Einschub: idiomatisches Haskell bei „Verzweigungen“

- Generell, statt Bedingungen xs == [ ] oder m == Nothing lieber null xs bzw. isNothing m.

- Außerdem, statt etwa:

```
let y = f x in
  if isJust y
    then something-involving-(fromJust y)-maybe-several-times
    else something-else
```

lieber:

```
case f x of
  Just y'   → the-first-something-from-above-but-with-y'-instead-of-(fromJust y)
  Nothing → something-else-as-above
```

(und ähnlich für analoge Situationen etwa bei Listen).

Strukturelle Rekursion auf Listen als Higher-Order Funktion

```
sum  [ ]      =  0
sum  (x : xs) =  x  +  sum  xs
```

```
prod  [ ]      =  1
prod  (x : xs) =  x  *  prod  xs
```

- Die Listenfunktionen zum Summieren bzw. Multiplizieren von Listenelementen weisen dasselbe Rekursionsmuster auf, das sich mit Hilfe einer vordefinierten Funktion zum „Falten“ von zweistelligen Operatoren in Listen realisieren lässt:

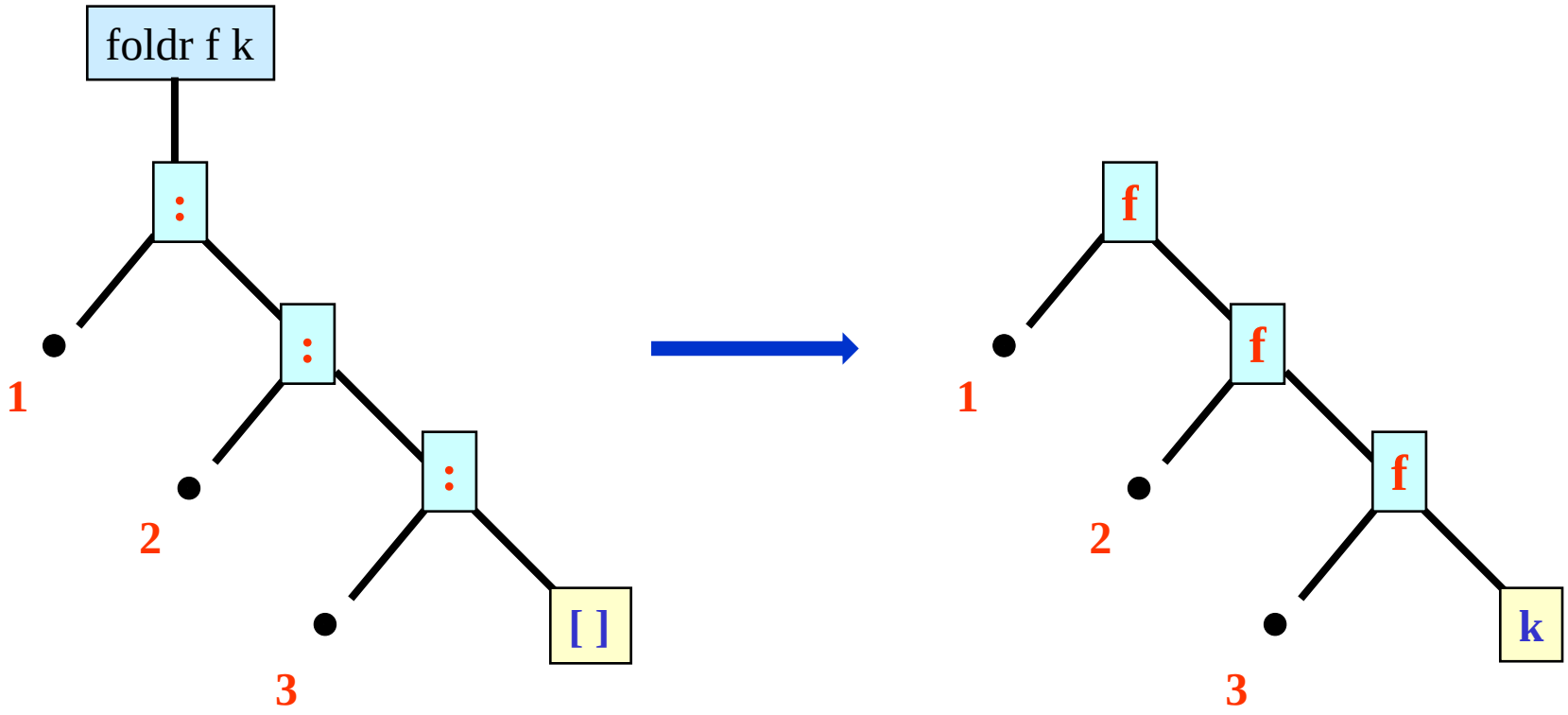
```
foldr  :: (a → b → b) → b → [a] → b
foldr  f  k  [ ]      =  k
foldr  f  k  (x : xs) =  f  x  (foldr  f  k  xs)
```

engl. „fold“: „falten“
(..r steht für „right“;
es gibt auch foldl)

