PROGRAMMIERUNG

ÜBUNG 5: UNIFIKATION & INDUKTION AUF LISTEN

Eric Kunze
eric.kunze@tu-dresden.de

TU Dresden, 11. Mai 2022

letzte Änderung:
11.05.2022, 16:21

INHALT

Unifikation & Unifikationsalgorithmus

Aufgabe 1

ERINNERUNG: TYPPOLYMORPHIE

- **bisher**: Funktionen mit konkreten Datentypen
  z.B. `length :: [Int] -> Int`
- **Problem**: Funktion würde auch auf anderen Datentypen funktionieren
  z.B. `length :: [Bool] -> Int` oder `length :: String -> Int`
- **Lösung**: Typvariablen und polymorphe Funktionen
  z.B. `length :: [a] -> Int`

ERINNERUNG: TYPPOLYMORPHIE

- **bisher**: Funktionen mit konkreten Datentypen
  z.B. `length :: [Int] -> Int`
- **Problem**: Funktion würde auch auf anderen Datentypen funktionieren
  z.B. `length :: [Bool] -> Int` oder `length :: String -> Int`
- **Lösung**: Typvariablen und polymorphe Funktionen
  z.B. `length :: [a] -> Int`

Bei konkreter Instanziierung wird Typvariable an entsprechenden Typbezeichner gebunden (z.B. `a = Int` oder `a = Bool`).

- Der Aufruf `length [1,5,2,7]` liefert für die Typvariable `a = Int`.
- Der Aufruf `length [True, False, True, True, False]` liefert die Belegung `a = Bool`.
- Der Aufruf `length "hello"` impliziert `a = Char`.

UNIFIKATION

Motivation: Typüberprüfung

```
f :: (t, Char) -> (t, [Char])
f (...) = ...

g :: (Int, [u]) -> Int
g (...) = ...

h = g . f
```

UNIFIKATION

Motivation: Typüberprüfung

```
f :: (t, Char) -> (t, [Char])
f (...) = ...

g :: (Int, [u]) -> Int
g (...) = ...

h = g . f
```

Wie müssen die Typvariablen `t` und `u` belegt werden, damit die Funktion `h` wohldefiniert ist, d.h. damit die Ergebnisse aus `f` wirklich in `g` eingesetzt werden dürfen?

TYPAUSDRUCK $\rightarrow$ TYPTERM

Ziel: theoretischere Form von Typausdrücken

Typausdrücke

- `Int`, `Bool`, `Float`, `Char`, `String`
- Typvariablen
- Listen, Tupel, Funktionen

Typterme

- Übersetzung *trans*: Typausdruck $\rightarrow$ Typterm

TYPAUSDRUCK $\rightarrow$ TYPTERM

Ziel: theoretischere Form von Typausdrücken

Typausdrücke

- `Int`, `Bool`, `Float`, `Char`, `String`
- Typvariablen
- Listen, Tupel, Funktionen

Typterme

- Übersetzung *trans*: Typausdruck $\rightarrow$ Typterm
- z.B.

$$\mathit{trans}(\texttt{(t,[Char])}) = ()^2(t, [](\mathit{Char}))$$
$$\mathit{trans}(\texttt{(Int, [u])}) = ()^2(\mathit{Int}, [](u))$$

TYPAUSDRUCK $\rightarrow$ TYPTERM

Ziel: theoretischere Form von Typausdrücken

Typausdrücke

- `Int`, `Bool`, `Float`, `Char`, `String`
- Typvariablen
- Listen, Tupel, Funktionen

Typterme

- Übersetzung *trans*: Typausdruck $\rightarrow$ Typterm
- z.B.

$$\mathit{trans}(\texttt{(t,[Char])}) = ()^2(t, [](\mathit{Char}))$$
$$\mathit{trans}(\texttt{(Int, [u])}) = ()^2(\mathit{Int}, [](u))$$

Beide Typausdrücke können in Übereinstimmung gebracht werden, wenn die Typterme *trans*(`(t,[Char])`) und *trans*(`(Int, [u])`) unifizierbar sind.

