Was ist generisches Programmieren ?

Während ein konventioneller Algorithmus nur ein Problem löst, löst ein generischer Algorithmus eine Klasse von Problemen.

Ein Beispiel für den Sinn und Zweck generischen Programmierens (1)

Gegeben seien zwei verschiedene Datentypen:

a) Liste natürlicher Zahlen:
   data List Nat = Nil | Cons Nat (List Nat)

b) Binärbaum natürlicher Zahlen:
   data Bin Nat = Tip Nat | Join (Bin Nat) (Bin Nat)

... und ein Problem:

Summiere die in den o.a. Strukturen enthaltenen natürlichen Zahlen! (= „Summation“)

Ein Beispiel für den Sinn und Zweck generischen Programmierens (2)

Konventionelle Lösung des Problems:

- Summation für Listen natürlicher Zahlen:

  $sum_{List}(Nil) = 0$
  $sum_{List}(Cons\ u\ us) = u + sum_{List}(us)$

- Summation für Binärbäume natürlicher Zahlen:

  $sum_{Bin}(Tip\ u) = u$
  $sum_{Bin}(Join\ x\ y) = sum_{Bin}(x) + sum_{Bin}(y)$

Ein Beispiel für den Sinn und Zweck generischen Programmierens (3)

Viel praktischer wäre es doch, für Listen und Binärbäume (und andere Datentypen) nur einen Algorithmus schreiben zu müssen!

Vorteile:

a) Wiederverwendbarkeit

b) Zuverlässigkeit

Voraussetzungen für generisches Programmieren

Damit generisches Programmieren möglich ist, muß eine Sprache folgendes bieten:

a) Die Möglichkeit, Algorithmen generisch, d.h. unabhängig vom Datentyp, zu definieren.

b) Einen Mechanismus, um solche Algorithmen auf eine beliebige Datenstruktur anzuwenden.

Gemeinsamkeiten von Datentypen

Wie können diese Voraussetzungen erfüllt werden ?

Die beiden Beispieldatentypen haben einige Gemeinsamkeiten.

a) Liste natürlicher Zahlen:
data List Nat = Nil | Cons Nat (List Nat)

b) Binärbaum Natürlicher Zahlen:
data Bin Nat = Tip Nat | Join (Bin Nat) (Bin Nat)

Datentypen als Algebren (1)
Was ist eine Algebra ?

Eine Algebra ist eine Menge und einige Operatoren/Funktionen auf dieser Menge, die Elemente der Menge zurückgeben.

Ein Beispiel:

(Nat, 0, +), wobei: $0 :: 1 \rightarrow Nat$, $+ :: Nat \times Nat \rightarrow Nat$

Datentypen als Algebren (2)

Rekursive Datentypen lassen sich auf natürliche Weise als Algebren auffassen:

data List Nat = Nil | Cons Nat (List Nat)

(List Nat, Nil, Cons),
wobei:
Nil :: 1 → List Nat,
Cons :: Nat x List Nat → List Nat

Homomorphismen (1)

Sowohl die natürlichen Zahlen als auch die Listen gehören zur selben Klasse von Algebren:

(A, e, f), wobei: $e :: 1 \rightarrow A$, $f :: B \times A \rightarrow A$

Funktionen zwischen Trägermengen von Algebren derselben Klasse, die „die Struktur der Klasse respektieren“ heißen Homomorphismen.

Homomorphismen (2)
Bedeutung von „respektiert die Struktur“

Für die Klasse unserer Beispiele:

(A, e, f), wobei: $e :: 1 \rightarrow A$, $f :: B \times A \rightarrow A$

... bedeutet „Homomorphismus“ bzw. „respektieren der Struktur“ folgendes:

$h :: (A_0, e_0, f_0) \rightarrow (A_1, e_1, f_1)$

$h(e_0) = e_1$

$h(f_0(x, y)) = f_1(x, h(y))$

Homomorphismen (3)
Das Beispiel „Summation“

Ein Beispiel für einen Homomorphismus zwischen List Nat und Nat ist die Summation:

sum :: (List Nat, Nil, Cons) → (Nat, 0, +)

sum(Nil) = 0

sum(Cons(x, y)) = x + sum(y)

Homomorphismen (4)
Eindeutigkeit

Eine Besonderheit der Algebra-Klasse:

(A, e, f), wobei: $e :: 1 \rightarrow A$, $f :: B \times A \rightarrow A$

... ist, daß Homomorphismen zwischen Algebren dieser Klasse eindeutig sind, d.h. z.B.:

Der Algorithmus, der Listen natürlicher Zahlen aufsummiert, ist eindeutig bestimmt (und läßt sich also „generisch“ erzeugen).

