Funktionale Programmierung

Grundlagen der funktionalen Programmierung II

Hans-Wolfgang Loidl, Andreas Abel

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

7. Mai 2009

Aus Grundlagen I

Terminänderung:
Vorlesung am **Mo 11.5. 12:15- in Raum .15** anstatt Do 14.5.
(fällt aus)

Aus Grundlagen I

Aus Grundlagen I kennen wir

- Basistypen
- Algebraische Datentypen mit patten matching
- Funktionen höherer Ordnung

Aus Grundlagen I

Aus Grundlagen I kennen wir

- Basistypen
- Algebraische Datentypen mit patten matching
- Funktionen höherer Ordnung

Zum Nachschlagen von Sprachdefinition und Libraries:

- Haskell Report: **http://haskell.org/onlinereport/**
- Haskell Libraries:
  http://www.haskell.org/ghc/dist/current/docs/libraries/

III. Polymorphismus

Haskell verwendet ein **statisches Typsystem**, d.h. alle Typen im Programm werden zur Compile-Zeit überprüft.

III. Polymorphismus

Haskell verwendet ein **statisches Typsystem**, d.h. alle Typen im Programm werden zur Compile-Zeit überprüft.
Vorteile statischer Typsysteme:

- Das Typsystem überprüft die korrekte Anwendung von Funktionen im Program als "sanity check":
  "*type correct programs don't go wrong*".
- Mit exakter Typinformation kann der Compiler mehr und bessere Optimierungen durchführen.

III. Polymorphismus

Haskell verwendet ein **statisches Typsystem**, d.h. alle Typen im Programm werden zur Compile-Zeit überprüft.
Vorteile statischer Typsysteme:

- Das Typsystem überprüft die korrekte Anwendung von Funktionen im Program als "sanity check": "*type correct programs don't go wrong*".
- Mit exakter Typinformation kann der Compiler mehr und bessere Optimierungen durchführen.

Nachteile statischer Typsysteme:

- Manche (sinnvolle) Programme können in der Sprache nicht ausgedrückt werden.
- Typinferenz kann sehr rechenintensiv werden.
- Fehlermeldungen sind teilweise schwer verständlich.

Parametrischer Polymorphismus

Das Typsystem von Haskell erlaubt es im Typ Variablen zu verwenden (**polymorpher** Typ). Diese Variablen sind universell quantifiziert, d.h. ein Typ mit einer Variablen steht für eine Menge konkreter Typen, die durch Instanzierung der Variablen durch konkrete Typen entstehen.

Parametrischer Polymorphismus

Das Typsystem von Haskell erlaubt es im Typ Variablen zu verwenden (**polymorpher** Typ). Diese Variablen sind universell quantifiziert, d.h. ein Typ mit einer Variablen steht für eine Menge konkreter Typen, die durch Instanzierung der Variablen durch konkrete Typen entstehen.
Funktionen mit polymorphem Typ verwenden **denselben** Code für alle Instanzen des Typs, z.B:

$$
\begin{aligned}
&\mathit{length} && :: [a] \to \mathit{Int} \\
&\mathit{length}\ [] && = 0 \\
&\mathit{length}\ (\_ : \mathit{xs}) && = 1 + (\mathit{length}\ \mathit{xs})
\end{aligned}
$$

Typklassen und ad-hoc Polymorphismus

In Haskell existiert das Konzept einer **(Typ-)Klasse**, die eine Familie von Funktionen zusammenfasst.
Klassen können für konkrete Typen instanziert werden.
Die dazugehörigen Funktionen müssen für den konkreten Typ definiert werden.
Damit erreicht man ad-hoc Polymorphismus ("**overloading**"), d.h. eine Funktion führt, in Abhängigkeit vom Typ, **verschiedenen** Code aus.
Ein Beispiel einer vordefinierten überladenen Funktion ist $+$

$$2 + 3 = 5$$
$$2.1 + 3.2 = 5.3$$

Typklassen und ad-hoc Polymorphismus

Beispiel für eine selbst definierte Klasse:

```
  — define a class with a size function
class HasSize a where
  sizeOf :: a → Int


  — instantiate this class for concrete types
instance HasSize [a] where
  sizeOf = length
instance HasSize (BinTree a) where
  sizeOf (Leaf _) = 1
  sizeOf (Node l r) = sizeOf l + sizeOf r
```

Instanzierung vordefinierter Klassen

Die einfachste Verwendung des Klassenkonzepts in Haskell ist die *automatische* Instanzierung für bestimmte vordefinierte Klassen mittels **deriving**.
Für folgende Klassen können automatisch Instanzen erzeugt werden: `Eq`, `Ord`, `Enum`, `Bounded`, `Show`, `Read`.
Als Beispiel erzeugt folgende Definition automatisch eine *show* Funktion, als Teil der Instanz der Klasse *Show*:

$$\textbf{data}\ \mathit{Complex4}\ a = a :+ a \\ \textbf{deriving}\ (\mathit{Eq}, \mathit{Show}, \mathit{Read})$$

