

MONADEN

von der Theorie zur Praxis

Frank Rosemeier

27. Mai 2004
3. Juni 2004

Gliederung

I. Theorie

II. Praxis

I. Theorie

Inhalt von Teil I

Monadologie

Kategorientheorie

Monoïde

Kategorielle Betrachtung

Monaden

Kleisli-Tripel

Kombinatorische Logik

Terme

Typen

Gottfried Wilhelm Leibniz

Monadologie

[Leibniz 1720]

Monadologie, Titel

Monadologie

Des Herrn Baron von Leibnitz

Lehr-Sätze von den Monaden, von der Seele des Menschen,
von seinem Systemate harmoniae praestabilitae zwischen der
Seele und dem Körper, von GOTT, seiner Existenz, seinen
anderen Vollkommenheiten und von der Harmonie zwischen
dem Reiche der Natur und dem Reiche der Gnade.

Monadologie, Beginn

1. Die *Monaden*, wovon wir hier reden werden, sind nichts anderes als *einfache* Substanzen, woraus die zusammengesetzten Dinge oder composita bestehen. Unter dem Wort *einfach* verstehtet man dasjenige, welches keine Teile hat.
2. Es müssen dergleichen einfache Substanzen sein, weil composita vorhanden sind; denn das *Zusammengesetzte* ist nichts anderes als eine Menge oder ein Aggregat von einfachen Substanzen.
3. Wo nun keine Teile vorhanden sind, daselbst kann auch weder eine Ausdehnung in die Länge, Breite und Tiefe, noch eine Figur, noch ein Zerteilung möglich sein. Und diese Monaden sind die wahrhaften Atomi der Natur und mit einem Worte die *Elemente der Dinge*.

Monadologie, Fußnote

Das Wort *Monade*, oder *Monas*, hat bekanntermaßen seinen Ursprung im Griechischen, und bedeutet eigentlich *Eines*. Man hat das Wort behalten, weil man vornehme Gelehrte zu Vorgänger hat, die die Kunst-Wörter der Kürze wegen behalten und mit einer teutschen Endigung nach der Gewohnheit der Engelländer und Franzosen gleichsam naturalisieren. Wenn man die Worte Serenaden, Cantaten, Elemente und dergleichen unzählige mehr in der teutschen Sprache beibehält, ohngeachtet es fremde Wörter sind; so habe ich geglaubet, daß es nicht inconvenient gehandelt sei, wenn ich mich um der Kürze willen des Worts, *Monade*, und anderer dergleichen Kunst-Wörter bediene. Viele Dinge scheinen Anfangs ungereimet, weil sie noch nicht gewöhnlich sind; ich halte aber davon, daß das ungewöhnliche, wenn es eine vernünftige Ursache zum Grunde hat, nicht für ungereimt könne gehalten werden.

Kategorientheorie

Bezeichnungen für Monaden

„Standard-Konstruktionen“ [Godement 1958]

„Tripel“ [Eilenberg/Moore 1965]

„Monaden“ [Mac Lane 1971]

Monoide, Definition

Definition.

Ein *Monoid* (M, e, m) besteht aus

- einer Menge M ,
 - einem Element $e \in M$,
 - einer Multiplikationsabbildung $m : M \times M \longrightarrow M$,

mit folgenden Bedingungen (wir schreiben xy für $m(x, y)$):

$$(M1) \quad ex = x = xe \quad (\text{neutrales Element})$$

$$(M2) \quad x(yz) = (xy)z \quad (\text{Assoziativgesetz})$$

für alle $x, y, z \in M$.

Monoide, Beispiel

Beispiel.

Für jede Menge A ist die Menge aller Listen

$$\mathcal{L}A := \{ [a_1, \dots, a_n] \mid a_1, \dots, a_n \in A, n \geq 0 \}$$

mit der Verknüpfung

$$[a_1, \dots, a_n] [b_1, \dots, b_m] := [a_1, \dots, a_n, b_1, \dots, b_m]$$

und der leeren Liste $[]$ als neutralem Element ein Monoid, das *Listen-Monoid*.

Bijektionen

Hinweis.

Wir identifizieren kartesische Produkte von Mengen mit Hilfe folgender Bijektionen

$$\{1\} \times M \cong M \cong M \times \{1\},$$

$$(1, x) \leftrightarrow x \leftrightarrow (x, 1),$$

$$(M \times M) \times M \cong M \times (M \times M),$$

$$((x, y), z) \leftrightarrow (x, (y, z)).$$

Monoide, Diagramm

Notiz.

