Zusammenfassung

Christian Höner zu Siederdissen
christian.hoener.zu.siederdissen@uni-jena.de

Theoretische Bioinformatik, Bioinformatik Uni Jena

01 Feb 2024

pure

Eine Funktion die "pure" ist wird für gleiche Eingaben immer gleiche Ausgaben produzieren. Ausserdem hat die Funktion keine Seiteneffekte

- `transparent x = x + 1` ist "pure"
- Alle "Funktionen" die nicht innerhalb eines monadischen Kontexts arbeiten sind "pure"
- wir ignorieren einfach mal `unsafePerformIO :: IO a -> a` und `accursedUnutterablePerformIO :: IO a -> a`, die haben ihren Namen aus gutem Grund!
- `opaque x = rmdir "$home">> return (x+1)` ist nicht "pure"!
- Viele monadische Funktionen verhalten sich "nach aussen" auch wie "pure" Funktionen. Warum?

lazy

auch bekannt als `call-by-need` rechnet eine Funktion so spät wie möglich aus. Dies erlaubt den Umgang mit unendlichen Datenstrukturen, solange immer nur ein endlicher Teil abgefragt wird.

Lazy Listen-Konstruktion.

```haskell
hd = take 10 [1..]
fib = 0:1:zipWith (+) fib (drop 1 fib)

ohoh :: IO Int
ohoh = do
  xs :: String <- readFile "big.file"
  return $ length xs
```

Lazy IO: Kombination von Datei lesen und Länge berechnen.

Datentypen

- Sammlung verwandter Werte
- Unterschied Typ- und Datenkonstructor
- Argumente: fix und variable (a2 vs Int)
- Viele (auch rekursive) Datentypen sind vorgegeben
- Jeder Ausdruck (Expression) hat einen Typ
- zur Kompilierzeit bekannt (Explizit oder Typinferenz)

```haskell
data Typkonstruktor a1 a2
  = DatenKonstruktor a1 | DK a2 | DaKo a1 a2 Int

data Maybe a = Nothing | Just a

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

Pattern Matching

- dekonstruiert Datentypen
- case oder Funktion
- _ zeigt an das das Argument uninteressant ist

```
isJust3 :: Maybe Int -> Bool
isJust3 x = case x of
  Just 3  -> True
  Nothing -> False
  Just _  -> False

isJust3 :: Maybe Int -> Bool
isJust3 (Just 3) = True
isJust3 _        = False
```

Rekursive Datentypen

```haskell
-- Typisch ist:
-- - "Terminierender Konstruktor"
-- - "Rekursiver Konstruktor"
data List a = Nil | Node a (List a)

data Tree a = Tip | Node (Tree a) a (Tree a)

-- Pattern Matching wie gehabt
sum :: List Int -> Int
sum = go 0
  where go acc Nil = acc
        go acc (Node a ls) = go (acc+a) ls
```

`acc` sollte im Zweifel `!acc` sein, damit immer auf WHNF reduziert wird.
Alternativ `= seq acc (go ...)`

Typklassen

- erlaubt “Overloading“ / “adhoc-polymorphism”
- Funktionen innerhalb einer Typklasse lassen für Datentypen überladen (Beispiel + in `Num`)
- erlaubt es generischen Code mit Constraints zu schreiben:
  `f a b = f+b :: Num a => a -> a -> a`
  `sort :: Ord a => [a] -> [a]`
- Viele “strukturelle” Typklassen lassen sich automatisch herleiten (`Eq`, `Ord`) oder ableiten (`Num`)

```
class Num a where
  (+) :: a -> a -> a

instance Num Int where
  a + b = hardwarePlus a b
instance Num Complex where
  C (ar,ai) + C (br,bi) = C (ar+br, ai+bi)
```

Funktionskombinationen

- Funktionen sind “first-class”: koennen als Argumente anderer Funktionen dienen, in Datenstrukturen verpackt werden
- Konstruktion komplexer Algorithmen aus einfachen Bausteinen