Nun können wir *show* auf Ausdrücke vom Typ *Complex4 Int* anwenden:

$$\textbf{let}\ c1 = 2 :+ 3 \\ \textbf{in}\ \mathit{show}\ c1$$

Wichtige vordefinierte Klassen

class $Eq\ \alpha$ **where**

$(==), (\neq) :: \alpha \to \alpha \to Bool$ — Types of class-functions

$x \neq y = not\ (x == y)$ — Default definition

instance $(Eq\ a) \Rightarrow Eq\ (Complex4\ a)$ **where**

$(r1 :+ i1) == (r2 :+ i2) = (r1 == r2)\ \&\&\ (i1 == i2)$

Beachte: $(Eq\ a) \Rightarrow$ fordert, dass der Typ a ebenfalls eine Instanz der Eq Klasse ist.

Wichtige vordefinierte Klassen

Wir erweitern unsere *HasSize* Klasse um eine *elemOf* Funktion:

```
class (Eq b) ⇒ MyClass a b | a → b where
      sizeOf :: a → Int
      elemOf :: b → a → Bool
```

Die Größe einer Datenstruktur soll jetzt auch vom Elementtyp abhängen:

```
instance (Eq a, MyClass a b) ⇒ MyClass [a] a where
  sizeOf []          = 0
  sizeOf xs@(x : _)  = length xs * sizeOf x
  elemOf x []                 = False
  elemOf x (y : ys) | x == y  = True
                    | otherwise = x `elemOf` ys
```

Module

Ein Modul *Test* das f von *TestAux* importiert und selbst g und f exportiert wird wie folgt definiert:

module *Test*(f, g) **where**
import *TestAux*(f)
$g = \ldots$

Nach dem Modulnamen kann eine Liste von Funktionen und Typen angegeben werden, die aus dem Modul *exportiert* werden. Per default werden alle Definitionen exportiert.
Im Gegensatz zu SMLs Funktoren gibt es keine parameterisierten Module in Haskell.
Definitionen anderer Module werden mittels **import** geladen.

Module

Beispiel: Definition einer *flatten* Funktion auf einem binären Baum:

module *TreeOps*(*flatten*) **where** — export only the function flatten
import *BinTree*(*BinTree*, *left*, *right*) — import data structure and fcts
$flatten :: BinTree\ a \rightarrow [a]$
$flatten\ (Node\ l\ r) = (flatten\ l) +\!\!+ (flatten\ r)$ — could also use left
$flatten\ (Leaf\ x) = [x]$

Haskell Prelude

Das Haskell Prelude ist die Basisbibliothek für Haskell, die automatisch immer geladen ist. Siehe
http://haskell.org/onlinereport/standard-prelude.html
sowie
http://www.haskell.org/ghc/dist/current/docs/libraries/base/Prelude.htr

Haskell Prelude

Das Haskell Prelude ist die Basisbibliothek für Haskell, die automatisch immer geladen ist. Siehe
http://haskell.org/onlinereport/standard-prelude.html
sowie
http://www.haskell.org/ghc/dist/current/docs/libraries/base/Prelude.htr
Einige nützliche Funktionen aus dem Prelude sind

$$
\begin{array}{ll}
take & :: Int \to [a] \to [a] \\
drop & :: Int \to [a] \to [a] \\
(++) & :: [a] \to [a] \to [a] \\
(!!) & :: [a] \to Int \to a \\
zip & :: [a] \to [b] \to [(a,b)] \\
enumFromTo & :: (Ord\ a) \Rightarrow a \to a \to [a] \\
filter & :: (a \to Bool) \to [a] \to [a] \\
fromIntegral & :: (Integral\ a, Num\ b) \Rightarrow a \to b \\
error & :: [Char] \to a
\end{array}
$$

Haskell Prelude

Es gelten folgende Äquivalenzen:

Auswertung

$take\ m\ [x_o, \ldots, x_n] \implies [x_o, \ldots, x_{m-1}]$

$drop\ m\ [x_o, \ldots, x_n] \implies [x_m, \ldots, x_n]$

$[x_o, \ldots, x_n] \mathbin{++} [y_0, \ldots, y_m] \implies [x_0, \ldots, x_n, y_0, \ldots, y_m]$

