Einleitung

Imperativ vs. Funktional

- ⇨ Imperativ: Zustände, Zustandsübergänge und zeitliche Abfolge («wie»)
- ⇨ Funktional: Nahe an mathematischen Notation (Funktion sind mathematische Objekte), Keine veränderliche Variablen (bezeichnen Werte und nicht Speicherbereiche)

Referenzielle Transparenz

Die Zuweisung x = 3 im Vergleich:

- Funktional: Der "Name" x benennt (in seinem Kontext), unabhängig von der Zeit, den **Wert** 3.
- Imperativ: Der "Name" x benennt einen **Ort** (Speicherbereich). Sein Wert ändert sich mit der Zeit, je nachdem was in besagtem Speicherbereich steht.

- ⇨ Konsequenz: Wert-Variable-Relation ist im funktionalen Paradigma zeitunabhängig (Variablen entsprechen eher Konstanten)
- ⇨ Nebeneffekte / Seiteneffekte in Funktionen kommen in der funktionalen Programmierung nicht vor
- ⇨ Externe Nebeneffekte (println etc.) werden in funktionalen Sprachen möglichst gut isoliert und explizit gekennzeichnet
- ⇨ Imperativ oft nicht ref. Transp. (bspw. x=3;x+=1;print x + 4; => print 3 + 4)
- ⇨ Da Nebeneffektfrei: Eine Variable kann in einem Ausdruck immer (im jeweiligen Kontext) mit ihrem Wert ersetzt werden (Ausdruck verändert sich dadurch nicht), Der Wert eines Ausdrucks hängt nur von seinen Teilausdrücken ab, Die Evaluation eines Ausdruckes ist unabhängig von Reihenfolge in der seine Teilausdrücke evaluiert werden
- ⇨ Vorteile von Referenzieller Transparenz: Einfache Programmverifikation (da wenige Abhängigkeiten), Flexibilität beim Auswerten von Ausdrücken (lazy evaluation), Erleichtert Beweisführung (bspw. für Code-Optimierungen), Erleichtert Programme zu verstehen

Typen und Typensysteme

- ⇨ Typen ~ Mengen
- ⇨ Bspw. anstatt x element von T schreibt man x :: t (in Haskell)
- ⇨ Sie dienen dazu eine «minimale Konsistenz» von Programmen zur Kompilierzeit sicherzustellen

Typensystem Spezifikationsschema:

- ⇨ Angabe einer Menge von «primitiven»- bzw. «Grundtypen» (z. B. Int, Char, Float etc.)
- ⇨ Syntaktische Elemente, um aus bestehenden Typen neue zu bauen
- ⇨ Algorithmen die überprüfen ob ein Programm zu einem gegebenen Typ passt (Type-Checker) oder sogar einen passenden Typ herleiten (Type-Inference)

Haskell Typensystem Beispiel (informell):

- ⇨ Grundtypen: Bool, Char, String …
- ⇨ Zusammengesetzte Listentypen: [«abc», «xyz»] ist vom Typ [String], [[‘a’]] vom Typ [[Char]] etc.
- ⇨ Zusammengesetzte Funktionstypen: Der Typ String -> Bool beinhaltet Funktionen, der Strings akzeptiert und True oder False zurückgibt etc. => Funktionstypen lassen sich beliebig verschachteln und eine Funktion vom Typ a -> Bool nennt man Prädikat auf a.
- ⇨ Zusammengesetzte Tupeltypen: (Int, String, Char) beinhaltet das Tripel (1, «1», ‘1’) etc. (lassen sich auch beliebig verschachteln).
- ⇨ Zusammengesetzte Recordtypen: Sind Tupel, in denen die Einträge Labels (Namen) tragen.
- ⇨ Zusammengesetzte Summentypen: Entspricht der Vereinigung von diskunkten Mengen (lassen sich mit Rekursion und Pattern dekonstruieren = Wert «herausnehmen»)

```haskell
-- Tiefe eines Baumes
depth :: Tree a -> Integer
depth (Node left _payload right) =
    1 + max (depth left) (depth right)
depth (Leaf _payload) = 1
```

```haskell
data Tree a
    = Node (Tree a) a (Tree a)
    | Leaf a
```

```haskell
data List a
    = Cons a (List a)
    | Nil
```

Typklassen:

