RWTH Aachen
Lehrgebiet Theoretische Informatik
Frohn–Reidl–Rossmanith–Sánchez–Ströder

WS 2013/14
Übungsblatt 11
20.01.2014

Übung zur Vorlesung Programmierung

Aufgabe T22

Aus Übungsblatt 10 kennen Sie bereits die Funktionen `minus10`, `listAdd` und `jedesZweite`. Reimplementieren Sie diese mit Hilfe von Funktionen höherer Ordnung, ohne explizit Rekursion zu verwenden!

Implementieren Sie darüber hinaus folgende Funktionen:

- `revDiv :: [Int] -> Int`
  Dividiert die Elemente der gegebenen Liste von rechts nach links. Um Division durch Null zu vermeiden, werden Nullen dabei ignoriert. Falls die gegebene Liste leer ist, darf sich die Funktion beliebig verhalten.
  Beispiel: `revDiv [2,0,3,12] = 2`, da $12/3/2 = 2$ gilt.
- `factorize :: [Int] -> [Int]`
  Entfernt alle Primzahlen aus der gegebenen Liste und teilt jedes Element der Liste, das keine Primzahl ist, durch einen seiner Primfaktoren.
  Beispiel: `factorize [2,3,4,5,6,7,8,9]` $\in \{$`[2,3,4,3],[2,2,4,3]`$\}$

Verzichten Sie dabei ebenfalls auf explizite Rekursion und nutzen Sie stattdessen Funktionen höherer Ordnung!

Hinweise zur API:

- `foldl :: (a -> b -> a) -> a -> [b] -> a`:
  Der Aufruf `foldl f a [b1,b2,...,bn]` entspricht dem Aufruf `f (... (f (f (f a b1) b2) b3 )... ) bn`, d.h. `f` wird sukzessive auf die Element der Liste Aufgerufen, wobei das Ergebnis des vorherigen Aufrufs als erster Parameter übergeben wird.
- `map :: (a -> b) -> [a] -> [b]`:
  Der Aufruf `map f [a1,a2,...]` ergibt die Liste `[f a1,f a2,...]`.
- `filter :: (a -> Bool) -> [a] -> [a]`:
  Der Aufruf `filter f [a1,a2,...]` liefert eine Liste derjenigen Elemente `ai`, für welche `f ai` wahr ist.
- `init :: [a] -> [a]`
  Die Funktion `init` liefert eine Liste ohne ihr letztes Element zurück.
- `last :: [a] -> a`
  Die Funktion `last` liefert das letzte Element einer Liste zurück.

- `zip :: [a] -> [b] -> [(a,b)]`
  Die Funktion zip bildet aus zwei Listen eine Liste von Paaren. Wenn eine Liste länger ist als die Andere, werden überschüssige Elemente ignoriert.
  Beispiel: `zip [1,2,3] [2,3,4,5] = [(1,2),(2,3),(3,4)]`.
- `foldr :: (a -> b -> b) -> b -> [a] -> b`
  Die Funktion `foldr` arbeitet wie `foldl`, arbeitet die Liste aber von rechts nach links statt von links nach rechts ab.
- Die Funktionen `‘div‘` und `‘mod‘` können zur Integer-Division bzw. zur Berechnung des Rests verwendet werden.

Lösungsvorschlag

```
minus10 :: [Int] -> [Int]
minus10 x = map (\a -> a - 10) x

listAdd :: Int -> [Int] -> [Int]
listAdd i x = map (\(a,b) -> a + b) (zip x (i:init x))

jedesZweite :: [Int] -> [Int]
jedesZweite x = map (\(a,_) -> a)
  (filter (\(_,b) -> (b ‘mod‘ 2) /= 0) (zip x [1..]))

revDiv :: [Int] -> Int
revDiv x = let filtered = filter (\a -> a /= 0) x in
           foldr (\a b -> b ‘div‘ a) (last filtered) (init filtered)

factorize :: [Int] -> [Int]
factorize x =
  let f i = foldr (\a b -> if (i ‘mod‘ a == 0) then a else b) i [2..i-1]
      x’ = map (\a -> a ‘div‘ f a) x in
  filter (\a -> a /= 1) x’
```

