Funktionale Programmierung

Folds und Nested Datatypes

Dulma Rodriguez, Hans-Wolfgang Loidl, Andreas Abel

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

2. Juli 2009

Fold für Listen

- Listen sind vordefiniert in Haskell: $[a]$
- Folgende *fold* Funktion ist auch vordefiniert.

$$
\begin{aligned}
&\mathit{fold} :: (a \to b \to b) \to b \to ([a] \to b)\\
&\mathit{fold}\ f\ v\ [\,] = v\\
&\mathit{fold}\ f\ v\ (x : xs) = f\ x\ (\mathit{fold}\ f\ v\ xs)
\end{aligned}
$$

- Damit können Funktionen über Listen definiert werden.
- *fold* nimmt als Argumente eine Funktion f für den "cons" Fall...
- ... und eine Konstante v für den "[]" Fall.

Funktionsdefinitionen mit Fold

- Elemente einer Liste aufsummieren (*sum*)

$$sum :: [Int] \to Int$$
$$sum = fold\ (+)\ 0$$

- Elemente einer Liste miteinander multiplizieren (*product*)

$$product :: [Int] \to Int$$
$$product = fold\ (*)\ 1$$

- Länge einer Liste berechnen (*length*)

$$length :: [a] \to Int$$
$$length = fold\ (\lambda x\ n \to 1 + n)\ 0$$

Funktionsdefinitionen mit Fold

- Elemente einer Liste umkehren (*reverse*)

$$reverse :: [a] \to [a]$$
$$reverse = fold\ (\lambda x\ xs \to xs +\!\!+ [x])\ []$$

- Eine Funktion auf allen Elementen einer Liste anwenden (*map*)

$$map :: (a \to b) \to ([a] \to [b])$$
$$map\ f = fold\ (\lambda x\ xs \to (f\ x) : xs)\ []$$

- Elemente aus einer Liste herausfiltern die eine Eigenschaft erfüllen (*filter*)

$$filter :: (a \to Bool) \to ([a] \to [a])$$
$$filter\ p = fold\ (\lambda x\ xs \to \textbf{if}\ p\ x\ \textbf{then}\ x : xs\ \textbf{else}\ xs)\ []$$

Universelle Eigenschaft von Fold

- Äquivalenz zwischen zwei Definitionen einer Funktion g, die Listen verarbeitet.

$$g\ [\,] = v$$
$$g\ (x : xs) = f\ x\ (g\ xs)$$

$$\Longleftrightarrow$$

$$g = \mathit{fold}\ f\ v$$

- $(\mathit{fold}\ f\ v)$ ist die *einzige* Lösung zu den Formeln die g definieren.
- Die universelle Eigenschaft kann benutzt werden um Funktionen mit *fold* zu definieren.

Definition von Funktionen mit Hilfe der universellen Eigenschaft

- Beispiel *sum*

$$
\begin{aligned}
&sum :: [Int] \to Int \\
&sum\ [\,] = 0 \\
&sum\ (x : xs) = x + sum\ xs
\end{aligned}
$$

- universelle Eigenschaft sagt:

$$
\begin{aligned}
&sum\ [\,] = v \\
&sum\ (x : xs) = f\ x\ (sum\ xs)
\end{aligned}
$$

- $\Rightarrow v = 0$, f wird wie folgt berechnet:

$$
\begin{array}{lcll}
sum\,(x : xs) & = & f\,x\,(sum\ xs) & \Longleftrightarrow \text{ Definition von } sum \\
x + sum\ xs & = & f\,x\,(sum\ xs) & \Longleftrightarrow \text{ General. vom } (sum\ xs) \text{ zum } y \\
x + y & = & f\,x\,y & \Longleftrightarrow \\
f & = & (+) &
\end{array}
$$

$$
sum = fold\ (+)\ 0
$$

Binäre Bäume

- Binäre Bäume mit Beschriftungen in den Blättern können in Haskell als Datentyp *BinTree* definiert werden.

$$\textbf{data}\ \mathit{BinTree}\ a = \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{BinTree}\ a)\ (\mathit{BinTree}\ a)$$

