ThProg Zusammenfassung

Von El Presidente

Inhalt

Klausurhinweise:

Aufgabentypen

- **TES:** Konfluenz (Schwierigkeit ++) und Terminierung (Schwierigkeit +)
- **System F** (Schwierigkeit ++)
- **Strukturelle Induktion (**Schwierigkeit ++) **und Folds** (Schwierigkeit +++)
- **Korekursion**(Schwierigkeit ++) **und Koinduktion** (Schwierigkeit +++)
- **Reguläre Sprachen / Reguläre Automaten** (Schwierigkeit +)

StuvePad:

https://pad.stuve.fau.de/p/thprog

Termesetzung (TES): Konfluenz und Terminierung (Übung 2+3)

Terme

Gegeben: Signatur Σ und Menge von Variablen V

Die Menge der Terme $T_\Sigma(V)$ ist induktiv wie folgt definiert:

- Variable v ∈ V ist Term, Konstante $c_{/0}$ ∈ Σ (nullstelliges Funktionssymbol) ist Term
- Sind $t_1, \ldots, t_n$ Term und ist $f_{/n}$ ∈ Σ n-stelliges Funktionssymbol, so ist $f(t_1, \ldots t_n)$ Term.

Substitution

Eine Substitution σ ist eine Funktion σ : V → $T_\Sigma(V)$. Angewandt auf Terme ersetzt sie Variablen durch Terme. Wir schreiben das Ergebnis der Anwendung von Substitution σ auf Term t wie folgt: tσ.

Termersetzungssystem (TES):

Eine Termersetzungsregel ist ein Paar (l, r) von Termen (wobei l nicht Variable ist und r nur Variablen aus l erwähnt). Ein Termersetzungssystem (TES) (Σ,→₀) besteht aus Σ und einer Menge von Regeln →₀ über Σ. Wir schreiben s → t wenn

> es eine Substitution σ und eine Regel (l, r) ∈→₀ gibt, so dass lσ in s enthalten ist und t an derselben Stelle rσ enthält und ansonsten mit s übereinstimmt.

Notation: →* (Reduktion), →⁺, ↔, ↔* (Konvertierung)

Termersetzung

Gegeben sei eine **Signatur Σ**, z.B. Σ = {A, B, ·, ….}. Diese besteht aus Konstanten (z.B. **A, B, C**,…) und Funktionssymbolen (z.B. ·). Zusätzlich gibt es ein **Termersetzungssystem (TES**, hat nichts mit Bethesda zu tun). Diese besteht aus **Produktionsregeln P** der Form Σ-Term→₀ Σ-Term. Anwenden dieser Regeln heißt **ableiten**.

Oft existiert auch noch eine **Variablenmenge V** ={x,y,…}. Funktionssymbole bekommen werden in **Σ** oft auch als z.B: ·/2 dargestellt. · ist die Funktion, und die 2 ist Anzahl der Parameter der Funktion.

Konstanten sind auch sozusagen Funktion mit 0 Parametern: c/0

TES Beispiel:		**Anmerkungen:**
$A \cdot x \rightarrow_0 B \cdot (C \cdot x)$	(1)	x und y sind Variablen (Variablenmenge **V = {x,y}**) und können stellvertretend für Σ-Term und Konstanten stehen.´ Ein endlicher Vorrat an Variablen heißt auch dass es nur endlich viele Kombinationen gibt.
$C \cdot (D \cdot x) \rightarrow_0 B \cdot (C \cdot x)$	(2)	
$B \cdot (x \cdot y) \rightarrow_0 A \cdot (D \cdot x)$	(3)	
$B \cdot (B \cdot x) \rightarrow_0 D \cdot x$	(4)	

Unendliche Derivation t → t_1 → t_2 → t_3:

Es existiert ein Σ-Term t, für den wir immer weiter unendlich ableiten können. Bei obrigen Beispiel ist das z.B für den Σ-Term t = $A \cdot (D \cdot C)$:

$A \cdot (D \cdot C) \rightarrow^{(1)} B \cdot (C \cdot (D \cdot C) \rightarrow^{(2)} B \cdot (B \cdot (C \cdot C)) \rightarrow^{(3)} B \cdot (A \cdot (D \cdot C)) \rightarrow^{(3)} D \cdot (C \cdot C)) \not\rightarrow$

t → t_1 → t_2 → t_3 ↳ Normalform

Normal-Form (Terminierung)

Normalform eines Terms: Ein Term t ist normal, wenn er nicht mehr abgeleitet werden kann.

Schwach normalisierender Term : Ein Term s kann zu mindestens einer Normalform abgelitten werden. Ein TES ist schwach normalisierend, wenn jeder Term des TES dies erfüllt.

Stark normalisierender Term : Term ist S

schwach normalisierend, und es gibt eine unendliche Derivation. Ein TES ist starknormalisierend, wenn jeder Term des TES dies erfüllt.

Stark Normaliserend widerlegen: Zeige dass das TES einen Term hat, den man unendlich ableiten kann.

Stark Normalisierend zeigen (mit Polynomordnung):

Polynome in die Polynomordnungen einsetzen • Domäne finden, bei der bei allen Polynomordnungen gilt: linke Seite >D rechte Seite (z.B: D= • Zusätzlich muss immer gelten: Jedes Polynom ist ganzzahlig positiv und streng monoton unter der Domäne D

Grammatik G(V, T, P, S):

Dient zum Erzeugen und Beschreiben von formalen Sprachen. Damit kann man Terme bzw. Wörter einer formalen Sprache beschreiben, und auch feststellen ob ein Term oder Wort Teil dieser formalen Sprache ist. Der uns bekannte Aufbau ist die Backus-Nauer-Form (BNF):

G:= regel|regel|regel|….

| ist ein Trennstrich für die Produktionsregeln. Diese sind oft rekursiv in der Grammatik verknüpfbar, und oft sind „Basisfälle“ angegeben.

Eine Grammatik besteht außerdem auch noch aus (siehe BfS, nicht so krass wichtig für ThProg):

- Signatur Σ,
- Produktionsregeln P,
- einem einer Menge von Symbolen V,
- einem Alphabet T,
- und einem Startsymbol S.

Kontext:

Ein Kontext ist ein Term C(·) mit genau einem „Loch“ (·). Formal sind Kontexte

definiert durch die Grammatik:

$$C(\cdot) ::= (\cdot) \mid f(t_1, \ldots, t_{i-1}, C(\cdot), t_{i+1}, \ldots, t_n)$$

Das Resultat C(t) der Einsetzung eines Terms t in einen Kontext C(·) ist rekursiv definiert durch

- $(\cdot)(t) = t$
- $f(t_1, \ldots, C(\cdot), \ldots, t_n)(t) = f(t_1, \ldots, C(t), \ldots, t_n)$

Strikte Ordnung, Asymmetrie; Irreflexivität, Transitivität einer Relation R

Sei $R \subseteq X \times X$ eine Relation.

Reflexiv: wenn xRx für alle $x \in X$

Irreflexiv: R ist irreflexiv genau dann wenn für alle x gilt: **¬xRx**;

Symmetrisch: wenn für alle $(x, y) \in R$ (also xRy) auch yRx gilt

Asymmetrisch: R ist asymmetrisch genau dann wenn für alle x, y gilt: **xRy ⇒ ¬yRx**

Transitivit: R ist transitiv genau dann wenn für alle x, y, z gilt: **xRy und yRz ⇒ xRz**

Strikte Ordnung (bzw. Prädordnung): R ist eine strikte Ordnung wenn R irreflexiv und transitiv ist;

- Eine Strikte Ordnung ist asymmetrisch
- Eine transitive asymmetrische Relation ist eine strikte Ordnung

Äquivalenzrelation oder einfach eine Äquivalenz, wenn R reflexiv, transitiv und symmetrisch ist

Polynomordnung: Beweisen das ein TES stark normalisierend ist

Eine polynomielle Interpretation ist ein Tupel $\mathcal{A} = \langle A, p_{f_1}, \ldots, p_{f_n} \rangle$ bestehend aus der Domäne $A \subseteq \mathbb{N}$ und einem streng monotonen Polynom $p_{f_i} \in \mathbb{N}[X_1, \ldots, X_m]$ mit

$$\forall a_1, \ldots, a_o \in A \Rightarrow p_{f_i}(a_1, \ldots, a_o) \in A$$

für jedes $f_{i/m} \in \Sigma$.
Die von $\mathcal{A}$ induzierte Polynomordnung $\succ_{\mathcal{A}}$ über Termen ist definiert als

$$t \succ_{\mathcal{A}} t' \text{ gdw. } p_t >_A p_{t'}$$

wobei $p_x = X$, $p_{f(t_1,\ldots,t_n)} = p_f(p_{t_1}, \ldots, p_{t_n})$ und

$$p >_A q \text{ gdw. } \forall a_1, \ldots, a_n \in A. p(a_1, \ldots, a_n) > q(a_1, \ldots, a_n).$$

Terminierung eines TES R mittels $\succ_{\mathcal{A}}$ zeigen:

1. Sicherstellen, daß $\mathcal{A}$ polynomielle Interpretation ist, und
2. zeigen, daß für jede Regel $(l, r) \in \to_0$ gilt: $l \succ_{\mathcal{A}} r$.