TYPAUSDRUCK $\rightarrow$ TYPTERM

Ziel: theoretischere Form von Typausdrücken

Typausdrücke

- `Int`, `Bool`, `Float`, `Char`, `String`
- Typvariablen
- Listen, Tupel, Funktionen

Typterme

- Übersetzung *trans*: Typausdruck $\rightarrow$ Typterm
- z.B.

$$\mathit{trans}(\texttt{(t,[Char])}) = ()^2(t, [](\mathit{Char}))$$
$$\mathit{trans}(\texttt{(Int, [u])}) = ()^2(\mathit{Int}, [](u))$$

Beide Typausdrücke können in Übereinstimmung gebracht werden, wenn die Typterme *trans*(`(t,[Char])`) und *trans*(`(Int, [u])`) unifizierbar sind.

$\rightarrow$ `t = Int` und `u = Char`

UNIFIKATIONSALGORITHMUS

- **gegeben:** zwei Typ*terme* t_1, t_2
- **Ziel:** entscheide, ob t_1 und t_2 unifizierbar sind

Wir notieren die beiden Typterme als Spalte:

$$\begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \qquad \text{bzw.} \qquad \begin{pmatrix} ()^2(t, [](Char)) \\ ()^2(Int, [](u)) \end{pmatrix}$$

Unifikationsalgorithmus erstellt eine Folge von Mengen M_i, wobei die M_{i+1} aus M_i hervorgeht, indem eine der vier Regeln angewendet wird.

$$M_1 := \left\{ \begin{pmatrix} t_1 \\ t_2 \end{pmatrix} \right\} \qquad \text{bzw.} \qquad M_1 := \left\{ \begin{pmatrix} ()^2(t, [](Char)) \\ ()^2(Int, [](u)) \end{pmatrix} \right\}$$

UNIFIKATIONSALGORITHMUS – REGELN

- **Dekomposition.** Sei $\delta \in \Sigma$ ein k-stelliger Konstruktor, $s_1, \ldots, s_k, t_1, \ldots, t_k$ Terme über Konstruktoren und Variablen.
$$\begin{pmatrix} \delta(s_1, \ldots, s_k) \\ \delta(t_1, \ldots, t_k) \end{pmatrix} \quad \leadsto \quad \begin{pmatrix} s_1 \\ t_1 \end{pmatrix}, \ldots, \begin{pmatrix} s_k \\ t_k \end{pmatrix}$$
- **Elimination.** Sei x eine Variable !
$$\begin{pmatrix} x \\ x \end{pmatrix} \quad \leadsto \quad \emptyset$$
- **Vertauschung.** Sei t keine Variable.
$$\begin{pmatrix} t \\ x \end{pmatrix} \quad \leadsto \quad \begin{pmatrix} x \\ t \end{pmatrix}$$
- **Substitution.** Sei x eine Variable, t keine Variable.
Occur Check: x kommt nicht in t vor
Dann ersetze in jedem anderen Term die Variable x durch t.
$$\begin{pmatrix} x \\ t \end{pmatrix}, \begin{pmatrix} y \\ s(x) \end{pmatrix} \quad \leadsto \quad \begin{pmatrix} x \\ t \end{pmatrix}, \begin{pmatrix} y \\ s(t) \end{pmatrix}$$

UNIFIKATIONSALGORITHMUS

Ende: keine Regel mehr anwendbar – Entscheidung:

- t_1, t_2 **unifizierbar**: M ist von der Form

  $$\left\{ \begin{pmatrix} u_1 \\ t_1 \end{pmatrix}, \begin{pmatrix} u_2 \\ t_2 \end{pmatrix}, \ldots, \begin{pmatrix} u_k \\ t_k \end{pmatrix} \right\}$$

  „Variablen“
  „Terme ohne Variablen“

  wobei $u_1, u_2, \ldots, u_k$ paarweise verschiedene Variablen sind und nicht in $t_1, t_2, \ldots, t_k$ vorkommen.

  allgemeinster Unifikator φ:

  $$\varphi(u_i) = t_i \qquad (i = 1, \ldots, k)$$

  $$\varphi(x) = x \quad \text{für alle nicht vorkommenden Variablen}$$

- t_1, t_2 sind **nicht unfizierbar**: M hat nicht diese Form und keine Regel ist anwendbar

Weitere Unifikatoren ψ erhält man durch Anwendung einer Substitution σ, sodass $\psi = \tilde{\sigma} \circ \varphi$, wobei $\tilde{\sigma}$ die Erweiterung von σ ist, die durch „Einsetzen“ definiert ist (siehe Skript).