- Zum Beispiel Definitionen von sum bzw. prod als Anwendung von foldr:

```
sum, prod :: [Int] → Int
sum  =  foldr  (+)  0
prod  =  foldr  (*)  1
```

Visualisierung von foldr



Weitere Beispiele für Verwendung von foldr

- Unter Verwendung von foldr sind vordefinierte logische Junktoren implementiert, die auf Listen von Booleschen Werten operieren:

```
and, or  :: [Bool] → Bool
and  =  foldr (&&) True
or   =  foldr (||) False
```

- „Quantoren“ über Listen sind als Verallgemeinerung dieser Junktoren mittels Komposition realisiert:

```
any, all  :: (a → Bool) → [a] → Bool
any p  =  or . map p
all p  =  and . map p
```

z.B.: `all (<100) [ x^2 | x ← [1 .. 19] ]`

Allgemeine Strategie zur Verwendung von foldr

- Wann kann eine Funktion mittels foldr ausgedrückt werden?
- Wann immer es möglich ist, sie in folgende Form zu bringen:

```
g [ ]       = k
g (x : xs)  = f  x (g xs)
```

für irgendwelche k und f

- Dann:

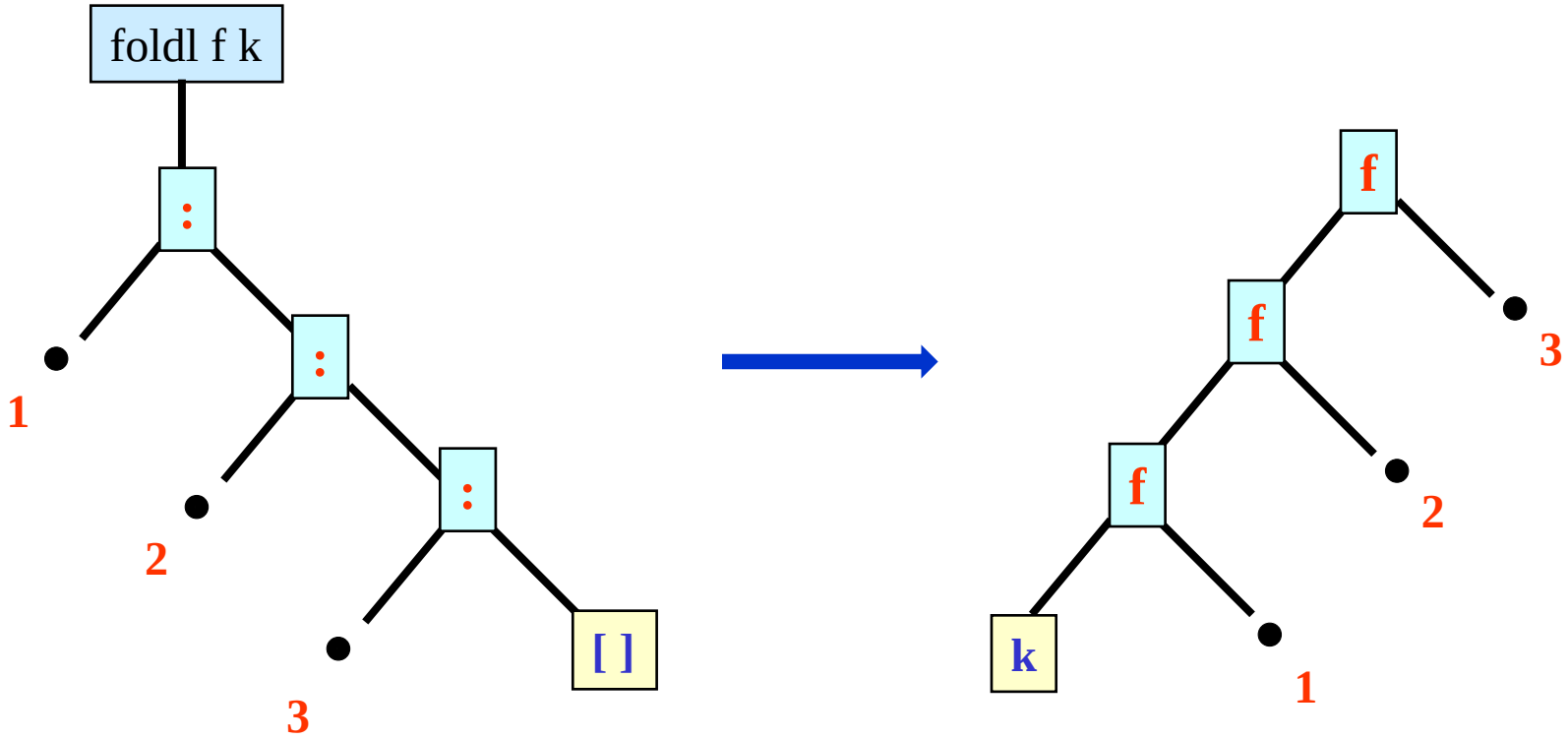
```
g  =  foldr  f  k
```

- Dies liefert eine einfache Charakterisierung strukturell rekursiver Funktionen auf Listen!

Eine linkslastige Variante von foldr

- Neben foldr gibt es:

```
foldl :: (b → a → b) → b → [a] → b
foldl f k [ ]       = k
foldl f k (x : xs)  = foldl f (f k x) xs
```



Variationen auf foldl und foldr

- Gibt auch alle Zwischenergebnisse von foldl aus:

```
scanl :: (b → a → b) → b → [a] → [b]
scanl f  k xs = k : case xs of
                      [ ]      → [ ]
                      x : xs' → scanl f  (f  k  x)  xs'
```

- Zum Beispiel:

```
> scanl (+) 0 [1 .. 5]
[0, 1, 3, 6, 10, 15]
```

- In gewissem Sinne dual zu foldr:

```
unfoldr :: (b → Maybe (a, b)) → b → [a]
unfoldr f b = case f b of
                  Nothing    → [ ]
                  Just (a, b') → a : unfoldr f b'
