Theorie der Programmierung SS 2015

Übungsblatt 3

Rev.9002

Abgabe der Lösungen: Tutorium in der Woche 06.07.-10.07.

Einige Annahmen

Wir nehmen einen Typ **Nat** von Konstanten 0,1,2,3,... und die die üblichen Grundoperationen $+,-,*,max,min,==,...$ für **Nat** als gegeben an. Weiter nehmen wir einen Typ **Bool** mit den Konstanten *True* und *False* sowie den Grundoperationen $\wedge,\vee,not$ etc. und dem **if** _ **then** _ **else** _-Konstrukt von Übungsblatt 2 als gegeben an. Mit () bezeichnen wir den Typ *unit*; er enthält nur einen einzigen Wert, den wir ebenfalls mit () bezeichnen.

Weiterhin schreiben wir in Beweisen der Gleichheit zweier gegebener Terme s und t die Aussage $s =_{\beta\delta} t$ abgekürzt als $s = t$. Beachten Sie, dass $=_{\beta\delta}$ hierbei die Konvertierbarkeitsrelation bezüglich β- und δ-Reduktionen ist und auch mit $\leftrightarrow^*_{\beta\delta}$ bezeichnet wird.

Übung 1 Listen natürlicher Zahlen

Wir betrachten die folgende algebraische Definition eines Datentyps für Listen natürlicher Zahlen:

data NatList where
NNil : () → **NatList**
NCons : **Nat** → **NatList** → **NatList**

1. Beschreiben Sie in eigenen Worten die durch die folgenden Terme gegebenen Listen natürlicher Zahlen:
   - *NNil*
   - *NCons* 5 *NNil*
   - *NCons* 5 (*NCons* 5 *NNil*)
   - *NCons* 1 (*NCons* 2 (*NCons* 3 (*NCons* 4 *NNil*)))
2. Es können nun Funktionen induktiv über der Struktur von **NatList** definiert werden, beispielsweise:

   sum NNil = 0
   sum (*NCons x xs*) = x + *sum xs*

   (a) Welchen Typ hat *sum*?
   (b) Werten Sie den Term *sum* (*NCons* 4 (*NCons* 89 (*NCons* 21 *NNil*))) aus.
3. Schreiben Sie eine Funktion *element* : **Nat** → **NatList** → **Bool**, so dass *element a xs* = *True* wenn a in xs vorkommt, und andernfalls *element a xs* = *False*.

Übung 2 Allgemeine Listen

Wir könnten ähnliche algebraische Datentypen *BoolList*, *NatListList* etc. definieren; das wäre allerdings extrem redundant. Anstelle dessen definieren wir einen *parametrischen* Typ **List** a, wobei a eine Typvariable ist.

data List a **where**
$Nil \ : \ () \to$ **List** a
$Cons : a \to$ **List** $a \to$ **List** a

1. Entscheiden Sie für jeden der folgenden Terme, ob er typisierbar ist, und geben Sie gegebenenfalls den zugehörigen Prinzipaltyp an.

   - *Cons True* (*Cons True Nil*)
   - *Cons True* (*Cons False Nil*)
   - *Cons True* (*Cons* 35 *Nil*)
   - *Cons True*
   - *Cons Nil* (*Cons* (*Cons* 35 *Nil*) *Nil*)
   - *Cons Nil* (*Cons* 35 *Nil*)
   - *Cons Nil*

   Notation: Wir schreiben beispielsweise anstelle von *Cons* x (*Cons* y (*Cons* z (*Cons* v *Nil*))) kurz $[x,y,z,v]$. Insbesondere entspricht $[]$ der leeren Liste *Nil*.

2. Definieren Sie induktiv die folgenden Funktionen über Listen:

   (a) *length* : **List** $a \to$ **Nat**, so dass:
   length $[] = 0$, *length* $[x] = 1$, *length* $[x,y] = 2$, ...

   (b) *snoc* : **List** $a \to a \to$ **List** a, so dass *snoc xs x* das Element x an das rechte Ende von *xs* anhängt; z.B.:
   snoc $[]$ $x = [x]$, *snoc* $[y]$ $x = [y,x]$, *snoc* $[y,z]$ $x = [y,z,x]$, ...