OCCUR CHECK

Um endlose Rekursionen zu unterbinden, benötigen die Regeln zum Vertauschen und zur Substitution gewisse Einschränkungen.

Occur Check: Gegeben sei ein Termpaar $\binom{x}{t}$, wobei x eine Variable und t ein Typterm sei.

- Kommt x in t vor, dann schlägt der Check fehl.
- Kommt x nicht in t vor, dann ist der Check okay.

OCCUR CHECK

Um endlose Rekursionen zu unterbinden, benötigen die Regeln zum Vertauschen und zur Substitution gewisse Einschränkungen.

Occur Check: Gegeben sei ein Termpaar $\binom{x}{t}$, wobei x eine Variable und t ein Typterm sei.

- Kommt x in t vor, dann schlägt der Check fehl.
- Kommt x nicht in t vor, dann ist der Check okay.

Beispiel:

- $\begin{pmatrix} x_1 \\ \gamma(x_1) \end{pmatrix} \rightsquigarrow$ Fehlschlag, da x_1 in $\gamma(x_1)$ vorkommt
- $\begin{pmatrix} x_1 \\ \gamma(x_2) \end{pmatrix} \rightsquigarrow$ okay, da x_1 nicht in $\gamma(x_2)$ vorkommt

OCCUR CHECK

Um endlose Rekursionen zu unterbinden, benötigen die Regeln zum Vertauschen und zur Substitution gewisse Einschränkungen.

Occur Check: Gegeben sei ein Termpaar $\binom{x}{t}$, wobei x eine Variable und t ein Typterm sei.

- Kommt x in t vor, dann schlägt der Check fehl.
- Kommt x nicht in t vor, dann ist der Check okay.

Beispiel:

- $\begin{pmatrix} x_1 \\ \gamma(x_1) \end{pmatrix} \rightsquigarrow$ Fehlschlag, da x_1 in $\gamma(x_1)$ vorkommt
- $\begin{pmatrix} x_1 \\ \gamma(x_2) \end{pmatrix} \rightsquigarrow$ okay, da x_1 nicht in $\gamma(x_2)$ vorkommt

Was passiert, wenn wir substituieren obwohl der Occur Check fehlschlägt?

$$\begin{pmatrix} x_1 \\ \gamma(x_1) \end{pmatrix}, \begin{pmatrix} x_2 \\ \gamma(x_1) \end{pmatrix} \Rightarrow \begin{pmatrix} x_1 \\ \gamma(x_1) \end{pmatrix}, \begin{pmatrix} x_2 \\ \gamma(\gamma(x_1)) \end{pmatrix} \Rightarrow \begin{pmatrix} x_1 \\ \gamma(x_1) \end{pmatrix}, \begin{pmatrix} x_2 \\ \gamma(\gamma(\gamma(x_1))) \end{pmatrix}$$

AUFGABE 1

$$\left\{\begin{pmatrix} \sigma(\sigma(& x_1 & ,\alpha), & \sigma(& \gamma(x_3) & ,x_3)) \\ \sigma(\sigma(& \gamma(x_2) & ,\alpha), & \sigma(& x_2 & ,x_3)) \end{pmatrix}\right\}$$

$$\overset{\text{Dek.}}{\Longrightarrow} \left\{\begin{pmatrix} \sigma(& x_1 & ,\alpha) \\ \sigma(& \gamma(x_2) & ,\alpha) \end{pmatrix}, \begin{pmatrix} \sigma(& \gamma(x_3) & ,x_3)) \\ \sigma(& x_2 & ,x_3)) \end{pmatrix}\right\}$$

$$\overset{\text{Dek.}}{\Longrightarrow}{}^2 \left\{\begin{pmatrix} x_1 \\ \gamma(x_2) \end{pmatrix}, \begin{pmatrix} \alpha \\ \alpha \end{pmatrix}, \begin{pmatrix} \gamma(x_3) \\ x_2 \end{pmatrix}, \begin{pmatrix} x_3 \\ x_3 \end{pmatrix}\right\}$$

