

---

# Heutiges Thema

Hintergrund und Grundlagen...

- **Teil 1:** Programmierung mit Monaden
  - Zusammenhang Monaden und Ein-/Ausgabe in Haskell
- **Teil 2:** Ausdrücke, Auswertung von Ausdrücken, Auswertungsstrategien
  - Schlagwörter:* applicative und normal order evaluation, eager und lazy evaluation, ...
- **Teil 3:**  $\lambda$ -Kalkül
  - ...formale Fundierung nahezu aller funktionalen Programmiersprachen

---

# Teil 1: Programmierung mit Monaden

...insbesondere

- Zusammenhang Monaden und Ein-/Ausgabe in Haskell

---

# Ein-/Ausgabe in Haskell und Monaden

Ein-/Ausgabe in Haskell...

- realisiert als Spezialfall eines allgemeineren Konzepts, des Konzepts der *Monade*.

Deshalb: Exkurs über Haskell's Monadenkonzept

---

# Monaden und Monadischer Programmierstil (1)

Monaden...

- erlauben die Reihenfolge, in der Operationen ausgeführt werden, explizit festzulegen.

*Beispiel:*

```
a-b -- Keine Festlegung der Auswertungsreihenfolge;  
    -- kritisch, falls z.B. Ein-/Ausgabe involviert ist.
```

```
do a <- getInt -- Reihenfolge explizit festgelegt  
    b <- getInt  
    return (a-b)
```

---

# Monaden (1)

...sind *Konstruktorklassen*, Familien von Typen `m a` über einem polymorphen Typkonstruktor `m` mit den Funktionen `(>>=)`, `return`, `(>>)` und `fail`.

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

  m >> k = m >>= \_ -> k      -- vordefiniert
  fail s = error s           -- vordefiniert
```

...wobei die Implementierungen der Funktionen gewissen Anforderungen genügen müssen.

---

## Monaden (2)

*Zentral:*

- Die Funktionen (`>>=`) und `return` (und das abgeleitete (und in der Anwendung oft bequemere) `do`-Konstrukt)

...für die die Konstruktorklasse `Monad m` keine Standardimplementierung vorsieht.

---

## Monaden (3)

Anforderungen (Gesetze) an die Monadenoperationen:

$$\text{return } a \gg= f = f \ a$$
$$c \gg= \text{return} = c$$
$$c \gg= (\lambda x \rightarrow (f \ x) \gg= g) = (c \gg= f) \gg= g$$

*Intuitiv:*

- `return` gibt den Wert zurück, ohne einen weiteren Effekt.
- durch `>>=` gegebene Sequenzierungen sind unabhängig von der Klammerung (assoziativ)

---

# Konstruktorklassen vs. Typklassen

Im Grundsatz ähnliche Konzepte, wobei...

- Konstruktorklassen
  - ...haben Typkonstruktoren als Elemente
- Typklassen (Eq a, Ord a, Num a, ...)
  - ...haben Typen als Elemente
- Typkonstruktoren sind...
  - ...Funktionen, die aus gegebenen Typen neue Typen erzeugen  
(Bsp.: Tupelkonstruktor ( ), Listenkonstruktor [ ], Funktionskonstruktor ->, aber auch: Ein-/Ausgabe IO,...)



---

## Der abgeleitete Operator (>@>)...

...ist folgendermaßen definiert:

$$\begin{aligned} (>@>) &:: \text{Monad } m \Rightarrow (a \rightarrow m b) \rightarrow \\ &\quad (b \rightarrow m c) \rightarrow \\ &\quad (a \rightarrow m c) \end{aligned}$$
$$f >@> g = \lambda x \rightarrow (f x) >>= g$$

*Hinweis:*

...return ist vom Typ  $a \rightarrow m a$ !

---

## Damit...

(1) `return >@> f = f`

(2) `f >@> return = f`

(3) `(f >@> g) >@> h = f >@> (g >@> h)`

Intuitiv...

- (1)&(2): `return` ist Einselement von `(>@>)`
- (3): `(>@>)` ist assoziativ

*Beachte:*

Obige Eigenschaften gelten nicht a priori, sondern sind durch die Implementierung sicherzustellen!

---

# Beispiele von Monaden (1)

Die Identitätsmonade (mehr dazu auf Folie 20, Teil 9):

...einfachste aller Monaden.

```
(>>=)  :: m a -> (a -> m b) -> m b  
m >>= f = f m
```

```
return :: a -> m a  
return = id
```

*Erinnerung:*

- ( $\gg$ ) und `fail` implizit durch die Standarddefinition festgelegt.

*Bemerkung:*

- In diesem Szenario...
  - ( $\circ\gg$ ) wird Vorwärtskomposition von Funktionen, ( $\>.\>$ ).
  - Beachte:* ( $\>.\>$ ) ist assoziativ mit Einselement `id`.

---

## Beispiele von Monaden (2)

Die Listenmonade:

```
instance Monad [] where
  xs >>= f = concat (map f xs)
  return x = [x]
  fail s   = []
```

---

## Beispiele von Monaden (3)

Die “Maybe”-Monade:

```
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= k = Nothing
  return   = Just
  fail s   = Nothing
```

---

## Beispiele von Monaden (4)

Die Ein-/Ausgabe-Monade:

```
instance Monad IO where
  (>>=) :: IO a -> (a -> IO b) -> IO b
  return :: a -> IO a    -- Rueckgabewerterzeugung
                        -- ohne Ein-/Ausgabe(aktion)
```

---

# Standardfunktionen über Monaden

Kombination von Monaden...

- ...zum Aufbau komplexerer Effekte

```
mapF :: Monad m => (a -> b) -> m a -> m b
mapF f m = do x <- m
            return (f x)
```

```
joinM :: Monad m => m (m a) -> m a
joinM m = do x <- m
            x
```

*Bemerkung:*

- Aus (1), (2) und (3) folgt:  
(4)  $\text{mapF } (f.g) = \text{mapF } f . \text{mapF } g$

---

# Ein-/Ausgabe und Monaden

*Erinnerung:*