```

Higher-Order Abstraktion auf der Ebene von Typen

- Einige der betrachteten Funktionen (map, foldr, …) lassen sich außer auf Listen auch auf anderen Datentypen sinnvoll realisieren. Dazu Verwendung des Typklassenkonzepts, aber nun higher-order Typvariablen:

```
class Functor f where
  fmap :: (a → b) → f a → f b
```

- Instanzen existieren neben Listen zum Beispiel für Maybe und andere vordefinierte Datentypen.

- Und mit neueren GHCs kann man sogar entsprechende Instanzen für viele nutzerdefinierte Datentypen deriven lassen.

Deskriptive Programmierung

Ein- und Ausgabe in Haskell

Deklarative Programmierung als Programmierparadigma

Die deklarative Programmierung ist ein Programmierparadigma, welches auf mathematischer, rechnerunabhängiger Theorie beruht.

Zu den deklarativen Programmiersprachen gehören:

- funktionale Sprachen (u.a. LISP, ML, Miranda, Gofer, Haskell)
- logische Sprachen (u.a. Prolog)
- funktional-logische Sprachen (u.a. Babel, Escher, Curry, Oz)
- Datenflusssprachen (wie Val oder Linda)

(aus Wikipedia, 07.04.08)

- In der Regel erlauben deklarative Sprachen in irgendeiner Form die Einbettung imperativer Programmteile, mehr oder weniger direkt und/oder „diszipliniert“.

- Andere Programmiersprachenkategorien, einigermaßen orthogonal zu dekl./imp.:
  - Objektorientierte oder ereignisorientierte Sprachen
  - Parallelverarbeitende/nebenläufige Sprachen
  - Stark oder schwach, statisch oder dynamisch, oder gar nicht getypte Sprachen

Ein-/Ausgabe in Haskell, ganz einfaches Beispiel

- In „reinen“ Funktionen ist keine Interaktion mit Betriebssystem/Nutzer/… möglich.
- Es gibt jedoch eine spezielle do-Notation, die Interaktion ermöglicht, und aus der man „normale“ Funktionen aufrufen kann.

