Kombinatorik

Das Ende der Katzen!

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

Theoretische Bioinformatik, Bioinformatik Uni Jena

Dec 08^{th}, 2022

Parsing mit coolen Typen[1]

Parser bisher:

```
pSumPNP :: [Token] -> Maybe (Expr,[Token])
```

generalisieren:

```
type Parser = [Token] -> Maybe (Expr,[Token])
```

$0 \ldots n$ Parses:

```
type Parser = [Token] -> [(Expr,[Token])]
```

Expr generalisieren:

```
type Parser a = [Token] -> [(a,[Token])]
```

Token generalisieren:

```
type Parser t a = [t] -> [(a,[t])]
```

[1] und schlechten Wortwitzen

Parsing mit coolen Typen[2]

Namen erfinden:

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

und vergleichen:

```haskell
type Parser t a = [t] -> [(a,[t])]

pSumPNP :: Parser Token Expr
pSumPNP :: Parser Char Expr
  == [Char] -> [(Expr,[Char])]
  == String -> [(Expr,String)]
```

Ein Parser ist eine Funktion die eine Eingabeliste $[t]$ von Token nimmt und eine Liste $[(a,[t])]$ von Parses a zusammen mit der restlichen Eingabe $[t]$ liefert.

[2] und schlechten Wortwitzen

Parsen eines Tokens

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

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



atomP :: Eq t => t -> Parser t t
atomP c = Parser go
  where go [] = []
        go (x:xs) | x/=c = []
        go (x:xs) = [(x,xs)]
```

```haskell
-- | This is Maybe
data Option a = Nul | Has a
  deriving (Show)
```

Functoren ?!

```haskell
instance Functor (Parser t) where
  fmap :: (a -> b) -> Parser t a -> Parser t b
  fmap f (Parser p) = Parser (\cs ->
    [(f a,ds) | (a,ds) <- p cs])
```

```haskell
-- Funktion f auf Elemente in Has anwenden
instance Functor Option where
  fmap :: (a->b) -> Option a -> Option b
  fmap f Nul     = Nul
  fmap f (Has a) = Has (f a)
```

Applicatives ???!

```
instance Applicative (Parser t) where
  pure :: a -> Parser t a
  pure x = Parser (\cs -> [(x,cs)])
  (<*>) :: Parser t (a -> b) -> Parser t a -> Parser t b
  Parser p <*> Parser q = Parser (\cs ->
    [ (f a,es) | (f,ds) <- p cs
    , (a,es) <- q ds])
```

```
-- Sowohl Funktion als auch Argument sind "eingepackt"
instance Applicative Option where
  pure :: a -> Option a
  pure = Has
  (<*>) :: Option (a->b) -> Option a -> Option b
  Nul <*> _ = Nul
  _ <*> Nul = Nul
  Has f <*> Has a = Has (f a)
```

Alternatives ?!

```haskell
instance Applicative (Parser t) => Alternative (Parser t)
where
  empty = noP
  Parser p <|> Parser q = Parser $ \cs -> p cs ++ q cs


instance (Monad (Parser t), Alternative (Parser t))
=> MonadPlus (Parser t) where
  mzero = empty
  mplus = (<|>)
```

```haskell
-- | Entweder lhs (bevorzugt) oder rhs nutzen
instance Applicative Option => Alternative Option where
  empty :: Option a
  empty = Nul
  (<|>) :: Option a -> Option a -> Option a
  Nul <|> option = option
  Has a <|> _     = Has a
```

Monads: Are you joking?

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

```
-- | a innerhalb eines "Option" Kontext bearbeiten
instance Monad Option where
  (>>=) :: Option a -> (a->Option b) -> Option b
  Nul >>= f = Nul
  Has a >>= f = f a
```

```
noP :: Parser t a
noP = Parser $ \cs -> []

satP :: (t -> Bool) -> Parser t t

satP c = Parser go
  where go [] = []
        go (x:xs) | c x = [(x,xs)]
        go _ = []

satP c = itemP >>= \x -> if c x then pure x else mzero

satP c = Parser goL >>= \x ->
  if c x then Parser (\cs -> [(x,cs)])
         else Parser (\cs -> [])
  where goL [] = []
        goL (x:xs) = [(x,xs)]
```

fuz rho *doh*

```haskell
satP c = do
  x <- itemP
  if c x then pure x else mzero

testPP =
  itemP >>= \x1 ->
  itemP >>= \x2 ->
  itemP >>
  itemP >>= \x4 ->
  return (x1,x2,x4)

testDO = do
  x1 <- itemP
  x2 <- itemP
  itemP
  x4 <- itemP
  return (x1,x2,x4)
```

Combinator-Time

