Woche 7 – Übersicht, Tricks & Aufgaben

(Disclaimer: Die hier vorzufindenden Notizen haben keinerlei Anspruch auf Korrektheit oder Vollständigkeit und sind nicht Teil des offiziellen Vorlesungsmaterials. Alle Angaben sind ohne Gewähr.)

Anmerkungen zu den Code-Expert Abgaben:

- Die Abgaben, die ich bekommen habe, sahen insgesamt sehr gut aus. (50% der Stud.)
- Ich empfehle **allen** dringend, die Induktionsaufgaben zu lösen.

Anmerkungen zu den Moodle Abgaben:

- Viele Abgaben waren sehr gut und fehlerfrei, vereinzelt gab es Probleme, die vermutlich auf Flüchtigkeitsfehler zurückzuführen sind. Falls nicht alles gestimmt hat, guckt auf jeden Fall noch einmal in die Musterlösung. Alle anderen, die die Aufgaben nicht abgegeben haben, sollten unbedingt auch in der Musterlösung gucken, da solche Evaluation-Strategy Aufgaben auch in vergangenen Jahren immer wieder an Klausuren abgefragt wurden (und da werden euch die Regeln dann üblicherweise nicht gegeben!)
- **Es gab auch diese Woche nur zu wenige (8) Abgaben. Löst unbedingt die Serien!**

Foldr nach Rezept aber ohne statische oder dynamische Argumente:

(Vermeintlicher) special Case, hier ist die Erklärung:

https://n.ethz.ch/~mschlegel/fmfp22/foldrexerc.pdf

Folds Pt.2:

Letzte Woche haben wir das Thema “Folds für beliebige Algebraic Datatypes” eingeführt. Heute möchte ich noch einmal die Intuition für das Lösen der typischen Aufgaben zu diesem Thema genauer erklären und anschliessend noch einige Beispiele geben.

Ein Fold reduziert eine Datenstruktur auf einen einzigen Wert. Sagen wir von nun an ausserdem, dass dieser Wert vom Type c sein soll. Also brauchen wir irgendeine Funktion, welche die Elemente der Datenstruktur als Eingabe nimmt und diesen gewünschten Wert zurückgibt (so wie wir in Funktionalen Programmiersprachen nun mal denken). Bei Lists könnte ein Fold bspw so aussehen:

$$foldList\ f\ z\ (k : v : []) = f\ \ k\ \ (f\ \ v\ \ (foldList\ f\ \ z\ \ []))$$

Wenn wir nun unsere eigene Liste definieren, ist das hier eigentlich genau das gleiche:

Definition: $$data\ List\ a = Cons\ a\ (List\ a)\ \mid\ Empty$$

$$foldList\ f\ z\ (Cons\ k\ (Cons\ v\ (Empy))) = f\ \ k\ \ (f\ \ v\ \ (foldList\ \ f\ \ z\ \ Empty))$$

(Wer sich auskennt, sieht hier sofort die Prelude-Funktion $foldr$, die selbstverständlich ein Fold ist)

Wenn wir uns das genau angucken, sehen wir, dass $foldList$ eine Funktion f verwendet, die zwei Argumente nimmt und dann einen Wert vom Type c zurückgibt. Wenn wir uns angucken, **wie genau $\boldsymbol{foldList}$** die Funktion auf die gegebene Datenstruktur (Liste) anwendet, erkennen wir folgendes: f nimmt als erstes Argument ein Element der Datenstruktur und als zweites Argument das Ergebnis der rekursiven Bearbeitung des Rests der Liste. So wir es wollen, gibt uns f einen Wert vom Typ c zurück. Das heisst, f's erstes Argument ist vom Type den die Listenelemente haben, nennen wir ihn a und f's „zweites Argument" (man erinnere sich an Higher Order Programming…) hat Type c – denn als zweites Argument nimmt f das Ergebnis des rekursiven Aufrufs – der ein Rückgabewert von f ist, und damit vom Type c ist.

Also hat f insgesamt den Type: $f :: a \to c \to c$.