```
(.) :: (b->c) -> (a->b) -> a->c
(.) f g = \x -> f (g x)
```

Listengeneratoren

- Erlauben es komplexe Kombinationen von Listen zu schreiben
- Gut als "kartesisches Produkt" zu verstehen
- Filtermöglichkeiten innerhalb des Generators
- lazy

```haskell
veryodd :: [(Int,Int)]
veryodd = [ (a,b) | a <- [1..], odd a
                  , b <- [1..a], even b
          ]
```

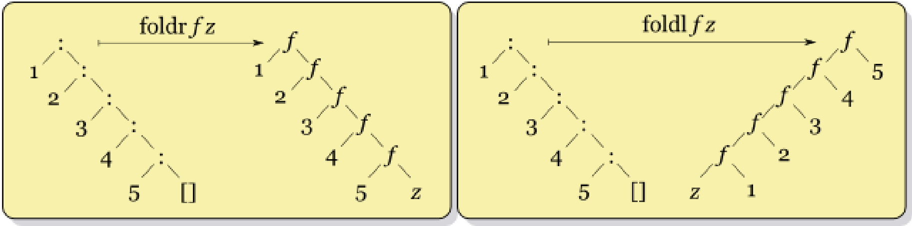
Strukturelle Rekursion: catamorphism

- `foldl`, `foldr` als grundlegende Operationen um Listen zu *reduzieren* (map-reduce in Neusprech)
- `foldr`: äquivalent dazu eine Binäroperation $\circ$ zwischen Elemente zu schreiben: $x_1 \circ x_2 \circ x_3 \circ \cdots \circ x_n$
  grundlegend: *lazy*, falls das Ergebnis "teilweise" schon genutzt werden kann
- `foldl`: äquivalent dazu in einem Wert zu akkumulieren. Kann effizienteren Code erzeugen, wenn man an der Reduzierung auf ein Ergebnis interessiert ist – nur die strikte Version nutzen

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

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

foldr vs foldl



https://en.wikipedia.org/wiki/Fold_(higher-order_function)

Entfaltung: anamorphism

- die inverse Operation zum `fold` ist das `unfold`
- entspricht Generatoren in, zB., Python

```
unfoldr :: (b -> Maybe (a,b)) -> b -> [a]
unfoldr f b = case f b of
  Nothing -> []
  Just (a,z) -> a : unfoldr f z
```

Design Pattern: Functor

- `Functor` repräsentiert Typen über denen Abbildungen existieren
- in der Praxis bedeutet es, ein Functor ist eine Funktion von $f\ a \to f\ b$, wobei nicht der Typ f verändert wird, sondern ihr Parameter und Wert a (nach b)

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

instance Functor [a] where
  fmap go [] = []
  fmap go (x:xs) = go x : fmap go xs

-- warum so? IO ist magic!
instance Functor (IO a) where
  fmap go x = x >>= (return . go)
```

Design Pattern: Monad

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

- Kombination von Funktion und “Berechnung”
- `a -> b` stellt die Funktion und
- `m` ... die Berechnung

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

instance Monad [a] where
  return x = [x]
  xs >>= f = [y | x <- xs, y <- f x]

instance Monad IO where
  return x = IO (\s -> (s, x))
  IO m >>= k = IO (\s -> case m s
    of IO (t, a) -> (k a) t)
```

Reader

- Erlaubt es ein Environment *r* zu transportieren
- *r* ist nur lesbar, nicht schreibbar (zB Konfiguration)

```haskell
data ReaderT r m a = ReaderT {runReaderT :: r -> m a}

instance Functor m => Functor (ReaderT r m) where
  fmap f = ReaderT . fmap f . runReaderT

instance Monad m => Monad (ReaderT r m) where
  return = lift . return
  m >>= k = ReaderT $ \r -> do
    a <- runReaderT m r
    runReaderT (k a) r
```

State

- klar definierte "mutable" Variablen
- kein IO!
- zu beachten: das Token *s* forciert Ordnung der Operationen *m* und *k* (warum?)

