Woche 5 – 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 zu den Fold-Aufgaben bekommen habe, haben mir alle sehr gut gefallen und zeigen, dass ihr das grundsätzliche Prinzip von Folds in Haskell sehr gut versteht.
- Insgesamt gab es aber (insbesondere bei der DFA-Aufgabe) diese Woche deutlich weniger Abgaben (ca. 15 weniger als sonst), weswegen ich euch hier noch einmal eindringlich darauf hinweisen möchte, mit dem Lösen der Aufgaben nicht aufzuhören. Es ist auch okay, wenn ihr die Aufgaben erst mit der Lösung löst, aber versucht es zumindest alleine – und falls das nicht klappt, versucht auf jeden Fall die Lösung zu verstehen (und bestenfalls auch noch einmal selbst zu implementieren, das hilft erfahrungsgemäß oft). Da die DFA-Aufgabe vermutlich für die meisten an der „lexicon"-Teilaufgabe gescheitert ist, hier noch einmal ein Lösungsvorschlag von mir (genauere Erklärung in der Übungsstunde):

```haskell
lexicon :: Alphabet a -> Int -> [Word a]
lexicon alphabet n = foldr (\x rec -> [front : old | front <- alphabet, old <- rec]) [[]] [1..n]
```

Anmerkungen zu den Theorie Abgaben:

- Alle Abgaben, die ich bekommen habe, haben mir sehr gut gefallen, ihr habt CYP/Induktion offensichtlich sehr gut verstanden :)

Foldr – Fortgeschritten:

Wir haben bereits in den letzten beiden Wochen gesehen, dass wir die überaus mächtige Funktion „foldr“ nutzen können, um andere, rekursive implementierte Funktionen neu zu schreiben. Heute lernen wir einen Formalen Ansatz kennen, der einfach nur Stur befolgt werden muss – und es uns erlaubt, die meisten rekursiven Prelude-Funktionen mit foldr auszudrücken:

Hier das Rezept und mehrere Beispielaufgaben, auch eine „ungewöhnliche“ (da ohne dynamische Argumente) (Quelle: Slides):

Rewrite `foldl` using `foldr`.

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

General procedure:

1. Identify recursive, dynamic, and static arguments.
   `foldl f z (x:xs) = foldl f (f z x) xs`
2. Write an aux function that has the recursive, then the dynamic arguments. Static arguments can still occur freely (and will come from the final context).
   ```
   aux [] z = z
   aux (x:xs) z = aux xs (f z x)
   ```
3. Write the dynamic arguments as lambdas.
   ```
   aux [] = \z -> z
   aux (x:xs) = \z -> aux xs (f z x)
   ```
4. Rewrite aux in terms of `foldr`. `x` and **`aux xs`** become arguments of the function for the recursive case.
   `aux = foldr (\x rec -> \z -> rec (f z x)) (\z -> z)`
5. Express the original function in terms of aux (reorder the dynamic and recursive arguments, if needed).
   `foldl f z xs = aux xs z`
6. Replace aux with its implementation.
   `foldl f z xs = foldr (\x rec z -> rec (f z x)) (\z -> z) xs z`

```haskell
-- Rekursive Definition von foldl:
myMap :: (a -> b) -> [a] -> [b]
myMap f [] = []
myMap f (x:xs) = (f x) : myMap f xs

aux1 [] = []
aux1 (x:xs) = (f x) : aux1 xs

-- Da es keine dynamischen Argumente gibt, ändert sich aux nicht
aux2 [] = []
aux2 (x:xs) = (f x) : aux1 xs

-- Nach dem gewohnten Rezept geht es weiter:
aux3 = foldr (\x acc -> (f x) : acc) []

myMapv2 f xs = aux3 xs

myMapv3 f xs = foldr (\x acc -> (f x) : acc) [] xs
```

```haskell
-- Zuerst wollen wir foldl mit foldr implementieren:
-- Rekursive Definition von foldl:
myFoldl :: (b -> a -> b) -> b -> [a] -> b
myFoldl f z [] = z
myFoldl f z (x:xs) = myFoldl f (f z x) xs

aux1 [] z = z
aux1 (x:xs) z = aux1 xs (f z x)

aux2 [] = \z -> z
aux2 (x:xs) = \z -> aux2 xs (f z x)

aux3 = foldr (\x acc -> \z -> acc (f z x)) (\z -> z)

myFoldlv2 f z xs = aux3 xs z

myFoldlv3 f z xs = foldr (\x acc -> \z -> acc (f z x)) (\z -> z) xs z

-- Jetzt wollen wir scanl mit foldr implementieren:
-- Rekursive Definition von scanl:
myScanl :: (b -> a -> b) -> b -> [a] -> [b]
myScanl f z [] = [z]
myScanl f z (x:xs) = z : myScanl f (f z x) xs

aux1 z [] = [z]
aux1 z (x:xs) = z : aux1 (f z x) xs

aux2 [] = \z -> [z]
aux2 (x:xs) = \z -> z : aux2 (f z x) xs

aux3 = foldr (\x acc -> \z -> z: acc (f z x)) (\z -> [z])

myScanlv2 f z xs = aux3 xs z

myScanlv3 f z xs = foldr (\x acc -> \z -> z: acc (f z x)) (\z -> [z]) xs z
```