Für jedes Monoid (M, e, m) ist

$$\begin{array}{ccccc}
 M \times M \times M & \xrightarrow{m \times 1_M} & M \times M & \xleftarrow{e \times 1_M} & M \\
 \downarrow 1_M \times m & & \downarrow m & & \downarrow 1_M \times e \\
 M \times M & \xrightarrow{m} & M & \xleftarrow{m} & M \times M
 \end{array}$$

ein kommutatives Diagramm mit

$$(m \times 1_M)(x, y, z) = (xy, z), \quad (e \times 1_M)x = (e, x)$$

$$(1_M \times m)(x, y, z) = (x, yz), \quad (1_M \times e)x = (x, e)$$

für alle $x, y, z \in M$.

Kategorische Betrachtung, Funktor

Feststellung.

Für jedes Monoid (M, e, m) haben wir einen **Funktor**

$$T : \text{Set} \longrightarrow \text{Set},$$

$$TA := M \times A,$$

$$Tf := M \times f \quad \text{für } f : A \longrightarrow B,$$

wobei $M \times f : M \times A \longrightarrow M \times B$ durch

$$(M \times f)(x, a) := (x, f(a))$$

definiert ist.

Kategorische Betrachtung, natürliche Transformationen

Bemerkung.

Für jedes Monoid (M, e, m) und jede Menge A liefern die Abbildungen

$$\eta_A : A \longrightarrow TA, \quad \eta_A(a) := (e, a)$$

$$\mu_A : TTA \longrightarrow TA, \quad \mu_A(x, (x', a)) := (xx', a)$$

für alle $a \in A, x, x' \in M$

natürliche Transformationen

$$\eta : \mathbf{1}_{\text{Set}} \longrightarrow T,$$

$$\mu : T \circ T \longrightarrow T.$$

Kategorische Betrachtung, Funktordiagramm

Beobachtung.

Für jedes Monoid (M, e, m) ist das folgende Diagramm

$$\begin{array}{ccccc}
 T \circ T \circ T & \xrightarrow{\mu \circ T} & T \circ T & \xleftarrow{\eta \circ T} & T \\
 T \circ \mu \downarrow & & \mu \downarrow & & \downarrow T \circ \eta \\
 T \circ T & \xrightarrow{\mu} & T & \xleftarrow{\mu} & T \circ T
 \end{array}$$

$\mathbf{1}_T$ ↗

kommutativ (mit den zuvor definierten T, η, μ).

Monaden, Definition

Definition.

Eine **Monade** (T, η, μ) auf einer Kategorie \mathcal{C} besteht aus

- einem Funktor $T : \mathcal{C} \rightarrow \mathcal{C}$,
- einer natürlichen Transformation $\eta : \mathbf{1}_{\mathcal{C}} \rightarrow T$,
- einer natürlichen Transformation $\mu : T \circ T \rightarrow T$,

die folgendes Diagramm

$$\begin{array}{ccccc}
 T \circ T \circ T & \xrightarrow{\mu \circ T} & T \circ T & \xleftarrow{\eta \circ T} & T \\
 T \circ \mu \downarrow & & \downarrow \mu & & \downarrow T \circ \eta \\
 T \circ T & \xrightarrow{\mu} & T & \xleftarrow{\eta} & T \circ T
 \end{array}$$

1_T 

kommutativ machen.

Monaden, Beispiel

Beispiel.

Der **Listen-Funktor** $\mathcal{L} : \text{Set} \longrightarrow \text{Set}$ ist definiert durch

$$\mathcal{L}A = \{ [a_1, \dots, a_n] \mid a_1, \dots, a_n \in A, n \geq 0 \}$$

$$\mathcal{L}f : \mathcal{L}A \longrightarrow \mathcal{L}B \quad \text{für } f : A \longrightarrow B,$$

$$\mathcal{L}f([a_1, \dots, a_n]) := [f(a_1), \dots, f(a_n)].$$

Mit den durch

$$\eta_A : A \longrightarrow \mathcal{L}A, \quad \eta_A(a) := [a],$$

$$\mu_A : \mathcal{L}\mathcal{L}A \longrightarrow \mathcal{L}A, \quad \mu_A([l_1, \dots, l_n]) := l_1 \cdots l_n,$$

definierten natürlichen Transformationen η und μ ist (\mathcal{L}, η, μ) eine Monade, die **Listen-Monade**.

Kleisli-Tripel, Definition

Definition.

Ein *Kleisli-Tripel* $(T, \eta, *)$ über einer Kategorie \mathcal{C} besteht aus

- einer Abbildung $T : \text{Obj } \mathcal{C} \longrightarrow \text{Obj } \mathcal{C}$,
- einer Familie $(\eta_A)_{A \in \text{Obj } \mathcal{C}}$ von Morphismen $\eta_A \in \text{Mor}_{\mathcal{C}}(A, TA)$,
- Abbildungen $* : \text{Mor}_{\mathcal{C}}(A, TB) \longrightarrow \text{Mor}_{\mathcal{C}}(TA, TB)$, $f \mapsto f^*$,

die folgende Bedingungen erfüllen

$$(K1) \quad \eta_A^* = \mathbf{1}_{TA} ,$$

$$(K2) \quad f^* \circ \eta_A = f ,$$

$$(K3) \quad g^* \circ f^* = (g^* \circ f)^* ,$$

für alle $f \in \text{Mor}_{\mathcal{C}}(A, TB)$, $g \in \text{Mor}_{\mathcal{C}}(B, TC)$, $A, B, C \in \text{Obj } \mathcal{C}$.

