InPlace

Programmieren wie bei Gandalf zu Hause

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

Theoretische Bioinformatik, Bioinformatik Uni Jena

Feb 02nd, 2023

Programmieren wie bei Gandalf?

- Was ist überhaupt das Problem das wir lösen wollen?
- *In-Place* Datenstrukturen und Algorithmen sind doch so alt wie Computer?
- Und Computer ist hier alles was rechnet, auch wenn es biologisch ist

1. Haskell ist *lazy*
2. Haskell ist *pure*
3. wir müssen *referential transparency* erhalten

Diese Eigenschaften machen IO und In-Place zu einem Abenteuer.

Referential Transparency

Eine Funktion ist dann "referentiell transparent" wenn ihr Aufruf durch den Wert des Aufrufs ersetzt werden kann, ohne das sich die Programmsemantik ändert

```
1  transparent :: Int -> Int
2  transparent x = x+1
3
4  opaque :: Int -> IO Int
5  opaque x = do
6    deleteDir "$HOME"
7    return $ x+1
8
9  main = do
10   print $ transparent 1
11   print $ 2
12   opaque 1 >>= print
13   pure 2 >>= print
```

pure

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

- `transparent` 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 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.

```
1  hd = take 10 [1..]
2  fib = 0:1:zipWith (+) fib (drop 1 fib)
3
4
5
6  ohoh :: IO Int
7  ohoh = do
8    xs :: String <- readFile "big.file"
9    return $ length xs
```

Die ST Monad

ST erlaubt es "mutablen" Code zu schreiben, ohne das wir die echte Welt manipulieren können.

```
1  sq xs = runST $ sqST xs
2
3  sqST :: [Int] -> ST s [Int]
4  sqST xs = do
5    let l = length xs
6    -- neues array
7    arr <- PA.newPrimArray l
8    -- fuellen
9    forM_ (zip [0..] xs) $ uncurry (PA.writePrimArray arr)
10   -- quadrieren (inplace)
11   forM_ [0..l-1] $ \i ->
12     PA.readPrimArray arr i >>= \x ->
13       PA.writePrimArray arr i (x^2)
14   -- auslesen in Liste
15   forM [0..l-1] (PA.readPrimArray arr)
```

ST "leckt" nicht

ST ist nur innerhalb eines "Kontextes" nutzbar. Wir können innerhalb von `runST` beliebig Code verknüpfen, aber keine Struktur kann aus dem `ST s` "ausbrechen".

```
1  sicher :: [Int] -> ST s (PA.MutablePrimArray s Int)
2  sicher xs = do
3    PA.newPrimArray (length xs)
4
5  leck :: [Int] -> PA.MutablePrimArray s Int
6  leck xs = runST $ sicher xs
7
8  -- Couldn't match type 's1' with 's'
9  -- Expected: ST s1 (PA.MutablePrimArray s Int)
10 --   Actual: ST s1 (PA.MutablePrimArray s1 Int)
```

Theorie zu State Threads (ST)

State Threads forcieren, analog zur State-Monad, die Hintereinanderausführung von Code.

Tokenübergabe verhindert das `k` vor `ST m` läuft.

```
1  data ST s a = State s -> (State s, a)
2
3  instance Monad (ST s) where
4  ST m >>= k = ST $ \s ->
5    -- nur @m s@ generiert neuen Token
6    -- der weiter gegeben wird
7    case m s of (t,r) ->
8      -- k r kann nur rechnen, sobald @r@
9      -- zur Verfuegung steht
10     case k r of ST k2 ->
11       -- der neue ST aus @k r@ bekommt den
12       -- neuen Token @t@ uebergeben
13       k2 t
```

(code vereinfacht!)

Theorie zu State Threads (ST)

Zusätzlich wird garantiert das ST code nicht `runST` verlassen kann.
Wie funktioniert das?

```
1  runST :: (forall s . ST s a) -> a
2  runST (ST s) = <...> s
```

- `forall s` innerhalb von `(forall s . ST s a)`
- das nennt man dann *Rank-2 polymorph*
- im Vergleich zu "State" liefert "ST" ein Interface zu *references*, quasi "Zeigern" oder "manipulierbare Speicheraddressen"
- diese sollen allerdings lokal bleiben und nicht beliebig zwischen Programmteilen herumgereicht werden
- `runST` hat nun den Constraint das Thread `s` nur innerhalb der *Scope* von `runST` zu sehen ist
- und damit jedes `s` einzigartig ist

Grundlegende Elemente

```
1  -- Neue Referenz vom Typ a
2  newSTRef :: a -> ST s (STRef s a)
3
4  -- Auslesen des aktuellen Wertes
5  readSTRef :: STRef s a -> ST s a
6  -- Achtung: vereinfacht!
7  readSTRef (STRef v) = ST $ \s -> readMutVar v s
8
9  -- Referenz mit neuem Wert ueberschreiben
10 writeSTRef :: STRef s a -> a -> ST s ()
11
12 -- Modifizieren
13 modifySTRef :: STRef s a -> (a -> a) -> ST s ()
14 modifySTRef ref f = writeSTRef ref . f =<< readSTRef ref
```

