Friedrich-Alexander-Universität
Technische Fakultät

FAU

Theorie der Programmierung

Übung 05 — Induktive Datentypen

Leon Vatthauer

30. Mai 2025

Aufgabe 1

Listen und Bäume

Wir betrachten die folgenden algebraischen Definitionen parametrischer Datentypen von Listen und Binärbäumen über einem Typparameter *a*:

```
data List a where
    Nil  : () → List a
    Cons : a → List a → List a
```

```
data Tree a where
    Leaf  : a → Tree a
    Inner : a → Tree a → Tree a → Tree a
```

Einige Annahmen

Wir nehmen einen Typ **Nat** von Konstanten $0, 1, 2, \ldots$ und die üblichen Grundoperationen $+, -, \times$, **max**, **min**, ==, ... für **Nat** als gegeben an. Weiter nehmen wir den Typ **Bool** mit den Konstanten *True* und *False* sowie den Grundoperationen &&, ||, **not** etc. und einem **if_then_else_** Konstrukt als gegeben an. Mit `()` bezeichnen wir den Typ *unit*; er enthält nur einen einzigen Wert, den wir ebenfalls mit `()` bezeichnen.

Aufgabe 1.1

Listen und Bäume

Wir betrachten die folgenden algebraischen Definitionen parametrischer Datentypen von Listen und Binärbäumen über einem Typparameter *a*:

```
data List a where
   Nil  : () → List a
   Cons : a → List a → List a
```

```
data Tree a where
   Leaf  : a → Tree a
   Inner : a → Tree a → Tree a → Tree a
```

Beschreiben Sie in eigenen Worten die durch die folgenden Terme gegebenen Listen und Binärbäume mit natürlichen Zahlen bzw. zeichnen Sie diese.

- Nil
- Cons 5 Nil
- Leaf 13
- Inner 5 (Leaf 3) (Leaf 9)
- Cons 5 (Cons 5 Nil)
- Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
- Inner 8 (Inner 4 (Leaf 1) (Leaf 20)) (Leaf 8)
- Inner 6 (Leaf 99) (Inner 1 (Leaf 4) (Leaf 6))

Aufgabe 1.2

Listen und Bäume

Wir betrachten die folgenden algebraischen Definitionen parametrischer Datentypen von Listen und Binärbäumen über einem Typparameter *a*:

```
data List a where
    Nil  : () → List a
    Cons : a → List a → List a
```

```
data Tree a where
    Leaf  : a → Tree a
    Inner : a → Tree a → Tree a → Tree a
```

Es können nun Funktionen induktiv über der Struktur von **List** *a* und **Tree** *a* definiert werden, beispielsweise:

```
length Nil         = 0
length (Cons x xs) = 1 + length xs
```

```
size (Leaf x)      = 1
size (Inner x l r) = 1 + size l + size r
```

Aufgabe 1.2

Listen und Bäume

Wir betrachten die folgenden algebraischen Definitionen parametrischer Datentypen von Listen und Binärbäumen über einem Typparameter *a*:

```
data List a where
    Nil  : () → List a
    Cons : a → List a → List a
```

```
data Tree a where
    Leaf  : a → Tree a
    Inner : a → Tree a → Tree a → Tree a
```

Es können nun Funktionen induktiv über der Struktur von **List** *a* und **Tree** *a* definiert werden, beispielsweise:

```
length Nil         = 0
length (Cons x xs) = 1 + length xs
```

```
size (Leaf x)      = 1
size (Inner x l r) = 1 + size l + size r
```

```
app Nil ys = ys
app (Cons x xs) ys = Cons x (app xs ys)
```

Aufgabe 1.2

Listen und Bäume

Wir betrachten die folgenden algebraischen Definitionen parametrischer Datentypen von Listen und Binärbäumen über einem Typparameter *a*:

```
data List a where
    Nil  : () → List a
    Cons : a → List a → List a
```

```
data Tree a where
    Leaf  : a → Tree a
    Inner : a → Tree a → Tree a → Tree a
```

Es können nun Funktionen induktiv über der Struktur von **List** *a* und **Tree** *a* definiert werden, beispielsweise:

```
length Nil         = 0
length (Cons x xs) = 1 + length xs
```

```
size (Leaf x)      = 1
size (Inner x l r) = 1 + size l + size r
```

```
app Nil ys = ys
app (Cons x xs) ys = Cons x (app xs ys)
```

(a) Welchen Typ hat `length`? Welchen hat `app`? Welchen hat `size`?

Aufgabe 1.2

Listen und Bäume

Wir betrachten die folgenden algebraischen Definitionen parametrischer Datentypen von Listen und Binärbäumen über einem Typparameter *a*:

```
data List a where
    Nil  : () → List a
    Cons : a → List a → List a
```

```
data Tree a where
    Leaf  : a → Tree a
    Inner : a → Tree a → Tree a → Tree a
```

Es können nun Funktionen induktiv über der Struktur von **List** *a* und **Tree** *a* definiert werden, beispielsweise:

```
length Nil         = 0
length (Cons x xs) = 1 + length xs
```

```
size (Leaf x)      = 1
size (Inner x l r) = 1 + size l + size r
```

```
app Nil ys = ys
app (Cons x xs) ys = Cons x (app xs ys)
```

(b) Werten Sie den Term `length (Cons 4 (Cons 89 (Cons 21 Nil)))` aus.