```
(>>=) :: IO a -> (a -> IO b) -> IO b  
return :: a -> IO a
```

wobei...

- (>>=): Wenn  $p$  und  $q$  Kommandos sind, dann ist  $p \gg= q$  das Kommando, das zunächst  $p$  ausführt, dabei den Rückgabewert  $x$  vom Typ  $a$  liefert, und daran anschließend  $q \ x$  ausführt und dabei den Rückgabewert  $y$  vom Typ  $b$  liefert.
- `return`: Rückgabewerterzeugung ohne Ein-/Ausgabe(aktion)

Somit...

- Ein-/Ausgabe in Haskell Monad über dem Typkonstruktor `IO`



---

# Programmieren mit Monaden (1)

*Gegeben:*

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

*Aufgabe:*

- Schreibe eine Funktion, die die Summe der Werte der Marken in einem Baum vom Typ `Tree Int` berechnet.

---

# Programmieren mit Monaden (2)

*Lösung 1: Monadenlos*

```
sTree :: Tree Int -> Int
sTree Nil                = 0
sTree (Node n t1 t2) = n + sTree t1 + sTree t2
```

*Beachte:*

- Die Reihenfolge der Berechnung ist weitgehend nicht festgelegt (Freiheitsgrade!)

---

# Programmieren mit Monaden (3)

*Lösung 1:* Monadenbehaftet

```
sumTree :: Tree Int -> Id Int
sumTree Nil = return 0
sumTree (Node n t1 t2) = do num <- return n
                             s1  <- sumTree t1
                             s2  <- sumTree t2
                             return (num + s1 + s2)
```

...wobei Id die Identitätsmonade bezeichnet.

*Beachte:*

- Die Reihenfolge der Berechnung explizit festgelegt (keine Freiheitsgrade!)

---

# Programmieren mit Monaden (4)

Die Identitätsmonade:

```
data Id a = Id a

instance Monad Id where
  (>>=) (Id x) f = f x
  return      = Id
```

---

# Programmieren mit Monaden (5)

Vergleich der monadenlosen und monadenbehafteten Lösung:

Es gilt:

- Anders als `sTree` hat `sumTree` einen “imperativen Anstrich” in etwa vergleichbar mit:

```
num    := n;  
sum1   := sumTree t1;  
sum2   := sumTree t2;  
return (num + sum1 + sum2);
```

---

# Programmieren mit Monaden: Ein Re-sümee (1)

Die Programmierung mit Monaden erlaubt...

- Berechnungsabläufe zu strukturieren.

Folgende Eigenschaften prädestinieren Monaden dafür in besonderer Weise:

- *Wohldefiniert*: ...Strategie sequentielle Programmteile systematisch zu spezifizieren.
- *Angemessen*: ...höhere Abstraktion durch Entkopplung der zugrundeliegenden Monade von der Struktur der Berechnung.
- *Fundiert*: ...Eigenschaften wie etwa (4) werden impliziert von den Monadforderungen (1), (2) und (3).

---

# Programmieren mit Monaden: Ein Re-sümee (2)

Monaden sind...

- ein in der Kategorientheorie geprägter Begriff  
~> ...zur formalen Beschreibung der Semantik von Programmiersprachen (Eugenio Moggi, 1989)
- (ohne obigen Hintergrund) populär in der Welt funktionaler Programmierung, insbesondere weil (Philip Wadler, 1992)
  - erlauben gewisse Aspekte imperativer Programmierung in die funktionale Programmierung zu übertragen
  - eignen sich insbesondere zur Integration von Ein-/Ausgabe, aber auch für weitergehende Anwendungsszenarien
  - geeignete Schnittstelle zwischen funktionaler und effektbehafteter, z.B. imperativer und objektorientierter Programmierung.

---

## **Teil 2: Ausdrücke, Auswertung von Ausdrücken, Auswertungsstrategien**

*Schlagwörter:*

applicative und normal order evaluation,  
eager und lazy evaluation, ...



---

# Auswerten von Ausdrücken und Funktionsaufrufen

$e :: \text{Float}$

$e = 2.71828$

$\text{res} = 2 * e * e \Rightarrow \dots$

$\text{simple} \quad \quad \quad :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

$\text{simple } x \ y \ z \quad \quad = (x + z) * (y + z)$

$\text{simple } 2 \ 3 \ 4 \Rightarrow \dots$

$\text{fac} \quad \quad \quad :: \text{Int} \rightarrow \text{Int}$

$\text{fac } n \quad \quad = \text{if } n == 0 \text{ then } 1 \text{ else } n * \text{fact } (n - 1)$

$\text{fac } 2 \Rightarrow \dots$

---

# Auswerten von Ausdrücken

## Einfache Ausdrücke

$$3 * (9 + 5) \Rightarrow 3 * 14 \Rightarrow 42$$

oder

$$3 * (9 + 5) \Rightarrow 3 * 9 + 3 * 5 \Rightarrow 27 + 3 * 5 \Rightarrow 27 + 15 \Rightarrow 42$$

oder ...

---

# Auswerten von Funktionsaufrufen (1)

Fundamental...

- *Expandieren*
- *Simplifizieren*

---

# Auswerten von Funktionsaufrufen (2)

*Beispiele:*

$$\text{simple } x \ y \ z = (x + z) * (y + z)$$

$$\text{simple } 2 \ 3 \ 4 \Rightarrow (2 + 4) * (3 + 4) \text{ (Expandieren)}$$

$$\Rightarrow 6 * (3 + 4) \text{ (Simplifizieren)}$$

$$\Rightarrow 6 * 7 \text{ (Simplifizieren)}$$

$$\Rightarrow 42 \text{ (Simplifizieren)}$$

oder

$$\text{simple } 2 \ 3 \ 4 \Rightarrow (2 + 4) * (3 + 4) \text{ (Expandieren)}$$

$$\Rightarrow (2 + 4) * 7 \text{ (Simplifizieren)}$$

$$\Rightarrow 6 * 7 \text{ (Simplifizieren)}$$

$$\Rightarrow 42 \text{ (Simplifizieren)}$$

oder ...

---

# Funktionsaufrufe

fac n = if n == 0 then 1 else (n \* fac (n - 1))

fac 2 ⇒ if 2 == 0 then 1 else (2 \* fac (2 - 1))

⇒ 2 \* fac (2 - 1)

Weiter mit a)

⇒ 2 \* fac 1

⇒ 2 \* (if 1 == 0 then 1 else (1 \* fac (1-1)))

⇒ analog fortführen...

...oder mit b)

⇒ 2 \* (if (2-1) == 0 then 1 else ((2-1) \* fac ((2-1)-1)))

⇒ analog fortführen...

Schließlich...

⇒ 2

---

## Auswertung gemäß Variante a)

fac n = if n == 0 then 1 else (n \* fac (n - 1))

fac 2 ⇒ if 2 == 0 then 1 else (2 \* fac (2 - 1))

⇒ 2 \* fac (2 - 1)

⇒ 2 \* fac 1

⇒ 2 \* (if 1 == 0 then 1 else (1 \* fac (1 - 1)))

⇒ 2 \* (1 \* fac (1 - 1))

⇒ 2 \* (1 \* fac 0)

⇒ 2 \* (1 \* (if 0 == 0 then 1 else (0 \* fac (0 - 1))))

⇒ 2 \* (1 \* 1)

⇒ 2 \* 1

⇒ 2

---

## Auswertung gemäß Variante b)

fac n = if n == 0 then 1 else (n \* fac (n - 1))

fac 2 ⇒ if 2 == 0 then 1 else (2 \* fac (2 - 1))

⇒ 2 \* fac (2 - 1)

⇒ 2 \* (if (2-1) == 0 then 1 else ((2-1) \* fac ((2-1)-1)))

⇒ 2 \* ((2-1) \* fac ((2-1)-1))

⇒ 2 \* (1 \* fac ((2-1)-1))

⇒ 2 \* (1 \* (if ((2-1)-1) == 0 then 1

⇒                   else ((2-1)-1) \* fac (((2-1)-1)-1)))

⇒ 2 \* (1 \* 1)

⇒ 2 \* 1

⇒ 2

---

# Freiheitsgrade...

Betrachte...