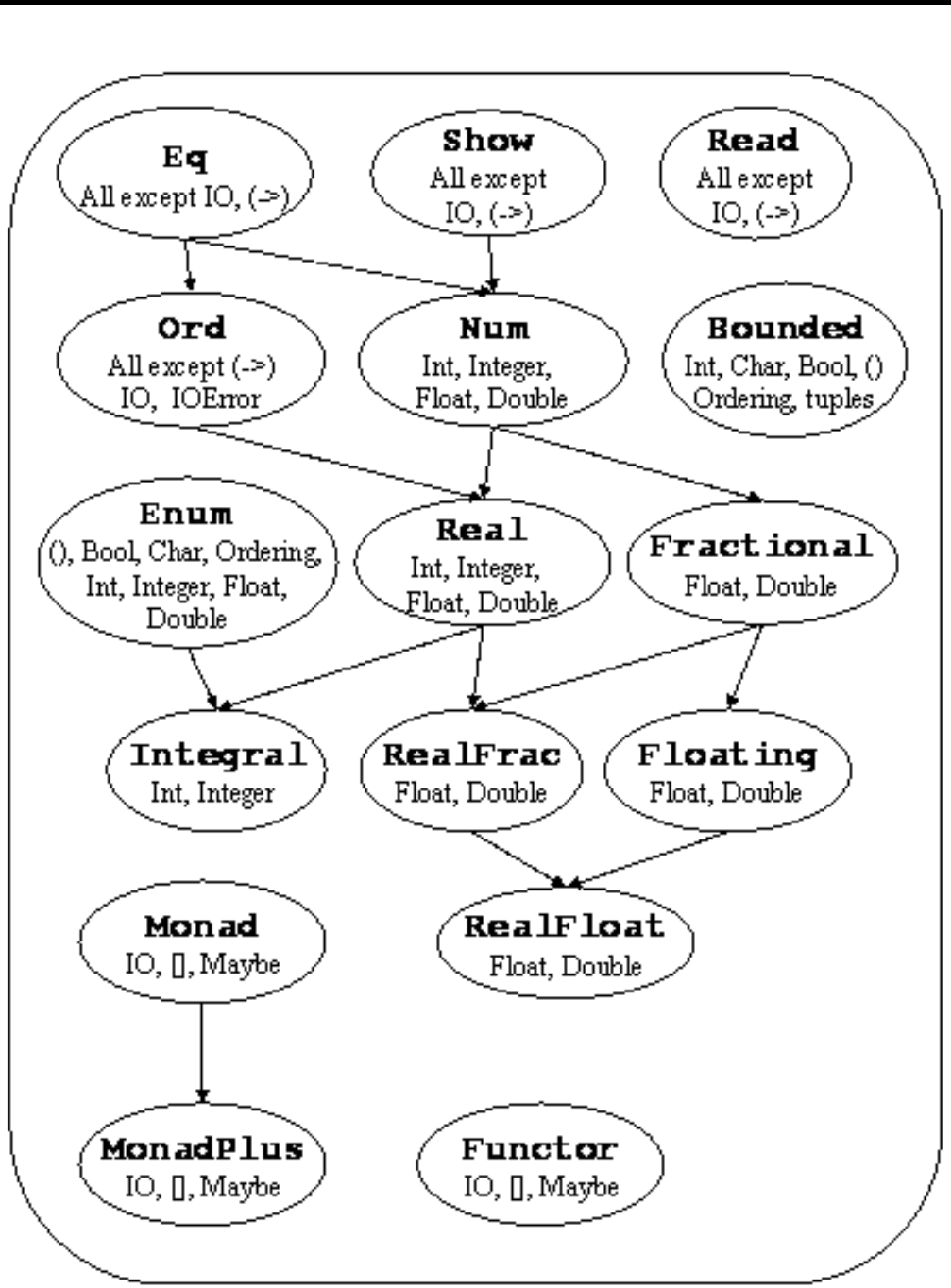
$[x_o, \ldots, x_{m-1}, x_m, x_{m+1}, \ldots, x_{n-1}] \mathbin{!!} m \implies x_m$

$zip\ [x_o, \ldots, x_n]\ [y_0, \ldots, y_n] \implies [(x_0, y_0), \ldots, (x_n, y_n)]$

$enumFromTo\ m\ n \implies [m, m+1, \ldots, n]$

$filter\ p\ xs \implies [x \mid x \leftarrow xs, p\ x]$

Kl


Eq
All except IO, (->)
Show
All except IO, (->)
Read
All except IO, (->)
Ord
All except (->) IO, IOError
Num
Int, Integer, Float, Double
Bounded
Int, Char, Bool, () Ordering, tuples
Enum
(), Bool, Char, Ordering, Int, Integer, Float, Double
Real
Int, Integer, Float, Double
Fractional
Float, Double
Integral
Int, Integer
RealFrac
Float, Double
Floating
Float, Double
Monad
IO, [], Maybe
RealFloat
Float, Double
MonadPlus
IO, [], Maybe
Functor
IO, [], Maybe

IV. Bedarfsauswertung ("lazy evaluation")

Bei der Auswertung eines Funktionsaufrufs verwendet Haskell **Bedarfsauswertung** (verzögerte Auswertung, "lazy evaluation"), d.h. die Argumente der Funktion werden erst ausgewertet wenn sie für die Berechnung benötigt werden.