$$\overset{\text{El.}}{\Longrightarrow} \left\{\begin{pmatrix} x_1 \\ \gamma(x_2) \end{pmatrix}, \begin{pmatrix} \alpha \\ \alpha \end{pmatrix}, \begin{pmatrix} \gamma(x_3) \\ x_2 \end{pmatrix}\right\}$$

$$\overset{\text{Dek.}}{\Longrightarrow} \left\{\begin{pmatrix} x_1 \\ \gamma(x_2) \end{pmatrix}, \begin{pmatrix} \gamma(x_3) \\ x_2 \end{pmatrix}\right\}$$

$$\overset{\text{Vert.}}{\Longrightarrow} \left\{\begin{pmatrix} x_1 \\ \gamma(x_2) \end{pmatrix}, \begin{pmatrix} x_2 \\ \gamma(x_3) \end{pmatrix}\right\} \quad x_2 \text{ kommt nicht in } \gamma(x_3) \text{ vor}$$

$$\overset{\text{Subst.}}{\Longrightarrow} \left\{\begin{pmatrix} x_1 \\ \gamma(\gamma(x_3)) \end{pmatrix}, \begin{pmatrix} x_2 \\ \gamma(x_3) \end{pmatrix}\right\}$$

AUFGABE 1

(a) **allgemeinster Unifikator:**

$$\varphi: \qquad x_1 \mapsto \gamma(\gamma(x_3)) \qquad x_2 \mapsto \gamma(x_3) \qquad x_3 \mapsto x_3$$

AUFGABE 1

(a) **allgemeinster Unifikator:**

$$\varphi : \quad x_1 \mapsto \gamma(\gamma(x_3)) \quad x_2 \mapsto \gamma(x_3) \quad x_3 \mapsto x_3$$

(b) **weitere Unifikatoren:**

$$\psi_1 : \quad x_1 \mapsto \gamma(\gamma(\alpha)) \quad x_2 \mapsto \gamma(\alpha) \quad x_3 \mapsto \alpha$$

$$\psi_2 : \quad x_1 \mapsto \gamma(\gamma(\gamma(\alpha))) \quad x_2 \mapsto \gamma(\gamma(\alpha)) \quad x_3 \mapsto \gamma(\alpha)$$

AUFGABE 1

(a) **allgemeinster Unifikator:**

$$\varphi: \quad x_1 \mapsto \gamma(\gamma(x_3)) \quad x_2 \mapsto \gamma(x_3) \quad x_3 \mapsto x_3$$

(b) **weitere Unifikatoren:**

$$\psi_1: \quad x_1 \mapsto \gamma(\gamma(\alpha)) \quad x_2 \mapsto \gamma(\alpha) \quad x_3 \mapsto \alpha$$

$$\psi_2: \quad x_1 \mapsto \gamma(\gamma(\gamma(\alpha))) \quad x_2 \mapsto \gamma(\gamma(\alpha)) \quad x_3 \mapsto \gamma(\alpha)$$

(c) **Fehlschlag beim occur-check:**

Alphabet: $\Sigma = \left\{\gamma^{(1)}\right\}$

$$t_1 = x_1$$

$$t_2 = \gamma(x_1)$$

AUFGABE 1 — TEIL (D)

$$t_1 = (\texttt{a}\ \ ,\ \ [\texttt{a}])$$

$$t_2 = (\texttt{Int}\ \ ,\ \ [\texttt{Double}])$$

$$t_3 = (\texttt{b}\ \ ,\ \ \texttt{c})$$

AUFGABE 1 — TEIL (D)

$$t_1 = (\texttt{a} \ , \ [\texttt{a}])$$

$$t_2 = (\texttt{Int} \ , \ [\texttt{Double}])$$

$$t_3 = (\texttt{b} \ , \ \texttt{c})$$

- t_1 und t_2 sind
- t_1 und t_3 sind
- t_2 und t_3 sind

AUFGABE 1 — TEIL (D)

$$t_1 = (\texttt{a}\ ,\ [\texttt{a}])$$

$$t_2 = (\texttt{Int}\ ,\ [\texttt{Double}])$$

$$t_3 = (\texttt{b}\ ,\ \texttt{c})$$

- t_1 und t_2 sind *nicht* unifizierbar
- t_1 und t_3 sind
- t_2 und t_3 sind

AUFGABE 1 — TEIL (D)

$$t_1 = (\texttt{a}\ ,\ [\texttt{a}])$$
$$t_2 = (\texttt{Int}\ ,\ [\texttt{Double}])$$
$$t_3 = (\texttt{b}\ ,\ \texttt{c})$$