Beispiel:	**Vorgehen:**
Schritt -1: Anmerkungen	
Aus dem oben genannten Tupel $\mathcal{A} = \langle A, p_{f_1}, \ldots . p_{f_n} \rangle$ (n = Anzahl der Funktionssymbole) ist A ⊆ ℕ unser Definitionsbereich aus Schritt 5. Manchmal ist A festgelegt, also muss dann der Definitionsbereich der Polynome auch für dieses vorgegebene A gelten! Ist keines vorgegeben, dürfen wir A dann selbst festlegen	
Schritt 0: TES und Signatur	
Signatur Σ = {-/1; []/3} (-(…) ist unäres Funktionssymbol, und wird mit -… abgekürzt) **Variablenmenge V** = {x,y,z} **TES:** [x, -y, z] →$_0$ [x, z, y] (1) -[x, y, z] →$_0$ [-x, -y, -z] (2) *Wir erkennen hier relativ gut, dass keine unendlichen Ableitungen stattfinden können, und das Normalformen möglich sind. Daraus entsteht der Verdacht dass es vielleicht stark normalisierend ist.*	Gegeben ist • eine **Signatur Σ** mit Funktionssymbolen, **Variablenmenge V** (nicht immer explizit angegeben), und das • **Termersetzungssystem (TES)** mit seinen Termersetzungen (bzw. Ableitungen)
Schritt 1: Polynome festlegen für jedes Funktionssymbol	
Signatur Σ = {-/1; []/3} **Polynom für** -/1: $p_-(x) = x + 1$ **Polynom für** []/3: $p_{[]}(x,y,z) = x + y + z$	Für jedes Funktionssymbol erstellen wir ein mathematisches, ganzzahliges und streng monotones **Polynom $p_{Funktionssymbol}$ (*Variablen*),** welches auf ℕ abbildet. Bei nullstelligen Funktionssymbolen weisen wir ein Konstante zu (z.B. $p_c() = 5$). **Geschicktes Wählen der Polynome:** 1. x + c (c ist Konstante) 2. Wenn das nicht klappt: x· *c* 3. Wenn das auch nicht klappt: x^2 4. Wenn das auch nicht klappt: $c \cdot x^2$ *Sind mehrere Variablen gegeben, dann z.B. x+ y + c etc.*
Schritt 2: Polynome in TES einsetzen	
Term (1): [x, -y, z] →$_0$ [x, z, y] • $T_{1,L}$ ->$_0$ $T_{1,R}$ • $T_{1,L} = p_{[]}(x, p_-(y), z) = x + (y+1) + z$ • $T_{1,R} = p_{[]}(x, y, z) = x + y + z$ **Term (2):** -[x, y, z] →$_0$ [-x, -y, -z] • $T_{2,L}$ ->$_0$ $T_{2,R}$ • $T_{2,L} = p_-(p_{[]}(x, y, z)) = (x + y + z)+1$	Jetzt setzen wir unser Polynome in die **rechte** und die **linke Seite** jeder Regel des TES ein. Achtung, es werden alle Funktionssymbole durch die Polynome ersetzt! Verschachtelungen entstehen logischerweise.

• $T_{2,R} = p_{[]}(p_{-}(x), p_{-}(y), p_{-}(z)) = x+1 + y+1 + z+1 = x + y + z +3$	
Schritt 3: $T_L > T_R$ für jeden Term des TES?	
Term (1): • $T_{1,L} > T_{1,R}$? • $x + (y+1) + z > x + y + z$ • **=> Ist erfüllt, für alle Zahlen in $\mathbb{N}$ (Defintitionsbereich)** **Term (2):** • $T_{2,L} > T_{2,R}$? • $x + y + z+1 \not> x + y + z +3$ • **=> Nie erfüllt!**	Wir haben nun alles durch mathematische Polynome ersetzt. Jetzt müssen wir für jeden Term prüfen, ob die linke Seite T_L größer ist als die rechte Seite T_R. Außerdem geben wir den Definitionsbereich an in dem das gilt.
Schritt 4: Polynome anpassen, falls nötig	
Signatur Σ = {-/1; []/3} **Polynom für** -/1 (Wird geändert): $p_{-}(x) = x^2$ **Polynom für** []/3 (Bleit gleich): $p_{[]}(x,y,z) = x + y + z$ Wenn wir jetzt wieder in die Terme einsetzen, und brav vergleichen, sehen wir folgendes: **Term (1):** • $T_{1,L} > T_{1,R}$? • $x + y^2 + z > x + y + z$ • **=> Ist erfüllt, für alle Zahlen in $\mathbb{N} > 1$ (= Definitionsbereich)** **Term (2):** • $T_{2,L} > T_{2,R}$? • $(x + y + z)^2 > x^2 + y^2 + z^2$ • **=> Ist erfüllt, für alle Zahlen in $\mathbb{N} > 1$ (= Definitionsbereich)**	Ist $T_L > T_R$ bei einem Term nicht erfüllt, müssen wir eines (oder mehrere) der Polynome neu definieren, solange bis $T_L > T_R$ für jeden Term gilt! ***Zur Erinnerung*** **Geschicktes Wählen der Polynome:** 1. x + c (c ist Konstante) 2. Wenn das nicht klappt: $x \cdot c$ 3. Wenn das auch nicht klappt: x^2 4. Wenn das auch nicht klappt: $c \cdot x^2$ *Sind mehrere Variablen gegeben, dann z.B. x+ y + c etc*
Schritt 5: Konklusion	
Das System ist stark normalisierend, mit Definitionsbereich => Ist erfüllt, für alle Zahlen in $\mathbb{N} \setminus \{0,1\}$ (Definitionsbereich). (Für $p_{-}(x) = x^2 +1$, wär es sogar in ganz $\mathbb{N}$ erfüllt) $\mathcal{A}$ **= < A, p_{-},$p_{[]}$>, mit Domäne A =** $\mathbb{N} \setminus \{0,1\}$ => Q.E.D	Jetzt kommt hier unsere Konklusion hin, und der Definitionsbereich(=Domäne). **<u>Die Domäne muss immer hingeschrieben werden</u>**, sie ist Teil des Beweises. Ansonsten gibt es Punktabzug! $\mathcal{A}$ **= <Domäne As, p_1,….,p_n>** Wobei p_i die Polynome sind

Allgemeinster Unifikator

Zwei Terme s und t sind unifizierbar gdw. Unif(s, t) := {σ | sσ = tσ} ≠∅.

Der allgemeinste Unifikator mgu(s, t) zweier unifizierbarer Terme ist definiert durch σ = mgu(s, t) ⇔ ∀τ.(τ ∈ Unif(s, t) ⇔ ∃τ'.τ = στ')

Menge der freien Variablen eines Terms t

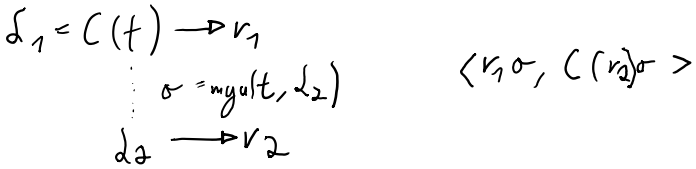
FV(t) ⊆ V bezeichne die Menge der freien Variablen eines Terms t

Kritisches Paar

Seien $l1 \to_0 r_1$ und $l2 \to_0 r_2$ zwei Umformungsregeln des TES

- mit $FV(l_1) \cap FV(l_2) = \emptyset$ (ggf. nach **Umbenennung der Variablen, muss man manchmal tun!**),
- und $l_1 = C(t)$ mit $t \notin V$ (t nicht trivial, also t ist nicht nur einfach eine Variable), und
- t und l_2 unifizierbar durch $\sigma = mgu(t, l_2)$.

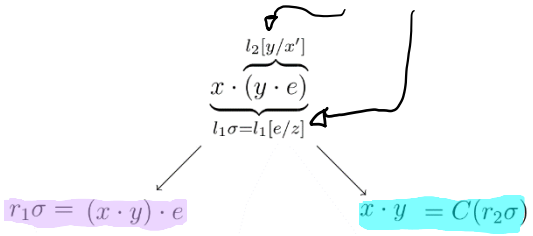
=> Dann ist $\langle r_1\sigma, C(r_2)\sigma \rangle$ kritisches Paar.



Auf germanisch: Das kritische Paar ergibt sich aus dem zu erstellenden Term, auf den man beide Regeln anwenden kann! Dies funktioniert über (geschickte) Substitution

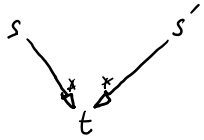
Beispiel: TES mit

- $(l_1 \to_0 r_1) = (\mathbf{x \cdot (y \cdot z)} \to_0 (x \cdot y) \cdot z)$ (1)
- $(l_2 \to_0 r_2) = (x' \cdot e \to_0 x')$ (2)
- Dann haben wir ein kritisches Paar $\langle r_1\sigma, C(r_2)\sigma \rangle$ durch $\sigma = mgu(t, l_2)$ mit
  - **$t = y \cdot z$**
  - **$C(\cdot) = x \cdot (\cdot)$**
- Also substituieren wir **$\sigma = [y/x', e/z]$**:



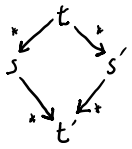
Zusammenführbarkeit von Termen

Zwei Terme s und s' sind (eines TES) zusammenführbar, wenn ein Term t existiert, so dass $s \to^* t$ und $s' \to^* t$.



