

Welche Möglichkeiten gibt es für $g :: [a] \rightarrow [a]$?

Zunächst wieder intuitiv:

- Die Ausgabe- kann nur Elemente der Eingabeliste enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von dieser Liste abhängen, und zwar von ihrer Länge.
- Folglich sind für eine feste Eingabelänge die Länge der Ausgabeliste sowie die Anordnung der Elemente darin stets fix.

Formal beschrieben:

Für jedes $g :: [a] \rightarrow [a]$ gibt es:

- eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$, so dass für jedes $n \in \mathbb{N}$ aus einer Liste der Länge n mit g stets eine Liste der Länge $f(n)$ wird;
- eine Funktion $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, so dass für jede Liste einer Länge $n \in \mathbb{N}$ und für jedes $0 \leq m < f(n)$, das m -te Element der Ausgabeliste das $h(n, m)$ -te Element der Eingabeliste ist.

Geht es vielleicht etwas formaler?

Strategie:

- zu jedem Typ angeben, welche möglichen Varianten (semantisch, nicht syntaktisch) von Elementen diesen Typs es gibt
- das Ganze möglichst kompositionell über den Aufbau von Typen
- später per Induktion über die Syntax beweisen, dass jede Funktion die semantischen Einschränkungen ihres Typs erfüllt

Geht es vielleicht etwas formaler?

Strategie:

- zu jedem Typ angeben, welche möglichen Varianten (semantisch, nicht syntaktisch) von Elementen diesen Typs es gibt
- das Ganze möglichst kompositionell über den Aufbau von Typen
- später per Induktion über die Syntax beweisen, dass jede Funktion die semantischen Einschränkungen ihres Typs erfüllt

Ein kompositioneller Ansatz

Frage: Welche g haben einen gewissen Typ τ ?

Ein kompositioneller Ansatz

Frage: Welche g haben einen gewissen Typ τ ?

Ansatz: Denotationen $\llbracket \tau \rrbracket$ von Typen als Mengen angeben.

Ein kompositioneller Ansatz

Frage: Welche g haben einen gewissen Typ τ ?

Ansatz: Denotationen $\llbracket \tau \rrbracket$ von Typen als Mengen angeben.

$$\llbracket \text{Bool} \rrbracket = \{\text{True}, \text{False}\}$$

$$\llbracket \text{Int} \rrbracket = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

Ein kompositioneller Ansatz

Frage: Welche g haben einen gewissen Typ τ ?

Ansatz: Denotationen $\llbracket \tau \rrbracket$ von Typen als Mengen angeben.

$$\llbracket \text{Bool} \rrbracket = \{\text{True}, \text{False}\}$$

$$\llbracket \text{Int} \rrbracket = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

$$\llbracket (\tau_1, \tau_2) \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$$

Ein kompositioneller Ansatz

Frage: Welche g haben einen gewissen Typ τ ?

Ansatz: Denotationen $\llbracket \tau \rrbracket$ von Typen als Mengen angeben.

$$\llbracket \text{Bool} \rrbracket = \{\text{True}, \text{False}\}$$

$$\llbracket \text{Int} \rrbracket = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

$$\llbracket (\tau_1, \tau_2) \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$$

$$\llbracket [\tau] \rrbracket = \{[x_1, \dots, x_n] \mid n \geq 0, x_i \in \llbracket \tau \rrbracket\}$$

Ein kompositioneller Ansatz

Frage: Welche g haben einen gewissen Typ τ ?

Ansatz: Denotationen $\llbracket \tau \rrbracket$ von Typen als Mengen angeben.

$$\llbracket \text{Bool} \rrbracket = \{\text{True}, \text{False}\}$$

$$\llbracket \text{Int} \rrbracket = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

$$\llbracket (\tau_1, \tau_2) \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$$

$$\llbracket [\tau] \rrbracket = \{[x_1, \dots, x_n] \mid n \geq 0, x_i \in \llbracket \tau \rrbracket\}$$

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \{f : \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket\}$$

Ein kompositioneller Ansatz

Frage: Welche g haben einen gewissen Typ τ ?

Ansatz: Denotationen $\llbracket \tau \rrbracket$ von Typen als Mengen angeben.

$$\llbracket \text{Bool} \rrbracket = \{\text{True}, \text{False}\}$$

$$\llbracket \text{Int} \rrbracket = \{\dots, -2, -1, 0, 1, 2, \dots\}$$

$$\llbracket (\tau_1, \tau_2) \rrbracket = \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$$

$$\llbracket [\tau] \rrbracket = \{[x_1, \dots, x_n] \mid n \geq 0, x_i \in \llbracket \tau \rrbracket\}$$

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \{f : \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket\}$$

$$\llbracket \forall \alpha. \tau \rrbracket = ?$$

Ein kompositioneller Ansatz

Frage: Welche g haben einen gewissen Typ τ ?