- Anzahl von Blättern berechnen (*size*)

$$\begin{aligned}&\mathit{size} :: \mathit{BinTree}\ a \to \mathit{Int}\\ &\mathit{size}\ (\mathit{Leaf}\ \_) = 1\\ &\mathit{size}\ (\mathit{Node}\ l\ r) = \mathit{size}\ l + \mathit{size}\ r\end{aligned}$$

- Konvertierung von binären Bäumen in Listen (*flatten*)

$$\begin{aligned}&\mathit{flatten} :: \mathit{BinTree}\ a \to [a]\\ &\mathit{flatten}\ (\mathit{Node}\ l\ r) = (\mathit{flatten}\ l) +\!\!+ (\mathit{flatten}\ r)\\ &\mathit{flatten}\ (\mathit{Leaf}\ x) \quad = [x]\end{aligned}$$

Fold fuer Binäre Bäume

- Wie für Listen man kann eine *fold* Funktion für binären Bäumen definieren.

$$
\begin{aligned}
&fold :: (a \to b) \to (b \to b \to b) \to BinTree\ a \to b \\
&fold\ f\ comb\ (Leaf\ l) = f\ l \\
&fold\ f\ comb\ (Node\ l\ r) = (fold\ f\ comb\ l)\ `comb`\ (fold\ f\ comb\ r)
\end{aligned}
$$

- *fold* nimmt als Argumente eine Funktion f um die Beschriftungen an den Blättern zu transformieren...
- ...und eine Funktion *comb* um die transformierten Zweigen zusammenzuführen.
- size

$$
\begin{aligned}
&size :: BinTree\ a \to Int \\
&size = fold\ (\lambda x \to 1)\ (+)
\end{aligned}
$$

- flatten

$$
\begin{aligned}
&flatten :: BinTree\ a \to [a] \\
&flatten = fold\ (\lambda x \to [x])\ (+\!\!+)
\end{aligned}
$$

Allgemeine Beschreibung von Datentypen

- Listen könnten in Haskell so definiert werden.

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

- Datentypen können im Allgemeinen als der kleinste Fixpunkt μ eines (parametrisierten) Funktors definiert werden.

$$\begin{array}{lcl} \mathit{ListF}\ X\ A & = & 1 + A \times X \\ \mathit{List}\ A & = & \mu X.\, \mathit{ListF}\ X\ A \end{array}$$

- Es gibt einen einzigen Typkonstruktor *in*:

$$\mathit{in} \quad : \quad \underbrace{\mathit{ListF}\ (\mathit{List}\ A)\ A}_{1 + A \times \mathit{List}\ A} \rightarrow \mathit{List}\ A$$

- *Nil* und *Cons* können mit Hilfe von *in* definiert werden.

$$\begin{array}{lcll} \mathit{Nil} & := & \mathit{in}\,(\mathit{inl}()) & : \quad \mathit{List}\ A \\ \mathit{Cons}\ x\ xs & := & \mathit{in}\,(\mathit{inr}(x, xs)) & : \quad A \rightarrow \mathit{List}\ A \rightarrow \mathit{List}\ A \end{array}$$

Allgemeine Beschreibung von Datentypen

- Binäre Bäumen haben wir in Haskell so definiert.

$$\textbf{data}\ \mathit{BinTree}\ a = \mathit{Leaf}\ a \mid \mathit{Node}\ (\mathit{BinTree}\ a)\ (\mathit{BinTree}\ a)$$

- Als kleinste Fixpunkt eines parametrisierten Funktors:

$$\begin{array}{lcl} \mathit{BinTreeF}\ X\ A & = & A + X \times X \\ \mathit{BinTree}\ A & = & \mu X.\, \mathit{BinTreeF}\ X\ A \end{array}$$

- Es gibt einen einzigen Typkonstruktor *in*:

$$in \quad : \quad \underbrace{\mathit{BinTreeF}\ (\mathit{BinTree}\ A)\ A}_{A + \mathit{BinTreeA} \times \mathit{BinTree}\ A} \rightarrow \mathit{BinTree}\ A$$

- *Leaf* und *Node* können mit Hilfe von *in* definiert werden.