Konfluenz eines TES T

T ist konfluent (Church-Rosser, CR), wenn für alle Terme t, s, s‘ mit t →* s und t →* s‘ die Terme s und s‘ zusammenführbar sind. => Ein TES ist konfluent, wenn alle Terme zusammenführbar sind



Newmans Lemma

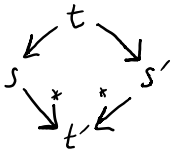
Ein TES, das stark normalisierend und lokal konfluent ist, ist konfluent.

Critical Pair Lemma

Ein TES ist lokal konfluent gdw. alle seine kritischen Paare zusammenführbar sind.

Lokale Konfluenz eines TES T

T ist lokal konfluent (weakly Church-Rosser, WCR), wenn für alle Terme t, s, s‘ mit t → s und t → s‘ die Terme s und s‘ zusammenführbar sind. (t → s bedeutet dass wir einmal ableiten, t →* s bedeutet dass wir mehrmals ableiten)=> Ein TES ist lokal konfluent, wenn alle Terme, die direkt aus dem Grundterm entstehen, zusammenführbar sind.



Lokale Konfluenz zeigen:

- Zu Zeigen: alle kritischen Paare zusammenführbar sind
- TES-Variablen umbenennen
- für jeden Term testen jeden anderen irgendwo einzusetzen, dazu müssen die Terme gemeinsame Operatoren haben
- Bsp: R1: -(v∧w) → (-v∨-w), R2: (u y) → -(-u -y) ∧ ∨ l1= -(v∧w), l2= (u∧y) σ=[u→v, y→w]; C(·)=-C(·) → t=(v w)) ∧ Nun Variablen auf der rechten seite der Regel R1 ersetzen: r1σ=(-u -y) ∨ Dann auf der linken Seite der Regel R1 die Variablen ersetzen und enstprechende Regel anwenden (bicht R1, sondern dann z.B> R2): C(r1σ)=--(-u∨-y), genauer Blatt04/A4
- weitere Regeln anwenden, um beide Paare zusammenzuführen => ist ein TES lokal konfluent und stark normalisierend, dann ist es auch stark konfluent

System F und λ-Kalkül (Übung 7 + 12)

Notationshinweise:

Wir schreiben λ-Terme den folgenden Regeln entsprechend abgekürzt:

- Applikation ist links-assoziativ. Beispielsweise kürzen wir ((x(yz))u)v zu x(yz)uv.
- Abstraktion reicht so weit wie möglich. Beispielsweise kürzen wir λx.(x(λy.(yx))) zu λx.x(λy.yx).
- Aufeinanderfolgende Abstraktionen werden zusammengefasst. Beispielsweise kürzen wir λx.λy.λz.yx zu λxyz.yx.

λ-Kalkül

Das λ-Kalkül ist so mächtig wie eine TM! Es beschreibt die Definition von Funktionen mit ihren gebundenen Parametern. Z.B: *λx.x+2* bedeutet einfach die Abbildung *x -> x+2* (also eig. *f(x) = x+2*).

Wenn t ein Term ist, dann bezeichnet *λx.t* die anonyme Funktion, die x auf t abbildet. Allerdings ist das λ-Kalkül im Grund auch ein TES.

Grammatik: t::= x| t_1t_2 | λx.t

Applikation Abstraktion

Kontext: C(·) ::= (·) | t C(·) | C(·) t | λx.C(·)

λ-Term

Die Menge der λ-Terme $T_\lambda(V)$ über V ist induktiv definiert:

- jede Variable x ∈ V ist λ-Term.
- **Abstraktion:** Ist t ein λ-Term und x eine Variable, so ist (λx.t) λ-Term.
- **Applikation:** Sind s und t λ-Terme, so ist (s t) λ-Term

Freie Variablen

Die Menge FV(t) der freien Variablen von t ist induktiv definiert:

- FV(x) = {x}
- FV(λx.t) = FV(t) − {x}
- FV(s t) = FV(s) ∪ FV(t)

Kollisionsfreie Substitution

Die kollisionsfreie Substitution sσ ist induktiv definiert:

- xσ = σ(x)
- (t s)σ = (tσ)(sσ)
- (λx.t)σ = λx.(tσ), wenn σ(x) = x und x ∉ FV(σ(y)) für y ∈ FV(t) mit σ(y) ≠ y; andernfalls: zuerst Umbenennung gebundener Variablen per α-Äquivalenz

α-Äquivalenz

λx.t =$_\alpha$ λy.t [y/x],·wenn y ∉FV(t). Also man kann unter dieser Bedingung Variablen in bestimmten Scopes umbenennen, und es ist trotzdem Äquivalenz gegeben.

Beispiel: ***(λx.xx)(λy.yy)) =$_\alpha$ (λx.xx)(λx.xx)*** gilt, weil hier Unterschiedliche Scopes vorhanden sind. Das linke **x** auf der rechten Seite der Äquivalenz ist eine andere Variable als das rechte **x**, und kann deshalb in **y** umbenannt werden.

β-Reduktion

$(\lambda x.s)t \rightarrow_{beta} s[t/x]$

δ-Reduktion

Man setzt einfach die Lambda-Definition eines Begriffs ein. Beispiel:

flip const twice* $\rightarrow_\delta$ *(λfxy.fyx) const twice , mit ***flip = λfxy.fyx***

System F (à la Curry): Typprüfung von Termen (Klausuraufgabe)

System F (siehe Übung 12) ist die Erweiterung der Typprüfung einfach getypter Terme (aus Übung 7)

Typen, Terme:

- Typen: $\alpha, \beta ::= a \mid \alpha \rightarrow \beta \mid \forall a.\alpha$ $\quad a \in V$
- Terme: $s, t ::= x \mid s\ t \mid \lambda x.s$ $\quad x \in V$

Typisierungsregeln:

Regeln:	**Beispiel:**
$(Ax)\ \dfrac{}{\Gamma, x: \alpha \vdash x: \alpha}$	$(Ax)\ \dfrac{}{\Gamma, n: \mathbb{N} \vdash one: \mathbb{N}}$
$(\rightarrow_i)\ \dfrac{\Gamma, x: \alpha \vdash s: \beta}{\Gamma \vdash \lambda x.\, s : \alpha \rightarrow \beta}$ **Anmerkung:** Falls so etwas wie $\Gamma \vdash \lambda x.\, s : \alpha$ gegeben ist, kann man $\alpha = (\gamma \rightarrow \beta)$ substituieren.	$(\rightarrow_i)\ \dfrac{\Gamma, n : \mathbb{N} \vdash n\ (mult\ two)one: \mathbb{N}}{\Gamma \vdash \lambda n.\, n\ (mult\ two)one: \mathbb{N} \rightarrow \mathbb{N}}$
$(\rightarrow_e)\ \dfrac{\Gamma \vdash s : \alpha \rightarrow \beta \qquad \Gamma \vdash t: \alpha}{\Gamma \vdash s\ t : \beta}$ *„Links zusammen, rechts alleine“*	$(\rightarrow_e)\ \dfrac{\Gamma, n : \mathbb{N} : n(mult, two): \_?\_ \rightarrow \mathbb{N} \qquad \Gamma, n: \mathbb{N} \vdash one: ?\ \_?\_}{\Gamma, n : \mathbb{N} \vdash n\ (mult\ two)one: \mathbb{N}}$
$(\forall_i)\ \dfrac{\Gamma \vdash s: \alpha \qquad a \notin FV(\Gamma)}{\Gamma \vdash s : \forall a.\, \alpha}$	$(2x\ \forall_i)\ \dfrac{\vdash \lambda p.\, p(\lambda xy.\, x) : (a\ x\ b) \rightarrow a}{\vdash \lambda p.\, p(\lambda xy.\, x) : \forall \mathbf{ab}.\, (a\ x\ b) \rightarrow a}$ *Hier fallen einfach die Quantoren weg.*
$(\forall_e)\ \dfrac{\Gamma \vdash s: \forall a.\, \alpha}{\Gamma \vdash s : (\alpha[\beta/a])}$	$(\forall_e)\ \dfrac{\Gamma, n : \mathbb{N} \vdash n: \forall a.\, (a \rightarrow a) \rightarrow a \rightarrow a}{\Gamma, n : \mathbb{N} : n: (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow \mathbb{N} \rightarrow \mathbb{N}}$

Bei $(\rightarrow_e)$: auch hier gilt z.B. bei einer Formel *f s t* :

Links bleibt zusammen (*f s*), rechts (*t*) bleibt alleine (Linksklammern)

Bei „->“ ist es aber umgekehrt! A -> B -> C = A -> (B->C)

Beispiel 1:

Schritt 0: Aufgabenstellung

Man erinnere sich, dass die Church-Numerale in System F den Typ $\mathbb{N}$ ***:= ∀a.(a → a) → a → a*** haben. Zeigen Sie unter der Annahme, dass ***one*** und ***two*** beide den Typ $\mathbb{N}$ haben und dass ***mult*** den Typ $\mathbb{N} \to \mathbb{N} \to \mathbb{N}$ hat, dass der Term ***λn. n (mult two) one*** den Typ $\mathbb{N} \to \mathbb{N}$ hat; d.h. geben sie eine Typherleitung in System F für

$\{mult : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, one : \mathbb{N}, two : \mathbb{N}\} \vdash \lambda\, n.\ n\ (mult\ two)\ one : \mathbb{N} \to \mathbb{N}$

an.

