

Abgrenzung

<pre>sum(L){l = length i,acc = 0; while (i < l){ acc = acc + L[i]; i = i + 1} return acc}</pre>	Imperatives Modell basiert auf Zuständen, Zustandsübergängen, deren zeitliche Abfolge Probleme in geeignete Arbeitsschritte zerlegt, in bestimmter Reihenfolge abgearbeitet
<pre>sum [] = 0 sum (a1:rest) = a1 + sum rest</pre>	Funktionales Modell nahe an mathematischer Notation, Definition/Beschreibung anstelle expliziter Anweisungen (deklarativ)

Referenzielle Transparenz

<pre>x = 3</pre>	“Variablen” in funktionaler Programmierung eher Konstanten (imperative Sicht) → Immutability Wert-Variable Relation im funktionalen Paradigma zeitunabhängig Beliebiger Ausdruck kann in jedem Kontext durch seinen unabänderlichen Wert ersetzt werden
<pre>int square(int x){ if (cntr < 3){ cntr++; return x} else { failwith ("enough is enough")}}</pre>	Interne Nebeneffekte ändern Zustand des Programms, kommen in rein funktionalen Sprachen nicht vor
<pre>f n = "Blink LED n times" greet name = println "Hello" ++ name</pre>	Externe Nebeneffekte verändern Zustand des Kontexts (Aussenwelt) in dem Programm eingebettet ist, vom Rest des Programmes isoliert

Solche Nebeneffekte in funktionalen Sprachen möglichst gut isoliert, explizit gekennzeichnet

Typen

<pre>3::Integer 'a'::Char "hi"::String "also hi"::[Char] 3.14::Float ["abc","xyz"]::[String] [1,2,3]::[Integer] [['a']]::[[Char]]</pre>	Typen $\approx$ Mengen, dienen dazu minimale Konsistenz von Programmen zur Kompilierzeit sicherzustellen Angabe einer Menge von Grundtypen, Syntaktische Elemente erlauben Bauen von neuen Typen aus bestehenden, Algorithmen überprüfen ob Programm zu gegebenem Typ passt (type-checker) oder leiten sogar passenden Typ her (type-inference)
<pre>data Customer = Customer {customerId :: Integer, name :: String}</pre>	Records = Tupel, in denen die Einträge Labels (Namen) tragen. Ähnlich zu Structs in C (aber immutable!)
<pre>customerId :: Customer -> Integer name :: Customer -> String</pre>	Records definieren automatisch Funktionen, um auf ihre Datenfelder zuzugreifen
<pre>data Shape = Rectangle Float Float Square Float Circle Float</pre>	In Mengenanalogie → Summentyp Vereinigung disjunkter Mengen In  können Summentypen direkt deklariert werden
<pre>data Tree a = Node (Tree a) a (Tree a) Leaf a</pre>	Summentypen können auch rekursiv sein
<pre>depth :: Tree a -> Integer depth (Node l p r) = 1 + max (depth l) (depth r) depth (Leaf p) = 1</pre>	Rekursive Summen lassen sich mit Rekursion und Patterns dekonstruieren (sog. pattern matching)