- t_1 und t_2 sind *nicht* unifizierbar
- t_1 und t_3 sind unifizierbar mit $\texttt{a} \mapsto \texttt{a}$, $\texttt{b} \mapsto \texttt{a}$, $\texttt{c} \mapsto [\texttt{a}]$
- t_2 und t_3 sind

AUFGABE 1 — TEIL (D)

$$t_1 = (\texttt{a}\ ,\ [\texttt{a}])$$

$$t_2 = (\texttt{Int}\ ,\ [\texttt{Double}])$$

$$t_3 = (\texttt{b}\ ,\ \texttt{c})$$

- t_1 und t_2 sind *nicht* unifizierbar
- t_1 und t_3 sind unifizierbar mit $\texttt{a} \mapsto \texttt{a}$, $\texttt{b} \mapsto \texttt{a}$, $\texttt{c} \mapsto [\texttt{a}]$
- t_2 und t_3 sind unifizierbar mit $\texttt{b} \mapsto \texttt{Int}$, $\texttt{c} \mapsto [\texttt{Double}]$

Induktionsbeweise

Aufgabe 2

VOLLSTÄNDIGE INDUKTION AUF $\mathbb{N}$

Definition: natürliche Zahlen $\mathbb{N} := \{0, 1, \ldots\}$

$$\begin{array}{rll} \text{Basisfall:} & 0 \in \mathbb{N} & \\ \text{Rekursionsfall:} & x + 1 \in \mathbb{N} & \text{für} \quad x \in \mathbb{N} \end{array}$$

Beweis von Eigenschaften: Eigenschaft $=$ Prädikat P

$$\boxed{\text{zu zeigen:} \qquad \text{für alle } x \in \mathbb{N} \text{ gilt } P(x)}$$

vollständige Induktion:

- Induktionsanfang:
  zeige $P(x)$ für $x = 0$
- Induktionsvoraussetzung:
  Sei $x \in \mathbb{N}$, sodass $P(x)$ gilt. $P(x)$ gilt noch nicht für *alle* $x \in \mathbb{N}$
- Induktionsschritt:
  zeige $P(x + 1)$ unter Nutzung der Induktionsvoraussetzung

INDUKTION AUF LISTEN

Erinnerung: Rekursion über Listen `xs`

Basisfall: $\texttt{xs} = \texttt{[]}$

Rekursionsfall: $\texttt{xs} = \texttt{(y:ys)}$ für `ys :: [a]`

INDUKTION AUF LISTEN

Erinnerung: Rekursion über Listen `xs`

Basisfall: $\texttt{xs} = \texttt{[]}$

Rekursionsfall: $\texttt{xs} = \texttt{(y:ys)}$ für $\texttt{ys :: [a]}$

Beweis von Programmeigenschaften: Eigenschaft = Prädikat P

$$\boxed{\text{zu zeigen:} \qquad \text{für alle } \texttt{xs :: [a]} \text{ gilt } P(\texttt{xs})}$$

INDUKTION AUF LISTEN

Erinnerung: Rekursion über Listen `xs`

$$\begin{aligned} \text{Basisfall:} &\quad \texttt{xs} = \texttt{[]} \\ \text{Rekursionsfall:} &\quad \texttt{xs} = \texttt{(y:ys)} \quad \text{für} \quad \texttt{ys :: [a]} \end{aligned}$$

Beweis von Programmeigenschaften: Eigenschaft = Prädikat P

$$\boxed{\text{zu zeigen:} \qquad \text{für alle } \texttt{xs :: [a]} \text{ gilt } P(\texttt{xs})}$$

Induktion auf Listen:

- Induktionsanfang:
  zeige $P(\texttt{xs})$ für `xs == []`

INDUKTION AUF LISTEN

Erinnerung: Rekursion über Listen `xs`

Basisfall: $\texttt{xs} = \texttt{[]}$

Rekursionsfall: $\texttt{xs} = \texttt{(y:ys)}$ für `ys :: [a]`

Beweis von Programmeigenschaften: Eigenschaft $=$ Prädikat P

zu zeigen:	für alle `xs :: [a]` gilt $P(\texttt{xs})$

Induktion auf Listen:

- Induktionsanfang:
  zeige $P(\texttt{xs})$ für `xs == []`
- Induktionsvoraussetzung:
  Sei `xs :: [a]` eine Liste für die $P(\texttt{xs})$ gilt.