```
fac(fac(square(2+2))) * fact(fac(square(3)))
```

Zentral...

- **Wo** im Ausdruck mit der Auswertung fortfahren?
- **Wie** mit (Funktions-) Argumenten umgehen?

Gretchenfrage...

- **Welcher** Einfluss auf das Ergebnis?



---

# Glücklicherweise...

## Theorem

Jede terminierende Auswertungsreihenfolge endet mit demselben Ergebnis.

...Alonzo Church/J. Barclay Rosser (1936)

---

# Auswertungsstrategien...

In der Praxis...

Um den Ausdruck  $f(exp)$  auszuwerten...

- a) ...berechne zunächst den Wert von  $exp$  (und setze diesen Wert dann im Rumpf von  $f$  ein)
  - ↪ applicative order evaluation, eager evaluation, call-by-value evaluation, leftmost-innermost evaluation
  
- b) ...setze  $exp$  unmittelbar im Rumpf von  $f$  ein und werte den so entstehenden Ausdruck aus
  - ↪ normal order evaluation, call-by-name evaluation, leftmost-outermost evaluation
  - ↪ *“Intelligente” Realisierung*: lazy evaluation, call-by-need evaluation

---

# Ein Beispiel...

Einige einfache Funktionen...

-- Die Funktion square zur Quadrierung einer ganzen Zahl

```
square :: Int -> Int
```

```
square n = n*n
```

-- Die Funktion first zur Projektion auf die erste Paarkomponente

```
first :: (Int,Int) -> Int
```

```
first (m,n) = m
```

-- Die Funktion infiniteInc zum "ewigen" Inkrement

```
infiniteInc :: Int
```

```
infiniteInc = 1 + infiniteInc
```

---

# Auswertung gemäß...

...applicative order (leftmost-innermost):

```
square(square(square(2)))  
⇒ square(square(2 * 2))  
⇒ square(square(4))  
⇒ square(4 * 4)  
⇒ square(16)  
⇒ 16 * 16  
⇒ 256
```

...6 Schritte.

---

# Auswertung gemäß...

...normal order (leftmost-outermost):

```
square(square(square(2)))  
⇒ square(square(2)) * square(square(2))  
⇒ (square(2) * square(2)) * square(square(2))  
⇒ ((2 * 2) * square(2)) * square(square(2))  
⇒ (4 * square(2)) * square(square(2))  
⇒ (4 * (2 * 2)) * square(square(2))  
⇒ (4 * 4) * square(square(2))  
⇒ 16 * square(square(2))  
⇒ ...  
⇒ 16 * 16  
⇒ 256
```

...1+6+6+1=14 Schritte.

---

# Applicative order effizienter?

Nicht immer...