Das Rezept muss einfach stur befolgt werden (und immer genau erkannt werden, welche Argumente statisch, welche dynamisch und welches das rekursive Argument ist), dann lassen sich entsprechende Aufgaben gewöhnlich sehr leicht lösen.

Type-Inference:

Nun wollen wir uns noch einmal mit der Type-Inference beschäftigen. Bereits in der letzten Stunde hatten wir das Thema grob angeschnitten. Hier noch einmal mein Rezept:

Weiter auf der nächsten Seite →

Am Beispiel **bestimme den most general type von map . (,)** mit

map :: (a->b)->[a]->[b],

(.) :: (b->c)->(a->b)->a->c,

(,):: a->b->(a,b)

Da wir gleich mehrere Male Typen „matchen“ müssen, sollten wir zunächst die ganzen freien Typenvariablen umbenennen:

map :: (e->f)->[e]->[f],

(.) :: (b->c)->(a->b)->a->c,

(,) :: x->y->(x,y)

Wie man sehen kann, stimmen die Typen immer noch, da ich die Variablen entsprechend dem vorgegebenen Muster geändert habe. Jetzt kommen wir aber nicht so durcheinander wenn wir über die einzelnen Typen der verschiedenen Funktionen sprechen, daher ist dieser Schritt oft wichtig.

Nun suche ich immer die „äußerste“ Funktion, in diesem Fall ist das der „Punkt“. Wir kennen ja von der Modulo-Funktion „mod“, dass sie auch „infix“ geschrieben werden kann, also x `mod` 2 == mod x 2. Genauso ist es mit dem Verknupfungsoperator, dem Punkt, auch:

Anstatt Infix-Notation können wir die geg. Funktion also auch in Präfix-Notation schreiben:

((.) map) (,) Wichtig: Das heisst jetzt nicht, dass map infix steht.

Also können wir jetzt so denken: (.) nimmt ein Argument map und gibt eine Funktion zurück, die ein Argument (,) nimmt. Wir sehen beim Typen von (.):

(.) :: (b->c)->(a->b)->a->c == (b->c) -> ((a->b)->(a-c)) (gem. Assoziativität)

Das Argument von (.) muss also vom Typen (b->c) sein. Für map gilt:

map :: (e->f)->[e]->[f] == (e->f) -> ([e]->[f]) (gem. Assoziativität)

Also wissen wir, da hier map als Argument an (.) übergeben wird und das Argument von (.) den Typen (b->c) hat, dass:

b :: (e->f) und **c :: ([e]->[f])**

Nun wissen wir, dass (.) einen Rückgabewert vom Typen **((a->b)->(a->c))** hat. Wir wissen, dass diesem Rückgabewert (der eine Funktion ist, die eine Funktion als Argument nimmt und eine Funktion zurückgibt) die Funktion **(,) :: x->y->(x,y) == x -> (y->(x,y))** übergeben wird. Also gilt nach dem gleichen Prinzip wie eben bei map:

a :: x und **b :: y -> (x,y)**

Wenn wir das nun mit unseren vorherigen Erkentnissen vereinen (wir wussten schon, dass **b :: (e->f)**, aber nichts spezifisches über e,f), erhalten wir dazu:

e :: y und **f :: (x,y)**

Nun ist der „finale“ Rückgabewert dieser Funktion, die der Rückgabewert von (.) mit Argument map war, halt nun mit Argument (,): **(a->c)**

Und wir wissen, was a und c für Typen sind, und zwar:

a :: x und **c :: ([e] -> [f]) == ([y] -> [(x,y)])** also: **map . (,) :: x -> ([y] -> [(x,y)])**