   (c) *reverse* : **List** $a \to$ **List** a, so dass:
   reverse $[] = []$, *reverse* $[x] = [x]$, *reverse* $[x,y] = [y,x]$, *reverse* $[x,y,z] = [z,y,x]$, ...
   Hinweis: Verwenden Sie *snoc*.

   (d) *drop* : $a \to$ **List** $a \to$ **List** a, so dass:
   drop x $[] = []$, *drop* x $[x,y] = [y]$, *drop* x $[x,y,z,x] = [y,z]$, ...

   (e) *elem* : $a \to$ **List** $a \to$ **Bool**, so dass:
   elem x $[] = False$, *elem* x $[y,z,q,v] = False$, *elem* x $[y,z,z,q,x,z] = True$, ...

   (f) *maximum* : **List Nat** $\to$ **Nat**, so dass:
   maximum $[] = 0$, *maximum* $[3] = 3$, *maximum* $[3,5,2,3] = 5$, ...

Übung 3 Beweise mittels struktureller Induktion

Die von Ihnen in Übung 2 definierten Funktionen sollten die folgenden plausiblen Eigenschaften erfüllen. Beweisen Sie dies jeweils durch Induktion über der Struktur der Argumentliste. Rechtfertigen Sie hierbei Ihre Schritte und geben Sie jeweils Ihre Induktionshypothese an. Falls es

Ihnen nicht gelingt, eine der Eigenschaften zu beweisen, so kann das darauf hinweisen, dass Ihre Implementierung fehlerhaft ist!

Hinweis: Wir erinnern daran, dass $s = t$ als $s =_{\beta\delta} t$ zu lesen ist. Ausserdem können Sie jederzeit zuvor bereits bewiesene Eigenschaften verwenden.

1. $\forall x\ xs.\ \mathit{length}\ (\mathit{snoc}\ xs\ x) = 1 + \mathit{length}\ xs$
2. $\forall xs.\ \mathit{length}\ (\mathit{reverse}\ xs) = \mathit{length}\ xs$
3. $\forall x\ xs.\ \mathit{reverse}\ (\mathit{snoc}\ xs\ x) = \mathit{Cons}\ x\ (\mathit{reverse}\ xs)$
4. $\forall xs.\ \mathit{reverse}\ (\mathit{reverse}\ xs) = xs$

Übung 4 Eine binäre Funktion: Listenkonkatenation

Wir betrachten die folgende Definition einer Funktion zur Listenkonkatenation:

$\mathit{Nil} \oplus ys = ys$
$(\mathit{Cons}\ x\ xs) \oplus ys = \mathit{Cons}\ x\ (xs \oplus ys)$

Wir möchten die folgende Eigenschaft mittels struktureller Induktion beweisen:

$\forall xs\ ys.\ \mathit{length}\ (xs \oplus ys) = \mathit{length}\ xs + \mathit{length}\ ys$

1. Über welche Liste(n) sollten wir induzieren, über das erste Argument von $(\_ \oplus \_)$, über das zweite, oder über beide? Warum?
2. Beweisen Sie die oben angegebene Eigenschaft; begründen Sie Ihre Schritte und geben Sie explizit die Induktionshypothese an.
3. Beweisen Sie die folgenden Eigenschaften mittels struktureller Induktion. Begründen Sie Ihre Schritte und geben Sie Ihre Induktionshypothese an:
   (a) $\forall xs.\ xs \oplus \mathit{Nil} = xs$
   (b) $\forall x\ xs.\ xs \oplus [x] = \mathit{snoc}\ xs\ x$
   (c) $\forall x\ xs\ ys.\ \mathit{snoc}\ (xs \oplus ys)\ x = xs \oplus (\mathit{snoc}\ ys\ x)$
   (d) $\forall xs\ ys.\ \mathit{reverse}\ (xs \oplus ys) = (\mathit{reverse}\ ys) \oplus (\mathit{reverse}\ xs)$
   (e) $\forall xs\ ys\ zs.\ (xs \oplus ys) \oplus zs = xs \oplus (ys \oplus zs)$
   (f) $\forall xs\ ys\ x.\ \mathit{elem}\ x\ (xs \oplus ys) = (\mathit{elem}\ x\ xs) \ \vee \ (\mathit{elem}\ x\ ys)$

   Hinweis: Nehmen Sie für den letzten Beweis $\mathit{True} \ \vee \ b = \mathit{True}$ und $\mathit{False} \ \vee \ b = b$ als bereits bewiesen an.