Schritt 1.1:

$$\frac{}{\Gamma, n : \mathbb{N} \vdash n\ (mult\ two) one : \mathbb{N}}\ (\to i)$$

$$\Gamma = \{mult : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, one : \mathbb{N}, two : \mathbb{N}\} \vdash \lambda n.\ n\ (mult\ two) one : \mathbb{N} \to \mathbb{N}$$

Schritt 1.2: **Links zusammen, rechts alleine**

$$\frac{\Gamma, n : \mathbb{N} \vdash n(mult, two) : \_?\_ \to \mathbb{N} \qquad \Gamma, n : \mathbb{N} \vdash one : \_?\_}{\Gamma, n : \mathbb{N} \vdash n\ (mult\ two) one : \mathbb{N}}\ (\to e)$$

$$\frac{}{\Gamma, n : \mathbb{N} \vdash n\ (mult\ two) one : \mathbb{N}}\ (\to i)$$

$$\Gamma = \{mult : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, one : \mathbb{N}, two : \mathbb{N}\} \vdash \lambda n.\ n\ (mult\ two)\ one : \mathbb{N} \to \mathbb{N}$$

Schritt 1.3:

$$\frac{\Gamma, n : \mathbb{N} \vdash n : \_?\_ \to \mathbb{N} \to \mathbb{N} \qquad \Gamma, n : \mathbb{N} \vdash mult\ two : \_?\_}{\Gamma, n : \mathbb{N} \vdash n(mult, two) : \mathbb{N} \to \mathbb{N}}\ (\to e) \qquad \frac{}{\Gamma, n : \mathbb{N} \vdash one : \mathbb{N}}\ \mathbf{A(x)}$$

$$\frac{}{\Gamma, n : \mathbb{N} \vdash n\ (mult\ two) one : \mathbb{N}}\ (\to e)$$

$$\frac{}{\Gamma, n : \mathbb{N} \vdash n\ (mult\ two) one : \mathbb{N}}\ (\to i)$$

$$\Gamma = \{mult : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, one : \mathbb{N}, two : \mathbb{N}\} \vdash \lambda n.\ n\ (mult\ two) one : \mathbb{N} \to \mathbb{N}$$

Schritt 1.4:

$$\frac{}{\Gamma, n : \mathbb{N} \vdash n : \forall a.(a \to a) \to a \to a}\ \mathbf{A(x)} \qquad \frac{}{\Gamma, n : \mathbb{N} \vdash mult : \mathbb{N} \to \mathbb{N} \to \mathbb{N}}\ \mathbf{A(x)} \qquad \frac{}{\Gamma, n : \mathbb{N} \vdash two : \mathbb{N}}\ \mathbf{A(x)}$$

$$\frac{}{\Gamma, n : \mathbb{N} \vdash n : (\mathbb{N} \to \mathbb{N}) \to \mathbb{N} \to \mathbb{N}}\ (\forall e) \qquad \frac{}{\Gamma, n : \mathbb{N} \vdash mult\ two : \mathbb{N} \to \mathbb{N}}\ (\to e)$$

$$\frac{}{\Gamma, n : \mathbb{N} \vdash n(mult, two) : \mathbb{N} \to \mathbb{N}}\ (\to e) \qquad \frac{}{\Gamma, n : \mathbb{N} \vdash one : \mathbb{N}}\ A(x)$$

$$\frac{}{\Gamma, n : \mathbb{N} \vdash n\ (mult\ two)\ one : \mathbb{N}}\ (\to e)$$

$$\frac{}{\Gamma = \{mult : \mathbb{N} \to \mathbb{N} \to \mathbb{N}, one : \mathbb{N}, two : \mathbb{N}\} \vdash \lambda n.\ n\ (mult\ two)\ one : \mathbb{N} \to \mathbb{N}}\ (\to i)$$

Anmerkung zu der hier angewandten (∀e)-Regel: Es gilt: ***N := ∀a. (a → a) → a → a*** *(siehe Church-Kodierung [(1)]). Das steht auch in der Aufgabenstellung (Danke Julian…)*

System F: Church Kodierungen

<table>
<tr><th>Natürliche Zahlen:</th><th>Paare:</th></tr>
<tr><td>

ℕ := ∀a. (a → a) → a → a (1)
zero : ℕ
zero : λfx. X
suc : ℕ → ℕ
suc = λnfx. f (n f x)
fold : ∀a.(a → a) → a → ℕ → a
fold = λfxn. n f x
add : ℕ → ℕ → ℕ
add = λn. fold suc n

</td><td>

(a × b) := ∀r. (a → b → r) → r
pair : ∀ab. a → b → (a × b)
pair = λxyf. f x y
fst : ∀ab. (a × b) → a
fst = λp. p (λxy. x)
snd : ∀ab. (a × b) → b
snd = λp. p (λxy. y)

</td></tr>
<tr><th colspan="2">Summen:</th></tr>
<tr><td colspan="2">

(a + b) := ∀r. (a → r) → (b → r) → r
inl : ∀ab. a → (a + b) $_{a}$-> inl = λ -> xfg. f x
inr : ∀ab. b → (a + b) $_{b}$-> inr = λ -> yfg. g y
case : ∀abs. (a → s) → (b → s) → (a + b) → s
case = λfgs. s f g

</td></tr>
</table>

Algorithmus W nach Hindley und Milner (Klausuraufgabe)

1. Algorithmus mit folgenden Regeln erschöpfend anwenden, so dass sich eine Baumartige Struktur ergibt (Bei dem „=“ soll eigentlich noch ein Punkt darüber sein, aber ich hab kein Symbol dafür…)

Wir bestimmen also die Menge: PT(Γ; s; α), für einen Term s, und einen Kontext Γ.

1. PT(Γ; x; α) = {α = β | x : β ∈ Γ}
   Bsp: PT(Γ, x:b, or,e) = {e= B-> B -> B}, da B-> B -> B der Typ von **or** ist
2. PT(Γ; λx. t; α) = PT((Γ, x : a); t; b) ∪ {a → b = α} mit a, b frisch
3. PT(Γ; ts; α) = PT(Γ; t; a → α) ∪ PT(Γ; s; a) mit a frisch

Man beachte: Variablen, sowohl bestehende als auch frische, sind Platzhalter. Es gibt zwar Notation da benutzt man an bestimmten Stellen lateinische und an anderen griechische Buchstaben, man kann es aber letztendlich machen wie man will

(Frisch bedeutet dass die Variable davor noch nicht da war, und das wir sie neu einführen. Dementsprechend führt man je nach Baumgröße nicht nur a und b, sondern auch c, d, e, f …. ein)

Wichtig: bei z.B. PT ((Γ; **f s t**; α) ist das wie PT ((Γ; **(f s) t**; α) zu interpretieren. „**Links bleibt zusammen, rechts bleibt allein.“**(Klingt fast politisch)

2. Danach wenden wir den Unifikationsalgorithmus (strukturiertes Gleichmachen von Termen) auf die Blätter des Baumes an. Damit bestimmten wir den Prinzipaltyp (allgemeinster Datentyp) von unseren Term s. Hier muss meistens nach α (von PT(Γ; s; α)) auflösen.

