HASKELL

KAPITEL 8

Bäume

Baum

rekursiv definierte Datenstruktur

nicht linear

vielerlei Varianten:

Struktur der Verzweigung,

Ort der gespeicherten Information

(Knoten, Kanten, Blätter)

Binärbaum

Jeder Knoten hat keinen oder zwei Nachfolger

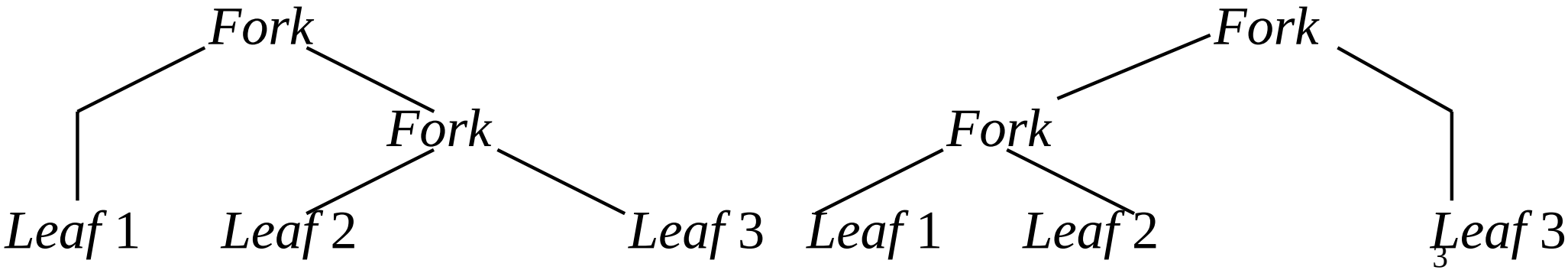
Ort der gespeicherten Information: Blätter

data *Btree* α = *Leaf* α | *Fork* (*Btree* α) (*Btree* α)

Beispiele:

Fork (*Leaf* 1) (*Fork* (*Leaf* 2) (*Leaf* 3))

Fork (*Fork* (*Leaf* 1) (*Leaf* 2)) (*Leaf* 3)



Binärbaum

Jeder Knoten hat keinen oder zwei Nachfolger

Ort der gespeicherten Information: Blätter

data *Btree* α = *Leaf* α | *Fork* *(Btree* α *)* *(Btree* α *)*

Beispiele:

Fork (*Leaf* 1) (*Fork* (*Leaf* 2) (*Leaf* 3))

Fork (*Fork* (*Leaf* 1) (*Leaf* 2)) (*Leaf* 3)

Darstellung der Beispiele als Klammerstruktur:

1 (2 3)

(1 2) 3

Induktion auf Binärbäumen

Aufgabe:

Zeige, dass eine Eigenschaft P für jeden Binärbaum xt gilt.

Verfahren:

1. Zeige $P(xt)$ für den Fall, dass xt ein Blatt ist.
2. Zeige $P(\mathit{Fork}\ (yt)\ (zt))$ unter der Annahme, dass yt und zt Bäume sind, und dass $P(yt)$ und $P(zt)$ gelten.

3. Falls xt unendlich wachsen kann: Zeige $P(\bot)$.

Blattzahl und Blattliste

$$size :: Btree\ \alpha \rightarrow Int$$

$$size\ (Leaf\ x) = 1$$

$$size\ (Fork\ xt\ yt) = size\ xt + size\ yt$$

$$flatten :: Btree\ \alpha \rightarrow [\alpha]$$

$$flatten\ (Leaf\ x) = [x]$$

$$flatten\ (Fork\ xt\ yt) = flatten\ xt\ ++\ flatten\ yt$$

Zahl innerer Knoten und Höhe

$$
\begin{aligned}
&\mathit{nodes} && :: \ \mathit{Btree}\ \alpha \rightarrow \mathit{Int} \\
&\mathit{nodes}\ (\mathit{Leaf}\ x) && = \ \ 0 \\
&\mathit{nodes}\ (\ \mathit{Fork}\ xt\ yt) && = \ \ 1 + \mathit{nodes}\ xt + \mathit{nodese}\ yt
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{height} && :: \ \mathit{Btree}\ \alpha \rightarrow \mathit{Int} \\
&\mathit{height}\ (\mathit{Leaf}\ x) && = \ \ 0 \\
&\mathit{height}(\ \mathit{Fork}\ xt\ yt) && = \ \ 1 + (\mathit{height}\ xt\ \ \mathbf{max}\ \ \mathit{height}\ \ yt)
\end{aligned}
$$