Übung 5 Higher-order-Programmierung

Wir betrachten die folgenden Definitionen von *higher-order* Funktionen:

$id\ x = x$

$f\ .\ g = \lambda x\ .\ f\ (g\ x)$

$map\ f\ Nil = Nil$
$map\ f\ (Cons\ x\ xs) = Cons\ (f\ x)\ (map\ f\ xs)$

1. Geben Sie die Prinzipaltypen von *id*, $(\_\ .\ \_)$ und *map* an.
2. Was berechnet der Term $maximum\ .\ (map\ length)$? Geben Sie den Prinzipaltypen dieses Terms an.
3. Beweisen Sie die folgenden – als Programmoptimierungen zu verstehenden – Eigenschaften mittels struktureller Induktion über der jeweiligen Argumentliste. Begründen Sie Ihre Schritte und geben Sie für jede Eigenschaft explizit Ihre Induktionshypothese an:
   (a) $\forall xs.\ map\ id\ xs = xs$
   (b) $\forall f\ g\ xs.\ (map\ f\ .\ map\ g)\ xs = map\ (f\ .\ g)\ xs$
   (c) $\forall xs\ ys\ f.\ map\ f\ (xs \oplus ys) = (map\ f\ xs) \oplus (map\ f\ ys)$
   (d) $\forall x\ y\ xs.\ x = y \Rightarrow map\ (drop\ x)\ (map\ (Cons\ y)\ xs) = map\ (drop\ x)\ xs$
   (e) $\forall x\ y\ xs.\ x \neq y \Rightarrow map\ (drop\ x)\ (map\ (Cons\ y)\ xs) = map\ (Cons\ y)\ (map\ (drop\ x)\ xs)$

   Hinweis: Wir nehmen an, dass die verwendeten Quantifizierungen jeweils die entsprechenden Typbeschränkungen respektieren und dass *drop* wie in Übung 2 und $\oplus$ wie in Übung 4 definiert ist.

Übung 6 Binäre Bäume

Die folgenden Definitionen definieren eine Datentyp für *binäre Bäume* sowie einige Grundfunktionen für diesen Typ:

data BinTree a **where**
  $Leaf\ :\ () \rightarrow$ **BinTree** a
  $Bin\ :$ **BinTree** $a \rightarrow a \rightarrow$ **BinTree** $a \rightarrow$ **BinTree** a

$mirror\ Leaf = Leaf$
$mirror\ (Bin\ l\ x\ r) = Bin\ (mirror\ r)\ x\ (mirror\ l)$

$inorder\ Leaf = Nil$
$inorder\ (Bin\ l\ x\ r) = inorder\ l \oplus Cons\ x\ (inorder\ r)$

Beweisen Sie mittels struktureller Induktion die folgenden Eigenschaften. Begründen Sie dabei Ihre Schritte und geben Sie Ihre Induktionshypothese(n) jeweils explizit an.

Hinweis: Sie sollten jeweils zwei Induktionshypothesen aufstellen! Ausserdem werden Sie vermutlich in vorherigen Übungen bereits bewiesene Eigenschaften benötigen.

1. $\forall t.\ mirror\ (mirror\ t) = t.$
2. $\forall t.\ inorder\ (mirror\ t) = reverse\ (inorder\ t).$

Übung 7 Delay-Listen

Wir nehmen an, dass wir einen Algorithmus programmieren möchten, der seine Eingabedaten in Form einer Liste von Elementen eines Typs a erhält, wobei die Eingabe in unregelmäßigen zeitlichen Abständen erfolgen darf; wir erlauben also, dass zwischen der Eingabe von je zwei Elementen x und y des Typs a ein beliebiger Zeitraum vergehen darf (beispielsweise, weil auf Benutzereingaben gewartet wird). Die folgende Definition definiert einen hierzu geeigneten mehrsortigen Datentyp für sogenannte *Delay-Listen*:

data DelayList a, **Delay** a **where**
  Nil : () $\to$ **DelayList** a
  $Cons$: $a \to$ **Delay** $a \to$ **DelayList** a
  Now : **DelayList** $a \to$ **Delay** a
  $Later$: **Delay** $a \to$ **Delay** a

Terme zu dieser Signatur entsprechen Listen bei denen zwischen zwei Elementen vom Typ a jeweils ein *Delay* (d.h. eine Verzögerung um n Schritte, für $n \in \mathbb{N}$) steht. Beispielsweise steht der Term $Cons\ x\ (Later\ (Later\ (Now\ (Cons\ y\ (Later\ (Now\ Nil))))))$ für die Liste $[x, 3, y, 2]$.