F-Algebren
(zur Formalisierung)

Formal können Algebren mit Hilfe sog. „Funktoren“ beschrieben werden. Man spricht dann von „F-Algebren“.

Definition:

Sei F ein Funktor. Eine F-Algebra ist dann ein Paar (A, α), so daß $\alpha \in F\ A \rightarrow A$.

Wie kann α eine Menge von verschiedenstelligen Operatoren bezeichnen ? Antwort: s. Funktor!

Definition „Funktor“

„Funktor“ ist ein Begriff aus der Kategorientheorie. Für unsere Zwecke genügt folgende Definition:

Ein Funktor ist ein Abbildungspaar (F, map_F).

F bildet ein n-tupel von Typen a_0, ..., a_n
auf einen Typ F a_0 a_n ab und

map_F bildet ein n-Tupel von Funktionen f_0, ..., f_n
auf eine Funktion F f_0 ... f_n ab.

Dabei sollen Typisierung, Komposition und Identität erhalten bleiben.

Datentypen als Funktoren (1)
F

Wie paßt die Definition eines Funktor mit unseren Datentypen zusammen ?

a) Der Datentyp selbst ist die Abbildung F auf Typen, z.B. F = List:

List a = Nil | Cons a List a

List bildet den Typ a auf den Typ List a ab!

Datentypen als Funktoren (2)
map_F

b) Die Funktion $map_{Datentyp}$ ist die Abbildung map_F auf Funktionen, z.B. $map_F = map_{List}$:

$map_{List} :: (a \rightarrow b) \rightarrow (List\ a \rightarrow List\ b)$

Unäre Funktoren
Typisierung bleibt erhalten

Wir beschränken uns auf unäre Funktoren, da alle unsere Beispiele unäre parametrisierte Datentypen sind (List a, Bin a).

Daß die Typisierung erhalten bleiben soll, bedeutet:

$$\frac{f :: a \to b}{map_F f :: Fa \to Fb}$$

Unäre Funktoren
Komposition und Identität bleibt erhalten

Daß die Komposition erhalten bleiben soll, bedeutet:

$$map_F(f \bullet g) = (map_F\ f) \bullet (map_F\ g)$$

Daß die Identität erhalten bleiben soll, bedeutet:

$$map_F\ id = id$$

Bemerkung: Der Bezeichner „F“ wird normalerweise mit F und map_F „überladen“, d.h. „F“ bezeichnet sowohl die Abbildung F als auch die Abbildung map_F.

Polynomial Functors (1)
(„Mehrnamige Funktoren“)

Ein Funktor wie List:

List = Nil | Cons a List a

... ist „mehrnamig“, d.h. er setzt sich aus verschiedenen einfacheren Funktoren zusammen.
Zum Beispiel lassen sich „|“, „a“ oder „List a“ als einfache Funktoren beschreiben.

Einfache Funktoren
Identitätsfunktor und Konstanter Funktor

Identitätsfunktor:

$$Id\ a = a$$

Konstanter Funktor:

$$Const_x\ a = x$$

Einfache Funktoren
Par und Rec

Par und Rec sind zwei „Extraktionsfunktoren“. Sie extrahieren aus einem Parameter-Paar einen der beiden Parameter:

```
Par a b = a

Rec a b = b
```

Einfache Funktoren
Summenfunktor (1)

Der Name des „Summenfunktors“ ist etwas irreführend, es handelt sich im Grunde um eine Fallunterscheidung. Der Summenfunktor wird mit „+“ bezeichnet und infix geschrieben:

a + b = inl a | inr b

Der englische Begriff für den Summenfunktor ist „disjoint sum“.

Einfache Funktoren
Summenfunktor (2), Konstruktoren

Die beiden Konstruktoren „inl“ und „inr“ im Summenfunktor:

a + b = inl a | inr b

.. bezeichnen gewissermaßen die Herkunft einer Instanz des Typs „a + b“.