Funktionen

<pre>a -> b -> c -> d -- read as a -> (b -> (c -> d))</pre>	Typ einer Funktion liest sich rechtsassoziativ
<pre>Prelude> c f g x = g (f x) Prelude> :t c c :: (t1-> t2)-> (t2-> t3)-> t1-> t3</pre>	Aus Funktionsdefinition (und Kontext) lässt sich Typ der Funktion oft ableiten Typ einer Funktion automatisch abgeleitet → sog. Typinferenz (type inference).
<pre>max1 :: (Int, Int) -> Int max1 (x, y) = if x > y then x else y</pre>	Zwei Sichtweisen auf n-stellige Funktionen: - Funktion, die n-Tupel als Eingabewerte akzeptiert, hat dann einen Typ von der Form $(a_1, \dots, a_n) \rightarrow b$ - Funktion, die $(n - 1)$-stellige Funktionen zurückgibt. Mehrstellige Funktion nach dieser Auffassung hat dann einen Typ von der Form $a_1 \rightarrow (a_2 \rightarrow \dots a_n) \rightarrow b$ sog. currying
<pre>max2 :: Int-> Int-> Int max2 x y = if x > y then x else y</pre>	
<pre>atleast_10 = max2 10</pre>	
$f(x, y) = \frac{x}{y}$	Partielle Funktion $f : X \rightarrow Y =$ Funktion $f : X' \rightarrow Y$, wobei $X' \subseteq X$ Gibt für gewisse Eingaben keinen Funktionswert zurück
<pre>data Maybe a = Just a Nothing</pre>	Viele funktionale Programmiersprachen stellen Typ zur Modellierung von optionalen/partiellen Rückgabewerten bereit Maybe Typ von  = einfacher Summentyp
<pre>f :: Fractional a => a -> a -> Maybe a f x y = case y of 0 -> Nothing _ -> Just \$ x/y</pre>	Partielle/optionale Funktionen können mit Maybe Typ explizit modelliert werden
<pre>//pseudocode, imperative if (someObj == null){ handle exception } else { do normal stuff})</pre>	Durch Verwendung eines Maybe Typs auf Typ-Ebene wird explizit gemacht, dass gegebene Funktion partiell ist.

Im Gegensatz dazu, Unterschied zwischen einer Null-Referenz und einem validen Objekt für das Typensystem “unsichtbar”.

Komponierbarkeit bleibt dank verschiedener Hilfsfunktionen erhalten

Formen der Rekursion

$2^0 = 1$ $2^{n+1} = 2^n \cdot 2$	Rekursive Definitionen zeichnen sich durch Bezugnahme auf das zu definierende Objekt in einer einfacheren Form aus.
$x^0 = 1, c(x) = 1$ $x^{n+1} = x \cdot x^n, G(a, x) = x \cdot a$	Schema der primitiven Rekursion: $f(0, \vec{x}) = c(\vec{x})$ $f(n, \vec{x}) = G(f(n, \vec{x}), n, \vec{x})$
<pre>fib(0) = 0 fib(1) = 1 fib(n) = fib(n - 1) + fib(n - 2)</pre>	Bei Wertverlaufsfunktion wird auf mehrere Vorgänger Bezug genommen Durch Codierung von endlichen Sequenzen als Zahlen, lässt sich jede Wertverlaufsrekursion auch als primitive Rekursion realisieren
$\text{col}(x) = \begin{cases} \frac{x}{2} & \text{falls } x \text{ gerade} \\ 3x+1 & \text{sonst} \end{cases}$	Allgemeine Rekursion Rekursion kann grundsätzlich entlang jeder Relation angewendet werden. Preis, den man dafür bezahlt → Möglichkeit, dass zu definierende Funktion nicht immer “wohldefiniert” ist, insbesondere manchmal keinen Rückgabewert liefert

<pre>length' :: [a] -> Integer length' [] = 0 length' (_:xs) = 1 + length' xs</pre>	Strukturelle Rekursion = Spezialfall der allg. Rekursion
<pre>foldFS :: (Name -> SizeKB -> b) -> (Name -> [b] -> b) -> FS -> b foldFS file dir fs = case</pre>	Dient dazu rekursive Typen zu definieren Verallg. Folds stellen für jeden Summentyp ein passendes Schema für die strukturelle Rekursion zur Verfügung

```
fs of
  File n s -> file n s
  Dir n subs -> dir n $
  map (fs file dir) subs
```

Vereinfacht gesagt → Alles was mit üblichen Mitteln der Programmierung erreicht werden kann auch mittels Rekursion umsetzbar

Aufgrund dieser Tatsache Formalismen, die vollständig auf Iterationen verzichten, aber Rekursion in einer allgemeinen Form unterstützen, trotzdem Turing-vollständig

```
sum_ :: [Integer]->
  Integer
sum_ [] = 0
sum_ (x:xs) = x +
  (sum_ xs)
```

Wichtig nicht für jeden rekursiven Funktionsaufruf den Aufrufstapel zu vergrößern um Stapelüberläufe bei tiefer Rekursion zu verhindern

Tail-call-optimierung behandelt genau diesen Fall

```
sumTR_ :: Integer->
  [Integer]-> Integer
sumTR_ acc [] = acc
sumTR_ acc (x:xs) =
  sumTR_ (x + acc) xs
sumTR = sumTR_ 0
```