Wir definieren die folgenden Funktionen mittels wechselseitiger Rekursion:

$countDelaysl\ Nil = 0$
$countDelaysl\ (Cons\ x\ d) = countDelaysd\ d$
$countDelaysd\ (Now\ dl) = 1 + (countDelaysl\ dl)$
$countDelaysd\ (Later\ d) = countDelaysd\ d$

$sumDelaysl\ Nil = 0$
$sumDelaysl\ (Cons\ x\ d) = sumDelaysd\ d$
$sumDelaysd\ (Now\ dl) = 1 + (sumDelaysl\ dl)$
$sumDelaysd\ (Later\ d) = 1 + (sumDelaysd\ d)$

Die Funktion $countDelaysl$: **DelayList** $a \to$ **Nat** bzw. $sumDelaysl$: **DelayList** $a \to$ **Nat** zählt bzw. addiert also die in einer Delay-Liste vorkommenden Delays.

$incDelaysl\ Nil = Nil$
$indDelaysl\ (Cons\ x\ d) = Cons\ x\ (incDelaysd\ d)$
$incDelaysd\ (Now\ dl) = Later\ (Now\ (incDelaysl\ dl))$
$incDelaysd\ (Later\ d) = Later\ (incDelaysd\ d)$

Die Funktion $incDelaysl$: **DelayList** $a \to$ **DelayList** a verlängert also alle in einer Delay-Liste vorkommenden Delays um einen Schritt.

1. Wir betrachten den Term $t = Cons\ p\ (Later\ (Later\ (Later\ (Now\ (Cons\ q\ (Later\ (Now\ Nil)))))))$. Berechnen Sie $countDelaysl\ t$, $sumDelaysl\ t$ und $incDelaysl\ t$.

2. Beweisen Sie mittels struktureller Induktion die folgenden Eigenschaften. Begründen Sie dabei Ihre Schritte und geben Sie Ihre Induktionshypothese jeweils explizit an.

   (a) $\forall\, dl.\ countDelaysl\ (incDelaysl\ dl) = countDelaysl\ dl$

   (b) $\forall\, dl.\ sumDelaysl\ (incDelaysl\ dl) = (sumDelaysl\ dl) + (countDelaysl\ dl)$

Übung 8 Ein koinduktives Animationsstudio

Eine *Animation* ist eine unendliche Sequenz von *Sprites*. Insbesondere ist eine *Animationsschleife* eine Animation, in der sich eine *endliche* Anzahl von Sprites unendlich oft wiederholt;

beispielsweise ließe sich ein nach rechts laufender Mensch mittels einer Animationsschleife darstellen. Eine Animation entspricht dann auf natürliche Weise einem *koinduktiven Typ*, und wir können mittels *Korekursion* eine Funktion $loop$: **List Sprite** → **Animation** definieren, die aus einer (als abstrakten Typ betrachteten) Liste von Sprites eine Animationsschleife erzeugt:

codata Animation where
  $sprite$: **Animation** → **Sprite**
  $advance$: **Animation** → **Animation**

$$sprite\ \ (loop\ (Cons\ s\ ss)) = s$$
$$advance\ (loop\ (Cons\ s\ ss)) = loop\ (snoc\ ss\ \ s)$$

$$sprite\ \ (loop\ \ Nil) = blankSprite$$
$$advance\ (loop\ \ Nil) = loop\ Nil$$

1. Es sei $walk\_right$ die aus sechs Sprites bestehende Liste $[s1, s2, s3, s4, s5, s6]$, die die folgende Sequenz bezeichnen soll:

   

   Was ist das Ergebnis der Auswertung des folgenden Terms?

