person -> date
```

2.4. Arbeiten mit algebraischen Typen (10 Prozent) Implementiert die folgenden Funktionen:

- `earlier?`, die überprüft, ob ein Wert von Typ `date` ein früheres Datum repräsentiert als ein anderer,
- `older?`, die überprüft, ob eine Person älter als eine andere ist,
- `birthday`, die den Geburtstag (also Tag und Monat) einer Person liefert,
- `sameName?`, die überprüft, ob zwei Personen den gleichen Nachnamen haben.

Lösung:

```
FUN earlier? : date ** date -> bool
DEF earlier? == \\x,y . IF year(x) = year(y)
                          THEN
                            IF month(x) = month(y) THEN day(x) < day(y)
                                                   ELSE month(x) < month(y)
                            FI
                          ELSE
                            year(x) < year(y)
                        FI

FUN older? : person ** person -> bool
DEF older? == \\x,y . earlier?(birthdate(x), birthdate(y))

FUN birthday : person -> nat ** nat
DEF birthday == \\x . LET birth == birthdate(x)
                      IN (day(birth), month(birth))

FUN sameName? : person ** person -> bool
DEF sameName? == \\x,y . surname(x) = surname(y)
```

3. Aufgabe (50 Prozent): Benutzerdefinierte Funktionale

Gegeben ist der folgende Datentyp:

```
TYPE safeNat == ok (value : nat)
                error (message : denotation)
```

3.1. Induzierte Signatur (10 Prozent) Welche Signatur induziert der Typ? Handelt es sich hierbei um einen Produkt- oder Summentyp?

Lösung:

```
SORT safeNat

-- Konstruktoren
```

```
FUN ok    : nat -> safeNat
FUN error : denotation -> safeNat

-- Diskriminatoren

DEF ok?    : safeNat -> bool
DEF error? : safeNat -> bool

-- Selektoren

DEF value   : safeNat -> nat
DEF message : safeNat -> denotation
```

Hier handelt es sich um einen Summentyp.

3.2. Verwendung von `safeNat` (Tut) Wozu kann dieser Typ verwendet werden? Implementiert die Funktion `safePred`, die den Vorgänger eines Werts vom Typ `safeNat` oder eine Fehlermeldung liefert.

Lösung:

Dieser Typ kann verwendet werden, um partielle *Funktionen, die natürliche Zahlen liefern, in totale Funktionen umzuwandeln. Wird eine solche Funktion mit Argumenten aufgerufen, die außerhalb ihres Definitionsbereichs liegen, so kann sie nun eine Fehlermeldung liefern.*

Beispiel:

```
FUN safePred : safeNat -> safeNat
DEF safePred(ok(n))      == IF n |= 0 THEN ok(pred(n))
                                      ELSE error("Vorgaenger von 0")
                            FI
DEF safePred(error(msg)) == error(msg)
```

Hier wird für den Fall, daß der Vorgänger undefiniert ist, eine Fehlermeldung geliefert. Ist während der Berechnung des Arguments bereits ein Fehler aufgetreten, so wird dieser einfach weitergereicht. Die Definition verwendet Patterns, da sie in diesem Fall die Lesbarkeit stark verbessern. In der Lambda-Notation würde diese Definition wie folgt aussehen:

```
DEF safePred == \\x . IF ok?(x) THEN LET n == value(x)
                                     IN IF n |= o
                                          THEN ok(pred(n))
                                          ELSE error("Vorgaenger von 0")
                      FI
                                  ELSE error(message(x))
                FI
```

3.3. Totalisierung einstelliger Funktionen (Tut) Implementiert die Funktion `safeLift`, die eine einstellige Funktion auf natürlichen Zahlen in eine Funktion auf `safeNat` umwandelt, wobei der Definitionsbereich der Funktion durch ein Prädikat angegeben werden kann. Implementiert die Funktion `safePred` aus der vorherigen Unteraufgabe sowie die Funktion `safeSucc` mit Hilfe des neuen Funktionals.

Lösung:

```
FUN safeLift : (nat -> nat) ** (nat -> bool) ** denotation
               -> safeNat -> safeNat
DEF safeLift(f, p, msg) (ok(x))    == IF p(x) THEN ok(f(x))
                                      ELSE error(msg)
                                   FI
DEF safeLift(_, _, _) (error(msg)) == error(msg)

DEF safePred == safeLift(pred, \\ x . x |= 0, "Vorgaenger von 0")
DEF safeSucc == safeLift(succ, \\ _ . true, "")
```

Die Implementierung von `safeLift` *ist eine Verallgemeinerung der Implementierung von* `safePred` *aus der vorherigen Aufgabe. Das Prädikat* `p` *überprüft dabei, ob das Argument innerhalb des Definitionsbereichs der Funktion liegt. Da* `succ` *eine totale Funktion ist, ist das Prädikat in der Definition von* `safeSucc` *immer erfüllt, und es wird hier keine Fehlermeldung benötigt.*

3.4. Totalisierung zweistelliger Funktionen (20 Prozent) Implementiert die Funktion `safeLift2`, die eine zweistellige Funktion auf natürlichen Zahlen in eine Funktion auf `safeNat` umwandelt, wobei der Definitionsbereich der Funktion durch ein zweistelliges Prädikat angegeben werden kann.

Lösung:

```
FUN safeLift2 : (nat ** nat -> nat) ** (nat ** nat -> bool) ** denotation
                -> safeNat ** safeNat -> safeNat
DEF safeLift2(f, p, msg)
             (ok(x), ok(y))             == IF p(x,y) THEN ok(f(x,y))
                                                  ELSE error(msg)
                                          FI
DEF safeLift2(_, _, _)
             (error(msg1), error(msg2)) == error(msg1 ++ " ; " ++ msg2)
DEF safeLift2(_, _, _)
             (error(msg), ok(_))        == error(msg)
DEF safeLift2(_, _, _)
             (ok(_), error(msg))        == error(msg)
```

3.5. Sichere Arithmetik (10 Prozent) Implementiert die Funktionen `safeAdd`, `safeSub`, `safeMult`, `safeDiv`, die die Addition, Subtraktion, Multiplikation und ganzzahlige Division für Werte vom Typ `safeNat` als totale Funktionen implementieren.

Hinweis: Verwendet wenn möglich die bereits definierten Funktionale.

Lösung:

```
FUN safeAdd safeSub safeMult safeDiv: safeNat ** safeNat -> safeNat
DEF safeAdd  == safeLift2(+, \\x, y . true, "")
DEF safeSub  == safeLift2(-, \\x, y . x >= y, "Negative Differenz")
DEF safeMult == safeLift2(*, \\x, y . true, "")
DEF safeDiv  == safeLift2(/, \\x, y . y |= 0, "Division durch 0")
```

3.6. `safeNat` in Sequenzen (10 Prozent) Implementiert das Funktional `safeMap`, das analog zu `map` eine Funktion auf alle natürlichen Zahlen in einer Sequenz anwendet und eine Sequenz von Werten vom Typ `safeNat` liefert. Der Definitionsbereich der Funktion kann durch ein Prädikat angegeben werden.

Hinweis: Diese Aufgabe kann bei Verwendung bereits definierter Funktionen durch einen Einzeiler gelöst werden.

Lösung:

```
FUN safeMap : (nat -> nat) ** (nat -> bool) ** denotation -> seq[nat]
                -> seq[safeNat]
DEF safeMap == \\f, p, m . \\s . map(safeLift(f, p, m))(map(ok)(s))
```