Deklaration liegt in **endrekursiver** Form (engl. tail-recursive) vor wenn Resultate von rekursiven Aufrufen direkt zurückgegeben werden → Rekursiver Aufruf ist der letzte

```
fakTR :: Integer-> Integer
fakTR = fakTR_ 1
  where
    fakTR_ :: Integer ->
      Integer-> Integer
    fakTR_ acc 0 = acc
    fakTR_ acc n =
      fakTR_ (n * acc) (n-1)
```

Muster um rekursive Funktion in endrekursive Form zu bringen → Zwischenresultat der Rekursion explizit in Akkumulator mitführen (sog. **Akkumulator Pattern**)

```
fakC :: Integer-> Integer
fakC = fakC_ (const 1)
  where
    fakC_ f n
    | n < 1 = f n
    | otherwise =
      fakC_ (\x->n*(f x))
        $ n - 1
```

Nicht alle rekursiven Funktionsdeklarationen lassen sich mittels (einfachen) Akkumulators in endrekursive Form bringen

Manchmal einfacher, anstelle eines Akkumulators (repräsentiert bereits geleistete Arbeit) Funktionsparameter mitzuführen (repräsentiert noch zu leistende Arbeit)

sog. **Continuation Pattern**

$2x = x$ definiert die Zahl 0

Fixpunkte bieten konstruktiven Zugang zu rekursiv definierten Formalismen Idee des Aufbaus einer rekursiven Struktur von Unten

Definiert als Elemente $x \in X \cup Y$ einer Funktion $F : X \rightarrow Y$ für die gilt $F(x) = x$

Fixpunkte von Funktionen höherer Ordnung

Es sei die Funktion f wie folgt definiert.
 $f(0, \vec{x}) = c(\vec{x})$

$f(n+1, \vec{x}) = g(f(n, \vec{x}), n, \vec{x})$

Dazu passendes (nicht rekursives!) Funktional

$F(f)(0, \vec{x}) = c(\vec{x})$

$F(f)(n+1, \vec{x}) = g(f(n, \vec{x}), n, \vec{x})$

```
expF f x
| x == 0 = 1
| otherwise =
  2 * f (x - 1)
```

```
fibonacciF :: (I -> I) ->
  I -> I
fibonacciF f x
| x <= 1 = 1
| otherwise = f (x-1) +
  f (x-2)
```

In  kann man eine Funktion fix zum finden von Fixpunkten definieren

```
fix :: (a -> a) -> a
fix f = f $ fix f
```

Durch Anwendung von fix auf ein massgeschneidertes (rekursiver Aufruf durch Funktionsparameter f ersetzt), nicht rekursives Funktional erhält man die dazugehörige rekursive Funktion

D.h. fix abstrahiert die Rekursion, sodass sie nur an einem einzigen Ort vorzufinden ist → Bringt Vorteile bei Anwendungen wie Memoization und State Monad

```
collatzF :: (I -> [I]) ->
  I -> [I]
collatzF f x
| x <= 1 = [1]
| otherwise =
  x:(f (next x))
  where
    next x
    | x `mod` 2 == 0 = x
    | `div` 2
    | otherwise = 3*x+1

collatz = fix collatzF

fibonacci = fix fibonacciF
```

Functor, Applicative & Monad

```
class Functor f where
```

```
fmap :: (a -> b) ->
```

```
  f a -> f b
```

```
(<$) = fmap
```

```
(+1) <$> [1,2,3]
```

```
-- [2,3,4]
```

Funktor = eine Typklasse

Alle Instanzen von Funktor müssen Funktion

fmap (<\$> als Infix) implementieren

Functor law identity:

```
id <$> Boxed x = Boxed (id
x) = Boxed x
```

- Identität: $\text{id } \langle \$ \rangle x = x$

- Komposition: $(f.g) \langle \$ \rangle x = f \langle \$ \rangle (g \langle \$ \rangle x)$

Functor law composition:

```
(f . g) <$> Boxed x =
```

```
Boxed $ (f . g) x
```

```
= Boxed $ (f (g x)) = f
```

```
<$> (Boxed (g x)) = f <$>
```

```
(g <$>Boxed x)
```