- ⇨ Zusammenfassung von verschiedenen Typen, die eine bestimmte Eigenschaft teilen (ähnlich einem Interface in Java)
- ⇨ Beispiel: Typklasse Eq enthält alle Typen, deren Elemente vergleichbar sind (bspw. Signatur der Funktion «==» mit (==) :: Eq a => a -> a -> Bool d.h. a muss eine Instanz von Eq sein)
- ⇨ Typ zu Instanz der Eq Typklasse machen: Man muss selbst die Funktionen der Typklasse implementieren oder automatisch mit deriving Instanz erzeugen
- ⇨ Bspw.: Wir definieren den Typ:
  data Person = Person String Int
  Manuell die Instanz schreiben mit:
  Instance Eq Person where (Person name1 age1) == (Person name2 age2) = name1 == name2 && age1 == age2
  (Personen sind demnach gleich wenn Alter und Name gleich => Man implementiert demnach == selbst)
  Oder direkt mit deriving:
  data Person = Person String Int deriving (Eq) (=> Hier wird automatisch eine sinnvolle Eq Instanz erzeugt)
- ⇨ (Person name1 age1) => name1 wird quasi so mit Patternmatching aus Person herausgeholt (definiert als String)

Funktionen (in Haskell)

Übersicht

- ⇨ Funktionstypen werden rechtsassoziativ gelesen (Bspw. a -> b -> c -> d wird als a -> (b -> (c -> d)) interpretiert)

Funktionstypen von Hand bestimmen

```
a b c d e = c (b d) (b e)
```

- d und e haben irgendwelche Typen: `d:D` und `e:E`
- Weil wir b sowohl auf d sowie auch auf e anwenden, ist `E = D` und der Typ von b von der Form `b:D->B`.
- Weil wir c auf zwei Argumente vom Typ B anwenden, ist der Typ von c von der Form `c:B -> B -> C` (insbesondere ist der Term `a b c d e` vom Typ C).
- Für den Term a erhalten wir daher durch Einsetzen den Typ `a: (D -> B) -> (B -> B -> C) -> D -> D -> C`

Beispiele:

```
a1 b = b (17 :: Int)
```
b: Int -> B
a1: (Int -> B) -> B

```
a2 b = b 15
```
b: A -> B
a2: (A -> B) -> B

```
a3 b c = c b
```
c: B -> C
b: B
a3: B -> (B -> C) -> C

```
x y z = y (z y)
```
z: Y -> Z
y: Z -> Y => KEINE TYPINFERENZ HIER MÖGLICH
x: (Z->Y) -> (Y->Z) -> Y

Begriffe

Currying / Uncurrying: Es gibt 2 Ansätze auf mehrstellige Funktionen (n Argumente und einen Return oder 1 Argument und n-1 returns (= Curry))

```
curry f a1 .. an = f (a1,..,an)
uncurry f (a1,.., an) = f a1 .. an
```

Partielle Anwendung: Eine curried Funktion nicht erschöpfend mit Argumenten zu versehen (Anwendung von mehrstelligen curried Funktionen) => Nichts zu tun mit partiellen Funktionen => Vor allem für generischen Code

Funktionen höherer Ordnung: Erhält oder gibt Funktionen zurück (bspw. Komposition ist eine Anwendung davon)

Folds (Faltung): Reduktion eines Typs in einen andern (bspw. Baum in einen Int). Beispiel:
=> data Tree = Leaf Int | Node Tree Tree
=> 1. Konstruktoren analysieren:
Leaf :: Int -> Tree
Node :: Tree -> Tree -> Tree
=> 2. Ersetze Tree durch einen Typparameter b
Leaf :: Int -> b
Node :: b -> b -> b
=> 3. Erstelle Funktion mit diesen Arg. Typen + Fold Funktion
foldTree :: (Int -> b) -> (b -> b -> b) -> Tree -> b

Partielle Funktion: Eine partielle Funktion $f : X' \to Y$ ist eine Funktion $f : X \to Y$, wobei $X' \subseteq X$

- ⇨ Gibt deshalb evtl. für gewisse Eingaben keinen Funktionswert zurück

- ⇨ Bspw. f(x,y) = x/y ist partiell, weil / 0 nicht definiert
- ⇨ Dies ist ein Anwendungsfall vom Maybe Datentyp (Optionale Werte):
  data Maybe a = Just a | Nothing
  (Summentyp)

```
f :: Fractional a => a -> a -> Maybe a
f x y = case y of
    0 -> Nothing
    _ -> Just (x / y)
```

- ⇨ Bei einer Null-Referenz denkt das Typensystem bspw. bei String name = null das name ein String ist => Es bleibt somit «unsichtbar» und crasht bspw. bei .length() Aufruf per Laufzeit. Mit Maybe zwingt man den Aufrufer mit dem Fall Nothing umzugehen (dass ist nicht mehr unsichtbar) => Dadurch werden alle Funktionen total
- ⇨ Als Erweiterung gibt es den Either Typ (um Informationen zu übergeben, wieso eine Funktion an einer bestimmten Stelle keinen geeigneten Returnwert liefert

```
data Either a b
    = Left a  -- Fehler mit Information a
    | Right b -- Das gute Resultat

data Result a b c =
    = Success a
    | XError b
    | YError c
```