Zwischenspiel: Magie, Maschinen, und die echte Welt

- ST ist also "nur" ein Wrapper für State Thread Tokens, die von einer State Monad getragen werden
- Innerhalb dessen werden direkte Haskell RunTime Funktionen aufgerufen
- IO selbst kann somit "einfach" konstruiert werden:

```
1  -- readMutVar ist nicht in Haskell selbst definiert
2  -- sondern wird beim Kompilieren durch direkte
3  -- Laufzeit-Umgebungs-Aufrufe ersetzt.
4  readMutVar = readMutVar
5
6  -- "IO" als "State Thread"
7  type IO a = ST RealWorld a
8  -- bzw.
9  newtype IO a = IO (State RealWorld -> (State RealWorld, a))
```

IO als Teilmenge von ST

Wir werden uns deshalb auf auf `ST` als Umgebung konzentrieren

- solange wir "nur" Algorithmen konstruieren wollen die durch Mutation effizienter sind, ist `ST` perfekt.
- dadurch sind Effekte begrenzt auf die jeweilige Ausführung in `runST`
- allerdings fallen Probleme wie Netzwerkkommunikation aus der Lösungsmenge heraus
- andererseits gibt es Notfalltüren. Diese sollte man aber nur benutzen wenn es nicht anders geht und man weiss was man tut

```
1  -- uebernimmt Funktionen in ST direkt in IO
2  stToIO :: ST RealWorld a -> IO a
3
4  -- bettet eine IO Aktion in ST ein
5  ioToST :: IO a -> ST RealWorld a
```

Einleitung

- Die folgenden Beispiele sollen hauptsächlich *nützlich* sein
- Insbesondere da es Algorithmen gibt, bei denen momentan die beste "pure" Implementation um einen log-Faktor langsamer ist, als die beste "mutierende" Implementation
- Mutierende Algorithmen sind allerdings leichter falsch zu implementieren (persönliche Einschätzung)
- Wir können trotzdem die Vorteile von Haskell, insbesondere das Typsystem nutzen

Listensumme

- Diese Funktion ist natürlich nichts weiter als `sum` oder `foldl' (+)`
- Der Akkumulator kann mutiert werden und jeder Schritt in Zeile 6 addiert das momentane Element in `xs` auf den Akkumulator.

```
 1  -- | Modifizierbarer Accumulator
 2  sumST :: Num a => [a] -> a
 3  sumST xs = runST $ do
 4    acc <- newSTRef 0
 5    forM_ xs $ modifySTRef' acc . (+)
 6    readSTRef acc
 7
 8
 9  readSTRef :: STRef s a -> ST s a
10  modifySTRef' :: STRef s a -> (a->a) -> ST s ()
11  forM_ :: [a] -> (a->m b) -> m ()
```

Nochmal Fibonacci

```
 1  fibST :: Integer -> Integer
 2  fibST n
 3    | n<2 = n
 4    | otherwise = runST $ do
 5        l <- newSTRef 0
 6        r <- newSTRef 1
 7        go n l r
 8    where
 9      go :: Integer -> STRef s Integer -> STRef s Integer
10          -> ST s Integer
11      go 0 now _ = readSTRef now
12      -- Achtung! Reihenfolge
13      go n now next = do
14        prev <- readSTRef now
15        here <- readSTRef next
16        writeSTRef now here
17        -- ($!) macht die rhs strikt, kurz thunks besprechen!
18        writeSTRef next $! prev + here
19        go (n-1) now next
```

Der unechte QuickSort, aber schön

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

```
1  funqs :: Ord a => [a] -> [a]
2  funqs [] = []
3  funqs (pivot:rest) =
4    -- kopiert jeweils *alle* Elemente, linearer Extra
5    -- Speicheraufwand
6    let smaller = funqs [a | a <- rest, a<=pivot]
7        larger  = funqs [a | a <- rest, a> pivot]
8    in smaller ++ [pivot] ++ larger
```

Der echte QuickSort: swapPA

- `MutablePrimArray s a` sind Arrays die primitive Elemente aufnehmen können
- Deshalb funktioniert diese Funktion nur mit Elementen `a` die den `Prim a` Constraint erfüllen: diese Elemente sind alles "Computerwörter" (Int, Double, etc)
- Wir merken uns schon einmal: es gibt wohl Arrays direkt auf Elementen (`Prim`) und Arrays für nicht-atomare Elemente

```
1  swapPA :: Prim a => MutablePrimArray s a
2          -> Int -> Int -> ST s ()
3  swapPA pa l r = do
4    x <- PA.readPrimArray pa l
5    y <- PA.readPrimArray pa r
6    PA.writePrimArray pa l y
7    PA.writePrimArray pa r x
```