Aufgabe T23

Binäre geordnete Bäume seinen, wie in der Vorlesung, folgendermaßen definiert:
data *OrdBintree a* = *Ext*|*In a* (*OrdBintree a*) (*OrdBintree a*)
deriving *Show*

Schreiben sie eine Funktion mit der gegebenen Signatur, die genau dann *True* zurückgibt, wenn die beiden gegebenen Bäume gleich sind. Andernfalls soll die Funktion *False* zurückgeben.

binTreeEqual::*Eq a* => *OrdBintree a*→*OrdBintree a*→*Bool*

Lösungsvorschlag

binTreeEqual::*Eq a* => *OrdBintree a*→*OrdBintree a*→*Bool*
binTreeEqual Ext Ext = *True*
binTreeEqual (*In v1 l1 r1*) (*In v2 l2 r2*) = *v1* == *v2* && *binTreeEqual l1 l2*

$\&\&\ binTreeEqual\ r1\ r2$

$binTreeEqual\ \_\ \_ = False$

Aufgabe T24

Implementieren sie die folgenden *fold*-Operationen für binäre Bäume:

$foldOrdBinTreeC::(a \to b \to a) \to a \to OrdBintree\ b \to a$
$foldOrdBinTreeL::(a \to b \to a) \to a \to OrdBintree\ b \to a$
$foldOrdBinTreeR::(a \to b \to a) \to a \to OrdBintree\ b \to a$

Diese drei Funktionen sollen auch die im ersten Argument gegeben Funktion f nacheinander auf die Elemente des Baumes anwenden, jedoch in unterschiedlicher Reihenfolge:

- *foldOrdBinTreeC f a t*: Zu Beginn wird f mit den Argumenten a und der Wurzel von t ausgewertet. Das Ergebnis *res* wird dann verwendet, um rekursiv erst den linken und dann den rechten Teilbaum auszuwerten.
- *foldOrdBinTreeL*: Zu Beginn wird f mit dem Argument a rekursiv auf den linken Teilbaum angewendet. Das Ergebnis *res* wird dann verwendet, um f mit dem Wert der Wurzel von t auszuwerten. Das Ergebnis wird wiederum auf den rechten Teilbaum angewendet.
- *foldOrdBinTreeR*: Zu Beginn wird f mit dem Argument a rekursiv auf den rechten Teilbaum angewendet. Das Ergebnis *res* wird dann verwendet, um f mit dem Wert der Wurzel von t auszuwerten. Das Ergebnis wird wiederum auf den linken Teilbaum angewendet.

Benutzen Sie diese Methoden, um eine Funktion $treetolist::OrdBintree\ a \to [a]$ zu schreiben, die aus einem geordneten Baum eine geordnete Liste erzeugt. Welche der drei Funktionen bietet sich dazu an?

Lösungsvorschlag

$foldOrdBinTreeC::(a \to b \to a) \to a \to OrdBintree\ b \to a$
$foldOrdBinTreeC\ \_\ x\ Ext = x$
$foldOrdBinTreeC\ f\ v\ (In\ x\ l\ r) = foldOrdBinTreeC\ f\ center\ r$
  where $center = foldOrdBinTreeC\ f\ (f\ v\ x)\ l$

$foldOrdBinTreeL::(a \to b \to a) \to a \to OrdBintree\ b \to a$
$foldOrdBinTreeL\ \_\ x\ Ext = x$
$foldOrdBinTreeL\ f\ v\ (In\ x\ l\ r) = foldOrdBinTreeL\ f\ (f\ left\ x)\ r$
  where $left = foldOrdBinTreeL\ f\ v\ l$