```haskell
theseP :: Eq t => [t] -> Parser t [t]
theseP [] = pure []
theseP (t:ts) = satP (t==) >> theseP ts

manyP p = someP  p <|> return []

someP p = do {x <- p; xs <- manyP p; return (x:xs)}

-- btw. "many" und "some" gibt es fuer *alle* Alternative's
```

In Haskell liegt die Kunst nicht darin moeglichst viele verschiedene Kombinatoren zu haben, sondern wenige, *generische* Kombinatoren die breite Anwendung finden.
Deshalb machen auch "Monaden" Sinn: sie beschreiben generische strukturelle Features

Listen, und Klammern

```
sepBy :: Parser t a -> Parser t b -> Parser t [a]
p `sepBy` s = (p `sepBy1` s) <|> return []

-- HEY! Das sind ja programmierbare Semikolons!

sepBy1 :: Parser t a -> Parser t b -> Parser t [a]
p `sepBy1` s = do {a <- p; as <- many (s >> p)
                  ;return (a:as)}

bracketedP :: Parser t l -> Parser t x -> Parser t r
  -> Parser t x
bracketedP lP xP rP = do
  _l <- lP
  x  <- xP
  _r <- rP
  return x
```

Operatoren und Operanden

```haskell
chainl :: Parser t a -> Parser t (a -> a -> a) -> a
  -> Parser t a
chainl p op a = (p `chainl1` op) <|> return a

chainl1 :: Parser t a -> Parser t (a -> a -> a)
  -> Parser t a
chainl1 p op = p >>= go
  where go a = do
          f <- op
          b <- p
          go (f a b)
          <|> return a
```

Noch schnell ein lexikalischer Parser

```haskell
spaceP :: Parser Char String
spaceP = many (satP isSpace)

tokenP :: Parser Char a -> Parser Char a
tokenP p = p <* spaceP

stringP :: String -> Parser Char String
stringP = tokenP . theseP
```

Und ein neuer Expr Parser

```haskell
digitP :: Parser Char Int
digitP = satP isDigit >>= \x -> pure (ord x - ord '0')

numberP :: Parser Char Expr
numberP = do
  ds <- some digitP
  spaceP
  return . Num $ foldl (\acc x -> 10*acc + x) 0 ds

bracketP :: Parser Char Expr
bracketP = bracketedP l exprP r
  where l = tokenP $ atomP '('
        r = tokenP $ atomP ')'
```

Dieser Parser braucht jetzt auch kein Tokenizing mehr! Und vesteht Leerzeichen!

Das ist ja einfach . . .

```
addopP, mulopP
  :: Parser Char (Expr -> Expr -> Expr)

addopP = (stringP "+" >> pure (App Add))
     <|> (stringP "-" >> pure (App Sub))

mulopP = (stringP "*" >> pure (App Mul))
     <|> (stringP "/" >> pure (App Div))
```

Der komplette Expr Parser

```
-- Expr's sind Terme mit addop's verbunden

exprP :: Parser Char Expr
exprP = termP `chainl1` addopP

-- Terme sind factors mit Multiplikationen verbunden

termP = factorP `chainl1` mulopP

-- factors sind Zahlen oder wohlgeformte Klammern

factorP = numberP <|> bracketP
```

Zusammenfassung

- Wir haben `Functor`, `Alternative`, `Applicative`, `Monad` als Abstraktionsmittel kennengelernt
- Jede dieser Abstraktionen erlaubt es eine Zahl vorgefertigter Kombinatoren zu nutzen
- Unser neuer Parser ist ein Beispiel fuer Monaden in Aktion
- Und auch fuer `do`-Notation, die aber nur syntaktischer Zucker ist
- Unser neuer Parser kann prinzipiell alle legalen Parses, nicht nur einen, erzeugen

Es folgt dann die Frage ob sich der "Monad" Aufwand lohnt? (Ja)
Und die Konstruktion eines effizienteren Countdown!

Memoisation

Wikipedia: Memoisation oder Memoisierung ist eine Technik, um Computerprogramme zu beschleunigen, indem Rückgabewerte von Funktionen zwischengespeichert anstatt neu berechnet werden.

Warum ist das von Interesse? Was macht `mkExpr`? Wie haeufig sehen wir Teilsequenzen in `mkExpr`?

`1,2,3,4,5,6`

fix und Open Recursion

Betrachten wir noch einmal *Fibonacci*

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
        ^^^         ^^^
```