Aufgabe 1.3

Listen und Bäume

Wir betrachten die folgenden algebraischen Definitionen parametrischer Datentypen von Listen und Binärbäumen über einem Typparameter *a*:

```
data List a where
    Nil  : () → List a
    Cons : a → List a → List a
```

```
data Tree a where
    Leaf  : a → Tree a
    Inner : a → Tree a → Tree a → Tree a
```

Es können nun Funktionen induktiv über der Struktur von **List** *a* und **Tree** *a* definiert werden, beispielsweise:

```
length Nil         = 0
length (Cons x xs) = 1 + length xs
```

```
size (Leaf x)      = 1
size (Inner x l r) = 1 + size l + size r
```

```
app Nil ys = ys
app (Cons x xs) ys = Cons x (app xs ys)
```

- Schreiben Sie eine Funktion `element` : **Nat** → **List Nat** → **Bool**, so dass `element a xs = True` wenn `a` in `xs` vorkommt, und andernfalls `element a xs = False`.

Aufgabe 2.1

Fold-Funktionen

Jeder induktive Datentyp besitzt eine Fold-Funktion, die sich aus der initialen Algebrastruktur des Typs ergibt. Der Typ und die Definitionen dieser Fold-Funktionen ergeben sich dabei allein aus den Typen der Konstruktoren. Beispielsweise ist die Fold-Funktion für den Datentyp **Nat** *a* aus der vorangegangenen Übung wie folgt definiert:

```
foldL : c → (a → c → c) → List a → c
foldL n f Nil = n
foldL n f (Cons x xs) = f x (foldL n f xs)
```

Hierbei entspricht der Typparameter *c* einem Ergebnistyp und die beiden Argumente *n* und *f* den Konstruktoren *Nil* und *Cons*, wobei die Typen der Argumente jeweils Operationen auf dem Ergebnistyp *c* beschreiben, mit dem eine Liste **List** *a* in einen einzelnen Wert vom Typ *c* „zusammengefaltet“ werden kann.

- Werten Sie den Term `foldL n f (Cons 2 (Cons 3 (Cons 6 Nil)))` so weit es geht aus.

Aufgabe 2.2

Fold-Funktionen

Jeder induktive Datentyp besitzt eine Fold-Funktion, die sich aus der initialen Algebrastruktur des Typs ergibt. Der Typ und die Definitionen dieser Fold-Funktionen ergeben sich dabei allein aus den Typen der Konstruktoren. Beispielsweise ist die Fold-Funktion für den Datentyp **Nat** *a* aus der vorangegangenen Übung wie folgt definiert:

```
foldL : c → (a → c → c) → List a → c
foldL n f Nil = n
foldL n f (Cons x xs) = f x (foldL n f xs)
```

Hierbei entspricht der Typparameter *c* einem Ergebnistyp und die beiden Argumente *n* und *f* den Konstruktoren *Nil* und *Cons*, wobei die Typen der Argumente jeweils Operationen auf dem Ergebnistyp *c* beschreiben, mit dem eine Liste **List** *a* in einen einzelnen Wert vom Typ *c* „zusammengefaltet“ werden kann.

- Finden Sie den Typ und die Definition der entsprechenden Fold-Funktion `foldT` des parametrischen Datentyps **Tree** *a*.

Aufgabe 2.3

Fold-Funktionen

Jeder induktive Datentyp besitzt eine Fold-Funktion, die sich aus der initialen Algebrastruktur des Typs ergibt. Der Typ und die Definitionen dieser Fold-Funktionen ergeben sich dabei allein aus den Typen der Konstruktoren. Beispielsweise ist die Fold-Funktion für den Datentyp **Nat** *a* aus der vorangegangenen Übung wie folgt definiert:

```
foldL : c → (a → c → c) → List a → c
foldL n f Nil = n
foldL n f (Cons x xs) = f x (foldL n f xs)
```

Hierbei entspricht der Typparameter *c* einem Ergebnistyp und die beiden Argumente *n* und *f* den Konstruktoren *Nil* und *Cons*, wobei die Typen der Argumente jeweils Operationen auf dem Ergebnistyp *c* beschreiben, mit dem eine Liste **List** *a* in einen einzelnen Wert vom Typ *c* „zusammengefaltet“ werden kann.

- Primitiv rekursive Funktionen auf **List** *a* können durch geeignete Instantiierung des Typparameters *c* und der Argumente *n* und *f* alternativ mittels `foldL` ausgedrückt werden. Drücken Sie die Funktionen `length` und `size` jeweils als Folds über Listen und Bäumen aus.