Dieses f wenden wir nun auf eine Liste an und bekommen dann abschliessend ein c zurück. Damit können wir nun einfach den Type von $foldList$ bestimmen:

$$foldList :: (a \to c \to c) \to c \to List\ a \to c$$

Denn $foldList$ nimmt als Argument f, dann den „Base-Case-Wert" z, der natürlich auch vom Type c sein muss, da er von f als „zweites Argument" genommen wird und eine Liste und gibt dann den gewünschten Rückgabewert zurück, der vom Type c ist.

Die definierte Fold-Funktion ($foldList$) ist nun eine valide Lösung für einen Fold, da sie Elemente vom Type $List\ a$ auf einen Wert reduziert.

Nun müssen nur noch die Definition von $foldList$ bestimmen. Aber das ist eigentlich relativ einfach: Im rekursiven Fall, also wenn die Liste so aussieht: $Cons\ x\ xs$, dann macht $foldList$ offensichtlich folgendes:

$$foldList\ \boldsymbol{f}\ z\ (Cons\ x\ xs) = \boldsymbol{f}\ x\ (foldList\ f\ z\ xs)$$

Also gibt sie das Ergebnis von f, angewendet auf x und das Ergebnis des rekursiven Aufrufs von sich selbst zurück.

Im „Base-Case" Fall, also wenn die Liste so aussieht: $Empty$, macht $foldList$ folgendes:

$$foldList\ f\ \boldsymbol{z}\ (Empty) = \boldsymbol{z}$$

Also wird im Grunde genommen im Case $(Cons\ x\ xs)$ die Funktion f angwendet auf irgendwas zurückgegeben und im Case $(Empty)$ der Wert z zurückgegeben. Theoretisch könnte z aber natürlich auch eine Funktion sein. Zusammengefasst also:

```haskell
foldList :: (a -> c -> c) -> c -> List a -> c
foldList f z (Cons x xs)  = f x (foldList f z xs)
foldList f z (Empty)      = z

-- Oder, einfach umgeschrieben:
foldList :: (a -> c -> c) -> c -> List a -> c
foldList f z = go
    where
        go (Cons x xs)    = f x (go xs)
        go (Empty)        = z
```

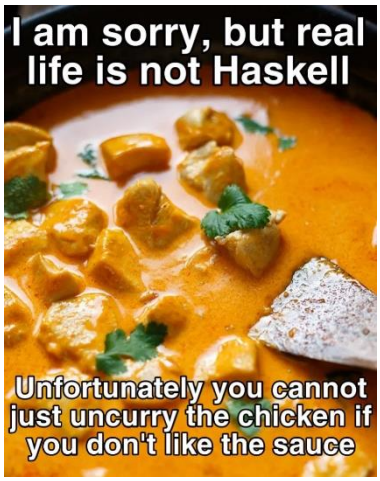
Ich benenne nun einmal f und z um, aber ändere sonst nichts, hier steht also exakt das gleiche, nur mit anderen Namen:

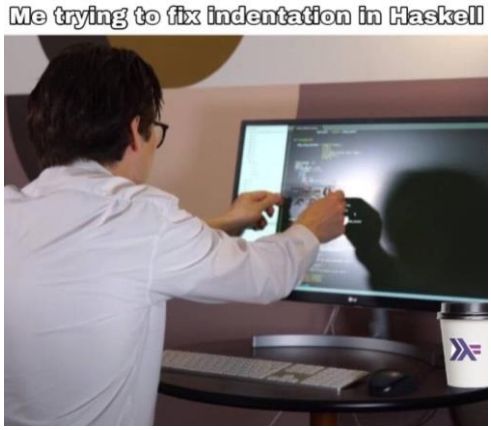
```haskell
foldList :: (a -> c -> c) -> c -> List a -> c
foldList fCons fEmpty = go
    where
        go (Cons x xs)    = fCons x (go xs)
        go (Empty)        = fEmpty
```

Und wir haben $foldList$ als Fold für den DataType $List\ a$ definiert.