Haskell Schlüsselwort:
Maximum in Infix-Notation

Tiefe der Blätter

ersetze jeden Knoten durch die Länge seines Weges
zur Wurzel

$$depth :: Btree\ \alpha \rightarrow Btree\ Int$$
$$depth = down\ 0$$

$$down :: Int \rightarrow Btree\ \alpha \rightarrow Btree\ Int$$
$$down\ n\ (Leaf\ x) = Leaf\ n$$
$$down\ n\ (Fork\ xt\ yt) = Fork\ (down\ (n+1)\ xt)\ (down\ (n+1)\ yt)$$

„größtes“ Blatt

$$maxBtree :: (Ord\ \alpha) \Rightarrow Btree\ \alpha \rightarrow \alpha$$

$$maxBtree(Leaf\ x) = x$$

$$maxBtree(\ Fork\ xt\ yt\ ) = (\ maxBtree\ xt\ )\ \mathbf{max}\ (\ maxBtree\ yt\ )$$

mapBtree

ersetze jedes Blatt a durch $f\,a$.

$$mapBtree :: (\alpha \to \beta) \to Btree\ \alpha \to Btree\beta$$

$$mapBtree\ f\ (Leaf\ x) = Leaf\ (f\ x)$$

$$mapBtree\ f\ (Fork\ xt\ yt) = Fork\ (mapBtree\ f\ xt)\ (mapBtree\ f\ yt)$$

foldBtree

eine vielseitig nutzbare Funktion.

Ergänzt *mapBtree* (für die Transformation von Blättern) um eine Transformation für Teilbäume.

$$foldBtree :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to Btree\ \alpha \to \beta$$

$$foldBtree\ f\ g\ (Leaf\ x) = f\ x$$

$$foldBtree\ f\ g\ (Fork\ xt\ yt) = g\ (foldBtree\ f\ g\ xt)\ (foldBtree\ f\ g\ yt)$$

size mit *foldBtree*

$$foldBtree :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to Btree\ \alpha \to \beta$$

$$foldBtree\ f\ g\ (Leaf\ x) = f\ x$$

$$foldBtree\ f\ g\ (Fork\ xt\ yt) = g\ (foldBtree\ f\ g\ xt)\ (foldBtree\ f\ g\ yt)$$

$$size :: Btree\ \alpha \to Int$$

$$size\ (Leaf\ x) = 1$$

$$size\ (Fork\ xt\ yt) = size\ xt + size\ yt$$

$$size = foldBtreee\ (const\ 1)\ (+)$$

$$\text{mit } const :: \alpha \to \beta \to \alpha$$

$$const\ x\ y = x$$

height mit *foldBtree*

$$foldBtree :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to Btree\ \alpha \to \beta$$

$$foldBtree\ f\ g\ (Leaf\ x) = f\ x$$

$$foldBtree\ f\ g\ (Fork\ xt\ yt) = g\ (foldBtree\ f\ g\ xt)\ (foldBtree\ f\ g\ yt)$$

$$height :: Btree\ \alpha \to Int$$

$$height\ (Leaf\ x) = 0$$

$$height\ (Fork\ xt\ yt) = 1 + (height\ xt\ \mathbf{max}\ height\ yt)$$

$$height = foldBtreee\ (const\ 0)\ (\oplus)$$

$$\mathbf{where}\ m \oplus n = 1 + (m\ \mathbf{max}\ n)$$

flatten mit *foldBtree*

$$foldBtree :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to Btree\ \alpha \to \beta$$

$$foldBtree\ f\ g\ (Leaf\ x) = f\ x$$

$$foldBtree\ f\ g\ (Fork\ xt\ yt) = g\ (foldBtree\ f\ g\ xt)\ (foldBtree\ f\ g\ yt)$$