```
first (42, square(square(square(2))))
```

...in applicative order

```
first (42, square(square(square(2))))
```

⇒ ...

⇒ first (42, 256)

⇒ 42

...1+6+1=8 Schritte.

...in normal order

```
first (42, square(square(square(2))))
```

⇒ 42

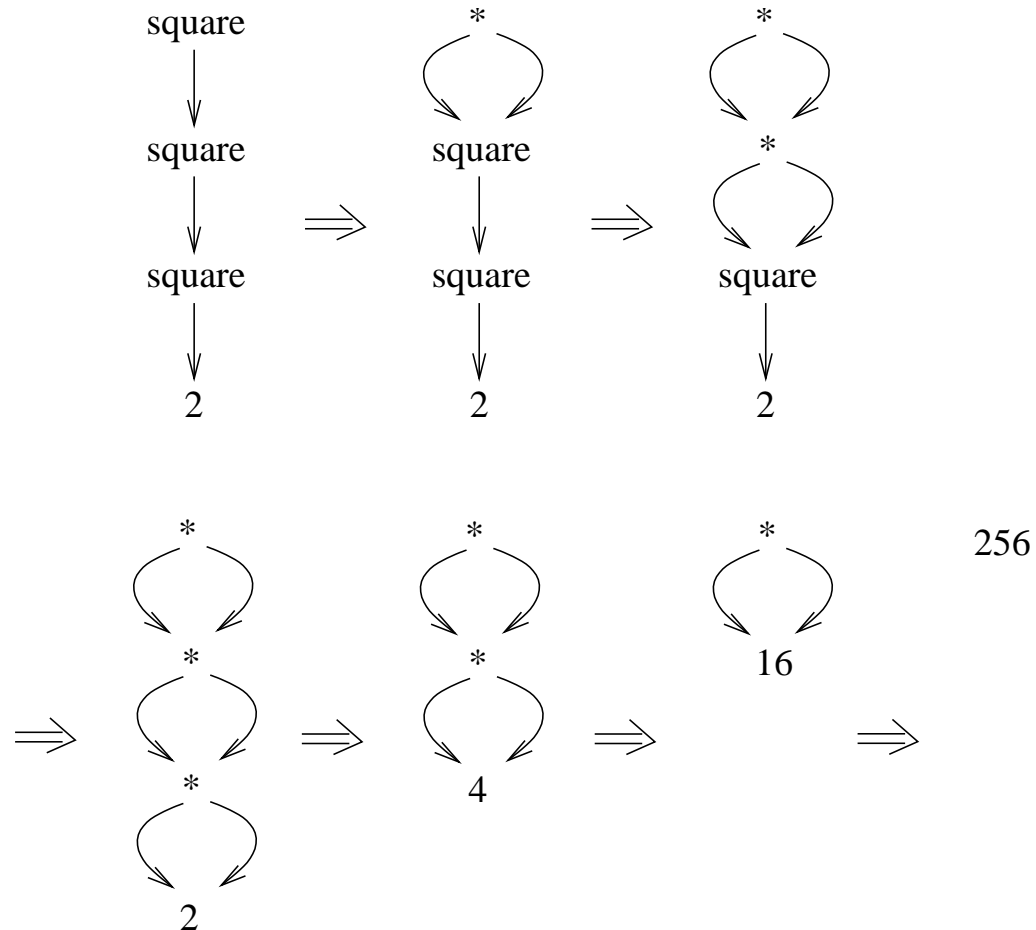
...1 Schritt.

---

# Von Normal Order zu Lazy Evaluation...

- *Problem*: Mehrfachauswertung von Ausdrücken bei *normal order Evaluation*
- *Ziel*: Vermeidung von Mehrfachauswertungen zur Effizienzsteigerung
- *Lösung*: *Lazy Evaluation!*  
...Ausdrucksdarstellung und -auswertung basierend auf Graphen und Graphtransformationen.

# Lazy evaluation (call-by-need)...



**...6 Schritte, aber Graphtransformationen!**



---

## Lazy evaluation (call-by-need)...

- ...beruht (implementierungstechnisch) auf Graphtransformationen
- ...garantiert, dass Argumente höchstens einmal (möglicherweise also gar nicht) ausgewertet werden

Insgesamt...

~> ...effiziente Realisierung der normal order Strategie!

---

# Zentrale Ergebnisse...

## Theorem

- ...alle terminierenden Auswertungsreihenfolgen enden mit demselben Ergebnis  
     $\rightsquigarrow$  Konfluenz- oder Diamanteigenschaft
- ...wenn es eine terminierende Auswertungsreihenfolge gibt, so terminiert auch die normal order Auswertungsreihenfolge

...Church/Rosser (1936)

*Insbesondere:*

Lazy evaluation “vergleichbar effizient” wie applicative (eager) order, falls alle Argumente benötigt werden.

---

## Frei nach Shakespeare...

*Eager or lazy evaluation: that is the question.*

*Quot capita, tot sensa.*

Oder: Die Antworten sind verschieden...

- eager evaluation (z.B. ML, Scheme (abgesehen von Makros),...)
- lazy evaluation (z.B. Haskell, Miranda,...)

---

# Lazy vs. Eager: Eine Abwägung (1)

## Lazy Evaluation

- Vorteile
  - terminiert mit Normalform, wenn es eine terminierende Auswertungsreihenfolge gibt
  - wertet Argumente nur aus, wenn nötig
  - elegante Behandlung potentiell unendlicher Datenstrukturen
- Nachteile
  - konzeptuell und implementierungstechnisch anspruchsvoller
    - \* partielle Auswertung von Ausdrücken (Seiteneffekte! Beachte: Letztere nicht in Haskell! In Scheme: Verantwortung beim Programmierer.)
    - \* Graphtransformationen
    - \* Ein-/Ausgabe
  - Volles Verständnis: Domain-Theorie und  $\lambda$ -Kalkül im Detail

---

# Lazy vs. Eager: Eine Abwägung (2)

## Eager Evaluation

- Vorteile
  - Konzeptuell und implementierungstechnisch einfacher
  - Vom mathematischen Standpunkt oft “natürlicher”  
(Beispiel: `first (42,infiniteInc)`)
  - (Einfache(re) Integration imperativer Konzepte)

*Mithin...* eager oder lazy – eine Frage des Anwendungsprofils!

---

## Zu guter Letzt

Wäre ein Haskell-Compiler (Interpreter) korrekt, der die Fakultätsfunktion applikativ auswertet?

Ja, weil die Funktion `fac` *strikt* in ihrem Argument ist...

~> eager evaluation oder auch strict evaluation!

---

## Teil 3: $\lambda$ -Kalkül

...formale Fundierung nahezu aller funktionalen Programmiersprachen

---

# Intuitive vs. formale Berechenbarkeit

*Ausgangspunkt...*

*Intuitiv berechenbar ...* “wenn es eine *irgendwie machbare effektive mechanische Methode* gibt, die zu jedem Argument aus dem Definitionsbereich nach endlich vielen Schritten den Funktionswert konstruiert und die für alle anderen Argumente entweder mit einem speziellen Fehlerwert oder nie abbricht” .

*Zentrale Frage...*

Lässt sich der Begriff “intuitiver Berechenbarkeit”  
formal fassen?

*Zur Beantwortung nötig...*

Formale Berechnungsmodelle!

...d.h. *Explikationen* des Begriffs “intuitiver Berechenbarkeit” .



---

# Der $\lambda$ -Kalkül... (1)

- ...ein spezielles formales Berechnungsmodell, wie viele andere auch, z.B.
  - allgemein rekursive Funktionen (Herbrand 1931, Gödel 1934, Kleene 1936)
  - Turing-Maschinen (Turing 1936)
  - $\mu$ -rekursive Funktionen (Kleene 1936)
  - Markov-Algorithmen (Markov 1951)
  - ...
- ...geht zurück auf Alonzo Church (1936)
- ...Berechnungen über Paaren, Listen, Bäumen, auch unendlichen, Funktionen höherer Ordnung einfach ausdrückbar
- ...in diesem Sinne “praxisnäher/realistischer” als andere formale Berechnungsmodelle

---

## Der $\lambda$ -Kalkül... (2)

### Church'sche These

Eine Funktion ist genau dann intuitiv berechenbar, wenn sie  $\lambda$ -definierbar ist (d.h. im  $\lambda$ -Kalkül ausdrückbar ist).

**Beweis?** ...schlechterdings unmöglich!

Aber...

---

## Der $\lambda$ -Kalkül... (3)

Man hat bewiesen...

- Alle der obigen Berechnungsmodelle sind gleich mächtig.

Das kann als Hinweis darauf verstanden werden, dass alle der obigen Berechnungsmodelle den Begriff wahrscheinlich “gut” charakterisieren!

**Aber:** es schließt nicht aus, dass morgen ein mächtigeres formales Berechnungsmodell gefunden wird, das dann den Begriff der intuitiven Berechenbarkeit “besser” charakterisierte.

*Präzedenzfall:* Primitiv rekursive Funktionen

- ...bis Ende der 20er-Jahre als adäquate Charakterisierung intuitiver Berechenbarkeit akzeptiert (auch von Hilbert)
- ...tatsächlich jedoch: echt schwächeres Berechnungsmodell
- ...Beweis: Ackermann-Funktion ist berechenbar, aber nicht primitiv rekursiv (Ackermann 1928)

---

# Die Ackermann-Funktion

...“berühmtberüchtigtes” Beispiel einer

- zweifellos (intuitiv) berechenbaren, aber nicht primitiv rekursiven Funktion!

`ack :: (Integer,Integer) -> Integer`

`ack (m,n)`

`| m == 0 = n+1`

`| (m > 0) && (n == 0) = ack (m-1,1)`

`| (m > 0) && (n /= 0) = ack (m-1,ack(m,n-1))`

~> ...hier in Haskell-Notation!