- `fib` nutzt (offensichtlich Rekursion)
- wir wollen jetzt die Rekursion "heraus ziehen" in eine eigene Funktion

```
fob :: (Int -> Int) -> Int -> Int
fob f 0 = 1
fob f 1 = 1
fob f n = f (n-1) + f (n-2)
         ^^^       ^^^

runfob :: Int -> Int
runfob = fib runfob
```

fix und Open Recursion

```
fob :: (Int -> Int) -> Int -> Int
fob f 0 = 1
fob f 1 = 1
fob f n = f (n-1) + f (n-2)
         ^^^       ^^^

runfob :: Int -> Int
runfob = fib runfob

-- fib (fib (fib (fib (...))))
```

Generalisieren von `runfob`

```
-- 'f' ist der Typ von Funktionen, zB
-- fix :: ((Int -> Int) -> (Int -> Int)) -> (Int -> Int)
fix :: (f -> f) -> f
fix f = let x = f x in x
```

fix und Open Recursion

```haskell
fob :: (Int -> Int) -> Int -> Int
fob f 0 = 1
fob f 1 = 1
fob f n = f (n-1) + f (n-2)
         ^^^       ^^^

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

runfob :: Int -> Int
runfob = fix fob
```

`fix` erlaubt es uns in jedem Rekursionsschritt interessante Dinge mit `fob` zu machen, wobei die "interessanten Dinge" auf jeder Ebene passieren!

fix und Open Recursion

```haskell
fob :: (Int -> Int) -> Int -> Int
fob f 0 = 1
fob f 1 = 1
fob f n = f (n-1) + f (n-2)
         ^^^       ^^^

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

-- fix fib 10
-- fix (\f -> fib ((1+) . f)) 10
```

Memotables

```
fob :: (Int -> Int) -> Int -> Int
fob f 0 = 1
fob f 1 = 1
fob f n = f (n-1) + f (n-2)

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

-- gib eine Liste von Indices fuer die Funktion f
-- und ihre Werte gespeichert werden
memoList :: [Int] -> (Int -> a) -> (Int -> a)
memoList ks f = (map f ks !!)

memofib :: Int -> Int
memofib = fix (memoList [0..1000] . fib)
-- === fix (fib . memoList [0..1000])
```

memo Expr

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

-- mkExprs, aber die Rekursion ist jetzt via "mk"
orExprs :: ([Int] -> [(Expr,Value)]) -> [Int] -> [(Expr, Val
orExprs mk [x] = [(Num x, Value x)]
orExprs mk xs =
  [ ev | (ys,zs) <- unmerges xs
  , l <- mk ys, r <- mk zs
  , ev <- combine l r ]

-- wie bei fib, wir machen die Rekursion explizit
recExprs :: [Int] -> [(Expr,Value)]
recExprs = orExprs recExprs

fixExprs :: [Int] -> [(Expr,Value)]
fixExprs = fix orExprs
```

memo Expr

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

-- mkExprs, aber die Rekursion ist jetzt via "mk"
orExprs :: ([Int] -> [(Expr,Value)]) -> [Int] -> [(Expr, Va
orExprs mk [x] = [(Num x, Value x)]
orExprs mk xs =
  [ ev | (ys,zs) <- unmerges xs
  , l <- mk ys, r <- mk zs
  , ev <- combine l r ]

-- Speichern fuer alle sequences, die man angibt
memoSeqs :: forall a . [[Int]] -> ([Int] -> a) -> ([Int] ->
memoSeqs sqs f = (tbl Map.!)
  where
    tbl :: Map.Map [Int] a
    tbl = Map.fromList [ (s,f s) | s <- sqs ]
```

Laufzeit?

```
length . concatMap mkExprs $ subseqs [1..6]
-- 5341067
-- (6.55 secs, 6,155,652,240 bytes)

length . concatMap recExprs $ subseqs [1..6]
-- 5341067
-- (6.68 secs, 6,182,532,000 bytes)

length . concatMap (fix orExprs) $ subseqs [1..6]
-- 5341067
-- (6.60 secs, 6,182,532,112 bytes)

length . concatMap (fix (orExprs . memoSeqs (subseqs [1..6]
-- 5341067
-- (4.45 secs, 4,110,812,488 bytes)
```

Zusammenfassung

- Rekursion kann durch `fix` den "least fixpoint operator" expliziter gemacht werden
- Dadurch koennen wir in jeder Ebene einer Rekursion Funktionalitaet "injizieren"
- Hier "ziehen" wir eine Datenstruktur durch, mit der wir fuer jede Eingabe die Ausgabe speichern (*memoisieren*)
- `fib` wird dadurch polynomiell
- `orExprs` und `memoSeqs` bringen weniger, da die Datenstrukturen viel groesser sind