Kleisli-Tripel, Proposition

Proposition.

(1) Zu jeder Monade (T, η, μ) ist $(T, \eta, *)$ ein Kleisli-Tripel mit

$$f^* := \mu_b \circ Tf \quad \text{für } f \in \text{Mor}_{\mathcal{C}}(A, TB), A, B \in \text{Obj } \mathcal{C}.$$

(2) Zu jedem Kleisli-Tripel $(T, \eta, *)$ erhalten wir eine Monade (T, η, μ) mit

$$Th := (\eta_B \circ h)^* \quad \text{für alle } h \in \text{Mor}_{\mathcal{C}}(A, B), A, B \in \text{Obj } \mathcal{C},$$

$$\mu_A := (\mathbf{1}_{TA})^* \quad \text{für alle } A \in \text{Obj } \mathcal{C}.$$

(3) Die Zuordnungen in (1) und (2) liefern eine *kanonische Bijektion* zwischen Monaden und Kleisli-Tripeln.

Kleisli-Tripel, Beispiel

Beispiel.

Bei dem Listen-Funktor $\mathcal{L} : \text{Set} \longrightarrow \text{Set}$ haben wir

$$f^*([a_1 \dots, a_n]) = f(a_1) \dots f(a_n)$$

für $f : A \longrightarrow \mathcal{L}B$, A, B Mengen.

Kombinatorische Logik

Terme

Terme, Kombinatoren

Definition.

Terme der kombinatorischen Logik sind

- (i) die *Konstanten* K und S
(und gegebenenfalls weitere Konstanten),
- (ii) *Variablen* wie f, g, h, x, y, z ,
- (iii) *Kombinationen* (MN) von Termen M und N .

Ein Term ohne Variablen wird *Kombinator* genannt
(sofern er keine anderen Konstanten enthält als K und S).

Kombinator-Gleichungen

Interpretation.

Versteh (MN) als Anwendung einer *Funktion M* auf ein *Argument N*.

Schreibweise.

Schreibe (LMN) für $((LM)N)$.

Äußere Klammern können wegfallen.

Grundgleichungen.

(K) $Kxy = x$, Konstanzfunktion,

(S) $Sxyz = (xz)(yz)$, Verschmelzungsfunktion.

Einige Kombinatoren

Problem.

Finde Kombinatoren I, B, E mit

- (I) $Ix = x$, *Identitätsfunktion* ,
- (B) $Bfgx = f(gx)$, *Kompositionsfunktion* ,
- (E) $Exy = yx$, *Einsetzungsfunktion* .

Lösung.

Folgende Kombinatoren haben die gewünschten Eigenschaften

$$\text{I} := \text{SKK} ,$$

$$\text{B} := \text{S(KS)K} ,$$

$$\text{E} := \text{B(SI)K} .$$

Funktionale Vollständigkeit

Satz (Funktionale Vollständigkeit).

Für jeden Term M und jede Variable x gibt es einen Term F mit $Fx = M$, wobei x in F nicht vorkommt.

Folgerung.

Für jeden Term M und für alle paarweise verschiedenen Variablen x_1, \dots, x_n gibt es einen Term G mit $Gx_1 \dots x_n = M$, wobei x_1, \dots, x_n in G nicht vorkommen.

Beweis

Beweis der funktionalen Vollständigkeit.

Gegeben sind M und x , gesucht F mit $Fx = M$.

- (i) Wenn x nicht in M vorkommt, setze $F := KM$.
- (ii) Gilt $M = x$, so setze $F := I$.
- (iii) Für $M = (M_1 M_2)$ mit $M_1 = F_1 x$ und $M_2 = F_2 x$ setzen wir $F := SF_1 F_2$.



λ -Notation

Definition (λ -Notation).

$\lambda x.x := I$,

$\lambda x.M := KM$, wenn x nicht in M vorkommt,

$\lambda x.(M_1M_2) := SM_1M_2$, wenn x in (M_1M_2) vorkommt.

Interpretation.

$\lambda x.M$ ist eine Funktion, die jedem Argument x den Wert M zuordnet (*Funktionsabstraktion*).

Typen

Typenschemata

Definition.

Typenschemata der kombinatorischen Logik sind

- (i) gegebenenfalls gewisse *Typkonstanten* (wie Bool, Float, Integer),
- (ii) *Typenvariablen* wie α, β, γ ,
- (iii) *Funktionsschemata* $(\sigma_1 \rightarrow \sigma_2)$ mit Typenschemata σ_1, σ_2 .

Ein Typenschema ohne Typvariablen wird *Typ* genannt.