(delete):
$$S \cup \{x \doteq x\} \quad \to S$$

(decomp):
$$S \cup \{f(E_1, \ldots, E_n) \doteq f(D_1, \ldots, D_n)\} \quad \to S \cup \{E_1 \doteq D_1, \ldots, E_n \doteq D_n\}$$

(conflict) :
$$S \cup \{f(E_1, \ldots, E_n) \doteq g(D_1, \ldots, D_k)\} \quad \to \bot \qquad (\text{für } f \neq g)$$

(orient) :
$$S \cup \{E \doteq x\} \quad \to S \cup \{x \doteq E\} \qquad (E \text{ keine Variable})$$

(occurs)/(elim) :
$$S \cup \{x \doteq E\} \quad \to \begin{cases} \bot & (x \in FV(E), x \neq E) \\ S[E/x] \cup \{x \doteq E\} & (x \notin FV(E), x \in FV(S)) \end{cases}$$

Beispiel

Schritt 0: Kontext, Term, Aufgabenstellung

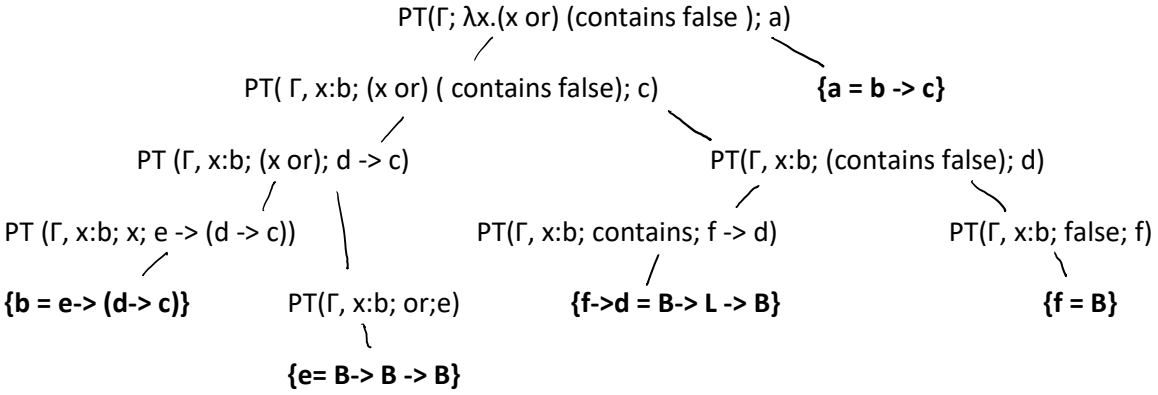
Nehmen Sie an, dass unsere Menge B von Basistypen die Typen **B** (Boolesche Werte) und *L*(Listen von Booleschen Werten) enthält.
Bestimmen Sie die **Menge PT(Γ; s; α)** aus dem Algorithmus von Hindley und Milner

- für den **Term s = λx . (x or) (contains false)**
- im **Kontext Γ** = {true : B, false : B, or : B → B → B, contains : B → L → B, nil : L, cons : B → L → L}

und geben Sie den **Prinzipaltyp von s** an

Schritt 1: Algorithmus W

Anmerkung: Wir verwenden hier nur lateinische Buchstaben, also starten wir mit **PT(Γ; s; a)**



Schritt 2: Unifizieren

{a = b -> c}, {b = e-> (d-> c)}, {f->d = B-> L -> B}, {f = B}, {e= B-> B -> B}

{a = b -> c}, {b = **e**-> (d-> c)}, {f->d = B-> L -> B}, {f = B}, {**e= B-> B -> B**}
Elim -> {a = b -> c}, {b = B-> B -> B -> (d-> c)}, {**f->d = B-> L -> B**}, {f = B}
Decomp -> {a = b -> c}, {b = B-> B -> B -> (**d**-> c)}, { **d = L -> B** },{f = B}
Elim -> {a = **b** -> c}, {**b = B-> B -> B -> (L -> B -> c)}**,{f = B}
Elim -> {a = B-> B -> B -> (L -> B -> c) -> c}, {f = B}

⇨ nach a => mgu =σ= [B-> B -> B -> (L -> B -> c) -> c} / a]

Strukturell Induktion und Folds

Typ a

SNOC und CONS

cons fügt ein Element zum Anfang einer Liste, snoc am Ende.

Definitionieren eines Datentyp:

Schreibweise: DATENTYP a = Konstr 1 | Konstr 2 | ….

Bsp: Definiere Datentyp NL a von nichtleeren Listen über Typ a

⇨ NL a = Base a| More a (NL a)

Strukturelle Induktion (Credit: Kuschelbunny <3) (unvollständig)

Beispiel 1	**Vorgehen**
Beispielaufgabe:	
data BinTree a where *Leaf : () -> BinTree a* *Bin : BinTree a -> a -> BinTree a -> 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 ⊕ (Cons x (inorder r))*	
Schritt 0	
Zu zeigende Formel: $\forall t.\ mirror\ (mirror\ t) = t$ Induktion über Variable t	Zu zeigen: FORMEL Induktion über VARIABLE. Falls es mehrere Variablen gibt: Meist erkennbar am Typ und daran dass der I.A. mit gegebenen Regeln funktioniert.
Schritt 1: I.A:	
I.A.: t = Leaf $mirror(mirror\ Leaf) = mirror\ Leaf$ $= Leaf = t$	Induktionsanfang: Für alle Konstruktoren, die den Datentyp selbst nicht als Argument haben. $I.\ A_{.1}$: VARIABLE = <Rückgabewert Konstruktor> (Beispiel Liste: Nil). Dann in Linke Seite einsetzen (LHS) und zur rechten Seite umformen. Falls nicht möglich nochmal von rechter Seite anfangen (RHS) bis zum Ergebnis der LHS einsetzen.
Schritt 2: I.V. bzw I.H.	
I.H.$_1$: ∃t1: $mirror(mirror\ t_1) = t_1$ I.H.$_2$: ∃t2: $mirror(mirror\ t_2) = t_2$	Induktionshypothese: Anzahl der Induktionsbehauptungen abhängig vom Induktionsschritt (wie viele Variablen hat der Induktionsschritt?). Wir nehmen die Induktionsbehauptung für die neuen Variablen an. $I.\ H_{.1}$: „Behauptung gilt für x“ (x ist eine neue Variable hier, gleicher Typ wie VARIABLE)

Schritt 3: I.S.	
I.S.: t = Bin t_1 a t_2 mirror (mirror t) = mirror(mirror(Bin t_1 a t_2) =mirror (Bin(mirror t2)) a (mirror t1)) = Bin mirror(mirror(t_1)) a mirror(mirror(t_2)) $=^{I.H.1+2}$ = Bin t_1 a t_2	Induktionsschritt: Für alle Konstruktoren durchführen, die nicht im Induktionsanfang sind. VARIABLE = „in den Konstruktor die Variablen aus I.H. eingesetzt (also x)“ VARIABLE in linke Seite der Induktionsbehauptung einsetzen -> Umformen bis man rechte Seite der Induktionsbehauptung hat (mit VARIABLE eingesetzt) Falls man rechte Seite nicht erreicht: Von linker Seite anfangen. (muss man in Klausuraufgaben (wahrscheinlich) immer machen)

Beispiel 2	**Vorgehen**
Beispielaufgabe:	
data Nat = Zero|Succ Zero plus :: Nat → Nat → Nat plus Zero n = n plus (Succ n) m = Succ (plus n m) mult :: Nat → Nat → Nat mult Zero n = n mult (Succ n) m = Succ (plus n m)	
Schritt 0	
Zu zeigende Formel: ∀n: Nat. mult Zero n = mult n Zero Induktion über Variable n	Zu zeigen: FORMEL Induktion über VARIABLE. Falls es mehrere Variablen gibt: Meist erkennbar am Typ und daran dass der I.A. mit gegebenen Regeln funktioniert.
Schritt 1: I.A:	
I.A.: n = Zero mult Zero n = n = Zero mult n Zero = Zero	Induktionsanfang: Für alle Konstruktoren, die den Datentyp selbst nicht als Argument haben. $I.A_{.1}$: VARIABLE = <Rückgabewert Konstruktor> (Beispiel Liste: Nil). Dann in Linke Seite einsetzen (LHS) und zur rechten Seite umformen. Falls nicht möglich nochmal von rechter Seite anfangen (RHS) bis zum Ergebnis der LHS einsetzen.
Schritt 2: I.V. bzw I.H.	
I.V: ∃n : mult zero n = mult n zero	Induktionshypothese: Anzahl der Induktionsbehauptungen abhängig vom Induktionsschritt (wie viele Variablen hat der Induktionsschritt?). Wir nehmen die Induktionsbehauptung für die neuen Variablen an. $I.H_{.1}$: „Behauptung gilt für x“ (x ist eine neue Variable hier, gleicher Typ wie VARIABLE)

Schritt 3: I.S.	
I.S.: n = succ , ⇨ Rückwärts aufwickeln z.Z. mult succ(m) zero = mult zero succ (m) succ (plus m zero) = succ (m) succ (plus zero m) = succ (m)	Induktionsschritt: Für alle Konstruktoren durchführen, die nicht im Induktionsanfang sind. VARIABLE = „in den Konstruktor die Variablen aus I.H. eingesetzt (also x)“ VARIABLE in linke Seite der Induktionsbehauptung einsetzen -> Umformen bis man rechte Seite der Induktionsbehauptung hat (mit VARIABLE eingesetzt) Falls man rechte Seite nicht erreicht: Von linker Seite anfangen. (muss man in Klausuraufgaben (wahrscheinlich) immer machen)
plus m zero = plus zero m (induktion über m) I.A: m = zero plus zero zero = plus zero zero I.V. ∃x : plus x zero = plus zero x IS: m = succ (x) plus (succ x) zero = succ (plus x zero) $=^{IV}$ succ (plus zero x) = succ(x) = m ⇨ plus zero m = m	Da wir nicht weiter kommen => Zwischenbeweis