Der echte QuickSort: loopPA

```
1  -- | Array, pivot, momentaner pivot, idx der zu vergleichen
2
3  loopPA :: (Ord a, Prim a) => MutablePrimArray s a
4         -> a -> Int -> Int -> ST s Int
5  loopPA pa pivot slot i = do
6    -- neues Element
7    val <- PA.readPrimArray pa i
8    if val < pivot
9    -- ist kleiner pivot, also i und slot vertauschen
10   -- neuer slot muss dann groesser werden
11   then swapPA pa i slot >> return (slot+1)
12   -- gleichen slot behalten
13   else return slot
```

Der echte QuickSort: partPA

- Partitioniert das Array, so das links vom slot alle Elemente kleiner als das pivot sind, rechts vom slot alle Elemente groesser als das pivot

```
1  -- | Gegeben array und bounds, ordner Elemente um das pivot
2
3  partPA :: (Show a, Ord a, Prim a) => MutablePrimArray s a
4          -> Int -> Int -> Int -> ST s Int
5  partPA pa begin end pidx = do
6    -- waehle pivot
7    pivot <- PA.readPrimArray pa pidx
8    swapPA pa pidx end
9    -- fuer alle begin..end-1
10   slot <- foldM (loopPA pa pivot) begin [begin..end-1]
11   swapPA pa slot end >> return slot
```

Quicksort

- Implementation von InPlace-Quicksort
- Diese Variante zählt auch die gemachten "goSchritte mit

```
 1  stQuickSort :: (Show a, Ord a, Prim a) => MutablePrimArray s
 2  stQuickSort pa = do
 3    let sz = PA.sizeofMutablePrimArray pa
 4    steps <- newSTRef 0
 5    go steps pa 0 (sz-1)
 6    readSTRef steps
 7    where
 8      go steps pa begin end = when (end>begin) $ do
 9        modifySTRef' steps (+1)
10        let pidx = begin + ((end-begin) `div` 2)
11        pidx <- partPA pa begin end pidx
12        go steps pa begin (pidx-1)
13        go steps pa (pidx+1) end
```

ST-Variante ordentlich verpacken

- Der Algorithmus ist nun vollständig und funktioniert auch
- Wir können hier noch einmal mit referential transparency beschäftigen

```
1  runstQuickSort :: (Show a, Ord a, Prim a)
2    => [a] -> (Int,[a])
3  runstQuickSort xs = runST $ do
4    let pa = PA.primArrayFromList xs
5    mpa <- PA.unsafeThawPrimArray pa
6    steps <- stQuickSort mpa
7    -- Zum testen mal @pa@ zurueck geben! ref.trans!
8    pa' <- PA.unsafeFreezePrimArray mpa
9    return (steps, PA.primArrayToList pa')
```

Ein Level höher

- `primitive` ist letztlich zu primitiv!
- Die Operationen auf solchen Arrays sind anders als was wir aus C oder java kennen
- Deshalb gibt es auf `primitive` aufbauend mehrere Libraries
- Heute soll `vector` von Interesse sein
- ausserdem noch Hilfsfunktionen in `vector-algorithms`
- `vector` kombiniert die Möglichkeit von InPlace Updates, einem `stream-fusion` Interface (stream fusion haben wir schon kennen gelernt!), und adaptiver Wahl der Datenstruktur (boxed, unboxed, Storable)

Data.Vector

Kombination von Listenartigen Operationen, Stream Fusion, und Arrays

- Temporäre `Vector` werden automatisch entfernt
- stream fusion verkoppelt einzelne Funktionen
- die endgültige Struktur ist extrem effizient: direkte Elemente hintereinander im Speicher

```
1  import qualified Data.Vector as M
2
3  V.foldl' (+) 0 . V.map (*2) . V.enumFromThenTo 1 3 99
4  == 5000
5
6  V.foldl' :: (a -> b -> a) -> a -> V.Vector b -> a
7  V.map :: (a -> b) -> V.Vector a -> V.Vector b
8  V.enumFromThenTo :: Enum a => a -> a -> a -> V.Vector a
```

MVector

- `MVector` ist die mutable Version eines `Vector`
- zwischen beiden Varianten kann mit `thaw` und `freeze` umgeschaltet werden
- `ST` erlaubt die Einbettung von mutierenden Algorithmen
- Achtung: unvorsichtige Nutzung der mutierenden Varianten kann *referential transparency* zerstören

Warum sollte man trotzdem viel “funktional” schreiben?

- wie lange rechnen die beiden Aufrufe unten?
- Warum?

```
1  -- nutze ST Monad und InPlace
2  print $ take 10 $ runSTquicksort [1..1000000]
3  -- nutze den "unechten" quicksort
4  print $ take 10 $ functionalQuicksort [1..1000000]
```

Weitere Datenstrukturen

- `hashtables` findet man ebenso in einer Variante für `ST`
- `repa` bezeichnet *high performance*, *regular*, *shape polymorphic*, *parallel arrays*; diese bauen auf `vector` auf und implementieren effiziente Operationen für Vektor- und Matrix-Algebra
- Konstruktionen wie (doubly) linked lists existieren. Man findet sie aber seltener da "normale" Listen bereits viele nötige Eigenschaften erfüllen (bis auf Updates)