$foldOrdBinTreeR::(a \to b \to a) \to a \to OrdBintree\ b \to a$
$foldOrdBinTreeR\ \_\ x\ Ext = x$
$foldOrdBinTreeR\ f\ v\ (In\ x\ l\ r) = foldOrdBinTreeR\ f\ (f\ right\ x)\ l$
  where $right = foldOrdBinTreeR\ f\ v\ r$

$treetolist :: OrdBintree\ a \rightarrow [a]$
$treetolist = foldOrdBinTreeR\ (\backslash x\ y \rightarrow y : x)\ [\,]$

$treetolist' :: OrdBintree\ a \rightarrow [a]$
$treetolist' = foldOrdBinTreeL\ (\backslash x\ y \rightarrow x ++ [y])\ [\,]$

Aufgabe T25

In dieser Aufgabe soll eine Transformation von logischen Formeln in Konjunktive Normalform (KNF) in Haskell implementiert werden. Die KNF ist nützlich, um die Erfüllbarkeit logischer Formeln („Ist es möglich, die Variablen einer Formel so zu belegen, dass die Formel wahr ist?“) in der Praxis effizient überprüfen zu können (sogenanntes SAT-Solving).

a) Entwerfen Sie mit `data` eine geeignete Datenstruktur für logische Formeln, welche die logischen Operatoren Und, Oder, Nicht und Gdw (genau dann, wenn) unterstützt. Außerdem sollen die logischen Konstanten Wahr und Falsch sowie Variablen unterstützt werden. Letztere sollen darüber hinaus einen Int Wert als Identifikator kapseln, um verschiedene Variablen voneinander unterscheiden zu können.

b) Implementieren Sie aufbauend auf der soeben entworfenen Datenstruktur eine Funktion `knf`, welche eine beliebige logische Formel, die mit Ihrer Datenstruktur darstellbar ist, in KNF überführt. Eine Formel ist in KNF gdw. innerhalb von Disjunktionen keine Konjunktionen vorkommen und Negationen ausschließlich auf Variablen angewendet werden. Um eine Formel in KNF zu überführen, sind die De Morganschen Gesetze ($\neg(\phi \wedge \psi) = \neg\phi \vee \neg\psi$ und $\neg(\phi \vee \psi) = \neg\phi \wedge \neg\psi$) und das Distributivgesetz ($\phi \vee (\psi \wedge \chi) = (\phi \vee \psi) \wedge (\phi \vee \chi)$) hilfreich. Geben Sie auch den Typ aller Funktionen an, die Sie implementieren (außer bei Lambda-Ausdrücken).

Lösungsvorschlag

```
data Formel = Und Formel Formel | Oder Formel Formel | Nicht Formel
           | Wahr | Falsch | Variable Int | Gdw Formel Formel
              deriving Show

knf :: Formel -> Formel
knf Wahr = Wahr
knf Falsch = Falsch
knf (Variable x) = Variable x
knf (Nicht Wahr) = Falsch
knf (Nicht Falsch) = Wahr
knf (Nicht (Variable x)) = Nicht (Variable x)
knf (Nicht (Und x y)) = distribute (knf (Nicht x)) (knf (Nicht y))
knf (Nicht (Oder x y)) = Und (knf (Nicht x)) (knf (Nicht y))
knf (Nicht (Nicht x)) = knf x
knf (Nicht (Gdw x y)) = knf (Oder (Und x (Nicht y)) (Und (Nicht x)
   y))
knf (Und x y) = Und (knf x) (knf y)
knf (Oder x y) = distribute (knf x) (knf y)
knf (Gdw x y) = knf (Oder (Und x y) (Und (Nicht x) (Nicht y)))
```

```
distribute :: Formel -> Formel -> Formel
distribute (Und x y) z = Und (distribute x z) (distribute y z)
distribute x (Und y z) = Und (distribute x y) (distribute x z)
distribute x y = Oder x y
```

Aufgabe H28 (1+2+2+2+2+3=12 Punkte)

Aus Übungsblatt 10 kennen Sie bereits die Funktionen `produkt`, `cleanEmpty`, `kleinstes` und `plaetten`. Reimplementieren Sie diese mit Hilfe von Funktionen höherer Ordnung, ohne explizit Rekursion zu verwenden! Verwenden Sie dabei *nicht* die vordefinierten Funktionen `product`, `minimum` und `concat`!