Beispiel 3	**Vorgehen**
Beispielaufgabe:	
data List a = Nil | Cons a (List a) snoc Nil y = Cons y Nil snoc (Cons x xs) y = Cons x (snoc xs y) Nil ⊕ ys = ys (Cons x xs) ys = Cons x (xs ⊕ ys)	
Schritt 0	
Zu zeigende Formel: ∀e,xs,ys: xs ⊕ (Cons e ys) = (snoc xs e) ⊕ ys. **Induktion über xs**	Zu zeigen: FORMEL Induktion über VARIABLE. Falls es mehrere Variablen gibt: Meist erkennbar am Typ und daran dass der I.A. mit gegebenen Regeln funktioniert.
Schritt 1: I.A:	
I.A.: xs = Nil xs ⊕ (Cons e ys) = Nil ⊕ (Cons e ys) =Cons e ys = Cons e (Nil ⊕ ys) = (Cons e Nil) ⊕ ys = (snoc Nil e) ⊕ ys	Induktionsanfang: Für alle Konstruktoren, die den Datentyp selbst nicht als Argument haben. $I.A_{.1}$: VARIABLE = <Rückgabewert Konstruktor> (Beispiel Liste: Nil). Dann in Linke Seite einsetzen (LHS) und zur rechten Seite umformen. Falls nicht möglich

	nochmal von rechter Seite anfangen (RHS) bis zum Ergebnis der LHS einsetzen.
Schritt 2: I.V. bzw I.H.	
I.V.: ∃ e, zs, ys = zs ⊕ (Cons e ys) = .. Wobei xs auf zs, also eine Konstante, reduzieren!	Induktionshypothese: Anzahl der Induktionsbehauptungen abhängig vom Induktionsschritt (wie viele Variablen hat der Induktionsschritt?). Wir nehmen die Induktionsbehauptung für die neuen Variablen an. $I.H._1$: „Behauptung gilt für x“ (x ist eine neue Variable hier, gleicher Typ wie VARIABLE)
Schritt 3: I.S.	
I.S.: xs = Cons z zs $(\text{cons z zs}) \oplus (\text{Cons e ys}) = \text{Cons z } (\text{zs} \oplus (\text{cons e ys})$ $=^{IV} \text{cons z } ((\text{snoc zs e}) \oplus \text{y}) = \text{Cons z } (\text{snoc zs e}) \oplus \text{ys} = (\text{snoc } (\text{Cons z zs}) \text{ e}) \oplus \text{ ys}$	Induktionsschritt: Für alle Konstruktoren durchführen, die nicht im Induktionsanfang sind. VARIABLE = „in den Konstruktor die Variablen aus I.H. eingesetzt (also x)“ VARIABLE in linke Seite der Induktionsbehauptung einsetzen -> Umformen bis man rechte Seite der Induktionsbehauptung hat (mit VARIABLE eingesetzt) Falls man rechte Seite nicht erreicht: Von linker Seite anfangen. (muss man in Klausuraufgaben (wahrscheinlich) immer machen)

Folds (evtl. noch zu Überarbeiten)

Vorgehen:

- fold braucht für jeden Konstruktor eine Funktion als Argument, und für jeden Konstruktor eine Definition (Konstruktoren ohne Argument ersetzt man dann dort einfach durch einen Buchstaben, ansonsten gilt die Curry -> Schreibweise)
- Falls eine Konstruktor in der Mitte ein Rekursionsaufruf hat, muss dort eine rekursiver Fold-Aufruf hin

Beispiel:
Schritt 0: Aufgabenstellung, Datentyp, fold
data Twins a = End a a | More a (**Twins** a) a Geben Sie die fold-Funktion für Twins, foldtw, inklusive ihres Typs an. Nutzen Sie diese, um eine Funktion: *identicalTwins* ***: Twins*** *a* → ***List*** *a* zu definieren, die Elemente nur dann in die Zielliste übernimmt, wenn die beiden Werte übereinstimmen. Beispielsweise soll gelten: *identicalTwins (More 1 (More 2 (More 1 (End 2 0) 1) 2) 3) = [2,1]* **Hinweis:** Sie dürfen die üblichen Konstrukte wie λ-Abstraktion, if . . . then . . . else verwenden und annehmen, dass Elemente mittels == auf Gleichheit überprüft werden können.
Schritt 0:
2 Konstruktoren, dass hei0t wir haben die Form foldtw : e m **Twins** a-> _, e für den 1. Konstruktor, m für den 2. Konstruktor, und _ ist die Rückgabe von den Funktionen e und m. **Den Typ Twins a benennen wir in b um, als Rückgabetyp (das muss gemacht werden)** **Bei Konstruktoren ohne Parameter schreibt man nur den Rückgabewert (=b) zurück!**
Schritt 1:
-End Konstruktor *(End* ***a a*** *)* hat zwei Argumente d.h. die Funktion e im fold muss auf beide Argumente angewendet werden, damit hat sie den Typ (**a**->**a**->b)
Schritt 2:
-More Konstruktor *(More a (****Twins*** *a) a)* bekommt in der Mitte ein Element des Typs Twins a, d.h. da muss bei der Definition der rekursive Aufruf von fold hin. -More Konstruktor hat 3 Argumente, wobei das mittlere den Typ b durch die rekursive Auswertung von fold hat, damit hat die Funktion m den Typ (a->b->a->b)
Schritt 3: fold f_1 f_2 <Datentyp>
Jetzt haben wir aus Konstruktor 1 (a->a->b) und aus Konstruktor 2(a->b->a->b) foldtw : (a->a->b)->(a->b->a->b)->**Twins** a-> b foldtw f_1 f_2 (End x y) := f_1 x y (1) foldtw f_1 f_2 (More x t y) := f_2 x (foldtw f_1 f_2 t) y (2) Hier muss dann (1) und (2) definiert werden, in dem wir die Konstruktoren in unser foldtw an dem dritten Parameter einsetzen, und dann das Verhalten beschreiben.
Schritt 4:
identicalTwins x = foldtw f_1 f_2 x (mit x : Twins a) f_1 x y = if x ==y then (Cons x Nil) else Nil f_2 x y z = if x==z then (Cons x (identicalTwins y)) else identicalTwins y)

Korekursion und Koinduktion

Anmerkungen

Bei diesem Aufgabentyp geht es immer um unendliche Datenstrukturen (Streams), und den aktuell betrachteten Schritt (mit hd), und den nächsten Schritt (mit tl).

Bisimulation

Eine Relation $R \subseteq A^\omega \times A^\omega$ heißt Bisimulation, wenn für alle $(s, t) \in R$ gilt:

- hd s = hd t (hd greift auf die geraden Elemente des Streams zu)
- (tl s) R (tl t) (tl greift auf die ungeraden Elemente des Streams zu)

Anwendung des Begriffs auf ein Beispiel:

codata IntStream where
head : IntStream → Int
tail : IntStream → IntStream.

Eine Relation R ⊆ IntStream x IntStream heißt Bisimulation, wenn für alle (s,t) aus R gilt:

- head s = head t
- (tail s) R (tail t)

Korekursion

Wir definieren rekursiv eine Funktion f mithilfe einer anderen Funktion g, deshalb **Ko**rekursion (wie auch in z.B. Koevolution, Kooperation etc.). Das ist meistens der einfache Teil.

Koinduktion

Hier beweisen wir unsere Definierung der Funktion aus der Korekursion. Das ist etwas komplizierter als Koreduktion, und gibt meistens auch mehr Punkte.

Vorgehen anhand Beispiel

Beispiel – Aufgabenstellung

Ein (digitales) Signal ist ein zeitlich veränderlicher Wert zwischen $-\infty$ und $+\infty$, wobei wir die Zeit als diskret behandeln, entsprechend etwa fortgesetztem Sampling. Solche Signale lassen sich in natürlicher Weise durch einen Kodatentyp repräsentieren:

codata Signal where
currentSample : **Signal → Int**
discardSample : **Signal → Signal**

Z.B. repräsentiert gemäß den folgenden korekursiven Definitionen flat x ein konstantes Signal mit Wert x und square x y ein zwischen x und y alternierendes Signal:

currentSample (flat x) = x
discardSample (flat x) = flat x
currentSample (square x y) = x
discardSample (square x y) = square y x

Unteraufgabe - Korekursion

Definieren Sie korekursiv eine Funktion *sampler :* **Signal → Signal → Signal**, so dass der Wert des Signals *sampler t* s der Wert von s ist, wenn der Wert von t größer als 0 ist, und 0 sonst. Insbesondere sollten z.B. die Gleichungen

sampler (square 0 1) (square x 0) = flat 0 ***(1)***

und (für alle x)

sampler (square 1 0) (flat x) = square x 0 ***(2)***

gelten. **Hinweis:** Sie dürfen bei der Definition die üblichen Operationen auf Zahlen (Addition, Vergleich etc.) und auf Booleans (and, or, if-then-else etc.) als gegeben annehmen.