Bedarfsauswertung ist die bevorzugte, aber nicht die einzig mögliche, Implementierung einer Sprache mit nicht-strikter Semantik.

In Sprachen mit strikter Semantik, z.B. SML, ist es möglich Bedarfsauswertung mittels expliziter Lambda Ausdrücke zu modellieren.

Bedarfsauswertung ("lazy evaluation")

Als Beispiel für Bedarfsauswertung, betrachten wir folgende Funktion:

$$\begin{aligned} \mathit{foo}\ x\ y\ z\ = \ & \textbf{if}\ \underline{x < 0}\ \textbf{then}\ \mathit{abs}\ x \\ & \textbf{else}\ x + y \end{aligned}$$

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von $\underline{x < 0}$ und dieses wiederum ein Auswerten des Arguments x.

Bedarfsauswertung ("lazy evaluation")

Als Beispiel für Bedarfsauswertung, betrachten wir folgende Funktion:

$$\begin{aligned} \mathit{foo}\ x\ y\ z\ = & \ \textbf{if}\ x < 0\ \textbf{then}\ \underline{\mathit{abs}\ x} \\ & \ \textbf{else}\ x + y \end{aligned}$$

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von $\underline{x < 0}$ und dieses wiederum ein Auswerten des Arguments x.
- Falls $x < 0$ wahr ist, wird der Wert von $\underline{\mathit{abs}\ x}$ zurückgegeben; weder y noch z werden ausgewertet.

Bedarfsauswertung ("lazy evaluation")

Als Beispiel für Bedarfsauswertung, betrachten wir folgende Funktion:

$$\begin{aligned} \mathit{foo}\ x\ y\ z\ = \ & \textbf{if}\ x < 0\ \textbf{then}\ \mathit{abs}\ x \\ & \textbf{else}\ \underline{x + y} \end{aligned}$$

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von $\underline{x < 0}$ und dieses wiederum ein Auswerten des Arguments x.
- Falls $x < 0$ wahr ist, wird der Wert von $\underline{\mathit{abs}\ x}$ zurückgegeben; weder y noch z werden ausgewertet.
- Falls $x < 0$ falsch ist, wird der Wert von $\underline{x + y}$ zurückgegeben; dies erfordert die Auswertung von y.

Bedarfsauswertung ("lazy evaluation")

Als Beispiel für Bedarfsauswertung, betrachten wir folgende Funktion:

$$\begin{aligned} foo\ x\ y\ z\ = \ & \textbf{if}\ x < 0\ \textbf{then}\ abs\ x \\ & \textbf{else}\ \underline{x + y} \end{aligned}$$

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von $\underline{x < 0}$ und dieses wiederum ein Auswerten des Arguments x.
- Falls $x < 0$ wahr ist, wird der Wert von $\underline{abs\ x}$ zurückgegeben; weder y noch z werden ausgewertet.
- Falls $x < 0$ falsch ist, wird der Wert von $\underline{x + y}$ zurückgegeben; dies erfordert die Auswertung von y.
- z wird in keinem Fall ausgewertet.

Bedarfsauswertung ("lazy evaluation")

Als Beispiel für Bedarfsauswertung, betrachten wir folgende Funktion:

$$\begin{aligned} \mathit{foo}\ x\ y\ z\ = \ & \textbf{if}\ x < 0\ \textbf{then}\ \mathit{abs}\ x \\ & \textbf{else}\ \underline{x + y} \end{aligned}$$

Auswertungsreihenfolge:

- Die Auswertung des If-Ausdrucks erfordert ein Auswerten von $\underline{x < 0}$ und dieses wiederum ein Auswerten des Arguments x.
- Falls $x < 0$ wahr ist, wird der Wert von $\underline{\mathit{abs}\ x}$ zurückgegeben; weder y noch z werden ausgewertet.
- Falls $x < 0$ falsch ist, wird der Wert von $\underline{x + y}$ zurückgegeben; dies erfordert die Auswertung von y.
- z wird in keinem Fall ausgewertet.
- Insbesondere ist der Ausdruck $\mathit{foo}\ 1\ 2\ (1\ `\mathit{div}`\ 0)$ wohldefiniert.

Definitionen

Wir verwenden im folgenden diese Funktionen:

$$
\begin{array}{l}
(!!) :: [a] \to Int \to a \\
[] \;!!\; \_ \quad\quad\quad = error\ "Empty\ list" \\
(x : \_) \;!!\; 0 \quad\quad = x \\
(\_ : xs) !!\; n \quad\quad = xs \;!!\; (n-1) \\
enumFromTo :: Int \to Int \to [Int] \\
enumFromTo\ m\ n \mid n < m = [] \\
\qquad\qquad\qquad\quad \mid otherwise = m : (enumFromTo\ (m+1)\ n)
\end{array}
$$

$[m..n]$ is "syntaktischer Zucker" für $enumFromTo\ m\ n$.