Ihr solltet nun die Intuition für die einzelnen Funktionen und ihren Typen entwickelt haben, nun gucken wir uns noch ein weiteres Beispiel (und eine schnellere Herleitung) an. Anschliessend noch einige weitere Beispiele:

Ärgerlicherweise ist mir kein Weg eingefallen, die Formatierung so hinzubekommen, dass hier nicht eine riesige Lücke entsteht, also FMFP-MEMES: (Quelle: der legendäre eth-memes channel auf discord)





Nun, da wir verstanden haben, wie man die Types für Listen definiert, hier noch ein weiteres Beispiel:

```haskell
data Tree a = Leaf a | Node a (Tree a) (Tree a)

{-

Wir haben also einen Tree gegeben. Nun überlegen wir uns, wie hier ein Fold funktionieren würde:
Wir brauchen eine Funktion, die einen Tree als Argument nimmt und - wie eben- einen gewünschten Wert
vom Type 'c' zurückgibt. Ein Tree kann entweder ein Leaf sein oder eine Node. Betrachten wir mal
zuerst den Fall, dass ein Tree eine Node ist. Sei folgender Tree gegeben: "Node x l r" und sei
fNode die Funktion, die das als Input nimmt und einen Wert zurückgibt. Für die Funktion, die den
Tree folded würde dann in diesem Fall also ungefähr das hier passieren:

foldTreeCaseNode fNode (Node x l r) = fNode x (foldTreeCaseNode fNode l) (foldTreeCaseNode fNode r)

fNode nimmt erst das Element in der Node, dann das Ergebnis des rekursiven Aufrufs auf dem linken Subtree l
und dann das Ergebnis des rekursiven Aufrufs auf dem rechten Subtree r. Da der Rückgabewert von foldTree
vom Type c sein soll und damit auch die Ergebnisse der rekursiven Aufrufe vom Type 'c' sein müssen, hat
fNode also offensichtlich folgenden Type: fNode :: a -> c -> c -> c

Nun betrachten wir den anderen Fall: Was ist, wenn der Tree ein Leaf a ist, bspw. "Leaf x"?
Dann würde eine Funktion, die den Tree folded, darauf folgendermassen arbeiten:

foldTreeCaseLeaf fLeaf (Leaf x) = fLeaf x

Die Funktion fLeaf hat also offensichtlich den Type: fLeaf :: a -> c (denn der Rückgabewert von foldTree
soll vom Type 'c' sein).

Da ein Tree aber sowohl aus Leafs als auch aus Nodes bestehen kann, müssen wir die beiden Definitionen
vereinen, damit es beim rekursiven Aufruf in foldTreeCaseNode nicht zu Problemen kommt. Das ist aber
sehr einfach, wir verwenden einfach Pattern Matching:

foldTree fNode fLeaf (Node x l r) = fNode x (foldTree l) (foldTree r)
foldTree fNode fLeaf (Leaf x)     = fLeaf x

Wir können nun auch sehr einfach den Type von foldTree bestimmen:

foldTree :: (a -> c -> c -> c) -> (a -> c) -> Tree a -> c

Und wenn wir die Funktion noch hübsch umschreiben, erhalten wir das typische Ergebnis:

-}

foldTree :: (a -> c -> c -> c) -> (a -> c) -> Tree a -> c
foldTree fNode fLeaf = go
    where
        go (Node x l r) = fNode x (go l) (go r)
        go (Leaf x)     = fLeaf x

{-

Wäre das auch schneller gegangen? JA: Am einfachsten ist folgendes Rezept: Type Definition angucken, für
jeden Data Constructor gibt es eine Funktion vom folgenden Type:
fdataConstructor :: <Types des data constructor> -> <Type des RückgabeWerts>
Wobei man bei <Types des data constructor> jedes Vorkommen des type constructors durch den Rückgabewert
ersetzt (da dies die Elemente sind, die rekursiv bearbeitet werden, also muss die Funktion
fdataConstructor hier nicht bspw. ein Element vom Type "Tree a" nehmen, sondern nimmt ein Argument das
den Type des Rückgabewerts der Fold-Funktion hat, da das Argument das Ergebnis eines rekursiven Aufrufs ist).

Hier z.B.: Node a (Tree a) (Tree a) gibt: fNode :: (a -> c -> c -> c)
Und        Leaf a gibt: fLeaf (a -> c)

Dann kann man einfach immer die foldFunktion nach diesem Schema schreiben:
foldDataType :: Type
foldDataType fdataConstructor1 fdataConstructor2 ... fdataConstructorN = go
    where
        go (dataConstructor1 ...) = fdataConstructor1 ...          (evtl. rekursive Aufrufe von go)
        go (dataConstructor2 ...) = fdataConstructor2 ...          (-//-)
        .                         .                                .
        .                         .                                .
        .                         .                                .
        go (dataConstructorN ...) = fdataConstructorN ...          (-//-)

-}
```

```haskell
data Mlist a = Bot | Node [a] (Mlist a)

foldMlist :: c -> ([a] -> c -> c) -> Mlist a -> c
foldMlist fBot fNode = go
    where
        go (Bot)            = fBot
        go (Node xs mlist)  = fNode xs (go mlist)
-------------------------------------------------------------------------------

data Dict k v = Empty | Join (Dict k v) k v (Dict k v)

foldDict :: c -> (c -> k -> v -> c -> c) -> Dict k v -> c
foldDict fEmpty fJoin = go
    where
        go (Empty)                  = fEmpty
        go (Join dict1 x y dict2)   = fJoin (go dict1) x y (go dict2)
-------------------------------------------------------------------------------

data Tree a b = Leaf a | Node (Tree a b) (Tree a b) | List [b] (Tree a b)

foldTree :: (a -> c) -> (c -> c -> c) -> ([b] -> c -> c) -> Tree a b -> c
foldTree fLeaf fNode fList = go
    where
        go (Leaf x)                 = fLeaf
        go (Node tree1 tree2)       = fNode (go tree1) (go tree2)
        go (List xs tree)           = fList xs (go tree)
-------------------------------------------------------------------------------

data Seq a = Empty | Single a | Concat (Seq a) (Seq a)

foldSeq :: c -> (a -> c) -> (c -> c -> c) -> Seq a -> c
foldSeq fEmpty fSingle fConcat = go
    where
        go (Empty)                  = fEmpty
        go (Single x)               = fSingle x
        go (Concat l r)             = fConcat (go l) (go r)
-------------------------------------------------------------------------------

data Interval a = V a a

foldInterval :: (a -> a -> c) -> Interval -> c
foldInterval fV = go
    where
        go (V x y) = fV x y
-------------------------------------------------------------------------------

data F a = AP a | Not (F a) | And (F a) (F a) | E (F a)

foldF :: (a -> c) -> (c -> c) -> (c -> c -> c) -> (c -> c) -> F a -> c
foldF fAP fNot fAnd fE = go
    where
        go (AP x)           = fAP x
        go (Not f)          = fNot (go f)
        go (And f1 f2)      = fAnd (go f1) (go f2)
        go (E f)            = fE (go f)
```