---

## Der $\lambda$ -Kalkül... (4)

...ausgezeichnet durch

- *Einfachheit*  
...nur wenige syntaktische Konstrukte, einfache Semantik
- *Ausdruckskraft*  
...Turing-mächtig, alle “intuitiv berechenbaren” Funktionen im  $\lambda$ -Kalkül ausdrückbar

Darüberhinaus...

↪ Bindeglied zwischen funktionalen Hochsprachen und ihren maschinennahen Implementierungen.

---

## Wir unterscheiden...

- *Reiner*  $\lambda$ -Kalkül
  - ...reduziert auf das “absolut Notwendige”
    - $\rightsquigarrow$  besonders bedeutsam in Untersuchungen zur Theorie der Berechenbarkeit
- *Angewandte*  $\lambda$ -Kalküle
  - ...syntaktisch angereichert, praxisnäher

---

# Reiner $\lambda$ -Kalkül: Syntax

Die Menge  $Exp$  der Ausdrücke des (reinen)  $\lambda$ -Kalküls, kurz  $\lambda$ -Ausdrücke, ist definiert durch:

- Jeder *Name* (*Identifizier*) ist in  $Exp$ .  
(Bsp:  $a, b, c, \dots, x, y, z, \dots$ )
- *Abstraktion*: Wenn  $x$  ein Name und  $e$  aus  $Exp$  ist, dann ist auch  $(\lambda x. e)$  in  $Exp$ . *Sprechweise*: Funktionsabstraktion mit formalem Parameter  $x$  und Rumpf  $e$ .  
(Bsp.:  $(\lambda x.(x\ x)), (\lambda x.(\lambda y.(\lambda z.(x\ (y\ z))))), \dots$ )
- *Applikation*: Wenn  $f$  und  $e$  in  $Exp$  sind, dann ist auch  $(f\ e)$  in  $Exp$ ; *Sprechweisen*: Anwendung von  $f$  auf  $e$ .  $f$  heißt auch *Rator*,  $e$  auch *Rand*.  
(Bsp.:  $((\lambda x.(x\ x))\ y), \dots$ )

---

## Alternativ...

...die Syntax in (modifizierter) Backus-Naur-Form (BNF):

$e ::=$		( $\lambda$ -Ausdrücke)
$::= x$		(Namen (Identifikatoren))
$::= \lambda x.e$		(Abstraktion)
$::= e e$		(Applikation)
$::= (e)$		



---

# Konventionen

- Überflüssige Klammern können weggelassen werden.  
Dabei gilt:
  - *Rechtsassoziativität* für  $\lambda$ -Sequenzen in Abstraktionen  
Bsp.: –  $\lambda x. \lambda y. \lambda z. (x (y z))$  kurz für  $(\lambda x. (\lambda y. (\lambda z. (x (y z))))$ ,  
–  $\lambda x. e$  kurz für  $(\lambda x. e)$
  - *Linksassoziativität* für Applikationssequenzen  
Bsp.: –  $e_1 e_2 e_3 \dots e_n$  kurz für  $(\dots ((e_1 e_2) e_3) \dots e_n)$ ,  
–  $(e_1 e_2)$  kurz für  $e_1 e_2$
- Der Rumpf einer  $\lambda$ -Abstraktion ist der längstmögliche dem Punkt folgende  $\lambda$ -Ausdruck  
Bsp.: –  $\lambda x. e f$  entspricht  $\lambda x. (e f)$ , nicht  $(\lambda x. e) f$

---

# Angewandte $\lambda$ -Kalküle

Angewandte  $\lambda$ -Kalküle sind syntaktisch angereichert.

Beispielsweise...

- Auch Konstanten, Funktionsnamen oder “übliche” Operatoren können Namen (im weiteren Sinn) sein  
(Bsp: 1, 3.14, *true*, *false*, +, \*, –, fac, simple, ...)
- Ausdrücke können...
  - komplexer sein  
(Bsp.: if e then e<sub>1</sub> else e<sub>2</sub> fi ... statt cond e e<sub>1</sub> e<sub>2</sub> für cond geeignete Funktion)
  - getypt sein  
(Bsp.: 1 : *IN*, *true* : *Boole*, ...)
- ...

$\lambda$ -Ausdrücke sind dann beispielsweise auch...

- Applikationen: fac 3, simple x y z (entspricht ((simple x) y) z), ...)
- Abstraktionen:  $\lambda x.(x + x)$ ,  $\lambda x.\lambda y.\lambda z.(x * (y - z))$ , 2 + 3,  
( $\lambda x.$ if odd x then x\*2 else x div 2 fi) 42, ...)

---

## In der Folge

...erlauben wir uns die Annehmlichkeit Ausdrücke, für die wir eine eingeführte Schreibweise haben (z.B.  $n * fac(n - 1)$ ) in dieser gewohnten Weise zu schreiben, auch wenn wir die folgenden Ergebnisse für den reinen  $\lambda$ -Kalkül formulieren.

Rechtfertigung...

- Resultate der theoretischen Informatik, insbesondere  
Alonzo Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941  
...zur Modellierung von ganzen Zahlen, Wahrheitswerten, etc. durch geeignete Ausdrücke des reinen  $\lambda$ -Kalküls

---

# Freie und gebundene Variablen (1)

...in  $\lambda$ -Ausdrücken

Die Menge der *frei* vorkommenden Variablen...

$free(x) = \{x\}$ , wenn  $x$  ein Variablenname ist

$free(\lambda x.e) = free(e) \setminus \{x\}$

$free(f e) = free(f) \cup free(e)$

Umgekehrt...

Die Menge der *gebunden* vorkommenden Variablen...

$bound(\lambda x.e) = bound(e) \cup \{x\}$

$bound(f e) = bound(f) \cup bound(e)$

*Beachte:* gebunden  $\neq$  nicht frei !

...sonst wäre etwa "x gebunden in y"

---

---

## Freie und gebundene Variablen (2)

Betrachte:  $(\lambda x. (x y)) x$

- Gesamtausdruck
  - $x$  kommt frei und gebunden in  $(\lambda x. (x y)) x$  vor
  - $y$  kommt frei in  $(\lambda x. (x y)) x$  vor
- Teilausdrücke
  - $x$  kommt gebunden in  $(\lambda x. (x y))$  vor und frei in  $(x y)$  und in  $x$
  - $y$  kommt frei in  $(\lambda x. (x y))$ ,  $(x y)$  und  $y$  vor

---

# Gebunden vs. gebunden an...

Wir müssen unterscheiden...

- Eine *Variable* ist *gebunden*...
- Ein *Variablenvorkommen* ist *gebunden an*...

*Gebunden* und *gebunden an* ...unterschiedliche Konzepte!

Letzteres meint:

- Ein (definierendes oder angewandtes) Variablenvorkommen ist an ein definierendes Variablenvorkommen gebunden

Definition

- *Definierendes* V.vorkommen ...Vorkommen unmittelbar nach einem  $\lambda$
- *Angewandtes* V.vorkommen ...jedes nicht definierende Vorkommen

---

# Der $\lambda$ -Kalkül: Vorwärts zur Semantik

Zentral sind folgende Begriffe...

- (Syntaktische) Substitution
- Konversionsregeln / Reduktionsregeln

---

# Syntaktische Substitution

Erster zentraler Begriff... (*syntaktische*) *Substitution*

$x[e/x] = e$ , wenn  $x$  ein Name ist

$y[e/x] = y$ , wenn  $y$  ein Name mit  $x \neq y$  ist

$(f\ g)[e/x] = (f[e/x])\ (g[e/x])$

$(\lambda x.f)[e/x] = \lambda x.f$

