Effizienzaspekte der funktionalen Programmierung

Heute! Hier!

Wie geht's nach Weihnachten weiter?

- Fallbeispiel: der Haskell Webserver.
- Grafik: `hOpenGL`
- GUIs: `wxHaskell`, `HTk`
- Fancy Types:
  - In Haskell: Existentielle Typen, Rang-2-Polymorphie, extendable records;
  - In anderen Sprachen: abhängige Typen.

Effizienzaspekte

- Beste Lösung: bessere Algorithmen.
- Zweitbeste Lösung: Büchereien nutzen.
- Effizenzverbesserungen durch
  - Endrekursion und Striktheit: Speicherlecks vermeiden
  - Der ewige Konflikt: Geschwindigkeit vs. Platz

Endrekursion

Eine Funktion ist endrekursiv, wenn kein rekursiver Aufruf in einem geschachtelten Ausdruck steht.

- D.h. darüber nur `if`, `case`, guards oder Fallunterscheidungen.
- Entspricht `goto` oder `while` in imperativen Sprachen.
- Wird in Schleifen übersetzt.
- Nicht-endrekursive Funktionen brauchen Platz auf dem Stack.

Beispiele

- `fac'` nicht endrekursiv:

```
fac' :: Integer-> Integer
fac' n  = if n == 0 then 1 else n * fac' (n-1)
```

- `fac` endrekursiv:

```
fac :: Integer-> Integer
fac n     = fac0 n 1 where
  fac0 n acc = if n == 0 then acc
                         else fac0 (n-1) (n*acc)
```

- `fac'` verbraucht Stackplatz, `fac` nicht. (Zeigen)

Beispiele

- Liste umdrehen, nicht endrekursiv:

```
rev' :: [a]-> [a]
rev' []       = []
rev' (x:xs) = rev' xs ++ [x]
```

  - Hängt auch noch hinten an — $O(n^2)$!

- Liste umdrehen, endrekursiv und $O(n)$: (Zeigen)

```
rev :: [a]-> [a]
rev xs = rev0 xs [] where
    rev0 []     ys = ys
    rev0 (x:xs) ys = rev0 xs (x:ys)
```

Überführung in Endrekursion

- Gegeben eine Funktion f' f': S-> T

```
f' x = if B x  then H x
               else φ (f' (K x)) (E x)
```

  - Mit K:S-> S, ϕ:T-> T-> T, E:S-> T, H:S-> T.

- Sei ϕ assoziativ ist, e:T neutrales Element

- Dann ist die endrekursive Form:

```
f: S-> T
f x = g x e where
 g x y = if B x  then φ (H x) y
                 else g (K x) (φ (E x) y)
```

Beispiel

- Länge einer Liste

```
length' :: [a]-> Int
length' xs = if (null xs) then 0
             else 1+ length' (tail xs)
```

- Zuordnung der Variablen:

$$\begin{array}{rcl} K(x) & \mapsto & \texttt{tail} \\ E(x) & \mapsto & 1 \\ \phi(x,y) & \mapsto & x+y \end{array} \qquad \begin{array}{rcl} B(x) & \mapsto & \texttt{null}x \\ H(x) & \mapsto & 0 \\ e & \mapsto & 0 \end{array}$$

- Es gilt: $\phi(x, e) = x + 0 = x$ (0 neutrales Element)

- Damit ergibt sich endrekursive Variante:

```
length :: [a]-> Int
length xs = len xs 0 where
  len xs y = if (null xs) then 0 -- was: 0+ 0
             else len (tail xs) (1+ y)
```

- Allgemeines Muster:
  - Monoid (ϕ, e): ϕ assoziativ, e neutrales Element.
  - Zustäzlicher Parameter akkumuliert Resultat.

Endrekursive Aktionen

Eine Aktion ist endrekursiv, wenn nach dem rekursiven Aufruf keine weiteren Aktionen folgen.