INDUKTION AUF LISTEN

Erinnerung: Rekursion über Listen `xs`

$$\begin{aligned} \text{Basisfall:} \quad & \texttt{xs} = \texttt{[]} \\ \text{Rekursionsfall:} \quad & \texttt{xs} = \texttt{(y:ys)} \quad \text{für} \quad \texttt{ys :: [a]} \end{aligned}$$

Beweis von Programmeigenschaften: Eigenschaft = Prädikat P

$$\boxed{\text{zu zeigen:} \qquad \text{für alle } \texttt{xs :: [a]} \text{ gilt } P(\texttt{xs})}$$

Induktion auf Listen:

- Induktionsanfang:
  zeige $P(\texttt{xs})$ für `xs == []`
- Induktionsvoraussetzung:
  Sei `xs :: [a]` eine Liste für die $P(\texttt{xs})$ gilt.
- Induktionsschritt:
  zeige $P(\texttt{x:xs})$ für alle `x :: a` unter Nutzung der Induktionsvoraussetzung

INDUKTION AUF LISTEN

Erinnerung: Rekursion über Listen `xs`

Basisfall: `xs` $=$ `[]`

Rekursionsfall: `xs` $=$ `(y:ys)` für `ys :: [a]`

Beweis von Programmeigenschaften: Eigenschaft $=$ Prädikat P

zu zeigen:	für alle `xs :: [a]` gilt P(`xs`)

Induktion auf Listen:

- Induktionsanfang:
  zeige P(`xs`) für `xs == []`
- Induktionsvoraussetzung:
  Sei `xs :: [a]` eine Liste für die P(`xs`) gilt.
- Induktionsschritt:
  zeige P(`x:xs`) für alle `x :: a` unter Nutzung der Induktionsvoraussetzung

Allgemeiner Hinweis: Es müssen immer alle Variablen quantifiziert werden!

STRUKTURELLE INDUKTION

Erinnerung: Rekursion über Bäume

Basisfall:	`Nil` oder `Leaf x` für `x :: a`
Rekursionsfall:	`Branch x l r` für `x :: a` und `l,r :: BinTree a`

STRUKTURELLE INDUKTION

Erinnerung: Rekursion über Bäume

Basisfall: `Nil` oder `Leaf x` für `x :: a`

Rekursionsfall: `Branch x l r` für `x :: a` und `l,r :: BinTree a`

zu zeigen:	für alle `t :: BinTree a` gilt $P($`t`$)$

STRUKTURELLE INDUKTION

Erinnerung: Rekursion über Bäume

Basisfall: `Nil` oder `Leaf x` für `x :: a`

Rekursionsfall: `Branch x l r` für `x :: a` und `l,r :: BinTree a`

zu zeigen: für alle `t :: BinTree a` gilt P(`t`)

strukturelle Induktion:

- Induktionsanfang:
  zeige P(`t`) für `t == Nil` oder `t == Leaf x` für alle `x :: a`

STRUKTURELLE INDUKTION

Erinnerung: Rekursion über Bäume

Basisfall: `Nil` oder `Leaf x` für `x :: a`

Rekursionsfall: `Branch x l r` für `x :: a` und `l,r :: BinTree a`

zu zeigen: für alle `t :: BinTree a` gilt $P(\texttt{t})$

strukturelle Induktion:

- Induktionsanfang:
  zeige $P(\texttt{t})$ für `t == Nil` oder `t == Leaf x` für alle `x :: a`
- Induktionsvoraussetzung:
  Seien `l, r :: BinTree a` zwei Bäume, sodass $P(\texttt{l})$ und $P(\texttt{r})$ gilt.