Zum Verständnis hier nun noch weitere einfachere Aufgaben teilweise ausführlich erklärt:

```
Weitere Beispiele:

\x -> \y -> x + y

Können wir umschreiben als:
\x y -> x + y

Wir wissen:
(+) :: (Num a) => a -> a -> a

Also folgt:
x :: (Num a) => a
y :: (Num a) => a

Also:
\x -> \y -> x + y :: (Num a) => a -> a -> a
------------------------------------------------------------

filter (<3)

Wir wissen:
filter :: (a -> Bool) -> [a] -> [a]
(<) :: (Ord a) => a -> a -> Bool
3 :: (Num a) => a

(Das würde man aus Zeitgründen normalerweise hier bei so einer einfachen Aufgabe
einfach weglassen, aber soll hier der Vollsätndigkeit halber dennoch getan werden):
Wir benennen die Variablen um, damit wir leichter "matchen" können:

filter :: (a -> Bool) -> [a] -> [a]
(<) :: (Ord b) => b -> b -> Bool
3 :: (Num c) => c

Nun fangen wir mit der "äussersten" Funktion an:

filter nimmnt als Argument eine Funktion vom Typen (a->Bool) und gibt eine Funktion
vom Typen [a] -> [a] zurück. An Filter übergeben wir das Argument (<3), das wir uns
nun noch genauer angucken werden:

(<) nimmt ein Argument vom Typen a (es muss einer geordenten Menge entspringen)
und gibt eine Funktion zurück, die ein Argument vom gleichen Typen a nimmt und ein
Bool zurückgibt. An (<) übergeben wir das Argument 3, vom Typen (Num a) => a, also:

(<3) :: (Ord c, Num c) => c -> Bool

Wenn wir das mit dem Typen des Arguments von Filter "matchen", sehen wir:
a :: (Num c) => c   Bool :: Bool

Also:
filter (<3) :: (Ord c, Num c) => [c] -> [c]
```

```
Gegeben:

head    :: [a] -> a                                    1       :: Num a => a
tail    :: [a] -> [a]                                  100     :: Num a => a
map     :: (a -> b) -> [a] -> [b]                      (+)     :: Num a => a -> a -> a
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]          (<)     :: Ord a => a -> a -> Bool


-------------------------------------------------------------------------------------------
:t (\x -> (head x) True)  ???

Wir wissen:                                 Also:
head :: [a] -> a                            x :: [a]

Ausserdem:                                  Also:
(head x) :: (Bool -> b)                     x :: [(Bool -> b)]

Also insgesamt:
(\x -> (head x) True) :: [(Bool -> b)] -> b  (ist das gleiche wie [Bool -> b] -> b)
-------------------------------------------------------------------------------------------
:t (\x y -> map (y<) x)  ???

Wir wisen:                                  Also:
map :: (a -> b) -> [a] -> [b]               x :: [a]

Ausserdem:                                  Also:
(<) :: Ord a => a -> a -> Bool              (y<) :: (Ord a) => a -> Bool

Wir sehen nun:
y :: Ord a => a
a :: Ord a => a
b :: Bool

Also insgesamt:
(\x y -> map (y<) x) :: Ord a => [a] -> a -> [Bool]
-------------------------------------------------------------------------------------------
:t (\x y -> head (x map) + 100)  ???

Wir wissen:                                 Also:
(+) :: Num a => a -> a -> a                 head (x map) :: Num a => a   (100 ist trivial)

Ausserdem:                                  Also:
head :: [a] -> a                            x map :: Num a => [a]

Daraus folgt:
x :: Num a => ((c -> d) -> [c] -> [d]) -> [a]

Also insgesamt:
(\x y -> head (x map) + 100) :: Num a => (((c -> d) -> [c] -> [d]) -> [a]) -> e -> a
-------------------------------------------------------------------------------------------
:t (\x -> x (<) + 1)  ???

Wir wissen:                                 Also:
(+) :: Num a => a -> a -> a                 x (<) :: Num a => a      (1 ist trivial)

Daraus folgt:
x :: Num a, Ord b => (b -> b -> Bool) -> a

Also insgesamt:
(\x -> x (<) + 1) :: Num a, Ord b => ((b -> b -> Bool) -> a) -> a
-------------------------------------------------------------------------------------------

Überprüfe es gerne von Hand per Proof oder mit ghci, das hier stimmt ;)
```

Der (vermeintliche) Type-Inference Endgegner: "(.) . (.)":

(Wichtige) Anmerkung: Beim Types matchen sollte eigentlich "=" stehen, statt "::".