   $$sprite\ \ (advance\ (advance\ (advance\ (loop\ \ walk\_right))))$$

2. Wir möchten nun eine gegebene Animation a um einen Frame *verzögern*. Das heisst, $delay\ \ a$ soll sich genau so wie a verhalten, allerdings soll der erste Frame von a zu Beginn von $delay\ \ a$ zweimal angezeigt werden. Definieren Sie eine korekursive Funktion

   $$delay\ :\ \textbf{Animation} \to \textbf{Animation},$$

   die die folgende Eigenschaft für alle $n \in \mathbb{N}$ erfüllt:

   $$sprite\ \ (advance^n\ \ (delay\ \ a)) = \begin{cases} sprite\ \ a & \text{falls } n = 0 \\ sprite\ \ (advance^{n-1}\ \ a) & \text{falls } n > 0 \end{cases}$$

3. Das Verzögern von Animationen ist nützlich, um die Geschwindigkeit einer Animation zu beeinflussen. Nehmen wir beispielsweise an, dass die Anzahl von Frames pro Sekunde fest vorgegeben ist, so können wir die Animationsgeschwindigkeit halbieren, indem wir *jeden* Frame (anstelle – wie oben – nur den ersten Frame) einmal verzögern. Definieren Sie eine korekursive Funktion $halfspeed$: **Animation** → **Animation**, die jeden Frame verzögert.

4. Analog ist es möglich, die Geschwindigkeit einer Animation zu verdoppeln, indem jeder zweite Frame übersprungen wird. Offensichtlich gibt es zwei Möglichkeiten, dies zu erreichen: Durch das Überspringen entweder aller Frames in *geraden* oder aber aller Frames in *ungeraden* Positionen. Definieren Sie beide Varianten als korekursive Funktionen, d.h. definieren Sie die zwei Funktionen $doublespeed\_e$, $doublespeed\_o$: **Animation** → **Animation** korekursiv.

   Hinweis: Möglicherweise werden Sie eine Hilfsfunktion definieren müssen.

5. Eine Methode zur Erzeugung komplexerer Animationen ist das Voranstellen einer (endlichen) Startsequenz vor eine gegebene Animation. Definieren Sie mittels Korekursion eine Funktion *prepend* : **List Sprite** → **Animation** → **Animation**, die solch eine einmalig abzuspielende Startsequenz vor einer Animation einfügt.

6. Animationsschleifen werden üblicherweise verwendet, um eine kontinuierliche Bewegung darzustellen, wie beispielsweise eine Person, die wartet, nach links oder rechts geht, nach links oder rechts rennt, klettert etc. Beim *Übergang* von einer Schleife *a1* zu einer anderen Schleife *a2* (beispielsweise beim Übergang vom Rennen zum Gehen) sollte darauf geachtet werden, das mit dem Übergang so lange gewartet wird, bis das "aktuelle" Sprite von *a1* *kompatibel* mit dem ersten Sprite von *a2* ist (ansonsten wird der Übergang zu abrupt erfolgen). Wir nehmen an, dass eine Funktion *compatible* : **Sprite** → **Sprite** → **Bool** gegeben ist. Definieren Sie eine Funktion *transition* : **Animation** → **Animation** → **Animation**, so dass *transition a1 a2* eine Animation ist, die so lange *a1* abspielt, bis ein mit dem ersten Sprite von *a2* kompatibles Sprite erreicht wird, und sodann zu *a2* übergeht.

Übung 9 Koinduktion

Um sicherzustellen, dass die Funktionen aus der vorherigen Übung korrekt sind, möchten wir nun beweisen, dass sie einige sinnvolle Eigenschaften erfüllen.

1. Es seien a und b zwei Terme des Typs **Animation**. Um mittels Koinduktion zu zeigen, dass $a = b$ gilt, müssen wir eine *Bisimulation* R (für den Typ **Animation**) finden, so dass $a\ R\ b$. Definieren Sie den Begriff einer Bisimulation für den Typ **Animation**.

2. Beweisen Sie die folgenden Eigenschaften mittels Koinduktion (falls nötig). Falls ein Beweisversuch fehlschlägt, so kann das daran liegen, dass die entsprechende Funktion nicht korrekt definiert ist!