Unendliche Datenstrukturen

Mit Bedarfsauswertung ist es möglich unendliche Datenstrukturen zu definieren. Es wird immer nur soviel von der Datenstruktur ausgewertet wie benötigt wird:

Auswertung

$[0..]!!2 \Longrightarrow 2$

Ein Beispiel für Bedarfsauswertung

Hier ist die Auswertungshistorie für [0..]!!2:

Auswertung

[0..]!!2 $\Longrightarrow$ *ist die Liste leer?*

Ein Beispiel für Bedarfsauswertung

Hier ist die Auswertungshistorie für [0..]!!2:

Auswertung

$$
\begin{array}{rcl}
[0..]!!2 & \Longrightarrow & \textit{ist die Liste leer?} \\
(0 : [1..])!!2 & \Longrightarrow & \textit{ist der Index 0?}
\end{array}
$$

Ein Beispiel für Bedarfsauswertung

Hier ist die Auswertungshistorie für [0..]!!2:

Auswertung

[0..]!!2	$\Longrightarrow$	*ist die Liste leer?*
(0 : [1..])!!2	$\Longrightarrow$	*ist der Index 0?*
[1..]!!1	$\Longrightarrow$	*ist die Liste leer?*

Ein Beispiel für Bedarfsauswertung

Hier ist die Auswertungshistorie für [0..]!!2:

Auswertung

[0..]!!2	$\Longrightarrow$	*ist die Liste leer?*
(0 : [1..])!!2	$\Longrightarrow$	*ist der Index 0?*
[1..]!!1	$\Longrightarrow$	*ist die Liste leer?*
(1 : [2..])!!1	$\Longrightarrow$	*ist der Index 0?*

Ein Beispiel für Bedarfsauswertung

Hier ist die Auswertungshistorie für [0..]!!2:

Auswertung

[0..]!!2	$\Longrightarrow$	*ist die Liste leer?*
(0 : [1..])!!2	$\Longrightarrow$	*ist der Index 0?*
[1..]!!1	$\Longrightarrow$	*ist die Liste leer?*
(1 : [2..])!!1	$\Longrightarrow$	*ist der Index 0?*
[2..]!!0	$\Longrightarrow$	*ist die Liste leer?*

Ein Beispiel für Bedarfsauswertung

Hier ist die Auswertungshistorie für [0..]!!2:

Auswertung

[0..]!!2	$\Longrightarrow$	*ist die Liste leer?*
(0 : [1..])!!2	$\Longrightarrow$	*ist der Index 0?*
[1..]!!1	$\Longrightarrow$	*ist die Liste leer?*
(1 : [2..])!!1	$\Longrightarrow$	*ist der Index 0?*
[2..]!!0	$\Longrightarrow$	*ist die Liste leer?*
(2 : [3..])!!0	$\Longrightarrow$	*ist der Index 0?*

Ein Beispiel für Bedarfsauswertung

Hier ist die Auswertungshistorie für [0..]!!2:

Auswertung

[0..]!!2	$\Longrightarrow$	*ist die Liste leer?*
(0 : [1..])!!2	$\Longrightarrow$	*ist der Index 0?*
[1..]!!1	$\Longrightarrow$	*ist die Liste leer?*
(1 : [2..])!!1	$\Longrightarrow$	*ist der Index 0?*
[2..]!!0	$\Longrightarrow$	*ist die Liste leer?*
(2 : [3..])!!0	$\Longrightarrow$	*ist der Index 0?*
2		

Ein Beispiel für Bedarfsauswertung

Hier ist die Auswertungshistorie für [0..]!!2:

Auswertung

[0..]!!2	$\Longrightarrow$	*ist die Liste leer?*
(0 : [1..])!!2	$\Longrightarrow$	*ist der Index 0?*
[1..]!!1	$\Longrightarrow$	*ist die Liste leer?*
(1 : [2..])!!1	$\Longrightarrow$	*ist der Index 0?*
[2..]!!0	$\Longrightarrow$	*ist die Liste leer?*
(2 : [3..])!!0	$\Longrightarrow$	*ist der Index 0?*
2		

Weitere Beispiele unendlicher Datenstrukturen werden wir in der Vorlesung zu zirkulären Datenstrukturen kennenlernen.

Why Functional Programming Matters

Funktionale Sprachen unterstützen Konzepte, die das Erstellen **modularer** Software erleichtern:

- Funktionen höherer Ordnung;
- Bedarfsauswertung

Why Functional Programming Matters

Funktionale Sprachen unterstützen Konzepte, die das Erstellen **modularer** Software erleichtern:

- Funktionen höherer Ordnung;
- Bedarfsauswertung

Funktionen höherer Ordnung abstrahieren Kontrollstrukturen und ermöglichen es einfache Funktionen zu komplexen zu verknüpfen.