Die schwierige Aufgabe: Most general type of (.) . (.)

Zunächst einmal markiere ich die 3 Funktionen farblich:

(.) . (.)

Wir wissen:

(.) :: (b -> c) -> (a -> b) -> a -> c

(.) :: (b -> c) -> (a -> b) -> a -> c

(.) :: (b -> c) -> (a -> b) -> a -> c

Zuerst "benenne" ich die Typen-Variablen um, damit es später weniger verwirrend wird:

(.) :: (b -> c) -> (a -> b) -> a -> c

(.) :: (n -> o) -> (m -> n) -> m -> o

(.) :: (y -> z) -> (x -> y) -> x -> z

Ausserdem wende ich die bekannten Assoziativitäts-Regeln an:

(.) :: (b -> c) -> ((a -> b) -> (a -> c))

(.) :: (n -> o) -> ((m -> n) -> (m -> o))

(.) :: (y -> z) -> ((x -> y) -> (x -> z))

(*Das wäre alles in einem einzigen Schritt gegangen.)

Jetzt gucken wir uns die geg. Funktion an: (.) . (.)

Anstatt Infix-Notation zu verwenden, können wir sie auch umschreiben (wie z.B. "x `mod` 3" ist das gleiche wie "mod x 3")

↑ Infix-Notation ↑ Prefix-Notation

Also:

((.) (.)) (.)

↑

Die "äusserste" Funktion.

$((.)\ (.))\ (.)$

Wir wissen: $(.)$ ist eine Funktion vom Typen:

$$(.) :: (b \to c) \to ((a \to b) \to (a \to c))$$

Sie nimmt als Argument also eine Funktion vom Typen $(b \to c)$ und gibt eine Funktion vom Typen $(a \to b) \to (a \to c)$ zurück. Als Argument wird ihr hier $(.)$ übergeben, mit

$$(.) :: (n \to o) \to ((m \to n) \to (m \to o))$$

Wir wissen, dass $(.)$ eine Funktion vom Type $(b \to c)$ als Argument nimmt, also folgt durch "matching":

$$b :: (n \to o)$$
$$c :: ((m \to n) \to (m \to o))$$

Wir wissen, dass der Rückgabewert von $(.)$ eine Funktion vom Typen $(a \to b) \to (a \to c)$ ist. Dieser Funktion übergeben wir das Argument $(.)$. Wir wissen: $(.) :: (y \to z) \to ((x \to y) \to (x \to z))$ also folgt (da das Argument der Funktion mit Typ $(a \to b) \to (a \to c)$ eine Funktion vom Typen $(a \to b)$ ist):

$$a :: (y \to z)$$
$$b :: ((x \to y) \to (x \to z))$$

Nun haben wir mehrere Dinge über den Type von b herausgefunden:

$$b :: (n \to o) \quad \text{und} \quad b :: ((x \to y) \to (x \to z))$$

Wenn wir das vereinen, erhalten wir also:

$$n :: (x \to y)$$
$$o :: (x \to z)$$

Fassen wir noch einmal zusammen: Die Funktion $(.)$ nimmt

eine Funktion als Argument und gibt als Rückgabewert eine Funktion zurück. Dieser Argument wurde gegeben, (.).
Die Funktion, die der Rückgabewert ist, ist eine Funktion vom Typen $(a \to b) \to (a \to c)$, die eine Funktion als Argument nimmt und eine Funktion zurückgibt. Dieses "zweite" Argument war auch gegeben, (.). Also ist alles was bleibt, der Rückgabewert hier, der den Typen $(a \to c)$ hat. Und mit unserer Analyse konnten wir bestimmen, was genau a und c sind.

$a :: (y \to z)$

$c :: (m \to n) \to (m \to o)$

$n :: (x \to y)$

$o :: (x \to z)$

Also: $c :: (m \to n) \to (m \to o) = (m \to (x \to y)) \to (m \to (x \to z))$

Also zusammen:

$(.).(.) :: a \to c = (y \to z) \to (m \to (x \to y)) \to (m \to (x \to z))$

Das können wir nun durch Umbenennen der Variablen noch "verschönern":

$(.).(.) :: (t2 \to t3) \to (t0 \to t1 \to t2) \to t0 \to t1 \to t3$

(Ich habe Klammern weggelassen, die man eigentlich nicht braucht

Type-Inference Proofs:

Um den letzten Abschnitt förmlich zu ergänzen, haben wir Type-Inference Proofs. Denn oftmals ist es schwierig, zu „sehen“, ob ein Type wirklich stimmt (oder ob eine Funktion überhaupt typeable ist). Dazu haben wir ein Proof-System, das so derart nah an ND-Proofs dran ist, dass euch spätestens jetzt klar sein sollte, weswegen die ND-Proofs eine sehr gute Übung waren:

Hier die Regeln (Quelle: Serie 5):

$$\frac{}{\ldots, x : \tau, \ldots \vdash x :: \tau}\ \textit{Var} \qquad \frac{\Gamma, x : \sigma \vdash t :: \tau}{\Gamma \vdash \lambda x.\, t :: \sigma \to \tau}\ \textit{Abs}$$

$$\frac{\Gamma \vdash t_1 :: \sigma \to \tau \qquad \Gamma \vdash t_2 :: \sigma}{\Gamma \vdash t_1\ t_2 :: \tau}\ \textit{App} \qquad \frac{\Gamma \vdash t :: Int}{\Gamma \vdash \textbf{iszero}\ t :: Bool}\ \textit{iszero}$$

$$\frac{}{\Gamma \vdash n :: Int}\ \textit{Int} \qquad \frac{}{\Gamma \vdash True :: Bool}\ \textit{True} \qquad \frac{}{\Gamma \vdash False :: Bool}\ \textit{False}$$

$$\frac{\Gamma \vdash t_1 :: Int \qquad \Gamma \vdash t_2 :: Int}{\Gamma \vdash (t_1\ \textbf{op}\ t_2) :: Int}\ \textit{BinOp} \qquad \text{for } \textbf{op} \in \{+, *\}$$

$$\frac{\Gamma \vdash t_0 :: Bool \qquad \Gamma \vdash t_1 :: \tau \qquad \Gamma \vdash t_2 :: \tau}{\Gamma \vdash \textbf{if}\ t_0\ \textbf{then}\ t_1\ \textbf{else}\ t_2 :: \tau}\ \textit{if}$$

$$\frac{\Gamma \vdash t_1 :: \tau_1 \qquad \Gamma \vdash t_2 :: \tau_2}{\Gamma \vdash (t_1, t_2) :: (\tau_1, \tau_2)}\ \textit{Tuple} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash \textbf{fst}\ t :: \tau_1}\ \textit{fst} \qquad \frac{\Gamma \vdash t :: (\tau_1, \tau_2)}{\Gamma \vdash \textbf{snd}\ t :: \tau_2}\ \textit{snd}$$

Die Regeln sollten fast alle klar verständlich sein (falls nein, schreibt mir). Die App-Regel ist die einzige, die eventuell weiterer Erklärung bedarf: Wenn aus unseren Annahmen folgt, dass die Funktion t_1 vom Type $\sigma \to \tau$ ist und aus unseren Annahmen folgt, dass das Argument t_2 vom Type σ ist, dann können wir herleiten, dass für die „Function Application“ (Deshalb „App“) gilt, dass der Rückgabewert dieser Function Application vom Type τ ist.

Hier nun eine Aufgabe: Man solle verifizieren, ob der gegebene Type korrekt ist. Dabei verwenden wir auch den Unification-Algorithmus (siehe Link auf Website).

Weiter auf der nächsten Seite →

(Final Exam FS14): Formally infer the type of "$\lambda x.\ x\ (\text{snd}\ (x\ 0))$"

$$\dfrac{\dfrac{}{x:t_1 \vdash x :: \text{Int} \to (t_4, t_3)}\ \text{Var},(3) \qquad \dfrac{}{x:t_1 \vdash 0 :: \text{Int}}\ \text{Int}}{x:t_1 \vdash x\ 0 :: (t_4, t_3)}\ \text{App}$$

$$\dfrac{\dfrac{}{x:t_1 \vdash x :: t_3 \to t_2}\ \text{Var},(2) \qquad \dfrac{x:t_1 \vdash x\ 0 :: (t_4, t_3)}{x:t_1 \vdash \text{snd}\ (x\ 0) :: t_3}\ \text{snd}}{x:t_1 \vdash x\ (\text{snd}\ (x\ 0)) :: t_2}\ \text{App}$$

$$\dfrac{x:t_1 \vdash x\ (\text{snd}\ (x\ 0)) :: t_2}{\vdash \lambda x.\ x\ (\text{snd}\ (x\ 0)) :: t_0}\ \text{Abs}^*,(1)$$