Rekursionsschemata

Einfaches Schema

$$f(0) = c$$
$$f(n+1) = G(f(n))$$

Primitive Rekursion (LOOP)

$$f(0, \vec{x}) = c(\vec{x})$$
$$f(n+1, \vec{x}) = G(f(n, \vec{x}), n, \vec{x})$$

wobei

- c eine Funktion (in den Variablen $\vec{x}$)
- G eine Funktion (ein Algorithmus)

Wertverlaufsrekursion

⇨ Bezug auf mehrere Vorgänger

$$fib(0) = 0$$
$$fib(1) = 1$$
$$fib(n) = fib(n-1) + fib(n-2)$$

⇨ Spezialfall der Wertverlaufsreduktion (dort kann sogar auf jeden Vorgänger Bezug genommen werden)

$$f(n) = G(f \restriction n)$$

$$f(n, \vec{x}) = G(f \restriction n, n, \vec{x})$$

⇨ Jede Wertverlaufsrekursion lässt sich auch durch eine primitiv rekursive Funktion darstellen (durch Kodierung)

Allgemeine Rekursion

⇨ Rekursion kann grundsätzlich entlang jeder Relation angewendet werden (dadurch besteht jedoch die Möglichkeit dass die definierende Funktion nicht immer wohldefiniert ist und manchmal keinen Rückgabewert liefert) Bspw. Collatz

⇨ Strukturelle Rekursion: Rekursive Typen dekonstruieren (bspw. _:xs)

⇨ Alle rekursiven Funktionen lassen sich in iterative umwandeln und sind deshalb Turing-vollständig

Endrekursion

Idee

⇨ Eine Code-Optimierung, um Rekursion in Iteration zu übersetzen

⇨ Wichtig ist dabei insbesondere, nicht für jeden rekursiven Funktionsaufruf den Aufrufstapel zu vergrössern => Verhinderung der Stapelüberläufe bei tiefer Rekursion

⇨ Endrekursion liegt vor, wenn Resultate von rekursiven Aufrufen direkt zurückgegeben werden.

⇨ Die meisten Funktionalsprachen-Compiler übersetzen Endrekursion so, dass bei deren Ausführung konstanter Stapelspeicher in Anspruch genommen wird

Akkumulator Pattern

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

⇨ Nicht endrekursiv, da rekursives Resultat weiter als Summand verwendet wird

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

sumTR = sumTR_ 0
```

⇨ Nun ist es endrekursiv mittels zusätzlichem Parameter «Akkumulator»

⇨ Idee: Zwischenresultat der Rekursion in einem Akkumulator mitzuführen

Beispiele:

```
pow x y
    | y < 1 = 1
    | otherwise = x * pow x (y - 1)
```

```
pow x y = powAcc 1 x y
 where
  powAcc acc x y
   | y < 1    = acc
   | otherwise = powAcc (x * acc) x (y - 1)
```

⇨ Wir starten mit acc=1 weil 1 neutrales Element der Multiplikation

⇨ Da immer ein Akkumulator vorhanden und der erste Funktionsaufruf rekursiv sein muss = Guter Startpunkt für Umwandlung

```
isPalindrome :: String -> Bool
isPalindrome w
    | l < 2 = True
    | otherwise =
        w0 == wE && isPalindrome w'
    where
        l  = length w
        w0 = head w
        wE = last w
        w' = tail $ init w
```

```
isPalindrome :: Integer -> String -> Bool
isPalindrome w = isPalindrome' True w
  where
    isPalindrome' acc w
      | not acc        = False (Optimierung)
      | l < 2 = acc
      | otherwise = isPalindrome' (w0 == wE && acc) w'
       where l = … (gleich)
```

⇨ True wird im nicht endrekursiven Fall erst beim Zurückkehren «hochgerechnet»

⇨ Beim Akkumulator Fall tragen wird die bisherige Wahrheit aktiv mit

Bemerkung:

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

⇨ Wir berechnen das Ergebnis direkt beim Aufbau des Call-Baums statt zu warten, bis wir wieder hochklettern.

Continuation Pattern

⇨ Anstatt wie im Akkumulator einen Parameter zu führen der die bereits geleistete Arbeit repräsentiert einen Parameter verwenden, der die noch zu leistende Arbeit darstellt

⇨ Man übergibt die Restberechnung (was danach noch passieren soll) als Funktion weiter

Beispiel:

Akkumulator:

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

Continuation:

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

Weitere Beispiele:

```
myMap :: (a -> b) -> [a] -> [b]
myMap f [] = []
myMap f (x:xs) = (f x):(myMap f xs)
```