Implementieren Sie darüber hinaus folgende Funktionen:

- `smallSum :: [[Int]] -> Int -> [[Int]]`
  Entfernt aus der gegebenen Liste von Listen von Integern (erstes Argument) alle Element, deren Summe nicht kleiner als das Zweite Argument ist.
  Beispiel: `smallSum [[2,3],[],[2,1],[7]] 4 = [[],[2,1]]`
- `divLists :: [Int] -> [Int] -> [Int]`
  Dividiert die Elemente der ersten Liste durch die Elemente der zweiten Liste. Falls die erste Liste kürzer ist als die zweite, wird für die fehlenden Elemente der ersten Liste der Wert 0 angenommen. Falls die zweite Liste kürzer ist als die erste, wird für die fehlenden Elemente der Wert 1 angenommen.
  Beispiel: `divLists [9,8,6,34,12] [4,1,3] = [2,8,2,34,12]`

Verzichten Sie dabei ebenfalls auf explizite Rekursion und nutzen Sie stattdessen Funktionen höherer Ordnung!

Hinweis: Haskell bietet Ihnen viele nützliche vordefinierte Funktionen an (siehe `http://hackage.haskell.org/package/base-4.6.0.1/docs/Data-List.html`), die Sie, mit Ausnahme der in der Aufgabenstellung explizit verbotenen Funktionen, verwenden können, um sich das Lösen der Aufgaben zu vereinfachen.

Lösungsvorschlag

```
produkt :: [Int] -> Int
produkt x = foldl (*) 1 x

cleanEmpty :: [[Int]] -> [[Int]]
cleanEmpty x = filter (not . null) x

kleinstes :: [Int] -> Int
kleinstes x = foldl (\a b -> if (a < b) then a else b) (head x) x

plaetten :: [[Int]] -> [Int]
plaetten x = foldl (\a b -> a ++ b) [] x

smallSum :: [[Int]] -> Int -> [[Int]]
smallSum x i = filter (\a -> foldl (+) 0 a < i) x

divLists :: [Int] -> [Int] -> [Int]
divLists x y = let (x',y') = if (length x < length y)
                             then (x ++ [0,0..],y)
```

```
                              else (x,y ++ [1,1..]) in
  map (\(a,b) -> a `div` b) (zip x' y')
```

Aufgabe H29 (5 Punkte)
Schreiben Sie eine *map*-Funktion für *OrdBinTree*.

$mapOrdBinTree :: (a \rightarrow b) \rightarrow OrdBintree\ a \rightarrow OrdBintree\ b$

Das Ergebnis des Aufrufes $mapOrdBinTree\ f\ t$ soll ein Binärbaum sein, der die gleiche Struktur wie t hat, in welchem jedoch jedes Element a durch $f\ a$ ersetzt wurde.

Lösungsvorschlag
$mapOrdBinTree :: (a \rightarrow b) \rightarrow OrdBintree\ a \rightarrow OrdBintree\ b$
$mapOrdBinTree\ \_\ Ext = Ext$
$mapOrdBinTree\ f\ (In\ v\ r\ l) = In\ (f\ v)\ (mapOrdBinTree\ f\ l)\ (mapOrdBinTree\ f\ r)$

Aufgabe H30 (10 Punkte)
Schreiben Sie eine Funktion $deleteFromOrdBinTree :: Int \rightarrow OrdBintree\ Int \rightarrow OrdBintree\ Int$, welche Werte aus geordneten Binärbäumen löscht. Das Entfernen eines Wertes aus einem Binärbaum funktioniert dabei (wie schon behandelt) wie folgt:

- Ist x in einem Blattknoten enthalten, hat also weder einen linken noch einen rechten Nachfolger, so entfernen wir schlicht dieses Blatt aus dem Baum.
- Hat der Knoten mit Inhalt x einen linken Nachfolger, so bestimmen wir das maximale Element y in diesem linken Teilbaum, entfernen es aus diesem und tauschen x gegen y im aktuellen Knoten aus.
- Hat der Knoten mit Inhalt x nur einen rechten Nachfolger, so bestimmen wir das minimale Element z in diesem rechten Teilbaum, entfernen es aus diesem und tauschen x gegen z im aktuellen Knoten aus.