Parser:

Ich empfehle stark, die Vorlesungslides genau durchzulesen und dabei immer wieder zu überlegen, was genau gerade eigentlich definiert wird. Eigentlich ist die Idee relativ einfach. Um die Übungsaufgaben der Serie 7 zu lösen, kann ich ausserdem die Code-Examples unter Code-Expert empfehlen.

Lazy / Eager Evaluation – Anmerkungen:

```
Wie würde man hier lazy evaluieren:
    (\x y -> x y) (\x -> x)

Zunächst einmal rufen wir uns folgendes über Haskell-Lambda-Funktionen in Erinnerung:
Grundsätzlich können wir Funktionen der Form (\x y -> ...) umschreiben zu
(\x -> \y -> ...). Das liegt - wie so oft - am Prinzip des Higher-Order Programmings.
Eine Funktion nimmt nicht zwei Argumente x,y und gibt ... zurück, sondern sie nimmt ein
Argument x und gibt eine Funktion zurück, die ein y nimmt und ... zurückgibt.
Das heisst, im fall (\x y -> ...) t2 substituieren wir t2 für x in t1 und erhalten die
Funktion (\y -> ...x_substituiert_mit_t2...).

Also bekommen wir für das Beispiel oben:

(\x y -> x y) (\x -> x)

(\y -> (\x -> x) y)
```