Beispiel:	**Vorgehen**
currentSample (sampler t s) = if currentSample(t) > 0 then currentSample (s) else 0 discardSample (sampler t s) = sampler (discardSample(z) discardSample(s))	Wir setzen nun die **definierende Funktion** (sampler) in die **gegebenen Funktionen** (currentSample = cS, discardSample = dS) ein. Typischerweise sind diese gegebene Funktion immer nach dem Prinzip „aktuellen Wert des Streams betrachten“ (cS = hd) und „nächsten Wert des Streams betrachten“ (dS = tl) Dann definieren wir mit den erlaubten Konstrukten wie sich nun die Funktionen verhalten sollen (wir programmieren sie).
Unteraufgabe - Koinduktion	
Beweisen Sie dann per Koinduktion, dass für Ihre Definition von *sampler* tatsächlich die beiden Gleichungen **(1)** und **(2)** gelten. Erläutern Sie alle Beweisschritte.	
Für (1) *sampler (square 0 1) (square x 0) = flat 0:* z.z. R ist Bisimulation Sei R = {sampler (sq 0 1)(sq x 0), flat 0}	Wir müssen für alle **Gleichungen** (also (1) und (2)) zeigen, das R eine Bisimulationen ist. R = {linke Seite der Gleichung, rechte Seite der Gleichung}
> cS (sampler (sq 0 1) (sq x 0)) = =if cS(sq 0 1) > 0 then cs (sq x 0) else 0 =if 0 > 0 then x else 0 = 0 =**cS**(flat 0)	Jetzt setzen wir diese Seiten der Gleichung in die **gegebenen Funktionen** (hier cS und dS) und setzen das für jede Funktion gleich, und lösen auf. Für cS geht das auch sehr schön auf, für dS nicht.
k = dS ((sq 0 1) (sq x 0)) = sampler (sq 1 0)(sq 0 x) h = ds (flat 0) = flat 0	Das es für dS nicht so einfach aufgeht, beweisen wir für die dS das ganze rekursiv über einen Hilfsbeweis, wieder über eine Bisimulation R‘=R ∪ {…,….} In diesem beweisen wir das dann auch wieder für beide gegebenen Funktion (dS und cS). Eventuell sind auch hier Hilfbeweise erforderlich!
Hilfsbeweis: z.z. R ist Bisimulation Sei R‘ = R ∪ {k, h}= R ∪{sampler (sq 1 0) (sq 0 x), flat 0} > cs (sampler (sq 1 0) (sq 0 x)) =…= 0 = cS(flat 0) >ds (sampler (sq 1 0) (sq 0 x)) = sampler (sq 0 1)(sq x 0) R flat 0 = dS (flat 0)	
k = …. = sampler (sq 1 0)(sq 0 x)R‘ flat 0 = ds (flat 0) = h	Jetzt haben wir auch für dS die Gültigkeit induktiv bewiesen.
Für (2) *sampler (square 1 0) (flat x) =square x 0:* Sei R = {sampler (sq 1 0) (flat x), square x 0}	Hier das ganze nochmal für die zweite Gleichung
> cs (sampler (sq 1 0) (flat x)) = if 1 > 0 then x else 0 = x = cS(square x 0)	
f = ds (sampler (sq 1 0)(flat x)) = = sampler (sq 0 1)(flat x) g = ds (sq x 0) = sq 0 x	
Hilfsbeweis: Sei R‘ = R ∪ {sampler (sq 0 1)(flat x), sq 0 x}	

> cs (sampler (sq 0 1) (flat x))= =if 0 > 0 then x else 0 = 0 = cs (sq 0 x) > ds (sampler (sq 0 1)(flat x)) = sampler (sq 1 0)(flat x) R‘ sq x 0 = ds(sq 0 x)	
f = … = sampler (sq 0 1) (flat x) R’sq 0 x = ds (sq x 0) = g	

Reguläre Sprachen / Reguläre Automaten (Übung 12)

Minimierung eines Automaten mit tabellarisch gegebenen Zustandsüberführungsfunktion

Minimieren des folgenden DFA A = ({ q, r, s, t, u, v }, { a, b }, δ, q, { t, u })Mit der tabellarisch gegebenen Zustandsüberführungsfunktion.

- **Q: Menge aller Zustände** ={ q, r, s, t, u, v },
- **Σ: Endliches Eingabealphabet** = { a, b },
- **δ: Übergangsfunktion (**bzw. hier als Tabelle) = siehe Tabelle
- **q_0: Startzustand** = q,
- **F: Endzustände** = { t, u }

Tabelle:

δ	***q***	***r***	***s***	***t***	***u***	***v***
a	r	r	t	z	t	q
b	s	s	u	r	q	s

Tabellarischer Algorithmus zur Minimierung eines DFA

(Minimiere einen DFA A = (Q ,Σ, δ, s, F)) Der Algorithmus verwendet eine globale Variable $R \subseteq Q \times Q$. Er läuft wie folgt:

1. Entferne aus Q alle nicht erreichbaren Zustände.
2. Initialisiere R auf $\{(q_1, q_2) \mid q_1 \in F \Leftrightarrow q_2 \in F\}$
3. Suche ein Paar $(q_1, q_2) \in R$ und einen Buchstaben $a \in \Sigma$ mit
   $(\delta(a, q_1), \delta(a, q_2)) \notin R$.

   3.1 Wenn kein solches Paar gefunden wird, gehe zu Schritt 4.

   3.2 Andernfalls entferne (q_1, q_2) aus R und fahre bei 3. fort.

4. Identifiziere alle Zustandspaare in R.

Wenn man den Algorithmus von Hand verwendet, sollte man ausnutzen, dass offenbar R stets symmetrisch ist und alle Paare (q, q) stets in R verbleiben. Es reicht also, eine dreieckige Tabelle zu führen, in der jedes Paar nur in einer der beiden möglichen Anordnungen vorkommt, und die Diagonalelemente (q, q) gar nicht. Man streicht dann zunächst die Zustandspaare, die hinsichtlich Finalität nicht übereinstimmen, und streicht dann wiederholt Paare heraus, die nach dem Kriterium in Schritt 3 aus R zu entfernen sind.

Felitzscher Algorithmus (Das ist einfach der obige Algorithmus auf Deutsch…):

Anmerkung: „Markiere“ bedeutet dass wir einfach eine Zahl herein schreiben. Diese kann z.B. die Schritt-Zahl des Algorithmus machen.

1. Entferne aus Q alle nicht erreichbaren Zustände => In der Tabelle können diese Zustände ignoriert werden
2. Markiere in der Tabelle den Eintrag (q_1, q_2) , falls gilt $\{(q_1, q_2) \mid q_1 \notin \text{und } q_2 \in F\}$
   Also muss ein Zustand ein Endzustand sein, der andere nicht! Die Markierung ist hier üblicherweise 0.

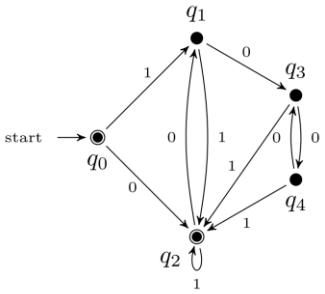
3. Jetzt betrachten wir alle Einträge (q_1, q_2), die noch nicht markiert worden sind. Markierung ist hier üblicherweise 1.
   Für jeden nicht-markierten Eintrag (q_1, q_2), mache, solange bis sich nichts mehr verändert! (***Das heißt schon betrachtete nicht markierte Einträge müssen mehrmals betrachtet werden***)
   Für jede Eingabe a, prüfe ob:
   Falls (δ(a, q_1), δ(a, q_2)) markiert ist, markiere auch (q_1, q_2).

   Gehe zu Schritt 4.

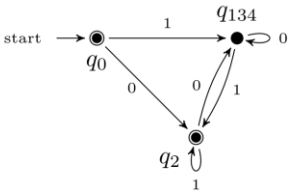
4. Identifiziere alle Zustandspaare in R. Stellen die noch unmarkiert sind geben die Knoten, die zusammengefasst werden, bekannt => Wenn an der stelle (q_0, q_1) in der Tabelle noch was frei ist, dann werden die Zustände q_0 und q_1 zusammen gefasst

Hilfreiches Video: https://www.youtube.com/watch?v=KkEoqDtyMAs

Beispiel:



q_1	0			
q_2	1	0		
q_3	0		0	
q_4	0		0	
	q_0	q_1	q_2	q_3



Anhand der Tabelle erkennt man: q_1, q_3 und q_4 sind gleich (haben gleiche Einträge in Tabelle)

⇨ Können zusammengefasst werden

Altklausuren

SS 2018

Aufgabe 1: Konfluenz und Terminierung

1.

$\Sigma = \{-/1;\ [\,]/3\}$

$p_-(x) = x + k \overset{k=1}{=} x+1$

$p_{[]}(x,y,z) = x+y+z$

$T_{1,L} \to x + \overbrace{(y+1)}^{p_-(y)} + z \quad = \quad p_{[]}(x, p_-(y), z)$

$T_{1,R} \to x+z+y$ Es muss gelten: $T_L > T_R$ und Definitionsbereich $\in \mathbb{N}$

$x+(y+1)+z > x+z+y$

$\Rightarrow$ kürzen $\Rightarrow 1 > 0 \Rightarrow$ für 1. Term gelb

$T_{2,L} \to p_-(p_{[]}(x,y,z)) = p_-(x+y+z) = (x+y+z)+1$

$T_{2,R} \to p_{[]}(p_-(x), p_-(y), p_-(z)) = p_{[]}(x+1, y+1, z+1) = x+1+y+1+z+1$

$= x+y+z+3$

$\Rightarrow$ NN klappt nicht

$p_-(x) = x^2$; $p_{[]}(x,y,z) = x+y+z$

$T_{1,L} \to x + y^2 + z \to T_{1,R} = x+y+z$ ✓ für alle $y \geq 1$

$T_{2,L} \to (x+y+z)^2 \to T_{2,R} = x^2+y^2+z^2$ ✓

$(= x^2+xy+x\cdot z+y\cdot x+y^2+y\cdot z+z\cdot x+z\cdot y+z^2)$

$x\cdot y + x\cdot z + y\cdot x + y\cdot z + z\cdot x + z\cdot y > 0$ $\to$ gilt für alle $x,y,z > 0$

$\Rightarrow$ ist stark normalisierend, wenn $y \geq 1$ u. $x,z > 0$, für $A = \mathbb{N} \setminus \{0\}$!!!