$$flatten :: Btree\ \alpha \to [\alpha]$$

$$flatten\ (Leaf\ x) = [x]$$

$$flatten\ (Fork\ xt\ yt) = flatten\ xt\ ++\ flatten\ yt$$

$$flatten = foldBtreee\ wrap\ (++)$$

mit $wrap :: \alpha \to [\alpha]$

$$wrap\ x = [x]$$

mapBtree mit *foldBtree*

$$foldBtree :: (\alpha \to \beta) \to (\beta \to \beta \to \beta) \to Btree\ \alpha \to \beta$$
$$foldBtree\ f\ g\ (Leaf\ x) = f\ x$$
$$foldBtree\ f\ g\ (Fork\ xt\ yt) = g\ (foldBtree\ f\ g\ xt)\ (foldBtree\ f\ g\ yt)$$

$$mapBtree :: (\alpha \to \beta) \to Btree\ \alpha \to Btree\ \beta$$
$$mapBtree\ f\ (Leaf\ x) = Leaf\ (f\ x)$$
$$mapBtree\ f\ (Fork\ xt\ yt) = Fork\ (mapBtree\ f\ xt)\ (mapBtree\ f\ yt)$$

$$Leaf :: \alpha \to Btree\ \alpha$$
$$Fork :: Btree\ \alpha \to Btree\ \alpha \to Btree\ \alpha$$
$$mapBtree\ f = foldBtreee\ (Leaf \bullet f)\ Fork$$

ergänzte Binärbäume

Jeder knoten trägt Information. Beispiel: Knotenzahl seines Teilbaumes:

data *Atree* α = *Leaf* α | *Fork In (Btree* α *) (Btree* α *)*

etc.

geht alles mit foldBtree

HASKELL

KAPITEL 9

Module

Modularisierung

Modul = Eine Struktureinheit im Programm, kapselt eine Menge von Funktionsdefinitionen, Typen, etc.

Modulersteller

- möchte nur einige der Funktionen für Nutzer freigegeben

-Implementation änderbar

Modulbenutzer

- braucht oft nur einige Funktionen
-mag evt. andere Namen

-will vielleicht importierte Funktionen weitergeben

Module dienen der Strukturierung von Programmen; sie haben eine klare Schnittstelle (Interface), in der festgelegt wird, was sie **exportieren** und was sie **importieren.**

Sie bieten:

- unabhängige Programmierung von Programmteilen
-
- separate Compilation von Programmteilen
-
- Bibliotheken von Komponenten können wieder verwendet werden

Beispiele:

```
module mymodule1 where
f = ...
g = ...
```

Default: alles wird exportiert

```
module mymodule2 where
import mymodule1
h = ...
```

Importiert f und g

```
module mymodule3 (f , ...) where
f = ...
g = ...
```

Exportiert nur f, ...

```
module mymodule4 (f , module bla) where
import bla
f = ...
g = ...
```

exportiert f und re-exportiert alles aus bla

import mymodule1 (f,...)

Importiert nur f,...

import mymodule1 **hiding** (f,...)

Importiert alles ausser f,...

import mymodule1 **renaming** (f to fff, g to ggg)

Import mit Umbenennung

HASKELL

KAPITEL 10

Zusammenfassung

Haskell hat ein Universum U von Werten.

U ist in Teilmengen partitioniert.

Jede solche Teilmenge von U ist ein *Typ*.

Vordefinierte (einfache) Typen: *Integer*, *Float*, *Bool*, *Char*

(natürliche Zahlen, Gleitkomma-Zahlen, die logischen Werte, die Symbole)

Zusammengesetzte Typen:

Funktionen, Listen, Bäume, etc. über gegebenen Typen

strenge Typisierung

Jeder Term gehört zu genau einem Typ.

Konsequenz:

quad :: Integer → Integer

quad x = square square x

erzeugt einen Typ-Fehler,

weil *square* für *Float definiert ist.*

Vorteil strenger Typisierung:

prima Erkennung von Fehlern.

Zwingt den Programmierer zu klarem Denken.