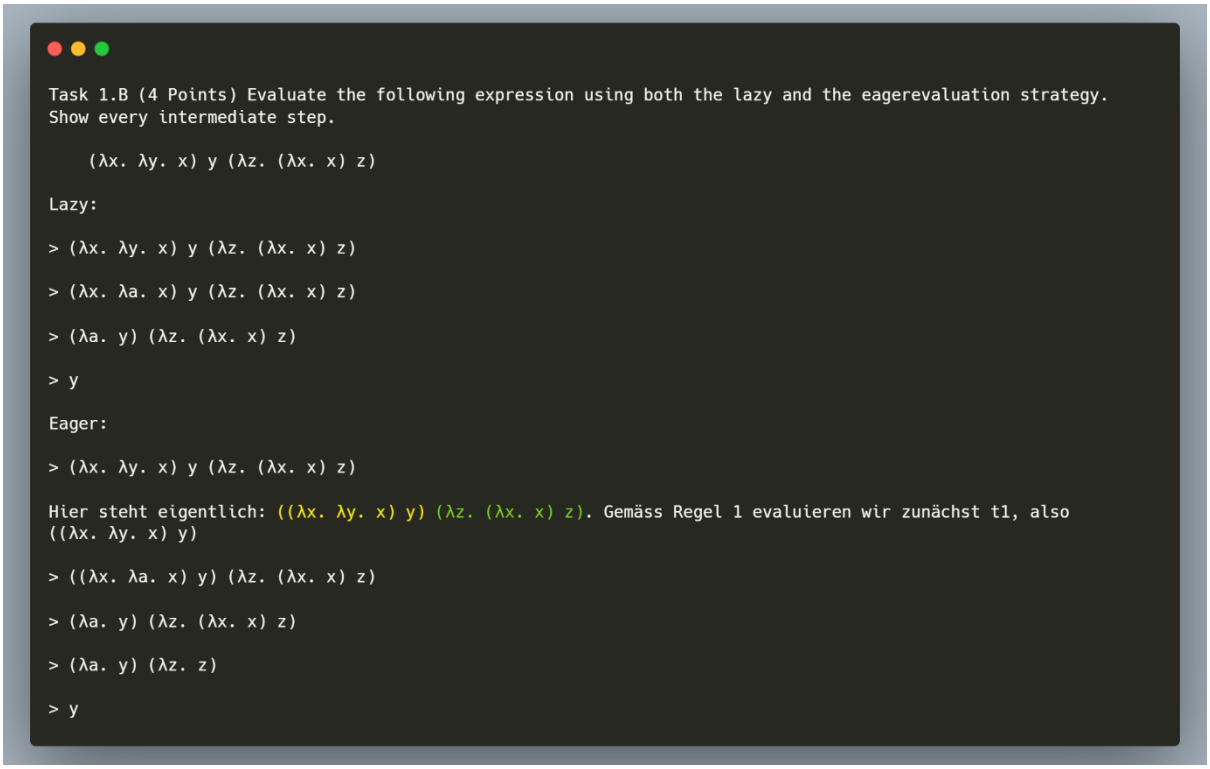
Credits: Artem, (Discord ETH-Memes Channel)