Zur (Funktor)klasse gehörende, obige Regeln sollten erfüllt werden (wird aber nicht von  überprüft)

```
predec :: Int -> Maybe Int
predec x
```

```
| x <= 1 = Nothing
```

```
| otherwise = Just $ x-1
```

```
h x y = (*) <$> predec x
```

```
<*> predec y
```

Applicative-Klasse hat folgende Regeln:

- Identität: $\text{pure id } \langle * \rangle vv = vv$

- Komposition: $\text{pure } (.) \langle * \rangle f \langle * \rangle g \langle * \rangle x = f \langle * \rangle (g \langle * \rangle x)$

- Homomorphismus: $\text{pure } f \langle * \rangle \text{pure } v = \text{pure } (f v)$

- Interchange: $f \langle * \rangle \text{pure } x = \text{pure } (\$ x) \langle * \rangle f$

```
buildUserB :: Profile ->
```

```
  CityBase -> Maybe User
```

```
buildUserB profile cities
```

```
= myLookup "name" profile
```

```
>>= \n -> myLookup
```

```
"email" profile
```

```
>>= \e -> myLookup
```

```
e cities
```

```
>>= \c -> pure $
```

```
User n e c
```

```
-- with do-notation
```

```
buildUserCM :: Profile ->
```

```
  CityBase -> Maybe User
```

```
buildUserCM profile cities
```

- Linke Identität: $\text{pure } a \gg= f = f a$

- Rechte Identität: $m \gg= \text{pure } = m$

- Assoziativität: $(m \gg= f) \gg= g = m \gg= (\backslash x \rightarrow f x \gg= g)$

Monad-klasse beschreibt Berechnungen mit Kontext (z.B. Fehler, Ein-/Ausgabe, Zustand oder Nichtdeterminismus).

Erlaubt Verkettungen von Operationen, deren Resultate bereits in einem Kontext eingebettet sind

Operator $\gg=$ (bind) führt Berechnung aus, entnimmt ihren Wert und übergibt ihn an

```
= do
```

```
name <- myLookup "name"
```

```
  profile
```

```
email <- myLookup "email"
```

```
  profile
```

```
city <- myLookup email
```

```
  cities
```

```
pure $ User name email
```

```
  city
```

nächste Berechnung, während Kontext automatisch weitergeführt wird

Funktion pure wird im Zusammenhang mit Monaden auch return genannt

λ -Kalkül

$\lambda x. \lambda y. xxy$

$(\lambda xy. xxy)wz \Rightarrow_{\beta} wwz$

—

x

y

$(\lambda x. x)$

Lambda-Kalküle bestehen aus folgendem:

- Terme
- Regeln zur Manipulation von Termen
- Evtl. Typensystem (typisierte Kalküle)

Terme: Variablen, Abstraktionen und Applikationen bilden die Grundbausteine des Kalküls.

$(\lambda x. x)y \Rightarrow_{\beta} y$

Regeln zur Manipulation von Termen:

Insbesondere die β -Reduktion beschreibt die Auswertung von Ausdrücken.

$\lambda x : T. x$

$(\lambda f : A \rightarrow B. \lambda x : A. fx)$

Typensystem: In typisierten λ -Kalkülen werden Variablen und Funktionen mit Typen versehen.

λ -Terme

x, y, π, e

$(x\pi), (xy)$

$\lambda x. (x\pi)y = y\pi$

λ -Terme sind folgendermassen gegeben:

- Jede Variable und jede Konstante ist ein Term
- Sind A und B Terme, dann ist auch (A B) ein Term (Applikation)
- Ist x eine Variable und A ein Term, dann ist auch $\lambda x. A$ ein Term (Abstraktion)

Konventionen bezgl. der Schreibweise von Termen:

- Äusserste Klammern können weggelassen werden
- Applikation linksassoziativ

$A = (A)$

$ABC = ((AB)C)$

$\lambda x. ABC = \lambda x. ((AB)C)$

- Bereich einer Variable grösstmöglich angenommen

Freie und gebundene Variablen

	Die Menge der freien Variablen eines λ - Terms A ist wie folgt gegeben:
$A = \pi, FV(A) = \emptyset$	• Ist A eine Konstante, dann ist $FV(A) = \emptyset$
$A = x, FV(A) = \{x\}$	• Ist A eine Variable v , dann gilt $FV(A) = \{v\}$
$A = xy, FV(A) = \{x, y\}$	• Ist $A = (BC)$, dann ist $FV(A) = FV(B) \cup FV(C)$
$A = \lambda x.(xy), FV(A) = \{y\}$	• Ist $A = \lambda x.B$, dann ist $FV(A) = FV(B) \setminus \{x\}$

Reduktionen

$((\lambda f g x.(f(gx))$ $(\text{add } 3))(\text{add } 2))0 =$ 5	Rechnen im λ -Kalkül im wesentlichen durch Substitution erreicht Vereinfachen eines Terms heisst Konversion (aka. Reduktion), vier Arten ($\alpha, \beta, \eta, \delta$)
$\lambda x.y(\lambda x.x) \Rightarrow_{\beta} y$	Term in Normalform wenn nicht mehr zu vereinfachen. Wir betrachten nur β -Normalformen
$\lambda f g x.(g f x) \Rightarrow_{\alpha} \lambda h g x.(g h x)$	α-Konversion formalisiert Idee, dass Bedeutung eines Terms nicht von verwendeten Variablen sondern nur von seiner Struktur abhängt
$\lambda x.(\text{Add } x x)z \Rightarrow_{\beta} \text{Add } z z$	β-Konversion formalisiert Idee der Funktionsanwendung: Ersetzen von Werten durch Funktionswerte
$\lambda x.(\text{Add } y x) \Rightarrow_{\eta} \text{Add } y$	η-Konversion formalisiert Idee, dass Funktionen, die dieselben Rückgabewerte produzieren gleich sind
$(\text{Add } 14)3 \Rightarrow_{\delta} 17$	δ-Konversion bestimmt die Bedeutung von Konstanten durch Berechnung

Reihenfolge in der Reduktionen angewendet werden können im Allg. nicht eindeutig

Normal Order Reduction entspricht Reduktionsstrategie von links nach rechts,
Lazy evaluation = Variante dieser Strategie

$$\lambda x.y(\lambda w.(w w)\lambda x.x) \Rightarrow_{\beta} y$$

$$\lambda x.y(\lambda w.(w w)\lambda x.x) \Rightarrow_{\beta}$$

$$\lambda x.y(\lambda x.x\lambda x.x) \Rightarrow_{\beta}$$

$$\lambda x.y\lambda x.x \Rightarrow_{\beta} y$$

Applicative Order Reduction entspricht Reduktionsstrategie von innen nach aussen,
Strict evaluation = Variante dieser Strategie

Rechnen mit Konversionen

$\lambda x.y(\lambda z.(z z z)\lambda z.(z z z)) \Rightarrow_{\beta} y$	Term besitzt β -Normalform $\Rightarrow$ kann immer mittels <i>Normal Order Reduction</i> gefunden werden
$\lambda x.y(\lambda z.(z z z)\lambda z.(z z z)) \Rightarrow_{\beta}$	
$\lambda x.y(\lambda z.(z z z)\lambda z.(z z z))$ $\lambda z.(z z z)) \Rightarrow_{\beta} \dots$	Es gibt Terme, die eine β -Normalform besitzen, welche nicht durch <i>applicative order reduction</i> gefunden werden kann
$Y! \equiv$ $\lambda f.(\lambda x.(f(x x))\lambda x.(f(x x)))$	Keine expliziten Konstrukte um rekursiv Funktionen zu deklarieren
$\text{fix} :: (a \rightarrow a) \rightarrow a$ $\text{fix } f = f \$ \text{fix } f$	Rekursion wird über Fixpunkte erreicht (äquivalent zu $\text{fix in } \lambda$)
$0 = \lambda f x.x$ $1 = \lambda f x.f x$ $2 = \lambda f x.f(f x)$ $\dots$	Konstrukte und Datentypen der Programmierung wie While-Schleifen(via Rekursion), Natürliche Zahlen (Church Numerele) lassen sich im λ -Kalkül simulieren