Paralleles Programmieren

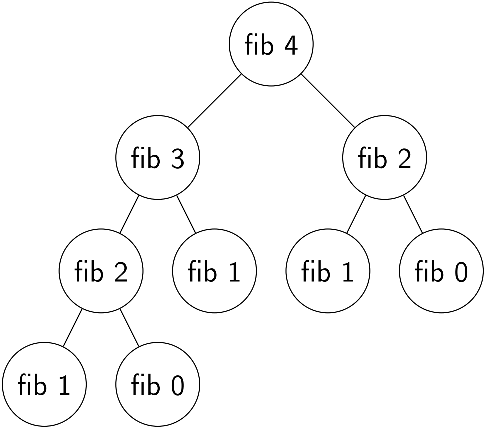
(removed graphics)

- Parallele Algorithmen nutzen mehrere Kerne gleichzeitig die "Wallclock" Zeit zu verringern
- In “puren” Sprachen (wie Haskell) ist Parallelismus einfacher, da Funktionen keinen *imliziten State* haben
- trotzdem sind parallel Algorithmen schwer zu entwickeln

Wir werden zwei Beispiele betrachten: `fib` und `countdown`

Betrachten wir `fib`

```haskell
seqFib :: Integer -> Integer
seqFib 0 = 1
seqFib 1 = 1
seqFib n = seqFib (n-1) + seqFib (n-2)
```



Paralleles fib

```haskell
seqFib :: Integer -> Integer
seqFib 0 = 1
seqFib 1 = 1
seqFib n = seqFib (n-1) + seqFib (n-2)



parFib :: Integer -> Integer
parFib n | n<=2 = seqFib n
parFib n = sum ([l,r] `using` parTraversable rdeepseq)
  where l = parFib (n-1)
        r = parFib (n-2)
```

... nutzlos?

```
./fib +RTS -s -N -RTS 0 37
39088169
This took 1.364737266s seconds
39088169
This took 1.002810075s seconds
```

```
  17,550,281,160 bytes allocated in the heap
      53,070,264 bytes copied during GC
       1,800,720 bytes maximum residency (23 sample(s))
         356,544 bytes maximum slop
              30 MiB total memory in use (0 MB lost due to

Parallel GC work balance: 53.19% (serial 0%, perfect 100%)

TASKS: 50 (1 bound, 49 peak workers (49 total), using -N24)

SPARKS: 81662776 (57077 converted, 0 overflowed, 0 dud,
  67384297 GC'd, 14221402 fizzled)
```

`parallel` in Haskell

- Haskell unterstützt parallele Kombinatoren
- diese müssen aber geschickt genutzt werden, sonst erstellt man Millionen (!) "threads" ohne Nutzen
- Frage: was kennzeichnet geschickte Nutzung?
- nicht "zu viel" Parallelismus, aber auch nicht "zu wenig" (da threads guenstig sind)
- zusaetzlich: genuegend Berechnungen, damit sich threads lohnen

Besseres Paralleles fib

```haskell
seqFib :: Integer -> Integer
seqFib 0 = 1
seqFib 1 = 1
seqFib n = seqFib (n-1) + seqFib (n-2)



parFib :: Integer -> Integer -> Integer
parFib c n | n<=2 || n<=c = seqFib n
parFib c n = l `par` r `pseq` l+r
--parFib c n = sum ([l,r] `using` parTraversable rdeepseq)
  where l = parFib c (n-1)
        r = parFib c (n-2)
```

```
./fib +RTS -s -N -RTS 33 40

165580141
This took 5.296301388s seconds

165580141
This took 0.454093625s seconds

  SPARKS: 33 ( 31 converted
             ,  0 overflowed
             ,  0 dud
             ,  0 GC'd
             ,  2 fizzled)
```

Grundlegende `parallel`-Kombinatoren

Berechne *a* parallel zu *b* und gebe *b* zurueck

```
par :: a -> b -> b
par a b = <magic> b
```

Berechne erst *a*, dann *b*, bevor *b* zurueck gegeben wird. Haskell hat sonst freie Wahl!

```
pseq :: a -> b -> b
pseq a b = <magic> b
```

Generiere threads fuer *l* und *r*, lasse diese ausrechnen und rechne dann $l + r$

```
l `par` r `pseq` l+r
```

Parallel-Kombinatoren

Es gibt eine Anzahl an Kombinatoren, die komplexe parallele Muster erlauben: `Control.Parallel.Strategies`

- `rseq` evaluiere zu WHNF (what?)
- `rpar` parallel spark (what?)
- `using` *a s*, berechne *a* mittels Strategie *s*
- `parTraversable` *x*, berechne die “Liste“ *x* mittels Strategie
- `parBuffer` *k x*, berechne rollend immer *k* Elemente in parallel in *x*
- `parMap` *f x* berechne *f* parallel ueber alle *x*