Aufgaben aus Alt-Klausuren FP-Part

Type Inference Exercise (HS21):

```
Task 1.A (3 Points) Recall the following functions from the Haskell Prelude library.

elem :: Eq a => a -> [a] -> Bool
map :: (a -> b) -> [a] -> [b]
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]

Determine whether the following expressions are well-typed or not, and if they are, state themost general type.
No other justification is needed.

#1:
zipWith elem

Wir sehen: zipWith :: (a -> b -> c) -> ([a] -> [b] -> [c])
Da zipWith als Argument die Funktion elem :: Eq q => q -> [q] -> Bool nimmt, können wir also sofort
bestimmen, was a,b,c in zipWith :: (a -> b -> c) -> ([a] -> [b] -> [c]) sind:

(a -> b -> c) = Eq q => q -> [q] -> Bool

Also: a = Eq q => q
      b = Eq q => [q]
      c = Bool

Da das zipWith sein Argument bekommen hat, ist der Return-Type das, was rechts vom Pfeil steht, also:
([a] -> [b] -> [c]). Und wir wissen ja jetzt, was a,b,c sind, also bekommen wir:

zipWith elem :: Eq q => [q] -> [[q]] -> [Bool]
-------------------------------------------------------------------------------------------------------

#2:
\x -> map x [x]

Wir sehen: map :: (a -> b) -> [a] -> [b]
Da map als Argument x bekommt, muss also per Unification x :: (a -> b) sein.
Als "zweites Argument" bekommt map ausserdem [x], also: x :: a

Nun versuchen wir das zu vereinen: a = (a -> b). Das klappt allerdings nicht, da es sich um einen sog.
"infinite type" handelt (Dieser Fehlermeldung habt ihr in Haskell höchstwahrscheinlich schon einmal
bekommen wenn ihr etwas illegales gemacht habt).
-------------------------------------------------------------------------------------------------------

#3:
\x y z -> (x z) (y z)

Wir sehen: Es gibt eine Function Application (x z) (y z). Also muss (x z) eine Funktion sein, die das Argument
(y z) nimmt und irgendwas zurückgibt.

Also: (x z) :: a -> b (für irgendwelche a,b)

Nun betrachten wir (x z) genauer. Auch hier handelt es sich um eine Function Application, wobei x eine Funktion
sein muss, die das Argument z nimmt und etwas vom Type (a -> b) zurückgibt (denn (x y) soll ja den Type
(a -> b) haben.

Also: x :: c -> (a -> b) = c -> a -> b
Also: z :: c (da z an x als Argument übergeben wird)

Nun gucken wir uns noch (y z) an: Auch hier handelt es sich um eine Function Application, also muss y eine
Funktion sein, die ein Argument vom Type von z, also c, nimmt und etwas vom Type a zurückgibt. Warum?
Denn (y z) wird als Argument an (x z) übergeben und wir hatten festgestellt, dass (x z) den Type a -> b hat.

Also: y :: c -> a

Zusammengefasst: \x y z -> (x z) (y z) :: (c -> a -> b) -> (c -> a) -> c -> b
```

Lazy / Eager Evaluation (HS21):



Task 1.B (4 Points) Evaluate the following expression using both the lazy and the eagerevaluation strategy. Show every intermediate step.

(λx. λy. x) y (λz. (λx. x) z)

Lazy:

> (λx. λy. x) y (λz. (λx. x) z)

> (λx. λa. x) y (λz. (λx. x) z)

> (λa. y) (λz. (λx. x) z)

> y

Eager:

> (λx. λy. x) y (λz. (λx. x) z)

Hier steht eigentlich: ((λx. λy. x) y) (λz. (λx. x) z). Gemäss Regel 1 evaluieren wir zunächst t1, also ((λx. λy. x) y)

> ((λx. λa. x) y) (λz. (λx. x) z)

> (λa. y) (λz. (λx. x) z)

> (λa. y) (λz. z)

> y

Induktion (FS 20):

Gegeben: data Direction = L | R, data Path = End | Node Int Direction Path

Task 2.B (8 Points) Consider the following Haskell declarations.

```
opp :: Direction -> Direction
opp L = R                                          -- opp.1
opp R = L                                          -- opp.2

mirror :: Path -> Path
mirror End          = End                          -- mirror.1
mirror (Node x d p) = Node x (opp d) (mirror p)    -- mirror.2

invert :: Path -> Path -> Path
invert p End          = p                          -- invert.1
invert p (Node x d q) = invert (Node (-x) d p) q   -- invert.2
```

Prove that for all finite paths `p::Path` the equality

```
mirror (invert End p) = invert End (mirror p)
```

holds. Structure your proof clearly and justify every proof step.

