Domain Specific Languages and Arrows

Heute Hier

- Arrows!
- Was ist das?
- Wozu ist das?
- Was ist eine Domain-Specific Language?

Domain Specific Languages (DSL)

- DSL: Sprache für speziellen Problembereich
  - Im Gegensatz zu universalen Programmiersprachen
  - Beispiel: LaTeX, Shell-Skripte, Spreadsheets, Emacs Lisp, . . .
- Implementierung von DSLs:
  - Einbettung in Haskell

Beispiel: Parserkombinatoren

- Typ für Parser:

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

- Monadeninstanz:

```
instance Monad (Parser a) where
  return a = \inp-> [(a, inp)]
  f >>= g  = \inp-> concat [g v out | (v, out)<- f inp]
```

- Problem: Alternierung zu kostspielig.

```
infixl 3 ‘alt‘
alt :: Parse a b-> Parse a b-> Parse a b
alt p1 p2 i = p1 i ++ p2 i
```

Voraussschauend Parsieren

- Lösung: LL(1) Parser mit lookahead
  - Können an ersten Token erkennen, ob es parsiert oder nicht.
  - Konstruktion einer Lookahead-Tabelle

```
data Lookahead  s = L  Bool [s]
data DynamicP s a = DP ([s]-> Maybe (a, [s]))
data Parser s a   = P (Lookahead s) (DynamicP s a)
```

- Kombinatorfunktionen, e.g.

```
token :: s-> Parser s s
token s = P (L False [s]) (DP (\ (x:xs)-> Just (s, xs))
```

  - Kein Test auf leere Eingabe oder `x == xs` nötig
- Damit effiziente Alternierung (Klasse `MonadPlus`):

```
instance MonadPlus Parser where
  P (L e1 s1) (DP p1) ‘mplus‘
  P (L e2 s2) (DP p2) =
    P (L (e1 || e2) (s1 ++ s2)
      (DP (\ x-> case x of
         [] -> if e1 then p1 [] else
               if e2 then p2 [] else Nothing
         (x:xs) -> if x ‘elem‘ s1 then p1 (x:xs) else
                   if x ‘elem’ s2 then p2 (x:xs) else
                   if e1 then p1 (x:xs) else
                   if e2 then p2 (x:xs) else Nothing)))
```

- Problem: Kein Instanz mehr für Monad:

  ```
  (>>=) :: Parser s a -> (a-> Parser s b)-> Parser s b
  ```

  - Zweiter Parser nicht mehr statisch konstruierbar.

Arrows!

- Idee: `Arrow a=> a b c` ist Berechnung mit Eingabe `b` und Ausgabe `c`
  - Internalisierung des Funktionstyps `b-> c`.

```
class Arrow a where
  arr :: (b -> c) -> a b c
  pure :: (b -> c) -> a b c
  (>>>) :: a b c -> a c d -> a b d
  first :: a b c -> a (b, d) (c, d)
  second :: a b c -> a (d, b) (d, c)
  (***) :: a b c -> a b' c' -> a (b, b') (c, c')
  (&&&) :: a b c -> a b c' -> a b (c, c')
```

- `arr` oder `pure`: Einbettung von Berechnungen
- `>>>`: Komposition
- `first`, `second`: Berchnung auf erstes (zweites) Argument anwenden.

```
second f = arr flip >>> first f >>> arr flip
```

- `***`: Beide Berechnungen anwenden.

```
f *** g = first f >>> second g
```

- `&&&`: Eingabe auf Berechnungen aufteilen

```
f &&& g = arr (\ b-> (b, b) >>> (f *** g))
```

Lifting

- `Lifting` (auf b): Funktion `a-> b` zu Funktion auf dem Datentypen `d`
  `a-> d b`
- Beispiel: monadische Addition:

```
addM :: Monad m=> m Int-> m Int-> m Int
addM a b = do x<- a; y<- b; return (x+ y)
```