```haskell
data StateT s m a = StateT {runStateT :: s -> m (a,s)}

instance Functor (StateT s m) where
  fmap f m = StateT $ \s ->
    fmap (\(a,t) -> (f a,t)) $ runStateT m s

instance Monad (StateT s m) where
  return a = StateT $ \s -> return (a,s)
  (>>=) :: m a -> (a -> m b) -> m b
  m >>= k = StateT $ \s -> do
    (a,t) <- runStateT m s
    runStateT (k a) t
```

(Vereinfachtes Tokenizing, Parsing)

- Was sind Ausdrucksbäume
- Parsing von "12+4711" in einen Solchen

```
data Token = Zahl Int | Op Char
token :: String -> [Token]
token [] = []
token (x:xs)
  | isDigit x = let (ls,rs) = span isDigit xs
                in Zahl (read (x:ls)) : token rs
  | x == '+'  = Op '+' : token xs
  | otherwise = error $ show (x,xs)
```

Monadisches Parsing

- Parser konsumieren Eingaben token für token
- Allerdings muß man manchmal "backtracken", wenn der Parse falsch läuft
- Ausserdem werden Parses kombiniert
- Monadisches Parsing vereinfacht die Beschreibung von Parsern

```haskell
data Parser t a
  = Parser {parse :: String -> [(a,String)]}

instance Monad (Parser t) where
  return x = Parser (\cs -> [(x,cs)])
  Parser p >>= pq = Parser $ \cs ->
    [ (b,es) | (a,ds) <- p cs
    , let Parser q = pq a
    , (b,es) <- q ds ]
```

Monadisches Parsing 2

- do-Notation vereinfacht die Beschreibung
- Einzelne Parser-Funktionen werden auch einfacher

```
itemP = Parser go
  where go [] = []
        go (x:xs) = [(x,xs)]

satP c = do
  x <- itemP
  if c x then return x else (Parser $ \cs -> [])

myParser = do
  a <- itemP
  b <- satP 'a'
  c <- itemP
  itemP
  e <- itemP
  return (a,b,e,c) -- Reihenfolge!
```

Memoisierung & least fixpoint operator

- Memo-Datenstrukturen: List, Array, Key-Value Bäume
- `fix :: (f -> f) -> f`
  - was erlaubt uns `fix`?
  - Implementation?
  - Memo-Systeme

```
fix :: (f -> f) -> f
fix f = let x = f x in x

fib n | n < 2 = 1
fib n = fib (n-1) + fib (n-2)

memoList :: [Int] -> (Int -> a) -> (Int -> a)
memoList ks f = (map f ks !!)

memofib :: Int -> Int
memofib = fix (memoList [0..1000] . go)
  where go f n = if n < 2 then 1 else f (n-1) + f (n-2)
```

Quantifizierung

- universell quantifizierte Funktionen: der Aufrufer entscheidet über den Typ der Variablen
- existentiell quantifizierte Funktionen: die aufgerufene Funktion beschränkt was mit den Variablen gemacht werden kann
- nützlich um interne Variablen (cf. stream fusion) vor dem Benutzer zu “verstecken”
- erlaubt Container heterogener Elemente, solange man sich nicht mehr für den Ursprungstyp interessiert

```
data Alles = forall a . Alles a

-- heterogen!
alle :: [Alles]
alle = [Alles 'a', Alles 1, Alles (+3)]
```

Call-Pattern Specialization (callspec)

- *callspec* dient uns als Beispiel für Programmtransformationen die der Compiler ausführen darf
- diese Transformationen erhalten die Semantik des Programms
- *callspec* ist vergleichsweise einfach und mechanisch, aber potentiell ausschlaggebend für effiziente Programmgeneration
- Beispiel für Nutzen: `sum . map (+1) . map (*2)`

```
case
  case x of
    A -> X
    B -> Y
of
  X -> 0     -- A->X->0
  Y -> 1     -- B->Y->1
```