```
myMapC :: ( a -> b ) -> [a ] -> [b]
myMapC f = mmc id
  where
    mmc g [] = g []
    mmc g (x : xs ) = mmc (\as -> g (( f x): as )) xs
```

Fixpunkte

Haskell Fixpunkt Funktional

- ⇨ Fixpunkt ist eine Funktion f(x) = x (x wird nicht verändert)
- ⇨ Beispiel eine f = Zahl halbieren solange grösser 10: f(20) = 10, f(10) = 10, f(5) = 5 (10 und 5 sind Fixpunkte)
- ⇨ Viele rekursive Definitionen lösen eine Gleichung der Form x = f(x), dabei ist x oft eine selbstreferenzierende Funktion
- ⇨ Um Rekursion zu definieren, such man also nach dem Fixpunkt der Funktion, die einen Schritt der Rekursion beschreibt

Haskell Funktional:

```
fix f = f $ fix f
```

- ⇨ fix nimmt eine Funktion f als Eingabe
- ⇨ fix f ist ein Wert x der genau so definiert ist, dass x = f x gilt (also ein Fixpunkt von f)
- ⇨ «Fixpunkt von f ist die Funktion f angewendet auf den Fixpunkt von f»: Haskell hat Lazy Evaluation => fix f erstellt eine potenzielle unendliche Struktur, die aber nur soweit ausgewertet wird, wie sie gebraucht wird (unendlich lange Laufzeit wenn Fixpunkt nicht existiert)

Functor, Applicative, Monad

Functor (Typklasse)

- ⇨ Functor Typklasse:

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

- ⇨ Functor ist ein abstrakter Datentyp, auf den man eine Funktion anwenden kann, ohne ihn zu entpacken.
- ⇨ Hier erlaubt fmap eine Funktion auf einen eingepackten Wert anzuwenden
- ⇨ Jeder Typ f der in Functor sein will muss eine Funktion fmap implementieren. fmap nimmt eine Funktion von a nach b sowie ein f a (also Wert vom Typ a der im Kontext f eingebettet ist) und gibt ein f b zurück (also ein Wert vom Typ b ebenfall im Kontext f)
- ⇨ Bspw. Maybe als Funktor:
  instance Functor Maybe where
  fmap f Nothing = Nothing
  fmap f (Just x) = Just (f x)
- ⇨ Wenn Nothing bleibt Nothing, wenn Just x dann wird die Funktion f auf x angewendet
- ⇨ fmap (+1) (Just 3) -- ergibt Just 4
- ⇨ fmap (+1) Nothing -- ergibt Nothing
- ⇨ f ist eine Art «Container» bzw. Kontext (bspw. Maybe, List, IO etc.). fmap erlaubt dann eine Funktion in diesem Container anzuwenden, ohne ihn zu öffnen
- ⇨ fmap f (Just x) = Just (f x)
  f ist (a -> b), Just ist f und x ist a (f x) gibt somit b
- ⇨ Mit einem Functor kann man eine Funktion (bspw. (+1) auf einen Wert anwenden der in einem Kontext steckt (bspw. Just 3) anwenden (also (+1) auf 3 anwenden) so dass man jedoch den Kontext also Just beibehält (Just ist selbst eine Funktion im Functor Kontext Maybe).
- ⇨ fmap als infix mit «<$>»
- ⇨ Weiteres Beispiel:
  (+1) [1 ,2 ,3] == [2 ,3 ,4]
- ⇨ [1,2,3] ist eine Liste und Listen sind Funktoren
- ⇨ instance Functor Datentyp where fmap … definiert den Datentyp so, dass ich map darauf anwenden kann
- ⇨ Listen fmap sind bspw. rekursiv definiert (mit map Funktion) auf jedes Element

■ Identität

```
fmap id = id
id <$> x = x
```

■ Komposition

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

FUP

Functor: Typ, auf den man eine Funktion mappen kann (Funktion auf Wert in einem Kontext anwenden ohne den Kontext zu verändern
Applicative: Erweiterung von Functor, mit der man Funktionen in einem Kontext auf Werte in einem Kontext anwenden kann + Kombi (die einzige Änderung ist, dass neben (a->b) auch ein f (applicative typ) mitgegeben wird, um Verkettung zu unterstützen da output auch Applicative Typ + b
Monad: Erweiterung von Applicative, bei der man den vorherigen Wert kennen darf, bevor man nächsten Berechnungsschritt macht (Im Monad is nun ein Wert im Kontext (m a) bspw. Just 4 und man definiert Funktion a->m b, die den 4 nutzt aber wieder einen Kontext zurück gibt)

Applicative Functor (Typklasse)

- ⇨ Problem:

```
h x y = (*) (predec x) (predec y)
```

Funktioniert nicht, weil * nicht direkt auf Int anwendbar ist (* erwartet Int aber predec x ist Maybe Int)

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

Funktioniert auch nicht, da (*) (predec x) einen Maybe (Int -> Int) zurückgibt aber predec y nur Maybe Int ist.

- ⇨ Applicative Functor Klasse bietet genau das an (eine Funktion <*> mit Maybe (Int->Int) nach Maybe Int nach Maybe Int)

```
class (Functor f) => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

- ⇨ => = f darf nur als Applicative verwendet werden wenn es auch die Functor Bedingung erfüllt (Implikation)
- ⇨ Maybe als Applicative:

```
pure = Just
Just f <*> x  = f <$> x
Nothing <*> _ = Nothing
```

- ⇨ pure nimmt einen normalen Wert und packt ihn in einen Kontext
- ⇨ n Stellige Funktionen:

```
mul3 x y z = x * y * z

-- d x y z = (x-1) * (y-1) * (z-1)
d x y z =
    mul3 <$> predec x <*> predec y <*> predec z
```

- ⇨ Applicative Functor Typ:

```
(<*>) :: Maybe (Int -> Int) -> Maybe Int ->
   Maybe Int
```

- ⇨ mul Beispiel: mul3 <$> predec x gibt uns eine Funktion Maybe (Int -> Int -> Int -> Int), die anschlissende Funktion <*> nimmt Int -> Int -> Int -> Int als a (rot markierter Teil) und b (blau markierter Teil). Das predec y (was ein Maybe Int ist) definiert den Int für a und gibt den Maybe (Int -> Int -> Int) zurück
- ⇨ = Partielle Anwendung => Bspw. predec x = Just 2 dann mul3 <$> Just 2 ist ein Funktor-Mapping mit a = 2, d. h. fmap mul3 (Just 2) = Just (mul3 2), wobei dann mul3 teilweise mit dem Wert 2 als erstes Argument bereits angewendet wurde und dann uns ein Just (Int -> Int -> Int) zurück gibt

Beispiel Verwendung (ohne und mit Applicative):

```
buildUser :: Profile -> Maybe User
buildUser prof = case myLookup "name" prof of
   Nothing -> Nothing
   Just name -> case myLookup "email" prof of
      Nothing -> Nothing
      Just email -> case myLookup "city" prof of
         Nothing -> Nothing
         Just city -> Just $ User name email city
```

```
buildUser profile = User
    <$> myLookup "name" profile
    <*> myLookup "email" profile
    <*> myLookup "city" profile
```

- ⇨ Geht, weil wenn nicht vorhanden = Nothing und Nothing <*> f = Nothing

- Identität

```
pure id <*> vv = vv
```

- Komposition

```
pure (.) <*> f <*> g <*> x = f <*> (g <*> x)
```

- Homomorphismus

```
pure f <*> pure v = pure (f v)
```

- Interchange

```
f <*> pure x = pure ($ x) <*> f
```

Monad (Typklasse)

- ⇨ Problem: Pipeline hat jeweils unabhängige Werte; Was ist wenn wir aber bspw. im Beispiel vorher die Stadt in einer Map speichern (mit Email als Key)? Dann bräuchte man Email bevor wir Stadt definieren. Von Hand:

```
buildUserC :: Profile -> CityBase -> Maybe User
buildUserC p cities = case myLookup "name" p of
   Nothing -> Nothing
   Just name -> case myLookup "email" p of
      Nothing -> Nothing
      Just email -> case myLookup email cities of
         Nothing -> Nothing
         Just city -> Just $
            User name email city
```

- ⇨ Wir brauchen dafür folgende Signatur: (bind = >>=)

```
bind :: Maybe String -> (String -> Maybe String)
   -> Maybe String
```

- ⇨ Damit kann man Funktionen vom Typ String -> Maybe String hintereinander ausführen
- ⇨ Monad Typklasse gibt uns das passende Interface:

```
class (Applicative m) => Monad m where
    (>>=) :: m a -> (a -> m b) -> m b
```

```
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
```

- ⇨ myLookup «name» profile gibt ein Maybe **name** zurück dass dann als Eingabe im \n -> … genutzt wird
- ⇨ Auch mit do Notation und <- möglich

- "left identity"

```
pure a >>= f = f a
```

- "Right identity"

```
m >>= pure = m
```

- Assoziativität:

```
(m >>= f) >>= g = m >>= (\x -> f x >>= g)
```

Bemerkung

Die Funktion `pure` wird im Zusammenhang mit Monaden auch `return` genannt.