Instanzen

Definition.

Instanzen eines Typenschemas σ entstehen aus σ durch Einsetzen gewisser Typenschemata für die Typenvariablen.

Beispiele.

$\alpha \rightarrow (\alpha \rightarrow \alpha)$ ist Instanz von $\alpha \rightarrow (\beta \rightarrow \gamma)$,

$\alpha \rightarrow ((\beta \rightarrow \alpha) \rightarrow \alpha)$ ist Instanz von $\alpha \rightarrow (\beta \rightarrow \alpha)$,

$(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$ ist Instanz von $\alpha \rightarrow \alpha$.

Interpretation für Typenschemata

Interpretation.

Typ τ \rightsquigarrow Menge A_τ von Termen

$M : \tau$ $\rightsquigarrow M \in A_\tau$

Term M hat Typ τ

Funktionstyp $(\tau_1 \rightarrow \tau_2)$ \rightsquigarrow Typ der Funktionen mit Argumenten in τ_1 und Werten in τ_2

Typenschema σ $\rightsquigarrow \{\tau \mid \tau \text{ ist Instanz von } \sigma\}$

$M : \sigma$ $\rightsquigarrow M : \tau$ gilt für alle

Term M hat Typenschema σ Instanzen τ von σ

Axiome, Regel

Schreibweise.

Schreibe $(\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3)$ für $(\sigma_1 \rightarrow (\sigma_2 \rightarrow \sigma_3))$.

Äußere Klammern können wegfallen.

Axiome.

$$K : \alpha \rightarrow \beta \rightarrow \alpha ,$$

$$S : (\alpha \rightarrow \beta \rightarrow \gamma) \rightarrow (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma) .$$

Regel.

$$M : \sigma_1 \rightarrow \sigma_2 \text{ und } N : \sigma_1 \implies (MN) : \sigma_2$$

Beispiel

Beispiel.

I : $\alpha \rightarrow \alpha$, denn

$$S : (\alpha \rightarrow (\beta \rightarrow \alpha) \rightarrow \alpha) \rightarrow (\alpha \rightarrow (\beta \rightarrow \alpha)) \rightarrow (\alpha \rightarrow \alpha)$$

$$K : \alpha \rightarrow (\beta \rightarrow \alpha) \rightarrow \alpha$$

$$\implies SK : (\alpha \rightarrow (\beta \rightarrow \alpha)) \rightarrow (\alpha \rightarrow \alpha) ,$$

$$K : \alpha \rightarrow (\beta \rightarrow \alpha)$$

$$\implies SKK : \alpha \rightarrow \alpha ,$$

und $SKK = I$.



Typinferenz

Bemerkung.

SII hat gar kein Typenschema ($\text{SII}x = xx$).

Satz [Hindley 1969].

- (1) Es gibt ein Entscheidungsverfahren um festzustellen, ob ein Kombinator C ein Typenschema hat oder nicht.
- (2) Gegebenenfalls lässt sich zu C ein allgemeinstes Typenschema σ berechnen, das heißt wir haben $C : \sigma$, und $C : \sigma'$ gilt nur für Instanzen σ' von σ .

II. Praxis

Inhalt von Teil II

Funktionales Programmieren

Begriff

Funktionale Programmiersprachen

Haskell

Werte

Typen

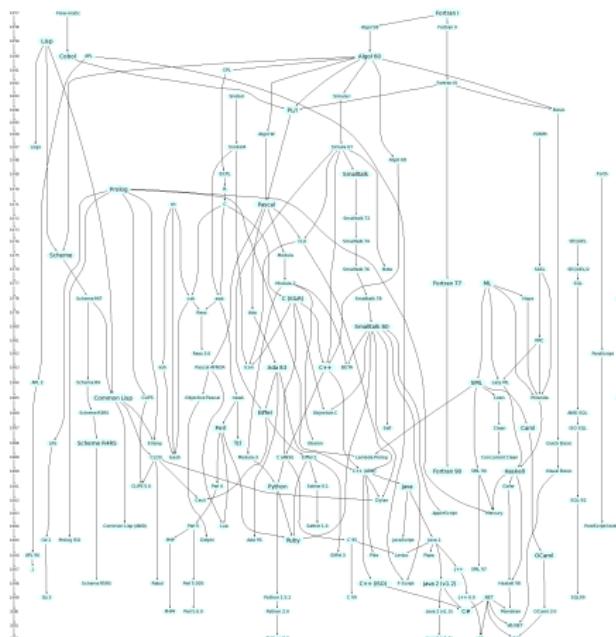
Typenklassen

Monaden

Programmier-Beispiele

Funktionales Programmieren

Programmiersprachen



Begriff

Imperatives Programmieren ist ein Programmierstil, der die *Ausführung von Befehlen* betont.