Einfache Funktoren
Summenfunktor (3), Beispiel für Konstruktoren

Sei a = Nat und b = Nat, dann ist a + b = Nat + Nat.

Gäbe es die Konstruktoren „inl“ und „inr“ nicht, wäre nicht festzustellen, ob beispielsweise die Instanz 3 ϵ Nat + Nat aus dem „linken“ oder dem „rechten“ Nat kommt.

Da es die Konstruktoren gibt, heißt eine Instanz entweder „inl 3“ oder „inr 3“, womit die Herkunft klar ist.

Einfache Funktoren
Summenfunktor (4), Abbildung auf Funktionen

Beim Summenfunktor ist insbesondere auch die Abbildung auf Funktionen interessant:

f + g = h, wobei:

h(inl u) = inl f(u)

h(inr v) = inr g(v)

Bedeutung: Fallunterscheidung!

Einfache Funktoren
Produktfunktor

Der Produktfunktor sieht wie folgt aus:

$a * b = (a, b)$

$f * g = h$, wobei:
$h(u,v) = (f(u), g(v))$

Der Prudktfunktor bildet aus einem gegebenen Paar ein neues Paar, indem er f auf die erste Komponente und g auf die zweite Komponente abbildet.

Komposition von Funktoren (1)

Es ist möglich, Funktoren zu kombinieren, z.B. ist die Liste natürlicher Zahlen eine Komposition aus den Funktoren List und Nat: List Nat

Es ist auch möglich, Funktoren verschiedener Stelligkeit mit Hilfe von Verdoppelung oder Spezialisierung zu kombinieren.

Komposition von Funktoren (2)
Verdoppelung und Spezialisierung

Beispiel Verdoppelung: Der binäre Funktor „+“ wird mit dem Funktor „List“ durch Verdoppelung von „List“ kombiniert:

List + List

Beispiel Spezialisierung: „+“ wird mit „List“ kombiniert, aber das erste Argument von „+“ wird spezialisiert, z.B. auf den Einheitstyp 1:

1 + List

Polynomial Functors (2)

Ein Funktor, der nur aus den vorgestellten einfachen Funktoren durch Komposition von Funktoren entstanden ist, heißt „Polynomial Functor“.

Ein Beispiel ist der (rekursiv definierte) Datentyp Nat:

```
data Nat = zero | succ Nat
```

der sich auch schreiben läßt als:

Nat = 1 + Nat (unter Vernachlässigung der Konstruktoren)

Polynomial Functors (3)
Bedeutung für das generische Programmieren

Es ist möglich, für jeden einfachen Funktor zu definieren, was z.B. Summe für diesen bedeutet.

Damit läßt sich auch eine Funktion schreiben, die mit Hilfe dieser Definitionen bestimmt, was Summe für einen zusammengesetzen Funktor bedeutet.

Wenn nun Datentypen durch Funktoren eindeutig bestimmt werden, ist es so möglich, generische Summenfunktionen zu schreiben. Damit ist die erste Voraussetzung für generisches Programmieren erfüllt.

Pattern Functors (1) oder
Was hat das mit Fixpunkten zu tun ?

Jeder rekursiv definierte Datentyp ist der Fixpunkt eines sog. „pattern functors“. Ein pattern functor ist eine „Funktion“, die einen bestimmten Datentyp liefert, wenn sie diesen Datentyp als Eingabe erhält.

(Die eher philosophische Bedeutung eines pattern functor ist, daß er einen Datentyp auf eindeutige Weise bestimmt.)

Pattern Functors (2)
Beispiel 1: Natürliche Zahlen

Definition der natürlichen Zahlen:

$$\text{Nat} = 1 + \text{Nat}$$

Der zugehörige pattern functor lautet:

$$N\ z = 1 + z, \text{ denn:}$$

$$N\ \text{Nat} = 1 + \text{Nat} = \text{Nat}$$

Pattern Functors (3)
Beispiel 2: Listen

Definition der Listen in Funktorschreibweise:

List a = 1 + (a * List)

Der zugehörige pattern functor lautet:

L z a = 1 + (a * z), denn:

L List a = 1 + (a * List) = List

Der Körper des pattern functors gleicht also dem Körper der Datentypdefinition in Funktorschreibweise.