   (a) $\forall a.\ \mathit{doublespeed\_e}\ (\mathit{halfspeed}\ a) = a$

   (b) $\forall a.\ \mathit{doublespeed\_e}\ a = \mathit{doublespeed\_o}\ (\mathit{delay}\ a)$

   (c) $\forall s\ t\ ts.\ \mathit{compatible}\ s\ t = \mathit{False} \Rightarrow$
   $\mathit{transition}\ (\mathit{loop}\ [s])\ (\mathit{loop}\ (\mathit{Cons}\ t\ ts)) = \mathit{loop}\ [s]$

   (d) $\forall s\ ss\ t\ ts.\ \mathit{compatible}\ s\ t = \mathit{True} \Rightarrow$
   $\mathit{transition}\ (\mathit{loop}\ (\mathit{Cons}\ s\ ss))\ (\mathit{loop}\ (\mathit{Cons}\ t\ ts)) = \mathit{prepend}\ [s]\ (\mathit{loop}\ (\mathit{snoc}\ ts\ t))$

   Hinweis: Falls Sie Ihren Beweis nicht vollenden können, Ihre Funktion aber für korrekt halten, so ist es vielleicht notwendig, dass sie den Kandidaten für Ihre Bisimulation *vergrössern*.

Übung 10 Ein koinduktiver zustandsloser Server

Ein HTTP-Server akzeptiert Verbindungen von Klienten; jeder Klient kann während der Dauer der Verbindung keine oder aber eine beliebige Anzahl von Requests stellen, so lange, bis die Verbindung beendet wird (falls sie je beendet wird). Weiterhin darf zwischen zwei Requests eines Klienten eine beliebige Zeitspanne liegen. Wir können die Verbindung *abfragen* (*pollen*), und das Ergebnis der Abfrage wird sein, dass die Verbindung entweder *beendet* (*closed*) ist, auf einen neuen Request oder die Verbindungbeendigung *wartet* (*waiting*), oder dass bereits ein neuer

Request erhalten wurde und die Verbindung *bereit* (*ready*) zum Transfer ist. Wir abstrahieren vom HTTP-Protokoll und definieren den koinduktiven Typ **Requests** von zustandslosen Requests auf einer Verbindung:

codata Requests a **where**
  eof : **Requests** $a@closed \to ()$
  $retry$: **Requests** $a@waiting \to$ **Requests** a
  $request$: **Requests** $a@ready \to a$
  $consume$: **Requests** $a@ready \to$ **Requests** a

Wir definieren drei Extremfälle von Verbindungen: Eine Verbindung, die unmittelbar beendet wird, eine Verbindung, die unendlich lange wartet, und eine Verbindung, die unendlich oft denselben Request wiederholt.

$$eof\ oops = ()$$

$$retry\ indecisive = indecisive$$

$$request\ (forever\ x) = x$$
$$consume\ (forever\ x) = forever\ x$$

Die Implementierung unseres Servers startet *worker*-Prozesse, um die **Requests** zu bearbeiten. Wir beobachten, dass wir Resourcen und Initialisierungszeit einsparen können, indem wir einen worker-Prozess mehr als eine Verbindung überwachen lassen. Dies kann auf transparente Art erfolgen, indem wir beispielsweise zwei Verbindungen seriell zu einer Verbindung verketten:

$$\begin{aligned}
eof\ \ &(serial\ xs@closed\ \ ys@closed) &&= () \\
retry\ \ &(serial\ xs@waiting\ ys) &&= serial\ (retry\ xs)\ ys \\
retry\ \ &(serial\ xs@closed\ \ ys@waiting) &&= retry\ ys \\
request\ &(serial\ xs@ready\ \ ys) &&= request\ xs \\
request\ &(serial\ xs@closed\ \ ys@ready) &&= request\ ys \\
consume\ &(serial\ xs@ready\ \ ys) &&= serial\ (consume\ xs)\ ys \\
consume\ &(serial\ xs@closed\ \ ys@ready) &&= consume\ ys
\end{aligned}$$

Wir möchten sicherstellen, dass *serial* sich erwartungsgemäss verhält.