Funktionales Programmieren ist ein Programmierstil, der die *Auswertung von Ausdrücken* betont; die Ausdrücke werden dabei unter Verwendung von *Funktionen* aus grundlegenden *Werten* kombiniert.

Funktionale Programmiersprachen unterstützen und ermutigen das Programmieren im funktionalen Stil.

[Hutton 2002]

Beispiel

Aufgabe.

Berechne die Summe der ganzen Zahlen von 1 bis 10.

Imperative Lösung.

C-Programm : summe = 0;
 for (i=1; i<=10; i++)
 summe += i;

Funktionale Lösung.

Haskell-Programm : sum [1..10]

Wurzeln

Kombinatorische Logik [Schönfinkel 1924] [Curry 1929] ...

Lambda-Kalkül [Church 1932–33] [Church 1941]

Typentheorie [Church 1940] [Hindley 1969]

Programmierung [Milner 1978] [Backus 1978]



“Can programming be liberated from the von Neumann style?”

Funktionale Programmiersprachen

1960	Lisp (John Mc Carthy) Dialekte: Common Lisp Scheme	<i>Listen</i> Künstliche Intelligenz
1978	FP (John Backus)	<i>Funktionen höherer Ordnung</i>
1978	ML (Robin Milner et al.) Dialekte: Standard ML Caml	<i>Typinferenz</i> Beweisassistenten: Isabelle, Nuprl, Coq
1985	Miranda_{TM} (David Turner)	<i>nicht-strikt</i>
1988	Erlang (Ericsson)	Telekommunikation
1990	Haskell	<i>Typenklassen, Monaden, rein funktional</i>

Haskell B. Curry



Haskell, Entwicklung

- 1987** Conference on Functional Programming Languages and Computer Architecture in Portland, Oregon: **Komitee** zum Entwurf einer funktionalen Standardsprache wird eingesetzt.

1990 Haskell Version 1.0 (benannt nach Haskell B. Curry)

1991 Haskell **Version 1.1** (Typenklassen)

1992 Haskell **Version 1.2** (polymorphe Klassenmethoden)

1996 Haskell **Version 1.3** (Typkonstruktorklassen, mit Monaden und monadischer do-Notation)

1997 Haskell **Version 1.4** (mit Monaden, mit „monadischer Mengen-Notation“)

1998 **Haskell 98** (mit Monaden, ohne „monadische Mengen-Notation“) **2002** überarbeitet

Haskell: Grundsätzliches

Grundsätzliches.

- ## 1. Haskell ist eine *rein funktionale Programmiersprache* :

Alle Berechnungen kommen durch *Auswertung* von Ausdrücken zustande, deren Ergebnis Werte sind.

2. In Haskell sind auch *Funktionen als Werte* uneingeschränkt zugelassen.

Insbesondere können *Funktionen höherer Ordnung* verwendet werden, also Funktionen, deren Argumente oder Ergebniswerte selbst Funktionen sind.

- ### 3. Haskell ist *statisch typisiert* :

Jeder Wert besitzt einen Typ, der vor Ausführung des Programmes bestimmt werden kann.

Haskell

Werte

Haskell-Werte

Primitive Werte.

- 6 ganzzahliger Wert
- 3.14 Gleitkomma-Wert
- 2.0E-5 Gleitkomma-Wert
- 'q' Schriftzeichen-Wert

Zusammengesetzte Werte.

- [7,8] Liste ganzer Zahlen
- ['x','y','z'] Liste von Schriftzeichen
- (9,10.0) Paar aus ganzer Zahl und GleitkommaZahl
- (11.0,'B',[13]) Tripel aus Gleitkomma-Zahl, Schriftzeichen und Liste ganzer Zahlen

Haskell-Funktionen

Definition durch Gleichungen.

inc n	=	n+1	Nachfolgerfunktion
constfour x	=	4	konstante Funktion
id y	=	y	identische Funktion
fst (x,y)	=	x	Projektionsfunktion

Definition mit λ -Notation.

inc	=	$\lambda n \rightarrow n+1$	$n \mapsto n + 1$
constfour	=	$\lambda x \rightarrow 4$	$x \mapsto 4$
id	=	$\lambda y \rightarrow y$	$y \mapsto y$
fst	=	$\lambda (x,y) \rightarrow x$	$(x,y) \mapsto x$

Funktionsanwendung

Funktionsanwendung.

inc 6	liefert den Wert	7
constfour 6	liefert den Wert	4
constfour 3.14	liefert den Wert	4
id 6	liefert den Wert	6
id 3.14	liefert den Wert	3.14
fst (9,10.0)	liefert den Wert	9

Funktionen höherer Ordnung

Funktionen als Werte.

`add r s = r+s`

$\rightsquigarrow \text{add} = \lambda r\ s \rightarrow r+s := \lambda r \rightarrow (\lambda s \rightarrow r+s)$

`const x y = x`

$\rightsquigarrow \text{const} = \lambda x\ y \rightarrow x := \lambda x \rightarrow (\lambda y \rightarrow x)$

Funktionen als Argumente.