Polymorphe Typen

Beispiel:

$square :: Integer \rightarrow Integer$

$sqrt :: Integer \rightarrow Float$

Daraus ableitbar:

$square \bullet square :: Integer \rightarrow Integer$

$sqrt \bullet square :: Integer \rightarrow Float$

Problem: braucht 2 verschiedene Def. von • :

$(\bullet) :: (Int \rightarrow Int) \rightarrow (Int \rightarrow Int) \rightarrow (Int \rightarrow Int)$

$(\bullet) :: (Int \rightarrow Float) \rightarrow (Int \rightarrow Int) \rightarrow (Int \rightarrow Float)$

Lösung: Typ-Variablen α, β, γ . Damit:

$(\bullet) :: (\beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$

Verwendung von Typ-Variablen

Beispiel: Die Funktion *error* soll ein (Text-)Argument auf unspezifizierte Weise weiterverarbeiten.

Nicht einmal der Typ des Resultats ist bekannt. Mit Typ-Variable sei

$error :: String \rightarrow \alpha$

Daraus konstruierbar:

$fact :: Integer \rightarrow Integer \rightarrow Integer$

```
fact n
        |x < 0  =  error „negative argument to fact“
        |x == 0  =  1
        |x > 0  =  n * fact (n – 1)
```

Grundlegende Fragen

Wie bezieht sich *Haskell* (und andere Programmiersprachen) auf die Realität?

Wo zweigt die Informatik von der Mathematik ab?

Was bedeuten rechentechnische Schranken für die Semantik der Sprachen?

Was ist die Rolle von „?“ ?

Hintergrund: Realität und Mathematik

Menge (nach Cantor), „Eine Zusammenfassung von Gegenständen unserer Anschauung oder unseres Denkens“

Struktur (nach Tsarski):

eine Menge („Träger“) der Struktur,

zusammen mit endlich vielen Konstante und endlich vielen Operationen.

Basis der Prädikatenlogik.

Programmieren: Strukturen konstruieren

.... so gut wie möglich

Haskell: Werte und *Typen*

Universum (*universe*)
die Menge der uns interessierenden Dinge.

Wert (*value*)
Elemente des Universums.
Für *Haskell* das einzig wichtige an Werten:
zwei Werte sind gleich oder verschieden

Typ (*Type*)
Struktur mit einer Teilmenge des Universums als Träger

Haskell: Umgang mit Typen

Einige Typen gibt *Haskell* vor.

Alle Typen sind implementierbar,

- einige um den Preis von Speicherüberlauf (*Integer*)
- oder ungenauen Rechnens (*Float*).

Weitere Typen kann der Programmierer selbst ableiten, aus gegebenen und bereits abgeleiteten Typen.

Haskell: irreguläres Verhalten

- Überschreiten von Speicherschranken
- Überschreiten von Zeitschranken
- inkonsistente Typen im Programm

was bedeutet das semantisch
was bedeutet das mathematisch?

alles abbilden auf $\perp$
! Sorgfalt im Umgang mit $\perp$
nicht-strikte Funktionen sind unausweichlich
aber verhalten sich schnell nicht so, wie man will

HASKELL

Ende

Ich wünsche allen Studentinnen und Studenten erfolgreiche Prüfungen und eine erholsame Semesterpause !

Modularisierung

Modul = Eine Struktureinheit im Programm, kapselt eine Menge von Funktionsdefinitionen, Typen, etc.

Modulersteller

- möchte nur einige der Funktionen für Nutzer freigegeben

-Implementation änderbar

Modulbenutzer

- braucht oft nur einige Funktionen
-mag evt. andere Namen

-will vielleicht importierte Funktionen weitergeben

Module dienen der Strukturierung von Programmen; sie haben eine klare Schnittstelle (Interface), in der festgelegt wird, was sie **exportieren** und was sie **importieren.**

Sie bieten:

- unabhängige Programmierung von Programmteilen
-
- separate Compilation von Programmteilen
-
- Bibliotheken von Komponenten können wieder verwendet werden

Beispiele:

```
module mymodule1 where
f = ...
g = ...
```

Default: alles wird exportiert

```
module mymodule2 where
import mymodule1
h = ...
```

Importiert f und g

```
module mymodule3 (f , ...) where
f = ...
g = ...
```

Exportiert nur f, ...

```
module mymodule4 (f , module bla) where
import bla
f = ...
g = ...
```

exportiert f und re-exportiert alles aus bla

import mymodule1 (f,...)

Importiert nur f,...

import mymodule1 **hiding** (f,...)

Importiert alles ausser f,...

import mymodule1 **renaming** (f to fff, g to ggg)

Import mit Umbenennung