DSL / EDSL und Kombinatoren-bibliotheken

DSL / EDSL

- ⇨ Idee: Industrifachleute können so mit speziellen Programmiersprachen (die die Fachwörter ihrer Domäne haben) mit automatisieren
- ⇨ EDSL = Embedded DSL
- ⇨ DSL ist eine eigenständige formale Sprache und haben eigene Compiler etc.
- ⇨ EDSL ist in einer bestehenden Programmiersprache eingebettet
- ⇨ Terme der EDSL sind auch Terme der Hostsprache (echte Teilmenge)
- ⇨ Typen der Hostsprache stehen für Elemente der abgebildeten Domäne, Funktionen werden zum kombinieren und manipulieren dieser Elemente verwendet
- ⇨ Haskell bietet geeignete Umgebung für EDSL da ausdruckstarkes Typensystem und Möglichkeit zur Abstraktion (bspw. Typklassen, höhere Funktionen etc. und nicht wie in Java main Funktion nötig etc.)

Beispiel EDSL für 2D-Grafiken

Grundformen:

```
-- | Basic Shapes

empty :: Shape

unitDisc :: Shape

unitSq :: Shape
```

Modifikatoren:

```
translate :: Vector -> Shape -> Shape
stretchX  :: Float  -> Shape -> Shape
stretchY  :: Float  -> Shape -> Shape
stretch   :: Float  -> Shape -> Shape
flipX     :: Shape  -> Shape
flipY     :: Shape  -> Shape
flip45    :: Shape  -> Shape
flip0     :: Shape  -> Shape
rotate    :: Float  -> Shape -> Shape
```

Kombinatoren:

```
intersect :: Shape -> Shape -> Shape

merge :: Shape -> Shape -> Shape

minus :: Shape -> Shape -> Shape
```

Syntax somit bereits beschriebn

Beispiel Form:

```
iShape = merge
    (stretchY 2 unitSq)
    (translate (0, 4) unitDisc)
```

Semantik:

⇨ Formen, die wir in Sprache beschreiben sollte als Bild wiedergegeben werden

```
type Point = (Float, Float)

newtype Shape = Shape { inside :: Point -> Bool }
```

⇨ Dies nennt man «Shallow» Embedding. Bei einem «Deep Embedding» entspricht die EDSL einem Datentyp in Haskell:

```
data Shape
    = Empty
    | UnitDisk
    | UnitSq
    | Translate Vector Shape
    | ...
```

⇨ Dadurch wird Syntax und Semantik klarer getrennt

Abstraktion mit Applicative / Monad

⇨ Anwendung beim Fehlermanagement

Viel "Handarbeit" beim Fehlerhandling:

```
evalE :: Exp -> Result Int
evalE e = case e of
    Const x -> Success x
    Div e1 e2 -> case evalE e1 of
      DivisionByZeroError -> DivisionByZeroError
      Success x1 -> case evalE e2 of
        DivisionByZeroError ->
           DivisionByZeroError
        Success x2
          | x2 /= 0 -> Success $ x1 `div` x2
          | otherwise -> DivisionByZeroError
    Add e1 e2 -> case evalE e1 of
      DivisionByZeroError -> DivisionByZeroError
      Success x1 -> case evalE e2 of
        DivisionByZeroError -> ...etc.
```

```haskell
instance Functor Result where
   fmap f (Success x) = Success $ f x
   fmap f _ = DivisionByZeroError
```

```haskell
instance Applicative Result where
   Success f <*> Success x = Success $ f x
   _ <*> _ = DivisionByZeroError
   pure = Success
```

```haskell
instance Monad Result where
   Success x >>= f = f x
   _ >>= _ = DivisionByZeroError
```

```haskell
eval2 :: Exp -> Result Int
eval2 e = case e of
  Const x -> pure x
  Add e1 e2 -> (+) <$> eval2 e1 <*> eval2 e2
  Mul e1 e2 -> (*) <$> eval2 e1 <*> eval2 e2
  Div e1 e2 -> do
    x1 <- eval2 e1
    x2 <- eval2 e2
    if x2 == 0 then
        DivisionByZeroError
    else
        return $ x1 `div` x2
```

Result Int

Monad

- ⇨ Durch Monad, weiss man, das man stoppen muss wenn DivisionByZeroError (man gibt dann kein x mehr weiter)
- ⇨ pure wendet quasi nur den Funktor Result mit x an (damit als Return der Typ Result Int wird)

λ-Kalkül

Idee

- ⇨ Implementieren Algorithmen als evaluation of expressions
- ⇨ Turing vollständig
- ⇨ Funktionale Sprachen = Varian. davon
- ⇨ Implementiert höhere Funktionen, Funktionsdefinitionen und -anwendungen
- ⇨ Einfach und ausdrucksstark
- ⇨ Können auch typisiert sein (Erweiter.)