Zusammenfassung

Ein wichtiges Konzept zur Modularisierung ist Funktionskomposition.
Dies ermöglicht es durch "Hintereinanderausführung" komplexere Funktionen zu erzeugen.

Zusammenfassung

Ein wichtiges Konzept zur Modularisierung ist Funktionskomposition.
Dies ermöglicht es durch "Hintereinanderausführung" komplexere Funktionen zu erzeugen.
Wir haben dies im Beispiel zur Summe der Quadrate von geraden Zahlen gesehen:

$$sqs :: [Int] \to Int$$
$$sqs = sum\ .\ map\ (\lambda x \to x * x)\ .\ (filter\ even)$$

Zusammenfassung

Ein wichtiges Konzept zur Modularisierung ist Funktionskomposition.
Dies ermöglicht es durch "Hintereinanderausführung" komplexere Funktionen zu erzeugen.
Wir haben dies im Beispiel zur Summe der Quadrate von geraden Zahlen gesehen:

$$sqs :: [Int] \to Int$$
$$sqs = sum\ .\ map\ (\lambda x \to x * x)\ .\ (filter\ even)$$

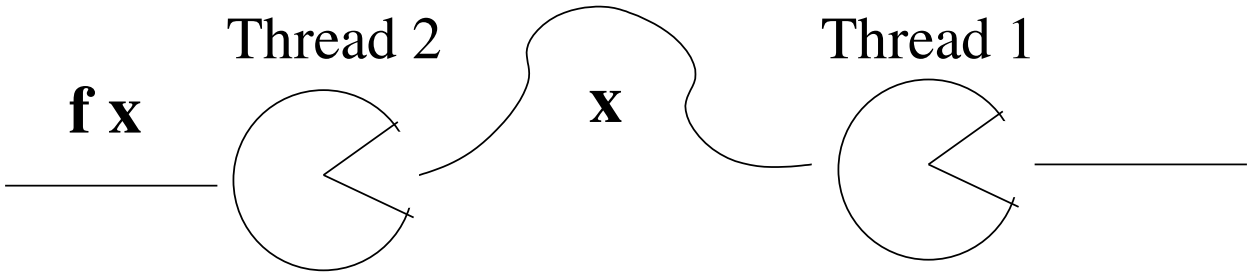
Erzeugt dieses Programm eine große Datenstruktur? **Nein:**

Zusammenfassung

Ein wichtiges Konzept zur Modularisierung ist Funktionskomposition.
Dies ermöglicht es durch "Hintereinanderausführung" komplexere Funktionen zu erzeugen.
Wir haben dies im Beispiel zur Summe der Quadrate von geraden Zahlen gesehen:

$$sqs :: [Int] \to Int$$
$$sqs = sum\ .\ map\ (\lambda x \to x * x)\ .\ (filter\ even)$$

Erzeugt dieses Programm eine große Datenstruktur? **Nein:**



Mittels Bedarfsauswertung ("lazy evaluation") ist es möglich Programme als Produzenten und Konsumenten zu strukturieren.

Beispiel: Alpha-Beta Suche

Gegeben: Position in einem Spiel.
Gesucht: Der beste Zug in dieser Position.

Beispiel: Alpha-Beta Suche

Gegeben: Position in einem Spiel.
Gesucht: Der beste Zug in dieser Position.
Idee:

1. Erzeuge einen Baum aller möglichen Positionen, mit der derzeitigen Position als Wurzel.
2. Ermittle den Wert jeder Position, mittels einer statischen Evaluierungsfunktion.
3. Suche einen Pfad durch den Baum, in dem abwechselnd der beste und der schlechteste Zug gewählt wird ("Minimax Suche").

1. Baumerzeugung

Wir verwenden folgende problemabhängige Datenstrukturen

$Position$ — abstract data type representing a position (eg. board)

data $NTree\ a = Branch\ a\ [NTree\ a]$ — tree structure
deriving $(Show, Read, Eq)$

und Funktionen:

— all possible moves from a given position
$moves :: Position \rightarrow [Position]$
— the static evaluation function
$static :: Position \rightarrow Int$

Wir erzeugen einen unendlichen Baum, durch wiederholtes anwenden einer Funktion vom Typ $a \rightarrow [a]$:

$repTree :: (a \rightarrow [a]) \rightarrow a \rightarrow (NTree\ a)$
$repTree\ f\ p = Branch\ p\ (map\ (repTree\ f)\ (f\ p))$

2. Statische Evaluierung

Wir abstrahieren die Kontrollstruktur einer map Funktion auf einen Baum:

$$\begin{aligned} &mapTree && :: (a \to b) \to (NTree\ a) \to (NTree\ b) \\ &mapTree\ f\ (Branch\ x\ xs) && = Branch\ (f\ x)\ (map\ (mapTree\ f)\ xs) \end{aligned}$$