`compose f g x = g(f x)`

$\rightsquigarrow \text{compose} = \lambda f\ g\ x \rightarrow g(f\ x)$

`twice f = compose f f`

$\rightsquigarrow \text{twice} = \lambda f\ x \rightarrow f(f\ x)$

`flip f = \x y \rightarrow f y x`

$\rightsquigarrow \text{flip} = \lambda f\ x\ y \rightarrow f\ y\ x$

Typen

Haskell-Typen

Schreibweise.

`w :: T` besagt, dass der Ausdruck `w` den Typ `T` hat.

Beispiele.

<code>6 :: Integer</code>	ganzzahliger Typ
<code>3.14 :: Float</code>	Gleitkomma-Typ
<code>'q' :: Char</code>	Schriftzeichen-Typ
<code>[7,8] :: [Integer]</code>	Listen-Typ
<code>(9,10.0) :: (Integer,Float)</code>	Paar-Typ
<code>inc :: Integer -> Integer</code>	Funktionstyp
<code>add :: Integer -> Integer -> Integer</code>	Funktionstyp höherer Ordnung

Polymorphe Haskell-Typen

Bemerkung.

Haskell-Typen können auch Typenvariablen enthalten, es handelt sich dann um *polymorphe Typen*.

Beispiele.

```
constfour  ::  a -> Integer
            id  ::  a -> a
            fst ::  (a,b) -> a
            const ::  a -> b -> a
            compose ::  (a -> b) -> (b -> c) -> (a -> c)
            twice  ::  (a -> a) -> (a -> a)
            flip   ::  (a -> b -> c) -> (b -> a -> c)
```

Typensynonyme

Typensynonyme werden durch Definition eines neuen Namens für häufig verwendete Typen eingeführt.

Beispiele.

```
type Alter          = Integer
type Name          = [Char]
type Person         = (Name, Alter)
type Datum          = (Integer, Integer, Integer)
type Liste a        = [a]
type Fun a b        = a -> b
type Paar a b       = (a, b)
type Tripel a b c  = (a, b, c)
```

Datentypen

Datentypen ermöglichen es, neue Typen durch Angabe von *Konstruktoren* einzuführen.

Beispiele.

```
data Farbe      = Blau | Gelb | Rot
```

```
data Jahreszeit = Fruehling
                | Sommer
                | Herbst
                | Winter
```

Vordefinierte Datentypen.

```
data Bool      = True | False
```

```
data Ordering  = LT | EQ | GT
```

Typkonstruktoren

Hinweis.

Datentypen können auch von Typvariablen abhängen, man spricht dann von *Typkonstruktoren*.

Beispiele.

```
data Punkt a = Pt a a  
             => Pt :: a -> a -> Punkt a
```

```
data Praedikat a = Praed (a -> Bool)  
                  => Praed :: (a -> Bool) -> Praedikat a
```

Typkonstruktoren

Vordefinierte Typkonstruktoren.

```
data Maybe a = Just a
              | Nothing
⇒ Just :: a -> Maybe a
Nothing :: Maybe a
```

```
data Either a b = Left a
                  | Right b
⇒ Left :: a -> Either a b
Right :: b -> Either a b
```

Rekursive Datentypen

Notiz.

Die Konstruktoren eines Datentyps dürfen Argumente haben, die von diesem Datentyp selber abhängen.

In diesem Fall spricht man von *rekursiven Datentypen*

Beispiele.

```
data Nat      = Zero
               | Succ Nat
⇒ Zero :: Nat
      Succ :: Nat -> Nat
```

```
data Tree a = Leaf a
               | Branch (Tree a) (Tree a)
⇒ Leaf   :: a -> Tree a
      Branch :: Tree a -> Tree a -> Tree a
```

Listen rekursiv definiert

Beobachtung.

Haskell-Listen verhalten sich wie die Werte des folgenden rekursiven Datentyps.

```
data List a = Nil
            | Cons a (List a)
⇒ Nil :: List a
Cons :: a -> List a -> List a
```

Notation.

`[]` steht für `Nil`

`x:l` steht für `Cons x l`

`[x1, x2, ..., xn]` steht für `x1: (x2: ... xn: Nil)`

Funktionen auf Datentypen

Definition durch Gleichungen.

not :: Bool \rightarrow Bool Negationsfunktion

not True = False

not False = True

pred :: Nat \rightarrow Nat Vorgängerfunktion

pred Zero = Zero

pred Succ n = n

Definition mit case-Ausdrücken.

not x = case x of True \rightarrow False
 False \rightarrow True

pred k = case k of Zero \rightarrow Zero
 Succ n \rightarrow n

Listen-Funktionen

Beispiele.