Pattern Functors (4)
Notationen

Während N den pattern functor von Nat und L den pattern functor von List bezeichnet, ist F die allgemeine Notation für pattern functors.

μF bezeichnet den Datentyp, z.B. μL = List.

Damit gilt: μF = F μF

Die Hilfsfunktion in_F (1)

Bei der Definition von Nat = 1 + Nat haben wir die Konstruktoren vernachlässigt:

Nat = zero 1 | succ Nat

Um für verschiedene Datentypen eine einheitliche Benennung der Konstruktoren zu haben, läßt sich auf generischem Wege aus jedem pattern functor eine Funktion in_F erzeugen, die die Konstruktoren „umbenennt“.

Die Hilfsfunktion in_F (2)

Tatsächlich macht die Funktion in_F aus einem Funktor einen Datentyp:

$$in_F :: F\ \mu F \rightarrow \mu F$$

Für den pattern functor N der natürlichen Zahlen lautet in_N:

$$in_N(\text{inl } u) = \text{zero } u$$
$$in_N(\text{inr } v) = \text{succ } v$$

Die Hilfsfunktion in_F (3) angewendet auf N

Am Beispiel der natürlichen Zahlen: N z = 1 + z

in_N(N Nat) = in_N(1 + Nat)

(wende Summenfunktor an: a + b = inl a | inr b)

in_N(1 + Nat) = in_N(inl 1 | inr Nat) = in_N(inl 1) | in_N(inr Nat)

(wende in_N an)

in_N(inl 1) | in_N(inr Nat) = zero 1 | succ Nat = Nat

Nochmal F-Algebren

Unsere Definition der F-Algebren war:

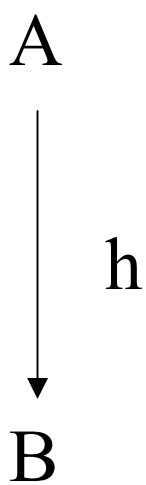
> Sei F ein Funktor. Eine F-Algebra ist dann ein Paar (A, α), so daß $\alpha \in F\ A \rightarrow A$.

Wenn F nun ein pattern functor ist, ist $\alpha = in_F$, z.B. für den pattern functor der natürlichen Zahlen N:

> Eine N-Algebra ist ein Paar (Nat, in_N).
> ($in_N \in N\ Nat \rightarrow Nat$, in_N = zero + succ)

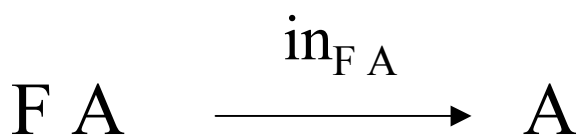
Homomorphismen für F-Algebren (1): Homomorphismen allgemein als Diagramm

Homomorphismen sind Abbildungen zwischen den Trägermengen von Algebren:



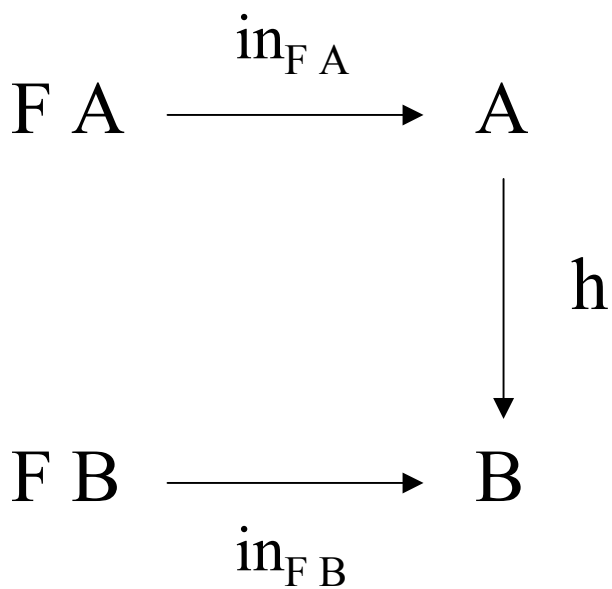
Homomorphismen für F-Algebren (2): F-Algebren als Diagramm

Die Trägermenge A einer F-Algebra wird beschrieben durch eine Funktion in_F, die als Eingabe den pattern functor F erhält:



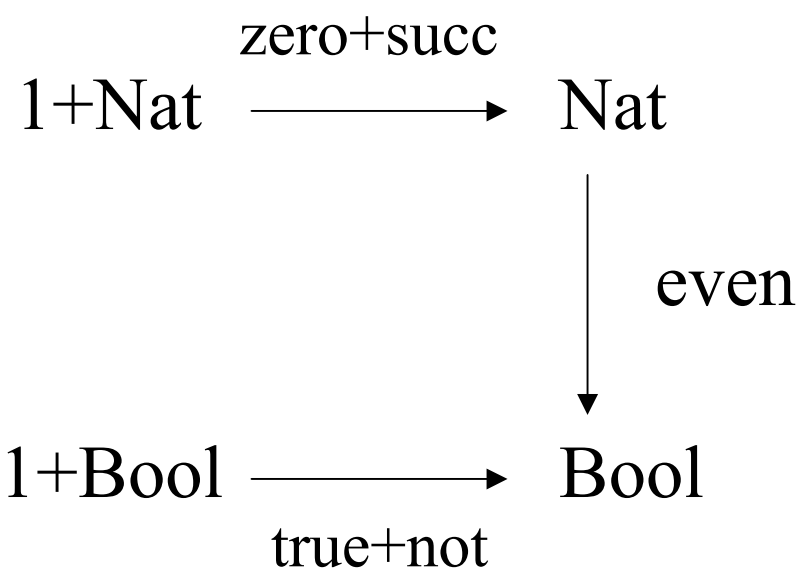
Homomorphismen für F-Algebren (3): Diagramm

Damit läßt sich ein Diagramm für einen Homomorphismus zwischen Trägermengen von F-Algebren wie folgt darstellen:



Homomorphismen für F-Algebren (4): Beispiel „even“ - Diagramm

Ein Beispiel ist der Homomorphismus „even“ zwischen den natürlichen Zahlen Nat und den Wahrheitswerten Bool:



Homomorphismen für F-Algebren (4): Definition von „even“

even läßt sich durch folgende Gleichungen bestimmen:

even(zero) = true

even(succ n) = not(even n)

zero ist die Funktion, die kein Argument nimmt und 0 liefert.

0 ist gerade, also ist even(zero) = even(0) = true

succ n liefert n + 1; Beispiel n = 0

even(succ 0) = not(even 0) = not(even zero)
= not(true) = false

Homomorphismen für F-Algebren (5): Ein beliebiger Homomorphismus h :: Nat → B

Abstrahieren wir von even :: Nat → Bool zu einem beliebigen Homomorphismus h :: Nat → B, mit $in_B = e_1 + f_1$. Die Gleichungen lauten dann:

$h(zero) = e_1$

$h(succ\ n) = f_1(h(n))$

Homomorphismen für F-Algebren (6): Ein beliebiger Homomorphismus h :: A → B

Abstrahieren wir außerdem von Nat zu einer Trägermenge einer beliebigen F-Algebra, so erhalten wir h:: A → B, $in_A = e_0 + f_0$.

Die Gleichungen, die h bestimmen, lauten nun:

$h(e_0) = e_1$

$h(f_0\ n) = f_1(\ h(n)\ )$

Diese Gleichungen müssen gelten, damit h ein Homomorphismus zwischen Trägermengen von F-Algebren ist! („die Struktur der Klasse respektiert“)

Homomorphismen für F-Algebren (7): Einzeilige Darstellung der Homomorphismus-Bedingung

Mit Hilfe der Fallunterscheidung „+“ lassen sich die Gleichungen in einer Zeile darstellen:

$h(e_0) + h(f_0\ n) = e_1 + f_1(\ h(n)\ )$

oder umgeformt (und ohne das n):

$h(e_0 + f_0) = e_1 + f_1(\ h\ )$

Homomorphismen für F-Algebren (8): Homomorphismus-Bedingung umgeformt

$h(e_0 + f_0) = e_1 + f_1(h)$ läßt sich nochmals umformen:

Die Funktion in_F der Algebra mit der Trägermenge A ist definiert als: $in_F = e_0 + f_0$, also gilt:

$$h(in_F) = e_1 + f_1(h)$$

Zur Verdeutlichung, daß dieses in_F etwas mit A zu tun hat, nennen wir es $in_{F\,A}$:

$$h(in_{F\,A}) = e_1 + f_1(h)$$

Homomorphismen für F-Algebren (9): Homomorphismus-Bedingung umgeformt

Läßt sich die rechte Seite von $h(in_{F\,A}) = e_1 + f_1(h)$ weiter umformen ?