2.

$[x, \bar{y}, z] \to_0 [x, z, y]$ $\quad = 4$ mögliche krit. Paare

$\overline{[u, v, w]} \to_0 [\bar{u}, \bar{v}, \bar{w}]$

Oder auch:

$[x, -y, z] \to_0 [x, z, y]$

$-[u, v, w] \to_0 [-u, -v, -w]$

2 mit 1 $\quad t_1 = [x, \bar{y}, z]$

$\quad \lhd t_2 = \overline{[u, v, w]}$ $\quad \text{mgu} = \sigma = [x/u, \bar{y}/v, z/w]$ $\quad p(\cdot)$

$\{[\bar{x}, \bar{\bar{y}}, \bar{z}],\ \overline{[x, z, y]}\}$ $\quad C(\cdot) =$

Die Teile des kritischen Paares sind nicht zusammen. Tabulation $\Rightarrow$ nicht lokal konfluent $\Rightarrow$ nicht konfluent

1 mit 2 $\quad \lhd t_2 \overline{[u, v, w]}$ $\quad \text{mgu} = \sigma = [[u, v, w]/\bar{y}]$

$= \sigma \langle [x, z, [u, v, w]], [x, [\bar{u}, \bar{v}, \bar{w}], z] \rangle$

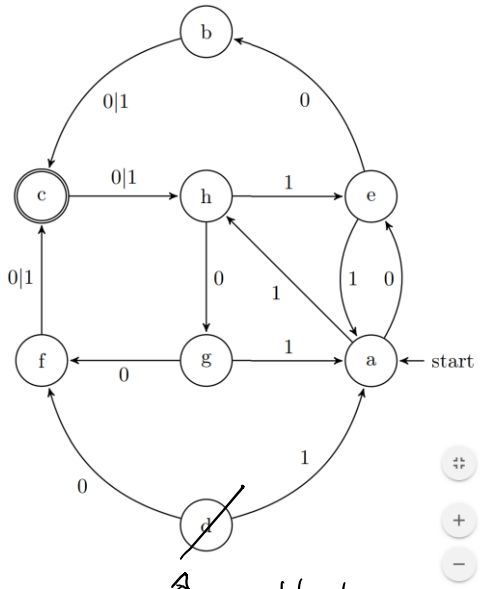
auch nicht zusammentabulierbar

Aufgabe 4: Korekursion und Kondiktion:

```
codata StreamNat a where
  hd : StreamNat a -> a
  tl : StreamNat a -> StreamNat a
```

```
hd(x,y) =
```

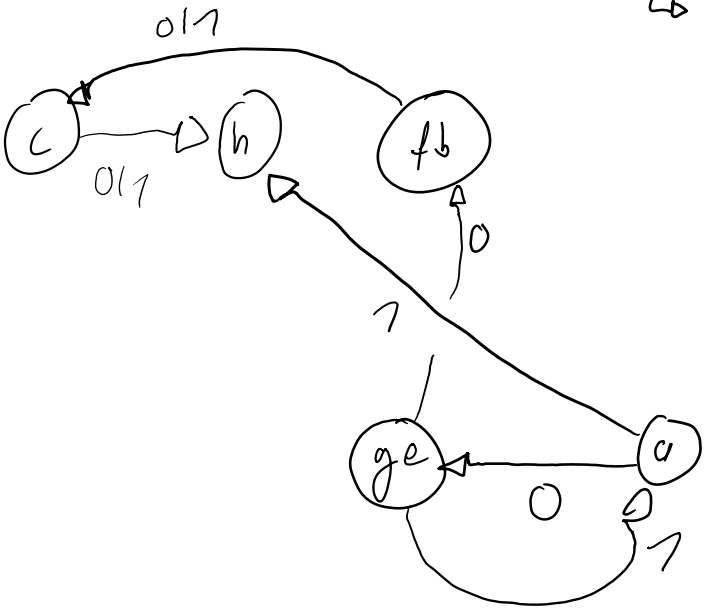
Aufgabe 5: Automat



unerreichbar!

	a	b	c	e	f	g	h
a							
b	1						
c	0	0					
e	1	1	0				
f	1	X	0	1		→ fb	
g	1	1	0	X	1		→ ge
h	1	1	0	1	1	1	

↳ Tabelle wurde überprüft



SS 2016

Aufgabe 1: Konfluenz

1.) $\Sigma = \{ / \; 2 \; ; \; o / 2 \}$

Für „/“: $p_/(x,y) = x + 2y$

Für „o“: $p_o(x,y) = x + y$

T1: $(a+2b) + (a+2c) \to_o b + (c+2a) \quad = D \quad b+c > 0$

T2: $(a+a) + 2b \to b+b \quad = D \quad a+a > 0$

T3: $a + 2(b+2c) \to_o a+2(b+c) \quad = D \quad 2c > 0$

wenn $(p_/(x,y) = x+2y+2$, dann auch für alle $///)$

2.) $(a/b) \circ (a/c) \to_o b(c/a)$ (1)

$(d \circ d)/e \to e \circ e$ (2)

$f/(g/h) \to f/(g \circ h)$ (3)

(3 und 5 kritisches Paar bilden) → nicht zusammenführbar

SS 2017

Aufgabe 1: Konfluenz und Terminierung

(μ x) ∵ y = y ∵ x (1.1)

(x ∵ y) ∵ z = z ∵ (y ∵ (μ x)) (1.2)

(x ∵ (μ (μ y))) = (x ∵ (μ y)) ∵ y (1.3)

1. Nicht stark normalisierend, weil:

Sei (μ x ∵ (μ(μ y)))

Dann: (μ x ∵ (μ(μ y))) –(1.3)-> (μ x ∵ μ y) ∵ y –(1.2)-> y∵ (μ y ∵ (μ (μ x)))

–(1.3)-> y ∵ ((μ y ∵ (μ x)) ∵ y) –(1.2)-> y ∵ (x∵(μ x ∵ μ (μ y))))

2. Jetzt gilt dieses TES. Es ist nun stark normalisierend. Ist es konfluent?

(μ x) ∵ y = y ∵ x (1.1)

(x ∵ y) ∵ z = z ∵ (y ∵ x) (1.2)

(x ∵ (μ (μ y))) = (x ∵ (μ y)) ∵ y (1.3)

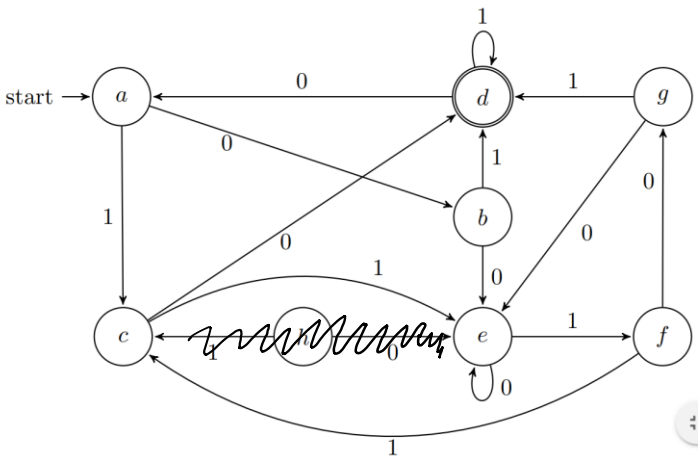
- ⇨ Newmans Lemma: Wenn ein TES stark normalisierend und lokal konfluent ist, ist es konfluent.
- ⇨ Lokale Konfluenz zeigen
- ⇨ Kritische Paare bilden

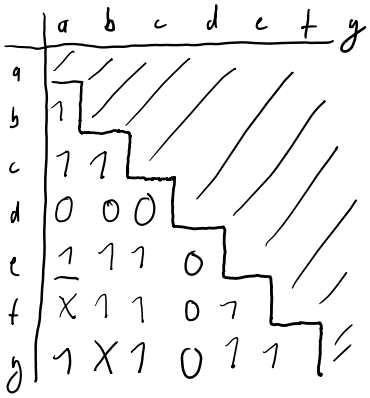
1) Variablen umbenennen

(μ x) ∵ y = y ∵ x (1.1)
(w ∵ v) ∵ z = z ∵ (v ∵ w) (1.2)
(t ∵ (μ (μ s))) = (t ∵ (μ s)) ∵ s (1.3)

2)

Aufgabe 5: Automaten





↳ Struveped hats auch so

⇒ f und g zusammen fassen

⇒ g und b