- Nicht endrekursiv:

```
getLines' :: IO String
getLines' = do str<- getLine
               if null str then return ""
                  else  do rest<- getLines'
                           return (str++ rest)
```

- Endrekursiv:

```
getLines :: IO String
getLines = getit "" where
   getit res = do str<- getLine
                  if null str then return res
                     else getit (res++ str)
```

Fortgeschrittene Endrekursion

- Akkumulation von Ergebniswerten durch closures
  - closure: partiell instantiierte Funktion
- Beispiel: die Klasse `Show`
  - Nur `show` wäre zu langsam ($O(n^2)$):

```
class Show a where
   show :: a-> String
```

  - Deshalb zusätzlich

```
   showsPrec :: Int-> a-> String-> String
   show x    = showsPrec 0 x ""
```

  - String wird erst aufgebaut, wenn er ausgewertet wird ($O(n)$).

- Damit zum Beispiel:

```
data Set a = Set [a] -- Mengen als Listen
instance Show a=> Show (Set a) where
  showsPrec i (Set elems) =
      \r-> r++ "{" ++ concat (intersperse ", "
                                    (map show elems)) ++ "}"
```

- Nicht perfekt— besser:

```
instance Show a=> Show (Set a) where
  showsPrec i (Set elems) = showElems elems where
    showElems []     = ("{}" ++)
    showElems (x:xs) = ('{' :) . shows x . showl xs
       where showl []     = ('}' :)
             showl (x:xs) = (',' :) . shows x . showl xs
```

Speicherlecks

- Garbage collection gibt unbenutzten Speicher wieder frei.
  - Nicht mehr benutzt: Bezeichner nicht mehr im Scope.
- Eine Funktion hat ein Speicherleck, wenn Speicher belegt wird, der nicht mehr benötigt wird.
- Beispiel: `getLines`, `fac`
  - Zwischenergebnisse werden nicht auswertet. (Zeigen.)
  - Insbesondere ärgerlich bei nicht-terminierenden Funktionen.

Striktheit

- Funktion `f` ist strikt in einem Argument `x` gdw.

$$\mathtt{x} \equiv \bot \Longrightarrow \mathtt{f(x)} \equiv \bot$$

- Beispiele:
  - `(+)` strikt in beiden Argumenten.
  - `(&&)` strikt im ersten, nicht-strikt im zweiten.

    `False && error` $\rightsquigarrow$ `False`
  - `silly` nicht-strikt im ersten. `silly error 3` $\rightsquigarrow$ `3`
  - Zeigen.
- Haskell ist
  - als nicht-strikt spezifiziert;
  - (meist) mit verzögerter Auswertung implementiert.

Striktheit

- Strikte Argumente erlauben Auswertung vor Aufruf
  - Dadurch konstanter Platz bei Endrekursion.
- Striktheit durch erzwungene Auswertung:
  - `seq :: a-> b-> b` wertet erstes Argument aus.
  - `($!) :: (a-> b)-> a-> b` strikte Funktionsanwendung

    ```
    f $! x = x `seq` f x
    ```
- Fakultät in konstantem Platzaufwand (zeigen):

```
fac2 n = fac0 n 1 where
  fac0 n acc = seq acc $ if n == 0 then acc
                         else fac0 (n-1) (n*acc)
```

`foldr` **vs.** `foldl`

- `foldr` ist nicht endrekursiv.

- `foldl` ist endrekursiv:

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

- `foldl' :: (a-> b-> a)-> a-> [b]-> a` ist endrekursiv und strikt.

- `foldl` endrekursiv, aber traversiert immer die ganze Liste.
- `foldl'` konstanter Platzaufwand, aber traversiert immer die ganze Liste.
- Wann welches `fold`?
  - Strikte Funktionen mit `foldl'` falten.
  - Wenn nicht die ganze Liste benötigt wird, `foldr`:

```
all :: (a-> Bool)-> [a]-> Bool
all p = foldr ((&&) . p) True
```

Gemeinsame Teilausdrücke

- Ausdrücke werden intern durch Termgraphen dargestellt.
- Argument wird nie mehr als einmal ausgewertet:
  - `trace :: String-> a-> a` druckt `String` bei Auswertung ((ghc) (Zeigen) `import IOExts (trace)`

```
f :: Int-> Int-> Int
f x y = x+ x
test1 = f (trace "Eins\n" (3+2))
          (trace "Zwei\n" (3+2))
```

- Sharing von Teilausdrücken
  - Explizit mit `where` und `let`
  - Implizit (ghc) (Beispiel)

Memoisation

- Fibonacci-Zahlen als Strom: linearer Aufwand.

```
fibs :: [Integer]
fibs = 1 : 1 : zipWith (+) fibs (tail fibs)
```

- Kleine Änderung der Fibonacci-Zahlen als Strom:

```
fibsFn :: () -> [Integer]
fibsFn () = 1 : 1 : zipWith (+) (fibsFn ())
                                (tail (fibsFn ()))
```

- Große Wirkung: Exponentiell. Warum?
  - Jeder Aufruf von `fibsFn()` bewirkt erneute Berechnung.