```
case x of
  A -> 0
  B -> 1
```

Lambda-Kalkül

- β-Reduktion reduziert Ausdrücke
- α-Konvertierung dient der (Semantik-erhaltenden) Umbenennung
- η-Reduktion entfernt Lambda-Abstraktionen
- δ-Regeln erlauben es "eingebaute" Funktionen zu nutzen
- Die Wahl der Reduktion ist wichtig (Terminierung, Laziness)

β-Reduktion

Formal schreibt man $(\lambda x.E)z \xrightarrow{\beta} E\left[\frac{z}{x}\right]$

- Im Ausdruck $(\lambda x.E)$ wird im Ausdruck E ersetzt
- Und zwar alle *freien* Variablen x durch z
- β-Reduktion gibt dann eine Kopie des (entsprechend ausgewerteten) Ausdrucks zurück

Beispiele:

- $(\lambda x.(+\,x\,1))\,2 \xrightarrow{\beta} (+\,2\,1) \xrightarrow{\delta} 3$

β-Reduktion

Formal schreibt man $(\lambda x.E)z \xrightarrow{\beta} E\left[\frac{z}{x}\right]$

- Im Ausdruck $(\lambda x.E)$ wird im Ausdruck E ersetzt
- Und zwar alle *freien* Variablen x durch z
- β-Reduktion gibt dann eine Kopie des (entsprechend ausgewerteten) Ausdrucks zurück

Beispiele:

- $(\lambda x.(+\,x\,1))\,2 \xrightarrow{\beta} (+\,2\,1) \xrightarrow{\delta} 3$
- $(\lambda x.(+\,x\,x))\,1 \xrightarrow{\beta} (+\,1\,1) \xrightarrow{\delta} 2$

β-Reduktion

Formal schreibt man $(\lambda x.E)z \xrightarrow{\beta} E\left[\frac{z}{x}\right]$

- Im Ausdruck $(\lambda x.E)$ wird im Ausdruck E ersetzt
- Und zwar alle *freien* Variablen x durch z
- β-Reduktion gibt dann eine Kopie des (entsprechend ausgewerteten) Ausdrucks zurück

Beispiele:

- $(\lambda x.(+\,x\,1))\,2 \xrightarrow{\beta} (+\,2\,1) \xrightarrow{\delta} 3$
- $(\lambda x.(+\,x\,x))\,1 \xrightarrow{\beta} (+\,1\,1) \xrightarrow{\delta} 2$
- $(\lambda x.(\lambda y.(+\,x\,y)))\,1\,2 \xrightarrow{\beta} (\lambda y.(+\,1\,y))\,2 \xrightarrow{\beta} (+\,1\,2) \xrightarrow{\delta} 3$

β-Reduktion

Formal schreibt man $(\lambda x.E)z \xrightarrow{\beta} E\left[\frac{z}{x}\right]$

- Im Ausdruck $(\lambda x.E)$ wird im Ausdruck E ersetzt
- Und zwar alle *freien* Variablen x durch z
- β-Reduktion gibt dann eine Kopie des (entsprechend ausgewerteten) Ausdrucks zurück

Beispiele:

- $(\lambda x.(+\,x\,1))\,2 \xrightarrow{\beta} (+\,2\,1) \xrightarrow{\delta} 3$
- $(\lambda x.(+\,x\,x))\,1 \xrightarrow{\beta} (+\,1\,1) \xrightarrow{\delta} 2$
- $(\lambda x.(\lambda y.(+\,x\,y)))\,1\,2 \xrightarrow{\beta} (\lambda y.(+\,1\,y))\,2 \xrightarrow{\beta} (+\,1\,2) \xrightarrow{\delta} 3$
- $(\lambda f.f\,1)\,(\lambda x.(+\,x\,2)) \xrightarrow{\beta} (\lambda x.(+\,x\,2))1 \xrightarrow{\beta} (+\,1\,2) \xrightarrow{\delta} 3$

Alpha(α)-Konvertierung

Was ist das Problem? Betrachte folgende β-Reduktion:

- $(\lambda f.\lambda x.f(f\,x))x$
- $\xrightarrow{\beta} \lambda x.x(x\,x)$

Durch die gleichen Namen ist jetzt eine falsche Funktion entstanden! Wir müssen den Parameter x umbenennen:

$$\lambda x.f(f\ x) \xrightarrow{\alpha} \lambda y.f(f\ y)$$

formal:

$$(\lambda x.E) \xrightarrow{\alpha} \lambda y.E\left[\frac{y}{x}\right]$$

- $(\lambda f.\lambda x.f(f\,x))x$
- $\xrightarrow{\alpha} (\lambda f.\lambda y.f(f\,y))x$
- $\xrightarrow{\beta} \lambda y.x(x\,y)$

Rekursions-Schemata

- Ersetze rekursive Funktionen auf rekursiven Datentypen
- durch nicht-rekursive Funktionen auf verwandten nicht-rekursiven Datentypen
- Diese Strukturen nennen wir Basis-Funktoren (base functor)