Auch die Algebra mit der Trägermenge B hat eine Funktion in_F, nämlich $in_{F\,B}$:

$in_{F\,B} = e_1 + f_1$

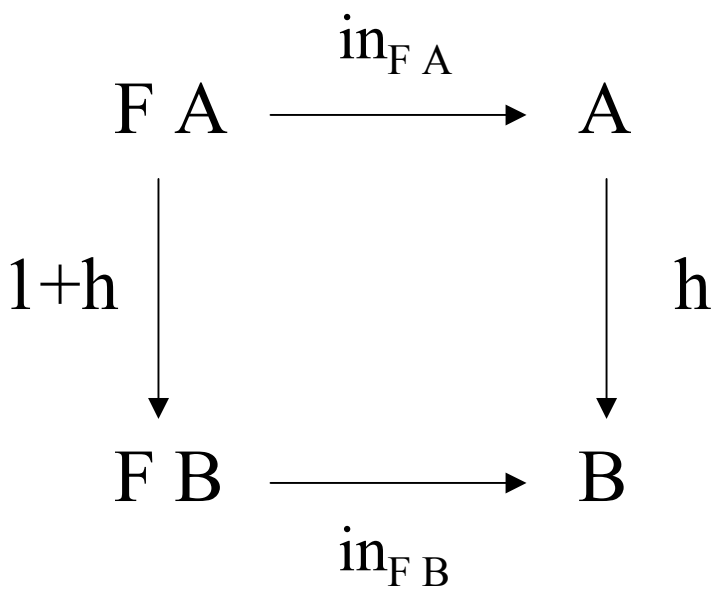
$in_{F\,B}(u) = e_1\ u$
$in_{F\,B}(v) = f_1\ v$

D.h. $in_{F\,B}(1 + h) = e_1 + f_1(h)$!

Homomorphismen für F-Algebren (10): Homomorphismus-Bedingung im Diagramm

Unsere Gleichung lautet inzwischen: $h(in_{F\,A}) = in_{F\,B}(1 + h)$

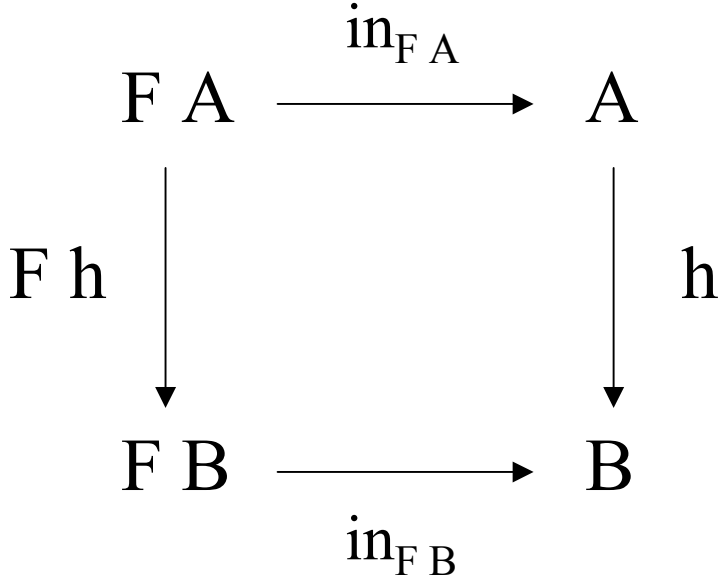
Damit können wir das Diagramm vervollständigen:



Homomorphismen für F-Algebren (11): Ein beliebiger Homomorphismus h :: B → A

Wenn wir definieren F h = 1 + h, erhalten wir die endgültige Form unserer Gleichung:

$$h(in_{F\ A}) = in_{F\ B}\ F\ h$$



Typbetrachtungen zur h-Bedingung

$$h\ in_{F\ A} = in_{F\ B}\ F\ h$$

Der Eingabewert von h ist offenbar der Ausgabewert von $in_{F\ A} :: F\ A \rightarrow A$, also ein Element der Menge A.