Wir haben hier den Fall, den ich in einer der Vorwochen schon intensiv besprochen habe: Wenn wir den Induktionsbeweis für die gegebene Aussage führen, bekommen wir ein Problem, da sie „zu spezifisch“ ist. Hier wird auf beiden Seiten der Gleichungen anstatt genereller Path Elemente explizit „End“ verwendet. Man kann leicht (durch einfaches hinschreiben des Proofs) sehen, dass man den Induktionsschritt so nicht vervollständigen kann.

Wir müssen also generalisieren: Das haben wir bislang immer so gemacht, indem wir das „spezifische Element“, bspw. die „0“ oder die leere Liste, „[]“, durch ein allgemeines Element, bspw. n::Nat oder ys::[a] ersetzt haben.

Dabei habe ich euch jedoch folgendes gezeigt gehabt: Pures Ersetzen durch ein „allgemeines Element“ führt meistens nicht zur korrekten, generalisierten Aussage. Man muss eher auf einer Seite das „spezifische Element“ durch ein „allgemeines Element“ ersetzen und dann überlegen, was dies für die andere Seite bedeutet. Hier wäre beispielsweise der Ansatz

$$\mathrm{P} \coloneqq \forall \mathrm{q} :: \mathrm{Path}\ .\ \mathrm{mirror}\ (\mathrm{invert}\ \mathrm{q}\ \mathrm{p}) = \mathrm{invert}\ \mathrm{q}\ (\mathrm{mirror}\ \mathrm{p})$$

falsch, wie man recht schnell sehen kann, wenn man sich die Funktionen genau anguckt. Anstatt dessen gehen wir folgendermaßen vor: Wir ersetzen auf der einen Seite und gucken dann nach, was das für die Gleichung bedeutet:

$$\mathrm{mirror}\ (\mathrm{invert}\ \mathrm{q}\ \mathrm{p})$$

Was genau passiert hier eigentlich? Fangen wir mir $\mathrm{invert}\ \mathrm{q}\ \mathrm{p}$ an:

Das Argument q wird von der Funktion invert gar nicht verändert. Dafür werden die Nodes vom Path p in umgedrehter Reihenfolge und mit gewechselten Vorzeichen „vorne“ an den Path q herangehängt.

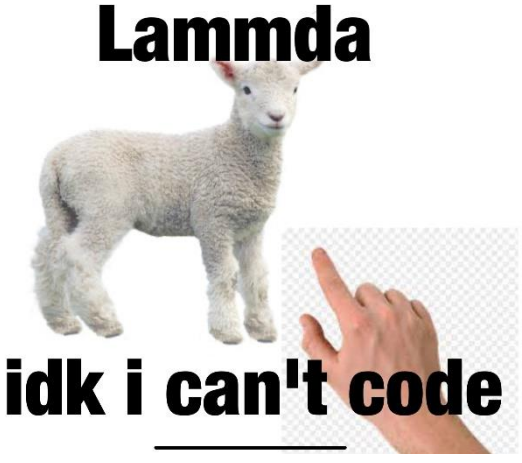
Die Funktion $mirror$ wechselt von diesem erhaltenen Pfad dann noch an jeder Node die Direction.

Okay – wie können wir das auf der RHS der Gleichung mit
$\mathrm{invert} < \mathrm{hier\_muss\_was\_hin} > (\mathrm{mirror}\ \mathrm{p})$ ausdrücken?