Folgendes sind Beispiele für Aufrufe der Funktion:

- $deleteFromOrdBinTree\ 6 (In\ 5 (In\ 3\ Ext\ (In\ 4\ Ext\ Ext))\ (In\ 7 (In\ 6\ Ext\ Ext)\ (In\ 8\ Ext\ Ext))) = In\ 5 (In\ 3\ Ext\ (In\ 4\ Ext\ Ext))\ (In\ 7\ Ext\ (In\ 8\ Ext\ Ext))$
- $deleteFromOrdBinTree\ 5 (In\ 5 (In\ 3\ Ext\ (In\ 4\ Ext\ Ext))\ (In\ 7 (In\ 6\ Ext\ Ext)\ (In\ 8\ Ext\ Ext))) = In\ 4 (In\ 3\ Ext\ Ext)\ (In\ 7 (In\ 6\ Ext\ Ext)\ (In\ 8\ Ext\ Ext))$
- $deleteFromOrdBinTree\ 3 (In\ 5 (In\ 3\ Ext\ (In\ 4\ Ext\ Ext))\ (In\ 7 (In\ 6\ Ext\ Ext)\ (In\ 8\ Ext\ Ext))) = In\ 5 (In\ 4\ Ext\ Ext)\ (In\ 7 (In\ 6\ Ext\ Ext)\ (In\ 8\ Ext\ Ext))$

Lösungsvorschlag

```
deleteFromOrdBinTree::Ord a => a→OrdBintree a→OrdBintree a
deleteFromOrdBinTree _ Ext = Ext
deleteFromOrdBinTree x (In v Ext Ext) =
    if (x == v)
    then Ext
    else (In v Ext Ext)
deleteFromOrdBinTree x (In v l r) =
    if (x < v)
    then In v (deleteFromOrdBinTree x l) r
    else if (x > v)
         then In v l (deleteFromOrdBinTree x r)
         else In (takeOne l) (changedL l) (changedR l)
    where takeOne Ext = minimumInTree r
          takeOne _ = maximumInTree l
          changedL Ext = l
          changedL _ = deleteFromOrdBinTree (takeOne l) l
          changedR Ext = deleteFromOrdBinTree (takeOne l) r
          changedR _ = r


minimumInTree::Ord a => OrdBintree a→a
minimumInTree t = foldOrdBinTreeR (\x y→if x < y then x else y) (headOrdBinTree t) t

maximumInTree::Ord a => OrdBintree a→a
maximumInTree t = foldOrdBinTreeR (\x y→if x > y then x else y) (headOrdBinTree t) t
```

Aufgabe H31 (10 Punkte)[1]

Betrachten Sie erneut eine Datenstruktur für logische Formeln in Haskell genau wie in Tutoraufgabe 25 (sie dürfen diese Datenstruktur wiederverwenden). Die Transformation solcher Formeln in KNF mittels der De Morganschen Gesetze und des Distributivgesetzes liefert zwar das gewünschte Ergebnis, aber ihre Laufzeit steigt im schlimmsten Fall exponentiell mit der Größe der Eingabeformel an. Somit würde man also für das SAT-Solving nichts gewinnen, da die effizientere Erfüllbarkeitsprüfung erst nach der teuren Transformation durchgeführt werden kann. Stattdessen wird in der Praxis eine andere Transformation genutzt, die von Grigorii Samuilovich Tseitin veröffentlicht wurde (Tseitin-Transformation). Dabei versucht man nicht, eine vollständig äquivalente Formel in KNF zu berechnen, sondern lediglich eine *erfüllbarkeitsäquivalente*. D. h. die transformierte Formel ist erfüllbar gdw. die ursprüngliche Formel erfüllbar ist. Statt im schlimmsten Fall exponentiell große transformierte Formeln zu erhalten, wachsen die Formeln bei der Tseitin-Transformation höchstens linear in der Größe der Eingabeformeln. Die Idee der Tseitin-Transformation ist die folgende:

Für jede logische Teilformel wird eine neue Variable erzeugt. Diese wird in Äquivalenzbeziehung zur Teilformel gesetzt, während die Teilformel in der restlichen Formel durch die neue Variable ersetzt wird. Dies wird solange wiederholt, bis wir nur noch eine Konjunktion von Äquivalenzen und Variablen haben (prinzipiell könnte man auch bei Negationen von Variablen oder Konstanten bereits halt machen, aber um das Verfahren möglichst

einfach zu halten, verzichten wir auf diese Optimierungen). Die Äquivalenzen enthalten aber höchstens drei Variablen, sodass sie unabhängig voneinander mit dem normalen Verfahren aus Tutoraufgabe 25 in KNF überführt werden können, ohne dass die asymptotische Laufzeit darunter leidet. Am Ende wird die neue Variable, die für die gesamte Formel steht, mit der restlichen Formel in Konjunktion gesetzt, um zu erzwingen, dass die Formel erfüllt wird. Das folgende Beispiel soll die Tseitin-Transformation illustrieren. Wir starten mit der Formel $X_1 \vee \neg(X_2 \wedge \neg X_3)$. Nun wird zunächst eine neue Variable für die äußerste Operation hinzugefügt: $X_4 \leftrightarrow X_1 \vee \neg(X_2 \wedge \neg X_3)$. Anschließend werden die Teilformeln (also die Argumente der Operation) betrachtet. Hier sind dies X_1 und $\neg(X_2 \wedge \neg X_3)$. X_1 ist bereits lediglich eine Variable, sodass wir hier fertig sind. Für das andere Argument wird wieder eine neue Variable eingeführt und wir erhalten $X_5 \leftrightarrow \neg(X_2 \wedge \neg X_3)$, wobei wir diese Teilformel in der ersten Äquivalenz durch die neue Variable ersetzen. Diese lautet nun also $X_4 \leftrightarrow X_1 \vee X_5$. Wir machen mit dem Argument der letzten Teilformel weiter. Dies ist $X_2 \wedge \neg X_3$. Wieder wird eine neue Variable eingeführt und die Teilformel in der übrigen Formel ersetzt, sodass wir $X_6 \leftrightarrow X_2 \wedge \neg X_3$ und $X_5 \leftrightarrow \neg X_6$ haben. Die Argumente der ersetzten Operation sind X_2 und $\neg X_3$. Das erste davon ist wiederum eine Variable, sodass wir nur für das zweite eine neue Variable einführen müssen. Damit erhalten wir als Zwischenstand insgesamt die Formel $(X_4 \leftrightarrow X_1 \vee X_5) \wedge (X_5 \leftrightarrow \neg X_6) \wedge (X_6 \leftrightarrow X_2 \wedge X_7) \wedge (X_7 \leftrightarrow \neg X_3)$. Schließlich wenden wir die normale Transformation in KNF auf die Äquivalenzen an, setzen das Ergebnis mit der ersten Hilfsvariable X_4 in Konjunktion und erhalten die folgende erfüllbarkeitsäquivalente Formel in KNF:
$((((((X_4 \vee \neg X_4) \wedge ((X_4 \vee \neg X_5) \wedge (X_4 \vee \neg X_1))) \wedge (((X_5 \vee X_1) \vee \neg X_4) \wedge (((X_5 \vee X_1) \vee \neg X_5) \wedge ((X_5 \vee X_1) \vee \neg X_1)))) \wedge X_1) \wedge ((((X_5 \vee \neg X_5) \wedge (X_5 \vee X_6)) \wedge ((\neg X_6 \vee \neg X_5) \wedge (\neg X_6 \vee X_6))) \wedge (((((X_6 \vee \neg X_6) \wedge (X_6 \vee (\neg X_7 \vee \neg X_2))) \wedge (((X_7 \vee \neg X_6) \wedge (X_7 \vee (\neg X_7 \vee \neg X_2))) \wedge ((X_2 \vee \neg X_6) \wedge (X_2 \vee (\neg X_7 \vee \neg X_2))))) \wedge X_2) \wedge ((((X_7 \vee \neg X_7) \wedge (X_7 \vee X_3)) \wedge ((\neg X_3 \vee \neg X_7) \wedge (\neg X_3 \vee X_3))) \wedge X_3)))) \wedge X_4)$
Implementieren Sie eine Funktion `tseitin`, welche eine Formel mittels der Tseitin-Transformation in eine erfüllbarkeitsäquivalente Formel in KNF transformiert. Geben Sie bei allen von Ihnen implementierten Funktionen (außer Lambda-Ausdrücken) auch deren Typ an.
Hinweise:

- Um das Erzeugen neuer Variablen zu vereinfachen, dürfen Sie annehmen, dass alle Identifikatoren in der Eingabeformel negativ sind. Somit können Sie frische Variablen durch einen positiven Zähler generieren.
- Um diesen Zähler als Argument mitführen zu können, ist es ratsam, eine Hilfsfunktion mit eben diesem zusätzlichen Argument zu verwenden.
- Um die Ersetzung der Teilformeln durch Variablen zu erleichtern, sollte die Hilfsfunktion Tripel zurückliefern: Die erste Komponente ist die transformierte Teilformel, die zweite Komponente ist die Variable, durch die diese Teilformel in der sonstigen Formel ersetzt wird, und die dritte Komponente ist der Zähler nach Transformation der Teilformel (achten Sie darauf, nicht den gleichen Zählerwert in der Rekursion für verschiedene Argumente zu verwenden).
- Sie dürfen die Datenstruktur und die Funktion `knf` aus Tutoraufgabe 25 als gegeben annehmen.

Lösungsvorschlag

```
tseitin :: Formel -> Formel
tseitin x = (\(a,b,c) -> (Und a b)) (tseitinHelp x 0)

tseitinHelp :: Formel -> Int -> (Formel, Formel, Int)
tseitinHelp Wahr n = (knf (Gdw (Variable n) Wahr), Variable n, n +
   1)
tseitinHelp Falsch n = (knf (Gdw (Variable n) Falsch), Variable n, n
    + 1)
tseitinHelp (Variable x) n = (Variable x, Variable x, n)
tseitinHelp (Nicht x) n = (Und (knf (Gdw (Variable m) (Nicht b))) a,
    Variable m, m + 1)
   where (a,b,m) = tseitinHelp x n
tseitinHelp (Und x y) n = (Und (Und (knf (Gdw (Variable m) (Und d
   b))) a) c, Variable m, m + 1)
   where (a,b,k) = tseitinHelp x n
       (c,d,m) = tseitinHelp y k
tseitinHelp (Oder x y) n = (Und (Und (knf (Gdw (Variable m) (Oder d
   b))) a) c, Variable m, m + 1)
   where (a,b,k) = tseitinHelp x n
       (c,d,m) = tseitinHelp y k
tseitinHelp (Gdw x y) n = (Und (Und (knf (Gdw (Variable m) (Gdw d
   b))) a) c, Variable m, m + 1)
   where (a,b,k) = tseitinHelp x n
       (c,d,m) = tseitinHelp y k
```

Abgabe zum 28.01.2014

[1] Bitte Quelltext für die Abgabe ausdrucken und zusätzlich per E-mail an den jeweiligen Tutor senden.