$(\lambda y.f)[e/x] = \lambda y.(f[e/x])$ , wenn  $x \neq y$  und  $y \notin \text{free}(e)$

$(\lambda y.f)[e/x] = \lambda z.((f[z/y])[e/x])$ , wenn  $x \neq y$  und  $y \in \text{free}(e)$ ,  
wobei  $x \neq z$  und  $z \notin \text{free}(e) \cup \text{free}(f)$



---

# (Syntaktische) Substitution

Einige Beispiele zur Illustration...

- $((x\ y)\ (y\ z))\ [a+b/y] = ((x\ (a+b))\ ((a+b)\ z))$
- $\lambda x. (x\ y)\ [a+b/y] = \lambda x. (x\ (a+b))$
- Aber:  $\lambda x. (x\ y)\ [a+b/x] = \lambda x. (x\ y)$
- Achtung:  $\lambda x. (x\ y)\ [x+b/y] \rightsquigarrow \lambda x. (x\ (x+b))$   
...ohne Umbenennung *Bindungsfehler!*

$$\begin{aligned} \text{Deshalb: } \lambda x. (x\ y)\ [x+b/y] &= \lambda z. ((x\ y)[z/x])\ [x+b/y] \\ &= \lambda z. (z\ y)\ [x+b/y] \\ &= \lambda z. (z\ (x+b)) \end{aligned}$$

...dank Umbenennung kein Bindungsfehler!

---

# Konversionsregeln

Zweiter zentraler Begriff:  $\lambda$ -Konversionen...

- $\alpha$ -Konversion (Umbenennung formaler Parameter)

$$\lambda x.e \Leftrightarrow \lambda y.e[y/x], \text{ wobei } y \notin \text{free}(e)$$

- $\beta$ -Konversion (Funktionsanwendung)

$$(\lambda x.f) e \Leftrightarrow f[e/x]$$

- $\eta$ -Konversion (Elimination redundanter Funktion)

$$\lambda x.(e x) \Leftrightarrow e, \text{ wobei } x \notin \text{free}(e)$$

$\leadsto$  führen auf eine operationelle Semantik des  $\lambda$ -Kalküls.

---

# Sprechweisen

...im Zusammenhang mit Konversionsregeln

- Von links nach rechts gerichtete Anwendungen der  $\beta$ - und  $\eta$ -Konversion heißen  $\beta$ - und  $\eta$ -Reduktion.
- Von rechts nach links gerichtete Anwendungen der  $\beta$ -Konversion heißen  $\beta$ -Abstraktion.

---

# Intuition hinter den Konversionsregeln

Noch einmal zusammengefasst...

- $\alpha$ -Konversion... erlaubt die konsistente Umbenennung formaler Parameter von  $\lambda$ -Abstraktionen
- $\beta$ -Konversion... erlaubt die Anwendung einer  $\lambda$ -Abstraktion auf ein Argument  
(Achtung: Gefahr von Bindungsfehlern! Abhilfe:  $\alpha$ -Konversion!)
- $\eta$ -Konversion... erlaubt die Elimination redundanter  $\lambda$ -Abstraktionen

Bsp.:  $(\lambda x. \lambda y. x + y) (y * 2) \Rightarrow \lambda y. (y * 2) + y$   
 $\rightsquigarrow$  Bindungsfehler ("y wird eingefangen")

---

## Beispiel für $\lambda$ -Reduktion

$$\begin{aligned} & (\lambda x. \lambda y. x * y) ((\lambda x. \lambda y. x + y) 9 5) 3 && (\beta\text{-Reduktion}) \\ \Rightarrow & (\lambda x. \lambda y. x * y) ((\lambda y. 9 + y) 5) 3 && (\beta\text{-Reduktion}) \\ \Rightarrow & (\lambda y. ((\lambda y. 9 + y) 5) * y) 3 && (\beta\text{-Reduktion}) \\ \Rightarrow & (\lambda y. (9 + 5) * y) 3 && (\beta\text{-Reduktion}) \\ \Rightarrow & (9 + 5) * 3 && (\beta\text{- und } \eta\text{-Reduktion nicht anwendbar}) \end{aligned}$$

$\rightsquigarrow$  Weitere Regeln zur Reduktion primitiver Operationen in erweitertem  $\lambda$ -Kalkül (Auswertung arithmetischer Ausdrücke, bedingte Anweisungen, Listenoperationen, . . . ), sog.  $\delta$ -Regeln.

$\rightsquigarrow$  solche Erweiterungen sind praktisch notwendig und einsichtig, aber für die Theorie (der Berechenbarkeit) kaum relevant.

---

# Reduktionsfolgen & Normalformen (1)

- Ein  $\lambda$ -Ausdruck ist in *Normalform*, wenn er durch  $\beta$ -Reduktion und  $\eta$ -Reduktion nicht weiter reduzierbar ist.
- (Praktisch relevante) Reduktionsstrategien
  - normal order (leftmost-outermost)
  - applicative order (leftmost-innermost)

---

## Reduktionsfolgen & Normalformen (2)

- *Beachte:* Nicht jeder  $\lambda$ -Ausdruck ist zu einem  $\lambda$ -Ausdruck in Normalform konvertierbar (Endlosrekursion).  
Bsp.: (1)  $\lambda x.(x x) \lambda x.(x x) \Rightarrow \lambda x.(x x) \lambda x.(x x) \Rightarrow \dots$   
(2)  $(\lambda x.42) (\lambda x.(x x) \lambda x.(x x))$  (hat Normalform!)

Zentrale Resultate:

- Wenn ein  $\lambda$ -Ausdruck zu einem  $\lambda$ -Ausdruck in Normalform konvertierbar ist, dann führt jede terminierende Reduktion des  $\lambda$ -Ausdrucks zum (bis auf  $\alpha$ -Konversion) selben  $\lambda$ -Ausdruck in Normalform (bis auf  $\alpha$ -Konversion).
- Durch Reduktionen im  $\lambda$ -Kalkül sind genau jene Funktionen berechenbar, die Turing-, Markov-, ... berechenbar sind!

---

# Church-Rosser-Theoreme

Seien  $e_1$  und  $e_2$  zwei  $\lambda$ -Ausdrücke...

## Theorem 1

Wenn  $e_1 \Leftrightarrow e_2$ , dann gibt es einen  $\lambda$ -Ausdruck  $e$  mit  $e_1 \Rightarrow^* e$  und  $e_2 \Rightarrow^* e$

(sog. *Konfluenzeigenschaft*, *Diamanteigenschaft*)

*Informell* ...wenn eine Normalform ex., dann ist sie eindeutig (bis auf  $\alpha$ -Konversion)!

## Theorem 2

Wenn  $e_1 \Rightarrow^* e_2$  und  $e_2$  in Normalform, dann gibt es eine normal order Reduktionsfolge von  $e_1$  nach  $e_2$

(sog. *Standardisierungstheorem*)

*Informell* ...normal order Reduktion terminiert am häufigsten!



---

# Semantik von $\lambda$ -Ausdrücken

- $\lambda$ -Ausdrücke in Normalform lassen sich (abgesehen von  $\alpha$ -Konversionen) nicht weiter vereinfachen (reduzieren)
- Nach dem 1. Church-Rosser-Theorem ist die Normalform eines  $\lambda$ -Ausdrucks eindeutig bestimmt, wenn sie existiert (wieder abgesehen von  $\alpha$ -Konversionen)

Das legt folgende Sichtweise nahe...

- Besitzt ein  $\lambda$ -Ausdruck eine Normalform, so ist dies sein Wert.
- *Umgekehrt*: Die *Semantik (Bedeutung)* eines  $\lambda$ -Ausdrucks ist seine Normalform, wenn sie existiert, ansonsten ist sie undefiniert.

---

# Rekursion. Und wie sie behandelt wird...

Erinnerung...