1. Definieren Sie den Begriff einer Bisimulation für den Typ **Requests** a.
2. Beweisen Sie die folgenden Eigenschaften mittels Koinduktion:
   (a) $\forall s.\ \ serial\ \ oops\ s = s$
   (b) $\forall r.\ \ serial\ \ r\ \ oops = r$
   (c) $\forall s.\ \ serial\ \ \ indecisive\ \ s = indecisive$
   (d) $\forall x\ s.\ \ serial\ \ (forever\ x)\ \ s = forever\ x$
3. Serielle Komposition ist nicht *fair* und erlaubt einfache DoS-Attacken, wie durch die letzten beiden Eigenschaften deutlich wird. Die *parallele* Komposition von zwei Verbindungen behebt dieses Problem indem sie Requests jeder einzelnen Verbindung unmittelbar ausliefert. Definieren Sie nun also eine Funktion par : **Requests** $a \to$ **Requests** $a \to$ **Requests** a, die zwei Verbindungen parallelisiert. Ihre Implementierung soll die folgenden Eigenschaften erfüllen. Beweisen Sie diese Eigenschaften mittels Koinduktion.
   (a) $\forall s.\ \ par\ oops\ s = s$
   (b) $\forall r.\ \ par\ r\ \ oops = r$
   (c) $\forall s.\ \ par\ \ indecisive\ \ s = serial\ \ s\ \ \ indecisive$

(d) $\forall r.\ par\ r\ indecisive = serial\ r\ indecisive$

(e) $\forall x\ y.\ par\ (intermittent\ x)\ (intermittent\ y) = alternate\ x\ y$, wobei

$request\ (alternate\ x\ y) = x$
$consume\ (alternate\ x\ y) = alternate\ y\ x$

$request\ (intermittent\ x) = x$
$consume\ (intermittent\ x) = waiting\ x$
where $retry\ (waiting\ x) = intermittent\ x$

Übung 11 *Allgemeine Bäume* *(20 Punkte)*

Wir nehmen für diese Übung die Funktionen $\oplus$ und *map* aus Übung 4 und Übung 5 als gegeben an und betrachten den folgenden mehrsortigen algebraischen Datentyp für beliebig verzweigende Bäume mit beschrifteten *Transitionen*:

data Tree a, **Forest** a **where**
$Empty : () \to$ **Forest** a
$Grow : a \to$ **Tree** $a \to$ **Forest** $a \to$ **Forest** a
$Leaf : () \to$ **Tree** a
$ToTree :$ **Forest** $a \to$ **Tree** a

Ein Baum ist also entweder ein Blatt oder er besteht aus einem Wald von Nachfolgern. Ein Wald ist entweder leer oder er besteht aus mindestens einem (mit einem Element von a beschrifteten) Baum und einem Restwald.

Die Funktionen $tracest :$ **Tree** $a \to$ **List** (**List** a) und $tracesf :$ **Forest** $a \to$ **List** (**List** a) erstellen Listen aller Pfade, die in einem Baum bzw. einem Wald zu einem Blatt bzw. einem leeren Wald führen.

$tracest\ Leaf = Cons\ Nil\ Nil$
$tracest\ (ToTree\ f) = tracesf\ f$
$tracesf\ Empty = Nil$
$tracesf\ (Grow\ x\ t\ f) = (map\ (Cons\ x)\ (tracest\ t)) \oplus (tracesf\ f)$

1. Stellen Sie den durch den folgenden Term gegebenen Baum grafisch dar:

   $t = ToTree\ (Grow\ x\ Leaf\ (Grow\ y\ (ToTree\ (Grow\ z\ Leaf\ (Grow\ q\ Leaf\ Empty)))\ (Grow\ y\ Leaf\ Empty)))$

   Berechnen Sie $tracest\ t$.

2. Definieren Sie eine Funktion $mapt : (a \to a) \to$ **Tree** $a \to$ **Tree** a, so dass $mapt\ g\ t$ die Funktion g auf die jeweilige Beschriftung jeder Transition des Baums t anwendet. Beispielsweise soll gelten:

   $mapt\ (add\ 1)\ (ToTree((Grow\ 3\ Leaf\ (Grow\ 7\ Leaf\ Empty))))$
   $= ToTree((Grow\ 4\ Leaf\ (Grow\ 8\ Leaf\ Empty)))$

   Hinweis: Definieren Sie hierzu auch eine Funktion $mapf$: $(a \to a) \to$ **Forest** $a \to$ **Forest** a.