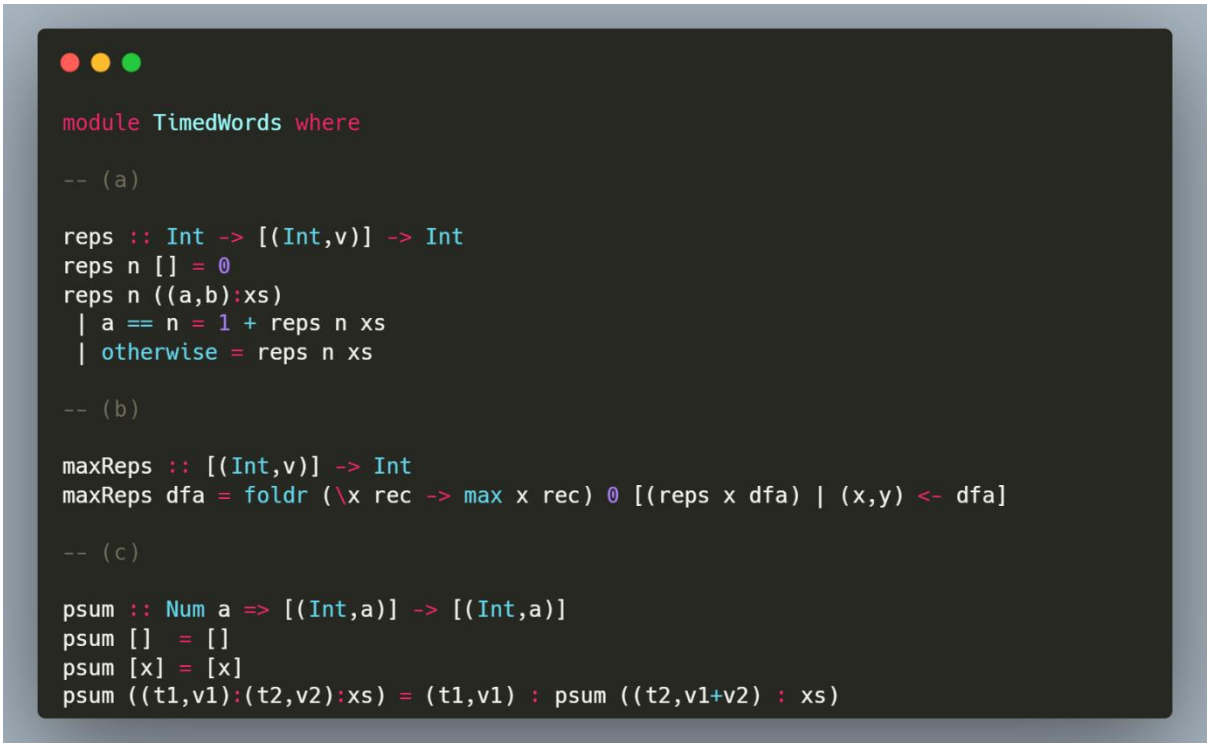
Eigentlich ist das relativ offensichtlich: invert wird die Nodes von $(\mathrm{mirror}\ \mathrm{p})$ in umgekehrter Reihenfolge (und mit umgekehrtem Vorzeichen an den Labels der Nodes) vorne an $< \mathrm{hier\_muss\_was\_hin} >$ ranhängen. Damit das Äquivalent zur LHS ist, muss also $< \mathrm{hier\_muss\_was\_hin} > = (\mathrm{mirror}\ \mathrm{q})$ sein. Denn dann erhalten wir einen Pfad, bei dem an jeder Node die Direction gewechselt wurde, und die Nodes von p vorne in verkehrter Reihenfolge und mit gewechseltem Vorzeichen an q angehängt werden.

Also ist die generalisierte Aussage:

$$\mathrm{P} \coloneqq \forall \mathrm{q} :: \mathrm{Path}\ .\ \mathrm{mirror}\ (\mathrm{invert}\ \mathrm{q}\ \mathrm{p}) = \mathrm{invert}\ (\mathrm{mirror}\ \mathrm{q})\ (\mathrm{mirror}\ \mathrm{p})$$



Timed Words (Code Examples):



```haskell
module TimedWords where

-- (a)

reps :: Int -> [(Int,v)] -> Int
reps n [] = 0
reps n ((a,b):xs)
 | a == n = 1 + reps n xs
 | otherwise = reps n xs

-- (b)

maxReps :: [(Int,v)] -> Int
maxReps dfa = foldr (\x rec -> max x rec) 0 [(reps x dfa) | (x,y) <- dfa]

-- (c)

psum :: Num a => [(Int,a)] -> [(Int,a)]
psum []  = []
psum [x] = [x]
psum ((t1,v1):(t2,v2):xs) = (t1,v1) : psum ((t2,v1+v2) : xs)
```