Terme (Programme)

- ⇨ Alphabet: Bestehend aus unendlich vielen Variablen x, y, z, v, v1, x1 …, dem Zeichen λ, dem Punkt und Klammern (Fakultativ Konstanten)

Regeln:

- ⇨ Jede Variable und jede Konstante ist ein Term
- ⇨ Sind A und B Terme, dann ist auch (A B) ein Term (= Applikation), (A B) bedeutet A angewendet auf B (wie in Haskell mit (f x)
- ⇨ Ist x eine Variable und A ein Term, dann ist auch λx.A ein Term (Abstraktion), Entspricht einer anonymen Funktion \x -> A (aufgrund der Haskell Typisierung lassen sich nicht alle Terme in Haskell realisieren (bspw. λx.(x x))

Terme als Haskell Datentyp:

```
data Term
   = Add
   | Var String
   | Applikation Term Term
   | Abstraktion String Term
   | N Integer
```

Konventionen:

- ⇨ Äusserste Klammern können weggelassen werden
- ⇨ Applikation ist linksassoziativ
- ⇨ Der Bereich einer quantifizierten Variable wird grösstmöglich angenommen (λx.A B C wird als λx.((A B) C) gelesen)
- ⇨ Terme von der Form λxyz.A werden als λx.λy.λz.A gelesen

Beispiel:

λxy.ABλu.A

=

λx.λy.((A B) (λu.A))

Regeln (zur Manipulation von Termen)

Freie und gebundene Variablen:

Die Menge der freien Variablen $FV(A)$ eines λ-Terms A ist wie folgt gegeben:

- Ist A eine Konstante, dann ist $FV(A) = \varnothing$.
- Ist A eine Variable v, dann gilt $FV(A) = \{v\}$.
- Ist $A = (B\ C)$, dann ist $FV(A) = FV(B) \cup FV(C)$.
- Ist $A = \lambda x.B$, dann ist $FV(A) = FV(B) \setminus \{x\}$.

Variablen, die in einem Term vorkommen aber nicht frei sind heissen gebundene Variablen.

- ⇨ Im untersten Beispiel wäre das x gebunden da in Argumentenliste

Substitution:

- ⇨ Den Term A[x:=B] erhält man aus dem Term A, indem man alle freien Vorkommen der Variablen x in A durch B ersetzt
- ⇨ Eine Substitution A[x:=B] ist zulässig, wenn keine der freien Variablen von B durch die Substitution gebunden werden
- ⇨ Bspw.:

- Eine unzulässige Substitution

$$\lambda xy.(x\ y\ z)[z := x]$$

- Eine zulässige Substitution

$$\lambda xy.(x\ y\ z)[z := u]$$

- ⇨ Durch geeignetes Umbenennen von Variablen lässt sich stets zulässig substituieren
- ⇨ Rechnen im Lambda Kalkül wird im wesentlichen durch Substitution erreicht:

anwenden

$$(((\lambda fgx.(f\ (g\ x))\ (add\ 3))\ (add\ 2))\ 0)$$

$\lambda fgx.(f\ (g\ x))\ [f := add\ 3]$
$= \lambda gx.((add\ 3)\ (g\ x))$

Etc...

- ⇨ «Anwendung» = Substitution

Reduktionen:

- ⇨ Ausrechnen = Substitutionen bis die Terme in einer vereinfachten Form vorliegen
- ⇨ Diese Vereinfachung nennt man auch Konversion und wir werden die folgenden vier Arten betrachten (α, β, η, δ).
- ⇨ Das Ziel dabei ist, einen nicht mehr zu vereinfachenden Term zu bekommen (= eine Normalform). Wir werden die β-Normalform betrachten

α-Reduktion:

Die α-Konversion:

$$\lambda x.A \Rightarrow_\alpha \lambda y.A[x := y]$$

wenn y nicht in A vorkommt.

- ⇨ Idee, dass die Bedeutung eines Terms nicht von den verwendeten Variablen sondern nur von seiner **Struktur** abhängt (äquivalente Terme wie λx.x und λy.y ineinander überführen)
- ⇨ Beispiel:

$$\lambda fgx.(g\ f\ x) \Rightarrow_\alpha \lambda hgx.(g\ h\ x)\ [f := h]$$

- ⇨ «Wenn eine gebundene Var in einer Funktion mit einer anderen freien Var kollidiert, kann man gebundene Var umbenennen» => y nicht in a

β-Reduktion:

Die β-Konversion:

$$(\lambda x.A\ B) \Rightarrow_\beta A[x := B]$$

wobei die Substitution zulässig sein muss.