Wir wenden die statische Evaluierungsfunktion auf jeden Knoten im Baum an:

$$statEval = mapTree\ static$$

3. Suche des besten Pfades

In jedem Schritt, wird aus allen möglichen Nachfolgern im Baum abwechselnd der beste oder schlechteste ausgewählt:

$$
\begin{aligned}
&\mathit{maximise} && :: \mathit{NTree\ Int} \rightarrow \mathit{Int} \\
&\mathit{maximise}\ (\mathit{Branch}\ x\ []) && = x \\
&\mathit{maximise}\ (\mathit{Branch}\ x\ ps) && = \mathit{maximum}\ (\mathit{map}\ \mathit{minimise}\ ps) \\
&\mathit{minimise} && :: \mathit{NTree\ Int} \rightarrow \mathit{Int} \\
&\mathit{minimise}\ (\mathit{Branch}\ x\ []) && = x \\
&\mathit{minimise}\ (\mathit{Branch}\ x\ ps) && = \mathit{minimum}\ (\mathit{map}\ \mathit{maximise}\ ps)
\end{aligned}
$$

3. Suche des besten Pfades

In jedem Schritt, wird aus allen möglichen Nachfolgern im Baum abwechselnd der beste oder schlechteste ausgewählt:

$$
\begin{aligned}
&\mathit{maximise} && :: \mathit{NTree\ Int} \rightarrow \mathit{Int} \\
&\mathit{maximise}\ (\mathit{Branch}\ x\ []) && = x \\
&\mathit{maximise}\ (\mathit{Branch}\ x\ ps) && = \mathit{maximum}\ (\mathit{map}\ \mathit{minimise}\ ps) \\
&\mathit{minimise} && :: \mathit{NTree\ Int} \rightarrow \mathit{Int} \\
&\mathit{minimise}\ (\mathit{Branch}\ x\ []) && = x \\
&\mathit{minimise}\ (\mathit{Branch}\ x\ ps) && = \mathit{minimum}\ (\mathit{map}\ \mathit{maximise}\ ps)
\end{aligned}
$$

Eine erste (nicht-funktionierende) Version des Algorithmus ist:

$$\mathit{evaluate} = \mathit{maximise} \,.\, \mathit{mapTree\ static} \,.\, \mathit{repTree\ moves}$$

Debugging

Diese Version muss den ganzen (möglicherweise unendlichen) Baum durchsuchen. Wir müssen zunächst die Tiefe des Suchbaums beschränken:

$$
\begin{array}{lcl}
\mathit{prune} & :: & \mathit{Int} \to (\mathit{NTree}\ a) \to (\mathit{NTree}\ a) \\
\mathit{prune}\ 0\ (\mathit{Branch}\ x\ xs) & = & \mathit{Branch}\ x\ [] \\
\mathit{prune}\ (n+1)\ (\mathit{Branch}\ x\ xs) & = & \mathit{Branch}\ x\ (\mathit{map}\ (\mathit{prune}\ n)\ xs)
\end{array}
$$

Debugging

Diese Version muss den ganzen (möglicherweise unendlichen) Baum durchsuchen. Wir müssen zunächst die Tiefe des Suchbaums beschränken:

$$
\begin{array}{lll}
\mathit{prune} & :: & \mathit{Int} \rightarrow (\mathit{NTree}\ a) \rightarrow (\mathit{NTree}\ a) \\
\mathit{prune}\ 0\ (\mathit{Branch}\ x\ xs) & = & \mathit{Branch}\ x\ [] \\
\mathit{prune}\ (n+1)\ (\mathit{Branch}\ x\ xs) & = & \mathit{Branch}\ x\ (\mathit{map}\ (\mathit{prune}\ n)\ xs)
\end{array}
$$

Eine erste funktionierende Version des Algorithmus ist:

$$
\mathit{evaluate} = \mathit{maximise} \,.\, \mathit{mapTree}\ \mathit{static} \,.\, \mathit{prune}\ 5 \,.\, \mathit{repTree}\ \mathit{moves}
$$

Tuning

Um Teilbäume, die nicht zum Resultat beitragen, abschneiden zu können, müssen wir *maximise* und *minimise* restrukturieren.

$$\begin{aligned} \mathit{maximise} &= \mathit{maximum} \,.\, \mathit{maximise}' \\ \mathit{maximise}'\ (\mathit{Branch}\ x\ xs) &= \mathit{map}\ \mathit{minimise}\ xs \end{aligned}$$

Tuning

Um Teilbäume, die nicht zum Resultat beitragen, abschneiden zu können, müssen wir *maximise* und *minimise* restrukturieren.

$$\begin{aligned} \mathit{maximise} &= \mathit{maximum} \,.\, \mathit{maximise}' \\ \mathit{maximise}'\ (\mathit{Branch}\ x\ xs) &= \mathit{map}\ \mathit{minimise}\ xs \end{aligned}$$