STRUKTURELLE INDUKTION

Erinnerung: Rekursion über Bäume

Basisfall: `Nil` oder `Leaf x` für `x :: a`

Rekursionsfall: `Branch x l r` für `x :: a` und `l,r :: BinTree a`

zu zeigen: für alle `t :: BinTree a` gilt P(`t`)

strukturelle Induktion:

- Induktionsanfang:
  zeige P(`t`) für `t == Nil` oder `t == Leaf x` für alle `x :: a`
- Induktionsvoraussetzung:
  Seien `l, r :: BinTree a` zwei Bäume, sodass P(`l`) und P(`r`) gilt.
- Induktionsschritt:
  zeige P(`Branch x l r`) für alle `x :: a` unter Nutzung der Induktionsvoraussetzung

STRUKTURELLE INDUKTION

Erinnerung: Rekursion über Bäume

Basisfall: `Nil` oder `Leaf x` für `x :: a`

Rekursionsfall: `Branch x l r` für `x :: a` und `l,r :: BinTree a`

zu zeigen: für alle `t :: BinTree a` gilt P(`t`)

strukturelle Induktion:

- Induktionsanfang:
  zeige P(`t`) für `t == Nil` oder `t == Leaf x` für alle `x :: a`
- Induktionsvoraussetzung:
  Seien `l, r :: BinTree a` zwei Bäume, sodass P(`l`) und P(`r`) gilt.
- Induktionsschritt:
  zeige P(`Branch x l r`) für alle `x :: a` unter Nutzung der Induktionsvoraussetzung

Allgemeiner Hinweis: Es müssen immer alle Variablen quantifiziert werden!

FEHLERQUELLEN

- kein Induktionsprinzip
- IV wird im Induktionsschritt nicht verwendet
- fehlende Quantifizierung (nur Gleichungen bringen kaum Punkte)
- *Missachtung freier Variablen*

FEHLERQUELLEN

- kein Induktionsprinzip
- IV wird im Induktionsschritt nicht verwendet
- fehlende Quantifizierung (nur Gleichungen bringen kaum Punkte)
- *Missachtung freier Variablen*
- zu beweisende Eigenschaft P wird für `xs` angenommen, um sie dann im Induktionsschritt nochmal für `xs` zu beweisen — eine Tautologie
- Annahme, dass P bereits für alle Listen gilt, um es dann für `x:xs` nochmal zu zeigen

AUFGABE 2

Zu zeigen ist die Gleichung

$$\texttt{sum (foo xs) = 2 * sum xs - length xs} \qquad \text{für alle } \texttt{xs :: Int}$$

mittels Induktion über Listen.

Induktionsanfang: Sei `xs == []`.

linke Seite:

$$\texttt{sum (foo [])} \overset{(2)}{=} \texttt{sum []} \overset{(6)}{=} \texttt{0}$$

rechte Seite:

$$\texttt{2 * sum [] - length []} \overset{(10)}{=} \texttt{2 * sum [] - 0} \overset{(6)}{=} \texttt{2 * 0 - 0 = 0}$$

Induktionsvoraussetzung: Sei `xs :: [Int]`, sodass

$$\texttt{sum (foo xs) = 2 * sum xs - length xs}$$

gilt.

AUFGABE 2 (FORTSETZUNG)

Induktionsschritt: Sei `x :: Int`. Es gilt

$$
\begin{aligned}
\texttt{sum (foo (x:xs))} &\overset{(3)}{=} \texttt{sum (x : x : (-1) : foo xs)} \\
&\overset{3\cdot(7)}{=} \texttt{x + x + (-1) + sum (foo xs)} \\
&\overset{(IV)}{=} \texttt{x + x + (-1) + 2 * sum xs - length xs} \\
&\overset{(Komm.)}{=} \texttt{2 * x + 2 * sum xs - 1 - length xs)} \\
&\overset{(Dist.)}{=} \texttt{2 * (x + sum xs) - (1 + length xs)} \\
&\overset{(7)}{=} \texttt{2 * sum (x:xs) - (1 + length xs)} \\
&\overset{(11)}{=} \texttt{2 * sum (x:xs) - length (x:xs)}
\end{aligned}
$$

ENDE

Fragen?