- ⇨ Idee: Formalisierung der Funktionsanwendung durch Ersetzen von Werten durch entsprechende Funktionswerte
- ⇨ Beispiel:

$$(\lambda x.(Add\ x\ x)\ z) \Rightarrow_\beta (Add\ z\ z)\ [x := z]$$

$$(\lambda x.x + 1)\ 4 \xrightarrow{\beta} 4 + 1$$

- ⇨ = Funktionsauswertung
- ⇨ «Wenn eine Funktion auf ein Arg angewendet wird, ersetzt man die gebundene Var durch das Arg»

η-Reduktion: (Eta)

Die η-Konversion:

$$(\lambda x.A\ x) \Rightarrow_\eta A$$

wobei $x \notin FV(A)$ gelten muss.

- ⇨ Idee: Funktionen, die gleiche Rückgabewerte produzieren sind gleich
- ⇨ Beispiel:

$$\lambda x.(Add\ y\ x) \Rightarrow_\eta (Add\ y)$$

- ⇨ «Wenn eine Funktion nichts anderes tut, als ein Argument an eine andere Funktion weiterzugeben, kann man sie weglassen»

δ-Reduktion:

- ⇨ Bestimmen Bedeutung von Konstanten
- ⇨ Beispiel:

$$(Add\ 14)\ 3 \Rightarrow_\delta 17$$

Rechnen mit Konversion:

$$\begin{aligned} & \lambda f.\lambda x.(f\ (f\ x))\ (Add\ 3)\ 2 \\ \Rightarrow_\beta\ & \lambda x.((Add\ 3)\ ((Add\ 3)\ x))\ 2 \\ \Rightarrow_\beta\ & ((Add\ 3)\ ((Add\ 3)\ 2)) \\ \Rightarrow_\delta\ & ((Add\ 3)\ 5) \\ \Rightarrow_\delta\ & 8 \end{aligned}$$

- ⇨ Einen Term, auf den wir eine β-Reduktion anwenden können nennen wir β-Redex
- ⇨ Die Übergänge sind die eigentlichen Reduktionen und die Konzepte Konversionen
- ⇨ Term ist in β-Normalform wenn kein β-Redex als Teilterm enthält und keine δ-Reduktion mehr möglich ist.
- ⇨ Ein Term A evaluiert zum Term B, wenn B in β-Normalform ist und eine endliche Sequenz von Reduktionen von A nach B existiert.
- ⇨ Anzahl der Redexe verringert sich mit einer Reduktion nicht notwendigerweise (bspw.: λf .(f f f) (λx.A) ⇒β (λx.A) (λx.A) (λx.A))
- ⇨ Nicht jeder Term kann zu einer β-Normalform reduziert werden
- ⇨ Reduktionsreihenfolge nicht eindeutig: => Normal order reduction = Links nach rechts (lazy evaluation ist eine Variante) => Applicative order reduction ist von innen nach aussen (Strict evaluation ist eine Variante), Die Argumente einer Funktion werden ausgewertet bevor die Funktion selbst ausgewertet wird
- ⇨ Wenn ein Term eine β-Normalform besitzt, dann wird diese immer durch normal order reduction gefunden.
- ⇨ Es gibt Terme, die eine β-Normalform besitzen, die nicht mit applicative order reduction gefunden werden kann.
- ⇨ Unabhängig von der gewählten Evaluationsstrategie, ist die (falls vorhanden) erhaltene β-Normalform bis auf α-Konversion eindeutig.

Vollständigkeit

- ⇨ Das Lambda Kalkül beinhaltet keine expliziten Konstrukte für Rekursionsdeklarationen
- ⇨ Rekursion wird über Fixpunkt erreicht
- ⇨ Fixpunktkombinator Y (entspricht im wesentlichen "fix")

$$Y :\equiv \lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))$$

- ⇨ Fixpunkteigenschaft:

$$\begin{aligned} (Y\ g) =_{Def.}\ & \lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))\ g \\ \Rightarrow_\beta\ & (\lambda x.(g\ (x\ x))\ \lambda x.(g\ (x\ x))) \\ \Rightarrow_\beta\ & g\ (\lambda x.(g\ (x\ x))\ \lambda x.(g\ (x\ x))) \\ \Rightarrow_\beta\ & g\ (\lambda f.(\lambda x.(f\ (x\ x))\ \lambda x.(f\ (x\ x)))\ g) \\ =_{Def}\ & g\ (Y\ g) \end{aligned}$$

- ⇨ Damit lässt sich Rekursion umsetzen (im untypisierten Lambda Kalkül)
- ⇨ Im Lambda Kalkül kann keine Funktion direkt benennen (also auch nicht sagen fac ruft fac (sich selbst) auf)
- ⇨ Fixpunktkombinator sorgt dafür dass man einer Funktion ihr eigenes «ich» übergeben kann = Rekursion