Nun wenden wir folgende Transformationen auf $\mathit{maximise}'$ an:

$$\begin{aligned} \mathit{maximise}'\ (\mathit{Branch}\ x\ xs) &= \mathit{map}\ \mathit{minimise}\ xs \\ &= \mathit{map}\ (\mathit{minimum} \,.\, \mathit{minimise}')\ xs \\ &= \mathit{map}\ \mathit{minimum}\ (\mathit{map}\ \mathit{minimise}'\ xs) \\ &= \mathit{mapmin}\ (\mathit{map}\ \mathit{minimise}'\ xs) \\ &\textbf{where}\ \mathit{mapmin} = \mathit{map}\ \mathit{minimum} \end{aligned}$$

Tuning

Wir sind nur am **maximum** der durch *mapmin* erzeugten Liste interessiert. Wir formen *mapmin* so um, dass der beste bisherige Wert weitergereicht wird:

$$\begin{aligned} \mathit{mapmin}\,(\mathit{xs} : \mathit{xss}) = {} & \mathit{curr} : (\mathit{omit}\ \mathit{curr}\ \mathit{xss}) \\ & \textbf{where}\ \mathit{curr} = \mathit{minimum}\ \mathit{xs} \end{aligned}$$

Tuning

Wir sind nur am **maximum** der durch *mapmin* erzeugten Liste interessiert. Wir formen *mapmin* so um, dass der beste bisherige Wert weitergereicht wird:

$$\begin{aligned} mapmin\ (xs : xss) = {} & curr : (omit\ curr\ xss) \\ & \textbf{where}\ curr = minimum\ xs \end{aligned}$$

omit ignoriert Sub-Bäume, deren Minima kleiner als *curr* sind:

$$\begin{aligned} & omit\ curr\ [] = [] \\ & omit\ curr\ (xs : xss) \quad \mid minleq\ xs\ curr = omit\ curr\ xss \\ & \qquad\qquad\qquad\qquad\quad \mid otherwise \quad = curr' : (omit\ curr'\ xss) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ curr' = minimum\ xs \end{aligned}$$

Tuning

Wir sind nur am **maximum** der durch *mapmin* erzeugten Liste interessiert. Wir formen *mapmin* so um, dass der beste bisherige Wert weitergereicht wird:

$$\begin{aligned} mapmin\ (xs : xss) = {} & curr : (omit\ curr\ xss) \\ & \textbf{where}\ curr = minimum\ xs \end{aligned}$$

omit ignoriert Sub-Bäume, deren Minima kleiner als *curr* sind:

$$\begin{aligned} & omit\ curr\ [] = [] \\ & omit\ curr\ (xs : xss) \quad \mid minleq\ xs\ curr = omit\ curr\ xss \\ & \qquad\qquad\qquad\qquad\quad \mid otherwise \quad = curr' : (omit\ curr'\ xss) \\ & \qquad\qquad\qquad\qquad\qquad\qquad \textbf{where}\ curr' = minimum\ xs \end{aligned}$$

minleq testet, ob das Minimum kleiner als *curr* ist:

$$\begin{aligned} & minleq\ []\ curr = [] \\ & minleq\ (x : xs)\ curr \quad \mid x < curr = True \\ & \qquad\qquad\qquad\qquad\quad \mid otherwise \quad = minleq\ xs\ curr \end{aligned}$$

Tuning

Die verbesserte Evaluierungsfunktion verwendet die umgeformte *maximise'* Funktion:

$$\begin{aligned} evaluate\ =& maximum\ .\ maximise'\ .\ mapTree\ static\ . \\ & prune\ 5\ .\ repTree\ moves \end{aligned}$$

Tuning

Die verbesserte Evaluierungsfunktion verwendet die umgeformte *maximise'* Funktion:

$$\begin{aligned} evaluate\ =& maximum\ .\ maximise'\ .\ mapTree\ static\ . \\ & prune\ 5\ .\ repTree\ moves \end{aligned}$$

Weitere Verbesserungen sind möglich, durch Berücksichtigung der statischen Evaluierungsfunktion in jedem Knoten:

- Sortieren der Nachfolger im Baum;
- Einschränken der Nachfolger im Baum auf die ersten n;
- Abschneiden von Teilbäumen mit sehr geringem Wert.

Zusammenfassung

- Typ-Klassen ermöglichen es Funktionen zu überladen ("overloading", "ad-hoc polymorphsim")
- Bedarfsauswertung ("lazy evaluation") ermöglicht die Verwendung unendlicher Datenstrukturen
- Funktionen höherer Ordnung ("higher-order functions") ermöglichen die Abstraktion von Berechnungsstrukturen
- Diese Konzepte sind nützliche für die **Modularisierung**

Mini-Projekt: Implementierung eines Strategiespiels mit künstlicher Intelligenz unter Verwendung der "Alpha-Beta Search" Heurisitik.