$^*\ x \notin \Gamma$

(1) $t_0 = t_1 \to t_2$

(2) $t_1 = t_3 \to t_2$ $\left.\begin{matrix} t_3 = \text{Int} \\ t_2 = (t_4, t_3) \end{matrix}\right\}$ $t_3 = \text{Int}$, $t_2 = (t_4, \text{Int})$

(3) $t_1 = \text{Int} \to (t_4, t_3)$

Also: $t_0 = t_1 \to t_2$, $t_1 = \text{Int} \to (t_4, \text{Int})$, $t_2 = (t_4, \text{Int})$, $t_3 = \text{Int}$

$\Rightarrow t_0 = t_1 \to t_2 = (\text{Int} \to (t_4, \text{Int})) \to (t_4, \text{Int})$

Also: $\lambda x.\ x\ (\text{snd}\ (x\ 0)) :: (\text{Int} \to (t_4, \text{Int})) \to (t_4, \text{Int})$

(Final Exam FS16): Formally infer type of

"$(\lambda y.\ \lambda x.\ \text{iszero}\ (y\ x))\ (\lambda x.\ \text{snd}\ x)$"

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{y:t_1, x:t_6 \vdash y :: t_6 \to \text{Int}}\ \text{Var},(5) \qquad \dfrac{}{y:t_1, x:t_6 \vdash x :: t_6}\ \text{Var}}{y:t_1, x:t_6 \vdash y\ x :: \text{Int}}\ \text{App}}{y:t_1, x:t_6 \vdash \text{iszero}\ (y\ x) :: t_7}\ \text{iszero},(4)}{y:t_1 \vdash \lambda x.\ \text{iszero}\ (y\ x) :: t_0}\ \text{Abs}^{*3},(3)}{\vdash (\lambda y.\ \lambda x.\ \text{iszero}\ (y\ x)) :: t_1 \to t_0}\ \text{Abs}^{*2} \qquad \dfrac{\dfrac{\dfrac{}{x:t_2 \vdash x :: (t_4, t_3)}\ \text{Var},(2)}{x:t_2 \vdash \text{snd}\ x :: t_3}\ \text{snd}}{\vdash (\lambda x.\ \text{snd}\ x) :: t_1}\ \text{Abs}^{*1},(1)$$

$$\vdash (\lambda y.\ \lambda x.\ \text{iszero}\ (y\ x))\ (\lambda x.\ \text{snd}\ x) :: t_0\ \text{App}$$

$^{*1}\ x \notin \Gamma$ $^{*2}\ y \notin \Gamma$ $^{*3}\ x \notin \Gamma$

Constraints: (1) $t_1 = t_2 \to t_3$
(2) $t_2 = (t_4, t_3)$
(3) $t_0 = t_6 \to t_7$
(4) $t_7 = \text{Bool}$
(5) $t_1 = t_6 \to \text{Int}$

Constraints: (1) $t_1 = t_2 \to t_3$
(2) $t_2 = (t_4, t_3)$
(3) $t_0 = t_6 \to t_7$
(4) $t_7 = \text{Bool}$
(5) $t_1 = t_6 \to \text{Int}$

Solvin the constraint system mit dem Algo:

Zuerst "mergen" wir so lange zeilen, bis auf der LHS alle Variablen unterschiedlich sind:

1. (1) und (5) "mergen":
   (1) $t_2 = t_6$
   (2) $t_3 = \text{Int}$
   (3) $t_2 = (t_4, t_3)$
   (4) $t_0 = t_6 \to t_7$
   (5) $t_7 = \text{Bool}$

2. (1) und (3) "mergen"
   (1) $t_6 = (t_4, t_3)$
   (2) $t_3 = \text{Int}$
   (3) $t_0 = t_6 \to t_7$
   (5) $t_7 = \text{Bool}$

3. Nun substituiren wir (2) in die anderen Gleichungen:
   (1) $t_6 = (t_4, \text{Int})$
   (2) $t_0 = t_6 \to t_7$
   (3) $t_7 = \text{Bool}$

4. Jetzt substituiren wir (3) in die anderen Gleichungen:
   (1) $t_6 = (t_4, \text{Int})$
   (2) $t_0 = t_6 \to \text{Bool}$

5. Nun substituiren wir (1) in die anderen Gleichungen:
   (1) $t_0 = (t_4, \text{Int}) \to \text{Bool}$

Also: $(\lambda y.\, \lambda x.\, \text{iszero}\ (y\, x))\ (\lambda x.\, \text{snd}\ x) :: (t_4, \text{Int}) \to \text{Bool}$