Der Ausgabewert von h wird durch $in_{F\ B}$ bestimmt, ist also ein Element der Menge B.

$$h :: A \rightarrow B$$

Eindeutige Bestimmung des Ausgabetyps von h durch $in_{F\,B}$

Wenn der Typ des Ausgabewertes von h durch:

$$in_{F\,B} = e_1 + f_1$$

bestimmt wird, wird er tatsächlich entweder durch die Funktion e_1 oder durch die Funktion f_1 bestimmt.

Ein solcher Homomorphismus, dessen Ausgabewert durch eine Funktion der Algebra seiner Wertemenge bestimmt wird, heißt Katamorphismus.

Katamorphismus

Eine andere Notation für den Katamorphismus h ist $([in_{F\ B}])$.

Dabei ist ([ ]) definiert als

$([\varphi]) = h$, wobei gilt: $h(in_F) = \varphi\ F\ h$

Katamorphismus
Bedeutung

Das heißt auf deutsch: Wenn ein Katamorphismus ([φ]) auf eine Struktur vom Typ A angewendet wird, wird er rekursiv auf die Komponenten der Struktur angewendet und die Ergebnisse werden kombiniert, indem sein „Körper“ φ auf diese angewendet wird.

Katamorphismus
Beispiel

Ein konkretes Beispiel für ([ ]) ist foldr für Listen.

foldr φ (x) wendet die Funktion φ zunächst auf das letzte und das vorletzte Element der Liste x an, dann auf das Resultat davon und das vorvorletzte Element der Liste usw.

foldr + (x) ist also die eingangs angesprochene Summation für Listen natürlicher Zahlen:

$sum_{List}(Nil) = 0$
$sum_{List}(Cons\ u\ us) = u + sum_{List}(us)$

([ ]) generisch

([ ]) ist also ein Mechanismus, um einen Algorithmus wie „+“ auf eine bestimmte Datenstruktur anzuwenden.

([ ]) läßt sich generisch erzeugen. Eine entsprechende Funktion greift auf den pattern functor eines Datentyps zurück und liefert ([ ]) für diesen Datentyp zu erzeugen.

Damit ist die zweite Bedingung für generisches Programmieren erfüllt.

Definition des Katamorphismus h

Definiert man eine zu in_F gegenteilige Funktion out_F, so läßt sich statt:

$$h(in_F) = \varphi \; F \; h$$

... folgendes schreiben:

$$h = \varphi \; F \; h \; out_F$$

Definition des Katamorphismus h anschaulich

Eine anschaulichere Schreibweise ist:

$$h = \varphi \;.\; \text{map id } h \;.\; \text{out}$$

Dabei ist φ der generische Algorithmus und map der Mechanismus, um φ auf eine bestimmte Datenstruktur anzuwenden.

Zusammenfassung

a) Rekursive Datentypen sind Fixpunkte ihres pattern functors und damit eindeutig bestimmt.

b) Ein pattern functor läßt sich auf einfache Funktoren zurückführen.

c) Mit Hilfe der einfachen Funktoren ist es möglich, sowohl Algorithmen als auch Mechanismen für deren Anwendung auf Datenstrukturen generisch zu erzeugen.

Ausblick (1)

a) Es gibt Datentypen, wie etwa den Rose Tree:

data Rose a = fork a (List(Rose a)),

die nicht durch polynomical functors beschrieben werden können.

Man kann aber sog. Typenfunktoren definieren, die zusammen mit den polynomical functors die Regulären Funktoren bilden. Generische Algorithmen müssen um die Verarbeitung von Typenfunktoren (neben einfachen Funktoren) erweitert werden.

Ausblick (2)

b) Die Komposition eines Homomorphismus mit einem Katamorphismus ist wieder ein Katamorphismus.

   Damit ist es möglich, eine gegebene Funktion mit einem gegebenen Katamorphismus zu verbinden, z.B. um die Effizienz zu erhöhen.

Ausblick (3)

c) Die vorgestellten Konzepte lassen sich praktisch z.B. in Haskell verwenden. Für Haskell gibt es eine Erweiterung namens PolyP, die die Definition generischer Algorithmen (und deren Anwendung) ermöglicht.