3. Beweisen Sie die folgende Eigenschaft:

   $\forall g\ x\ xs.\ map\ (Cons\ (g\ x))\ (map\ (map\ g)\ xs) = map\ (map\ g)\ (map\ (Cons\ x)\ xs)$

   Zeigen Sie dann, dass Ihre Funktionen *mapt* und *mapf* korrekt sind, indem Sie zeigen, dass sie die folgende Eigenschaft erfüllen:

$\forall g\ t.\ \ tracest\ \ (mapt\ g\ t) = map\ \ (map\ \ g)\ \ (tracest\ t)$

Rechtfertigen Sie alle Ihre Schritte und geben Sie Ihre Induktionshypothese(n) jeweils explizit an!

Hinweis: Beweisen Sie zunächst die erste Eigenschaft und verwenden Sie sie dann als Lemma im Beweis der zweiten Eigenschaft. Verwenden Sie, falls nötig, auch das folgende in Übung 5 bewiesene Lemma:

$\forall g\ \ xs\ \ ys.\ \ map\ \ g\ \ (xs \oplus ys) = (map\ \ g\ \ xs) \oplus (map\ \ g\ \ ys)$

Übung 12 *IRC-Bots considered coinductive* *(20 Punkte)*

Ein IRC-Bot ist ein Programm, das sich zu einem IRC-Channel verbindet und dort Nachrichten liest oder schreibt, auf Anfrage Informationen zur Verfügung stellt etc. Wir können einen IRC-Bot (oder eine beliebige andere Art von Chat-Bot) durch einen koinduktiven Typ **Chatter** $a\ b$ modellieren, wobei a und b die Typen der Eingabe- bzw. Ausgabe-Nachrichten sind (bei einem IRC-Bot wären a und b jeweils Strings):

codata Chatter $a\ b$ **where**
  $listen$: **Chatter** $a\ b$*@listening* $\to (a \to$ **Chatter** $a\ b)$
  $says$: **Chatter** $a\ b$*@speaking* $\to b$
  $continue$: **Chatter** $a\ b$*@speaking* $\to$ **Chatter** $a\ b$

$listen\ \ mute = \lambda x\ .\ mute$

$listen\ \ parrot = \lambda x\ .\ \ repeat\ \ x$
  where $says\ \ \ \ (repeat\ \ y) = y$
  $continue\ (repeat\ \ y) = parrot$

$listen\ \ (const\ \ (Cons\ x\ xs)) = \lambda y\ .\ say\ \ (Cons\ x\ xs)$
  where $says\ \ \ \ (say\ \ (Cons\ x\ xs)) = x$
  $continue\ (say\ \ (Cons\ x\ xs)) = const\ (snoc\ \ xs\ \ x)$

$says\ \ (spam\ \ (Cons\ x\ xs)) = x$
$continue\ \ (spam\ \ (Cons\ x\ xs)) = spam\ \ (snoc\ xs\ x)$

Die Definition beinhaltet vier einfache Bots: Der Bot *mute* liest mit, ohne je etwas zu sagen und der Bot *parrot*: **Chatter** $a\ a$ wiederholt alles, was er liest; der Bot *const* ignoriert jeden Input und gibt einfach immer wieder seine Argumentliste aus und der Bot *spam* gibt permanent seine Argumentliste aus, ohne je auf Eingaben zu warten.

1. Definieren Sie den Begriff einer Bisimulation für den Typ **Chatter** $a\ b$.
2. Definieren Sie eine Funktion *compose* : **Chatter** $a\ a \to$ **Chatter** $a\ a \to$ **Chatter** $a\ a$, die zwei kompatible Bots c und d derart verkettet, dass d liest, was c sagt, und umgekehrt. Der zusammengesetzte Bot soll dann zwischen dem Verhalten von c und d alternieren.
3. Beweisen Sie, dass Ihre Funktion die folgenden Eigenschaften erfüllt:
   (a) $\forall x.\ \ compose\ mute\ x = mute$,
   (b) $\forall x\ y\ z.\ \ listen\ \ (compose\ \ (const\ \ [x])\ \ (const\ \ [y]))\ \ z = spam\ \ [x,y]$
   (c) $\forall y.\ \ compose\ \ (say\ \ [y])\ \ parrot = spam\ \ (twice\ \ [y]))$

wobei

$$twice\ Nil = Nil$$
$$twice\ (Cons\ x\ xs) = Cons\ x\ (Cons\ x\ (twice\ xs))$$