Memoisation

- ◦ Die Abhilfe: Memoisation

- Memoisation: Bereits berechnete Ergebnisse speichern.

- In Hugs: Aus dem Modul `Memo`:

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

- Damit ist alles wieder gut (oder?)

```
fibsFn' :: () -> [Integer]
fibsFn' = memo (\()-> 1 : 1 : zipWith (+)
                                (fibsFn' ())
                                (tail (fibsFn' ())))
```

- `GHC` kann das automatisch.

Überladene Funktionen sind langsam.

- Typklassen sind elegant aber langsam.
  - Implementierung von Typklassen: dictionaries von Klassenfunktionen.
  - Überladung wird zur Laufzeit aufgelöst.
- Bei kritischen Funktionen durch Angabe der Signatur Spezialisierung erzwingen.
- NB: Zahlen (numerische Literale) sind in Haskell überladen!
  - Bsp: `facs` hat den Typ `Num a=> a-> a`

```
facs n = if n == 0 then 1 else n* facs (n-1)
```

Listen als Performance-Falle

- Listen sind keine Felder.
- Listen:
  - Beliebig lang
  - Zugriff auf n-tes Element in linearer Zeit.
  - Abstrakt: frei erzeugter Datentyp aus Kopf und Rest
- Felder:
  - Feste Länge
  - Zugriff auf n-tes Element in konstanter Zeit.
  - Abstrakt: Abbildung Index auf Daten

- Modul `Array` aus der Standardbücherei

```
data Ix a=> Array a b  -- abstract
array     :: (Ix a) => (a,a) -> [(a,b)]
                                   -> Array a b
listArray :: (Ix a) => (a,a) -> [b] -> Array a b
(!)       :: (Ix a) => Array a b -> a -> b
(//)      :: (Ix a) => Array a b -> [(a,b)]
                                   -> Array a b
```

- Als Indexbereich geeignete Typen (Klasse `Ix`): `Int`, `Integer`, `Char`, `Bool`, Tupel davon, Aufzählungstypen.
- In Hugs/GHC vordefiniert (als “primitiver” Datentyp)

Listen sind keine `FiniteMap`s!

- `FiniteMap`: Assoziationslisten oder endliche Abbildungen
  - Effizient, weil durch balancierte Bäume implementiert.

```
data Ord key => FiniteMap key elt
```

- Alles fängt mit `emptyFM` an.
- Hinzufügen mit `addToFM`, `addToFM_C`.
  - Verhalten bei Überschreiben kann spezifiziert werden.
- Auslesen mit `lookupToFM`, `lookupWithDefaultFM`.

Unboxed Types

- boxed type: Elemente Objekte auf dem Heap
  - Vgl. Wrapper-Klassen in `Java`
- unboxed type: direkte Repräsentation als Maschinenworte
  - z.B. `Int#` (`long int`), `Double#` (`double`), `Addr#` (`void *`);
  - Operationen z.B. `(+#)` entspricht Addition auf Maschinenebene.
  - Objekte nterliegen nicht der garbage collection.
- Verschiedene Restriktionen (nicht für polymorphe Funktionen etc.)
- Nützlich: für numerik-intensive Berechnungen, innere Schleifen.

Zusammenfassung

- Endrekursion: `while` für Haskell.
  - Überführung in Endrekursion meist möglich.
  - Noch besser sind strikte Funktionen.
- Speicherlecks vermeiden: Striktheit, Endrekursion und Memoisation.
- Überladene Funktionen sind langsam.
- Listen sind keine Arrays.
- Referenzen in Haskell: `IORef`