Einfaches Beispiel:

```
prod :: [Int] → Int
prod [ ]       =  1
prod (x : xs)  =  x * prod xs
```

reine Funktion

```
main = do n ← readLn
          m ← readLn
          print (prod [n .. m])
```

„Hauptprogramm“

Programmablauf

```
Eingabe → 5
Eingabe → 8
Ausgabe → 1680
```

Prinzipielles zu Ein-/Ausgabe in Haskell: IO-Typen …

- Es gibt einen vordefinierten Typkonstruktor IO, so dass sich für jeden konkreten Typ Int, Bool, [(Int, Tree Bool)] etc. der Typ IO Int, IO Bool, … bilden lässt.

- Die Interpretation eines Typs IO a ist, dass Elemente dieses Typs nicht selbst konkrete Werte sind, sondern „nur“ potentiell beliebig komplexe Sequenzen von Ein- und Ausgabeoperationen sowie von dabei eingelesenen Werten abhängigen Berechnungen, bei denen schließlich ein Wert des Typs a entsteht.

- Ein (eigenständig lauffähiges) Haskell-Programm insgesamt hat immer einen „IO-Typ“, in der Regel einfach main :: IO ().

- Zum Erzeugen von „IO-Werten“ gibt es vordefinierte Primitiven (und man sieht ihnen diesen IO-Charakter an Hand ihres Typs an):

```
getChar :: IO Char
getLine :: IO String
readLn :: Read a ⇒ IO a
```

```
putChar :: Char → IO ()
putStr, putStrLn :: String → IO ()
print :: Show a ⇒ a → IO ()
```

Prinzipielles zu Ein-/Ausgabe in Haskell: … und do-Notation

- Um IO-behaftete Berechnungen zu kombinieren (also basierend auf den IO-Primitiven komplexere Aktionssequenzen zu „bauen“), gibt es die do-Notation.

- Allgemeine Form:

$$
\begin{array}{l}
\text{do } cmd_1 \\
\quad x_2 \leftarrow cmd_2 \\
\quad x_3 \leftarrow cmd_3 \\
\quad cmd_4 \\
\quad x_5 \leftarrow cmd_5 \\
\quad \ldots
\end{array}
$$

Der do-Block insgesamt hat den Typ des letzten cmd_n. Zu diesem ist generell kein x_n vorhanden.

wobei die cmd_i jeweils IO-Typen haben und an die x_i (sofern explizit vorhanden) jeweils ein Wert des in cmd_i gekapselten Typs gebunden wird (und ab dieser Stelle im gesamten do-Block verwendet werden kann), nämlich gerade das Ergebnis der Ausführung von cmd_i.

- Oftmals auch noch nützlich (z.B. am Ende eines do-Blocks), vordefinierte Funktion return :: a → IO a, die ohne jegliche Ein-/Ausgabe einfach ihr Argument liefert.

Prinzipielles zu Ein-/Ausgabe in Haskell: IO-Typen und do-Notation

- Natürlich stehen auch im Kontext von IO-behafteten Berechnungen alle Features und Abstraktionsmittel von Haskell zur Verfügung, also wir definieren Funktionen mit Rekursion, verwenden Datentypen, Polymorphie, Higher-Order, …

- Ein „komplexeres“ Beispiel:

```
dialog = do putStr "Eingabe: "
            s ← getLine
            if s == "end"
               then return ()
               else do let n = read s
                       putStrLn ("Ausgabe: " ++ show (n * n))
                       dialog
```

- Was „nein, nicht, auf keinen Fall“ geht, ist aus einem IO-Wert direkt (abseits der expliziten Sequenzialisierung und Bindung in einem do-Block) den gekapselten Wert zu entnehmen.

- Neben den gesehenen Primitiven für Ein-/Ausgabe per Terminal gibt es Primitiven und Bibliotheken für File-IO, Netzwerkkommunikation, GUIs, …

Eigene „Kontrollstrukturen“

- Wie schon betont, stehen auch im Kontext von IO-behafteten Berechnungen alle Abstraktionsmittel von Haskell zur Verfügung, also insbesondere Polymorphie und Definition von Funktionen höherer Ordnung.

- Dies wird gern benutzt für Dinge wie:

```
while :: (a → Bool) → (a → IO a) → (a → IO a)
while p body  =  loop
   where  loop x  =  if  p x  then  do  x' ← body x
                                        loop x'
                                 else  return x
```

- Was ist dann wohl die Ausgabe folgenden Aufrufs?

```
> while (< 10) (\n → do {print n; return (n + 1)}) 0
```