```
fac n = if n == 0 then 1 else (n * fac (n - 1))
```

oder alternativ:

```
fac = λn.if n == 0 then 1 else (n * fac (n - 1))
```

**“Problem”** ...

λ-Abstraktionen sind *anonym*.

---

# Lösung: Der Y-Kombinator

Y-Kombinator:  $Y = \lambda f.(\lambda x.(f (x x)) \lambda x.(f (x x)))$

Es gilt: Für jeden  $\lambda$ -Ausdruck  $e$  ist  $(Y e)$  zu  $(e (Y e))$  konvertierbar:

$$\begin{aligned} Y e &\Rightarrow \lambda x.(e (x x)) \lambda x.(e (x x)) \\ &\Rightarrow e (\lambda x.(e (x x)) \lambda x.(e (x x))) \\ &\Leftrightarrow e (Y e) \end{aligned}$$

Mithilfe des Y-Kombinators lässt sich Rekursion realisieren.

*Intuition:*

$$\begin{aligned} f &= \dots f \dots \text{ (rekursive Darstellung)} \\ \rightsquigarrow f &= \lambda f.(\dots f \dots) f \text{ (}\lambda\text{-Abstraktion)} \\ \rightsquigarrow f &= Y \lambda f.(\dots f \dots) \text{ (nicht-rekursive Darstellung)} \end{aligned}$$

*Bemerkung:*

$\lambda$ -Terme ohne freie Variablen heißen *Kombinatoren*.

---

# Zur Übung empfohlen

...Anwendung des Y-Kombinators

Betrachte...

$$\text{fac} = Y \lambda f.(\lambda n.\text{if } n == 0 \text{ then } 1 \text{ else } n * f (n - 1))$$

Rechne nach...

$$\text{fac } 1 \Rightarrow \dots \Rightarrow 1$$

Überprüfe dabei...

Der Y-Kombinator realisiert Rekursion  
durch wiederholtes Kopieren

---

# Praktisch relevant: Typisierte $\lambda$ -Kalküle

Jedem  $\lambda$ -Ausdruck ist ein Typ zugeordnet

Beispiele:

$$\begin{aligned} 3 &:: \text{Integer} \\ (*) &:: \text{Integer} \rightarrow \text{Integer} \rightarrow \text{Integer} \\ (\lambda x. 2 * x) &:: \text{Integer} \rightarrow \text{Integer} \\ (\lambda x. 2 * x) 3 &:: \text{Integer} \end{aligned}$$

*Einschränkung:* Typen müssen konsistent sein (wohltypisiert)

*Problem jetzt:* Selbstanwendung im  $Y$ -Kombinator

$\rightsquigarrow$   $Y$  nicht endlich typisierbar!

*Abhilfe:* explizite Rekursion zum Kalkül hinzufügen mittels Hinzunahme der Reduktionsregel  $Y e \Rightarrow e (Y e)$   
...nebenbei: zweckmäßig auch aus Effizienzgründen!

---

## Zurück zu Haskell...

- Haskell beruht auf typisiertem  $\lambda$ -Kalkül
- Übersetzer/Interpretierer überprüft, ob alle Typen konsistent sind
- Programmierer kann Typdeklarationen angeben (Sicherheit), muss aber nicht (bequem, manchmal unerwartete Ergebnisse)
- fehlende Typinformation wird vom Übersetzer inferiert (berechnet)
- Rekursive Funktionen direkt verwendbar (daher in Haskell kein Y-Kombinator notwendig)

---

## Ergänzende und weiterführende Literaturhinweise

- A. Church. *The Calculi of Lambda-Conversion*. Annals of Mathematical Studies, Vol. 6, Princeton University Press, 1941.
- H.P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. (Revised Edn.), North Holland, 1984.

---

# Zum Abschluss für heute

...λ-artige Funktionsnotation in Haskell

...am Beispiel der Fakultätsfunktion:

```
fac :: Int -> Int
```

```
fac = \n -> (if n == 0 then 1 else (n * fac (n - 1)))
```

Mithin in Haskell: “\” statt “λ” und “->” statt “.”

*Anekdote* (vgl. P. Pepper [4]):

$$(n.\widehat{n} + 1) \rightsquigarrow (\wedge n.n + 1) \rightsquigarrow (\lambda n.n + 1) \rightsquigarrow \backslash n \rightarrow n + 1$$



---

# Vorschau auf die noch kommenden Aufgabenblätter...

Ausgabe des...

- achten Aufgabenblatts: Mo, den 04.12.2006  
...Abgabetermine: Di, den 12.12.2006, und Di, den 09.01.2007, jeweils 15:00 Uhr
- neunten Aufgabenblatts: Mo, den 11.12.2006  
...Abgabetermine: Di, den 09.01.2007, und Di, den 16.01.2007, jeweils 15:00 Uhr (letztes Aufgabenblatt)

---

## Vorschau auf den abschließenden Vorlesungstermin...

- Do, 07.12.2006, Vorlesung von 16:30 Uhr bis 18:00 Uhr im Radinger-Hörsaal (letzte Vorlesung!)

(Der für den 12.12.2006 angekündigte Termin wird nicht benötigt.)