`append :: [a] -> [a] -> [a]` Monoid-Verknüpfung

`append Nil l' = l'`

`append (x:l) l' = x : (append l l')`

`map :: (a -> b) -> ([a] -> [b])` Funktorkfunktion

`map f Nil = Nil`

`map f (x:l) = (f x) : (map f l)`

`concat :: [[a]] -> [a]` Transformationsfunktion

`concat Nil = Nil`

`concat (x:l) = append x (concat l)`

Typenklassen

Ad-hoc-Polymorphismus

Philip Wadler, Stephen Blott:

“How to make ad-hoc polymorphism less ad hoc.”

[Wadler/Blott 1989]

Parametrischer Polymorphismus.

Polymorphe Funktionen haben eine *gleichartige Definition* für verschiedene Typen.

Ad-hoc-Polymorphismus.

Überladene Funktionen, die für mehrere „gleichartige“ Typen definiert sind, haben *verschiedene Definitionen* für verschiedene Typen.

Ad-hoc-Polymorphismus

Beispiel.

Parametrischer Polymorphismus

```
id :: a -> a
id :: Farbe -> Farbe    id :: Bool -> Bool
id x = x                  id x = x
```

Ad-hoc-Polymorphismus

```
eq :: Farbe -> Farbe -> Bool
eq Blau Blau      =  True
eq Gelb Gelb      =  True
eq Rot Rot      =  True
eq _ _           =  False

eq :: Bool -> Bool -> Bool
eq True True      =  True
eq False False     =  True
eq _ _           =  False
```

Haskell-Typenklassen

In Haskell kann Ad-hoc-Polymorphismus (das Überladen von Funktionen) durch einen *Typenklassen-Mechanismus* erreicht werden.

Bei der Definition einer *Typenklasse* die zugehörigen werden *Klassenmethoden* angegeben.

Beispiel.

```
class Eq a where
  eq      :: a -> a -> Bool
  noteq   :: a -> a -> Bool
```

Die Klassenmethoden haben den Typ

```
eq      :: Eq a => a -> a -> Bool
noteq  :: Eq a => a -> a -> Bool
```

(*eingeschränkter Polymorphismus*).

Klasseninstanzen

Um zu einen Typ als zu einer Klasse gehörig zu erkennen wird die betreffende *Instanz* der Klasse gebildet, indem für jede Methode der Klasse eine passende Definition angegeben wird.

Beispiel.

```
instance Eq Farbe where
  eq Blau Blau          =  True
  eq Gelb Gelb          =  True
  eq Rot Rot             =  True
  eq _ _                  =  False
  noteq x y              =  not (eq x y)
```

Typkonstruktor-Klassen

Seit Version 1.3 ist es in Haskell auch erlaubt **Klassen und Instanzen für Typkonstruktoren** zu definieren.

Beispiel.

```
class Functor f where
    fmap :: (a -> b) -> (f a -> f b)

instance Functor Maybe where
    fmap f Nothing    =  Nothing
    fmap f (Just x)   =  Just (f x)

instance Functor List where
    fmap = map
```

Warnung

Hinweis.

Die **Funktorgesetze**

(F1) $\text{fmap id} = \text{id}$

(F2) $\text{fmap} (\text{compose } g \text{ } f) = \text{compose} (\text{fmap } g) \text{ } (\text{fmap } f)$

werden von Haskell *nicht* überprüft .

Haskell-Monaden

Literaturhinweise

Eugenio Moggi Computational lambda-calculus and monads.

[Moggi 1989]

Philip Wadler Comprehending monads.

[Wadler 1992]

Philip Wadler Monads for functional programming.

[Wadler 1995]

Eine Monaden-Klasse

Nun können wir **Monaden** in Haskell definieren.

Beispiel.

```
class Monade m where
    unit :: a -> m a
    join :: m (m a) -> m a

instance Monade Maybe where
    unit = Just
    join Nothing = Nothing
    join (Just x) = x

instance Monade List where
    unit x = [x]
    join = concat
```

Warnung

Hinweis.

Die **Monadengesetze** werden **von Haskell *nicht* überprüft** .

Monaden in Haskell

Tatsächlich wird die *Monaden-Klasse* in Haskell eher wie ein Kleisli-Tripel definiert.

Definition.

```
class Monad m where
    bind   :: m a -> (a -> m b) -> m b
    bindconst :: m a -> m b -> m b
    return :: a -> m a
    fail   :: String -> m a
```

Maybe-Monade

```
instance Monad Maybe where

    bind Nothing f      =  Nothing
    bind (Just x) f    =  f x

    bindconst u v = bind u (const v)

    return = Just

    fail s = Nothing
```

List-Monade

```
instance Monad List where

    bind l f = concat (map f l)

    bindconst l l' = bind l (const l')

    return x = [x]

    fail s = Nil
```

Monadische do-Notation

Um monadische Programme übersichtlicher schreiben zu können wird die *monadische do-Notation* verwendet.