- Allgemein:

```
liftM2 :: Monad m=> (a-> b-> c)-> m a-> m b-> m c
liftM2 f a b = do x<- a; y<- b; return (f x y)
```

- Auch für Arrows:

```
liftA2 :: Arrow a=> (b-> c-> d)-> a e b-> a e c-> a e d
liftA2 op f g = (f &&& g) >>> arr (\ (b, c)-> op b c)
```

Parserkombinatoren als Arrows

- Brauchen extra Parameter — Eingabetyp.
  - Lookahead nicht mehr von Eingaben zur Parsierzeit abhänging.

```
data DynamicP s a b = DP ((a, [s])-> Maybe (b, [s]))
data Parser   s a b = P (Lookahead s) (DynamicP s a b)
```

- Damit Instanz wie folgt:

```
instance Arrow (Parser s) where
  arr f = P (L True []) (DP (\ (b, s)-> Just (f b, s)))
  P (L e1 s1) (DP p1) >>> P (L e2 s2) (DP p2) =
    P (L (e1 || e2) (s1 `union` if e1 then s2 else []))
      (DP (\ (b,s)-> case p1 (b, s) of
                       Just (c, s')-> p2 (c, s')
                       Nothing     -> Nothing))
  first (P l (DP p)) =
    P l (\ ((b, d), s)-> case p (b, s) of
                           Just (c, s')-> Just ((c, d), s')
                           Nothing     -> Nothing)
```

Streams als Arrows

- Beispiel: Ströme als Berechnungen
- `SP a b` ist ein *stream processor* von `a` nach `b` mit

```
put :: b-> SP a b-> SP a b     -- b ausgeben, dann weiter
get :: (a-> SP a b)-> SP a b   -- a konsumieren, dann weiter
```

- Mögl. Implementation:

```
data SP a b = Put b (SP a b) | Get (a-> SP a b)
put = Put
get = Get
```

- Damit Instanz für `Arrow`

```
instance Arrow SP where
  arr f = Get (\ (x-> Put (f x) (arr f))
  sp1 >>> Put c sp2   = Put c (sp1 >>> sp2)
  Put b sp1 >>> Get f = sp1 >>> f b
  Get f1 >>> Get f2   = Get (\a-> f1 a >>> Get f2)
```

- Für `first`:

```
first f = bypass [] [] f
bypass [] ds (Get f) = Get (\ (b, d)-> bypass [] (ds++ [d]
bypass (b:bs) [] (Get f) = bypass bs [] (f b)
bypass bs [] (Put c sp)  =
  Get (\ (b, d)-> Put (c, d) (bypass bs++ [d]) [] sp)
bypass [] (b:bs) (Put c sp) =
  Put (c, d) (bypass [] ds sp)
```

  - Ungelesene Eingaben puffern, bis Ausgaben produziert werden
  - Argumente für `bypass`: ungelesene Eingaben (mind. eine leer)
- Damit zum Beispiel Fibonacci-Zahlen:

```
fibs  = put 0 fibs'
fibs' = put 1 (liftA2 (+) fibs fibs')
```

Arrow Laws

Folgende Gesetze gelten für `Arrow`:

- Assoziativität
- `id` ist rechts und links neutrales Element
- Funktorialität für `left`, `right`

Weitere Operationen

- Fallunterscheidungen:

```
class (Arrow a) => ArrowChoice a where
  left :: a b c -> a (Either b d) (Either c d)
  right :: a b c -> a (Either d b) (Either d c)
  (+++) :: a b c -> a b' c' -> a (Either b b') (Either c c')
  (|||) :: a b d -> a c d -> a (Either b c) d
```

- Feedback (Rekursion)

```
class (Arrow a) => ArrowLoop a where
  loop :: a (b, d) (c, d) -> a b c
```

Zusammenfassung

- Arrows sind Verallgemeinerung von Monaden
- 'Internalisierung' des Funktionstyps $\Rightarrow$ besonders nützlich für DSLs.
- Exotischer als Monaden
  - Monaden sind unerläßlich, Arrows sind optional
- Nützlich?