$$\begin{array}{lcll} \mathit{Leaf}\ a & := & in\,(inl\ a) & : \quad A \rightarrow \mathit{BinTree}\ A \\ \mathit{Node}\ l\ r & := & in\,(inr(l, r)) & : \quad \mathit{BinTree}\ A \rightarrow \mathit{BinTree}\ A \rightarrow \mathit{BinTree}\ A \end{array}$$

Allgemeine Beschreibung von Datentypen

- Gegeben ein Funktor F, man kann einen Datentyp definieren mit μF.
- Der allgemeine Konstruktor *in* hat den Typ $in : F\,(\mu F) \rightarrow \mu F$.
- Die speziellen Datenkonstruktoren wie *Nil*, *Cons*, *Leaf* und *Node* kann man mit Hilfe von *in* definieren.
- Man kann auch *fold* Funktionen allgemein für alle Datentypen definieren mit Hilfe von Iteratoren.

Fold fuer Listen aus dem Iterationsschema

- Ein allgemeiner Iterator oder *fold* Funktion hat diesen Typ:

$$fold : (F\ B \to B) \to \mu F \to B$$

- Listen

$$\begin{array}{lcl} ListF\ X\ A & = & 1 + A \times X \\ List\ A & = & \mu\, X.\, ListF\ X\ A \end{array}$$

- Fold fuer Listen aus dem Iterationsschema

$$\begin{array}{ll} fold : (\underbrace{1 + A \times B}_{F\ B} \to B) \to \underbrace{List\ A}_{\mu F} \to B & \iff \\ fold : (1 \to B) \to (A \times B \to B) \to List\ A \to B & \iff \\ fold : B \to (A \to B \to B) \to List\ A \to B & \end{array}$$

Fold fuer Binäre Bäume aus dem Iterationsschema

- Ein allgemeiner Iterator oder *fold* Funktion hat diesen Typ:

$$fold : (F\ B \to B) \to \mu F \to B$$

- Binäre Baume

$$\begin{aligned} BinTreeF\ A &= \lambda X\ .\ A + X \times X \\ BinTree\ A &= \mu\ X\ .BinTreeF\ X\ A \end{aligned}$$

- Fold für binäre Bäumen aus dem Iterationsschema

$$\begin{aligned} & fold : (\underbrace{A + B \times B}_{F\ B} \to B) \to \underbrace{BinTree\ A}_{\mu F} \to B & \iff \\ & fold : (A \to B) \to (B \times B \to B) \to BinTree\ A \to B & \iff \\ & fold : (A \to B) \to (B \to B \to B) \to BinTree\ A \to B \end{aligned}$$

Verschränkte Datentypen (Nested Datatypes)

- Nested Datatypes sind Datentypen höherer Ordnung.
- Sind Datentypen mit einem Typparameter
  $\Rightarrow$ eine Familie von Datentypen.
- Nested Datatypes können benutzt werden um Invarianten in den Typen zu definieren.
- Beispiel: Powerlisten

  data *PList a* = *Zero a* | *Succ* (*PList* (*a*, *a*))
- Powerlisten sind perfekt ausbalancierte binären Bäumen mit 2^n Elemente.
- Beispiel

  numPList :: *PList Int*
  numPList = *Succ* (*Succ* (*Zero* ((1, 2), (3, 4))))

Programmieren mit Powerlisten

- Elemente einer Powerliste aufsummieren (*psum*)

$$psum' :: (a \to Int) \to PList\ a \to Int$$
$$psum'\ f\ (Zero\ a) = f\ a$$
$$psum'\ f\ (Succ\ l) = psum'\ (\lambda(a1, a2) \to (f\ a1) + (f\ a2))\ l$$
$$psum :: (PList\ Int \to Int)$$
$$psum\ l = psum'\ (\lambda x \to x)\ l$$

- Powerliste aus 2^n Einsen.

$$ones' :: Int \to a \to PList\ a$$
$$ones'\ 0\ v = Zero\ v$$
$$ones'\ n\ v = Succ\ (ones'\ (n-1)\ (v, v))$$
$$ones :: Int \to PList\ Int$$
$$ones\ n = ones'\ n\ 1$$