```
type family Base t :: * -> *

data ListF a b = NilF | ConsF a b

instance Functor (ListF a) where
  fmap f NilF = NilF
  fmap f (ConsF a b) = Cons a (f b)

type instance Base [a] = ListF a
```

Base ist eine Funktion von Typ- nicht Datenargumenten!

Recursive

```haskell
class Functor (Base t) => Recursive t where
  project :: t -> Base t t

instance Recursive [a] where
  project [] = NilF
  project (x:xs) = ConsF x xs
```

Hinweis: durch (momentane) Magie kann `Recursive` automatisch generiert werden.

fold / catamorphism

Layer um Layer wird die Struktur zu einem Wert zusammengefaltet

```
cata :: Recursive t => (Base t a -> a) -> t -> a
cata f = c
  where c = f . fmap c . project

t = cata (\x -> case x of { NilF -> 0; ConsF a b -> a+b})
         [1..4]
```

Paramorphismus

Layer um Layer wird die Struktur zu einem Wert zusammengefaltet; in jedem Layer steht der noch zu verarbeitende Teil der Struktur zur Verfügung

```
para :: Recursive t => (Base t (t,a) -> a) -> t -> a
para t = p
  where p x = t . fmap ((,) <*> p) $ project x

  testpara2 = para (\x -> case x of
    NilF -> [[]]
    ConsF a (rs,as) -> [rs,[a]] : as
  ) [1..3]

[ [[2,3] , [1]]
, [[3]   , [2]]
, [[]    , [3]]
, [  ]
]
```

Konstruktion von Algorithmen

- Einfache Algorithmen; Beispiel “kleinste freie Zahl”
- Basierend auf der jeweiligen Problembeschreibung
- Am Beispiel: Gegeben Liste $L \subset \mathbb{N}$
  Finde: Kleinste Zahl x mit: $x \notin L$

```
-- L als Liste
kleinste :: [Int] -> Int
kleinste ls = head ([0,1,2..] \\ ls)

xs \\ ys = filter (notElem ys) xs

notElem [] r = True
notElem (l:ls) r = l/=r && notElem ls r
```

Der unechte QuickSort, aber schön

- Diese Variante ist *nicht* in-place
- Dafür extrem einfach zu schreiben

```haskell
funqs :: Ord a => [a] -> [a]
funqs [] = []
funqs (pivot:rest) =
  -- kopiert jeweils *alle* Elemente, linearer Extra
  -- Speicheraufwand
  let smaller = funqs [a | a <- rest, a<=pivot]
      larger  = funqs [a | a <- rest, a> pivot]
  in smaller ++ [pivot] ++ larger
```

Paralleles Programmieren

- Parallelismus für *pure* Algorithmen
- `par` parallelisiert, `pseq` ordnet
- weitere Kombinatoren werden auf Basis dieser grundlegenden Kombinatoren gebaut: `parTraversable`, `parMap`, `parBuffer` implementieren paralleles Ausrechnen mit “vielen” Threads von kompletten Datenstrukturen

```haskell
par :: a -> b -> b
par a b -- 'a' rechnet im Hintergrund

pseq :: a -> b -> b
pseq a b -- stelle sicher das 'a' fertig
  -- bevor 'b' angefangen wird

l `par` r `pseq` l+r
  -- als "grundlegende Idee"
```