Schreibweise.

do *x <- e* steht für `bind e (\x -> do Zeilen)`
Zeilen

do *e* steht für `bindconst e (do Zeilen)`
Zeilen

do *e* steht für *e*

Programmier-Beispiele

Funktionale Programmierung

Aufgabe.

Finde heraus, ob in einer Liste $l :: \text{List } a$ ein Element x mit einer bestimmten Eigenschaft $p :: a \rightarrow \text{Bool}$ vorkommt, und berechne gegebenenfalls ein solches x .

Bibliotheksfunktion.

```
filter :: (a->Bool) -> [a] -> [a]
filter p Nil    = Nil
filter p (x:l) = case p x of
                    True  -> x : (filter p l)
                    False -> filter p l
```

Funktionale Programmierung

Haskell-Programm.

```
listToMaybe :: [a] -> Maybe a
```

```
listToMaybe Nil    = Nothing
```

```
listToMaybe (x:l) = Just x
```

```
find :: (a->Bool) -> [a] -> Maybe a
```

```
find p = compose listToMaybe (filter p)
```

Monadische Programmierung

Aufgabe.

Finde heraus, ob es in einer Liste $l :: \text{List } a$ Elemente x und x' gibt, die den Eigenschaften $p :: a \rightarrow \text{Bool}$ beziehungsweise $p' :: a \rightarrow \text{Bool}$ genügen, und berechne gegebenenfalls ein solches Paar (x, x') .

Haskell-Programm mit do-Notation.

```
find2 :: (a->Bool) -> (a->Bool) -> [a] -> Maybe(a,a)
find2 p p' l = do  x  <- find p l
                    x' <- find p' l
                    return (x,x')
```

Monadische Programmierung ohne do-Notation

Haskell-Programm ohne do-Notation.

```
find2 :: (a->Bool) -> (a->Bool) -> [a] -> Maybe(a,a)
```

```
find2 p p' l = bind (find p l) (\x ->
                                bind (find p' l) (\x' ->
                                return (x,x')))
```

Ende

Literatur

- ▶ John Backus:
Can programming be liberated from the von Neuman style?
Comm. ACM **21** (1978) 613–641.
- ▶ Alonzo Church: *A set of postulates for the foundation of logic.*
Annals of Math. **33** (1932) 346–366, Annals of Math. **34** (1933)
839–864.
- ▶ Alonzo Church: *A formulation of the simple theory of types.*
J. Symbolic Logic **5** (1940) 56–68.
- ▶ Alonzo Church: *The calculi of lambda-conversion.*
Princeton Univ. Press, Princeton 1941.
[Reprinted by Kraus Reprint Corporation, New York 1965.]
- ▶ Haskell B. Curry: *An Analysis of Logical Substitution.*
Amer. J. Math. **51** (1929) 363–384.

- ▶ Samuel Eilenberg, John C. Moore: *Adjoint functors and triples*.
III. J. Math. **9** (1965) 381–398.
- ▶ Roger Godement: *Théorie des Faisceaux*.
Hermann, Paris 1958.
- ▶ Roger Hindley:
The Principal type-scheme of an object in combinatory logic.
Trans. Amer. Math. Soc. **146** (1969), 29–60.
- ▶ Graham Hutton:
Frequently Asked Questions for comp.lang.functional.
November 2002, Internet:
<http://www.cs.nott.ac.uk/~gmh/faq.html>.
- ▶ H. Kleisli: *Every standard construction is induced by a pair of adjoint functors*.
Proc. Amer. Math. Soc. **16** (1965) 544–546.

- ▶ Gottfried Wilhelm Leibniz: *Monadologie*.
Franckfurth und Leipzig 1720.
[Posthum herausgegebene Übersetzung aus dem Französischen
von Heinrich Köhler (1685–1737).]
- ▶ Saunders Mac Lane: *Categories for the Working Mathematician*.
Springer-Verlag, New York 1971.
- ▶ Robin Milner: *A Theory of Type Polymorphism in Programming*.
Journal of Computer and System Sciences **17** (1978) 348–375.
- ▶ Eugenio Moggi: *Computational lambda-calculus and monads*.
Fourth Annual Symposium on Logic in Computer Science
Proceedings, IEEE 1989.

- ▶ Moses Schönfinkel:
Über die Bausteine der mathematischen Logik.
Math. Annalen **92** (1924) 305–316.
- ▶ Philip Wadler, Stephen Blott:
How to make ad-hoc polymorphism less ad hoc.
ACM Symposium on Principles of Programming Languages,
Austin, Texas, January 1989.
- ▶ Philip Wadler: *Comprehending monads.*
P. 461–493 in Mathematical Structures in Computer Science,
Vol. 2, Cambridge Univ. Press, Cambridge 1992.
- ▶ Philip Wadler: *Monads for functional programming.*
P. 24–52 in Johann Jeuring, Erik Meijer (eds.): Advanced
Functional Programming, Springer 1995 (LNCS 925).