Mathematische Definition von Powerlisten

- Nested Datatypes werden definiert als der kleinste Fixpunkt eines Funktors höherer Ordnung.
- Datentyp *PList*

$$\begin{aligned} PListF\ X\ A &:= A + X\,(A \times A) \\ PList &:= \mu X.PListF \end{aligned}$$

- Der allgemeine Datenkonstruktor *in* hat den Typ:

$$\begin{aligned} in &: \forall A.F\,(\mu F)\,A \to \mu\,F\,A \Rightarrow \\ in &: \forall A.\ PListF\ \ PList\ \ A \to PList\,A \iff \\ in &: \forall A.\ A + PList(A \times A) \to PList\,A \iff \end{aligned}$$

- *Zero* und *Succ* werden dann so definiert:

$$\begin{aligned} Zero\ a &:= in\,(inl\,a) : \forall A\,.\,A \to PList\,A \\ Succ\ l &:= in\,(inr\,l) : \forall A\,.\,PList\,(A \times A) \to PList\,A \end{aligned}$$

Fold fuer Powerlisten aus dem Iterationsschema

- Ein allgemeiner Iterator oder *fold* Funktion hat diesen Typ:

$$fold : (\forall A.F\ G\ A \to G\ A) \to (\forall A.\mu\ F\ A \to G\ A)$$

- Powerlisten

$$PListF\ X\ A \quad := \quad A + X\ (A \times A)$$

- Fold für Powerlisten aus dem Iterationsschema

$$fold : (\forall A.\underbrace{A + G(A \times A)}_{F\ G\ A} \to G\ A) \to (\forall A.\underbrace{PList\ A}_{\mu F} \to G\ A)$$

$$fold : (\forall A.(A \to G\ A)) \to (\forall A.G(A \times A) \to G\ A) \to (\forall A.PList\ A \to G\ A)$$

Fold fuer Powerlisten aus dem Iterationsschema

- Fold für Powerlisten

$$
\begin{aligned}
&fold :: (forall\ a, a \to g\ a) \to \\
&\quad (forall\ a, g\ (a, a) \to g\ a) \to PList\ a \to g\ a \\
&fold\ z\ s\ (Zero\ a) = z\ a \\
&fold\ z\ s\ (Succ\ l) = s\ (fold\ z\ s\ l)
\end{aligned}
$$

$$
\mathbf{newtype}\ G\ a = G\ \{unG :: (a \to Int) \to Int\}
$$

- Definition von sum mit Hilfe von fold

$$
\begin{aligned}
&sumZ\ a = G\ \$\ \lambda k \to k\ a \\
&sumS\ (G\ h) = G\ \$\ \lambda k \to h\ \$\ \lambda(a, a') \to k\ a + k\ a' \\
&sumPL :: PList\ Int \to Int \\
&sumPL\ l = unG\ (fold\ sumZ\ sumS\ l)\ id
\end{aligned}
$$

Zusammenfassung

- Für viele Programmieraufgaben sind *fold* Funktionen hilfreich.
- *fold* Funktionen sind oft vordefiniert.
- Es gibt allgemeine typtheoretische Methoden um sie zu definieren.
- Die theoretischen Analysen basieren auf einer Darstellung von Datentypen als der kleinste Fixpunkt eines Funktors.
- Es gibt Datentypen höheren Ordnung: nested datataypes.
- Nested Datatypes sind nützlich um Invarianten in den Typen festzulegen.
- Programmieren mit nested datatypes ist komplizierter als mit einfachen Datentyen weil alle Elemente der Familie in Beziehung zueinander sind.
- Man muss Funktionen definieren die für alle Elemente der Familie gelten, also stark polymorphisch.

Literatur


Andreas Abel, Ralph Matthes, and Tarmo Uustalu.
Iteration and coiteration schemes for higher-order and nested datatypes.
Theor. Comput. Sci., 333(1-2):3–66, 2005.


Graham Hutton.
A tutorial on the universality and expressiveness of fold.
Journal of Functional Programming, 9:355–372, 1993.