Ansatz: Denotationen $\llbracket \tau \rrbracket$ von Typen als Mengen angeben.

$$\begin{aligned}\llbracket \text{Bool} \rrbracket &= \{\text{True}, \text{False}\} \\ \llbracket \text{Int} \rrbracket &= \{\dots, -2, -1, 0, 1, 2, \dots\} \\ \llbracket (\tau_1, \tau_2) \rrbracket &= \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket \\ \llbracket [\tau] \rrbracket &= \{[x_1, \dots, x_n] \mid n \geq 0, x_i \in \llbracket \tau \rrbracket\} \\ \llbracket \tau_1 \rightarrow \tau_2 \rrbracket &= \{f : \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket\} \\ \llbracket \forall \alpha. \tau \rrbracket &= ?\end{aligned}$$

- $g \in \llbracket \forall \alpha. \tau \rrbracket$ müsste eine ganze „Familie“ von Werten sein: für jeden Typ τ' , eine Instanz mit Typ $\tau[\tau'/\alpha]$.

Ein kompositioneller Ansatz

Frage: Welche g haben einen gewissen Typ τ ?

Ansatz: Denotationen $\llbracket \tau \rrbracket$ von Typen als Mengen angeben.

$$\begin{aligned}\llbracket \text{Bool} \rrbracket &= \{\text{True}, \text{False}\} \\ \llbracket \text{Int} \rrbracket &= \{\dots, -2, -1, 0, 1, 2, \dots\} \\ \llbracket (\tau_1, \tau_2) \rrbracket &= \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket \\ \llbracket [\tau] \rrbracket &= \{[x_1, \dots, x_n] \mid n \geq 0, x_i \in \llbracket \tau \rrbracket\} \\ \llbracket \tau_1 \rightarrow \tau_2 \rrbracket &= \{f : \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket\} \\ \llbracket \forall \alpha. \tau \rrbracket &= ?\end{aligned}$$

- $g \in \llbracket \forall \alpha. \tau \rrbracket$ müsste eine ganze „Familie“ von Werten sein: für jeden Typ τ' , eine Instanz mit Typ $\tau[\tau'/\alpha]$.
- $\llbracket \forall \alpha. \tau \rrbracket = \{g \mid \forall \tau'. g_{\tau'} \in \llbracket \tau[\tau'/\alpha] \rrbracket\} ?$

Ein kompositioneller Ansatz

Frage: Welche g haben einen gewissen Typ τ ?

Ansatz: Denotationen $\llbracket \tau \rrbracket$ von Typen als Mengen angeben.

$$\begin{aligned}\llbracket \text{Bool} \rrbracket &= \{\text{True}, \text{False}\} \\ \llbracket \text{Int} \rrbracket &= \{\dots, -2, -1, 0, 1, 2, \dots\} \\ \llbracket (\tau_1, \tau_2) \rrbracket &= \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket \\ \llbracket [\tau] \rrbracket &= \{[x_1, \dots, x_n] \mid n \geq 0, x_i \in \llbracket \tau \rrbracket\} \\ \llbracket \tau_1 \rightarrow \tau_2 \rrbracket &= \{f : \llbracket \tau_1 \rrbracket \rightarrow \llbracket \tau_2 \rrbracket\} \\ \llbracket \forall \alpha. \tau \rrbracket &= ?\end{aligned}$$

- $g \in \llbracket \forall \alpha. \tau \rrbracket$ müsste eine ganze „Familie“ von Werten sein: für jeden Typ τ' , eine Instanz mit Typ $\tau[\tau'/\alpha]$.
- $\llbracket \forall \alpha. \tau \rrbracket = \{g \mid \forall \tau'. g_{\tau'} \in \llbracket \tau[\tau'/\alpha] \rrbracket\} ?$
- Aber das schließt „ad-hoc-polymorphe“ Funktionen ein!

Unerwünschte Ad-Hoc-Polymorphie am Beispiel

- Mit der vorgeschlagenen Definition,

$$\llbracket \forall \alpha. (\alpha, \alpha) \rightarrow \alpha \rrbracket = \{g \mid \forall \tau. g_\tau : \llbracket \tau \rrbracket \times \llbracket \tau \rrbracket \rightarrow \llbracket \tau \rrbracket\}.$$

Unerwünschte Ad-Hoc-Polymorphie am Beispiel

- Mit der vorgeschlagenen Definition,
$$\llbracket \forall \alpha. (\alpha, \alpha) \rightarrow \alpha \rrbracket = \{g \mid \forall \tau. g_\tau : \llbracket \tau \rrbracket \times \llbracket \tau \rrbracket \rightarrow \llbracket \tau \rrbracket\}.$$
- Aber das erlaubt auch

$$\begin{aligned} g_{\text{Bool}}(x, y) &= \neg x \\ g_{\text{Int}}(x, y) &= y + 1, \end{aligned}$$

was in Haskell beim Typ $\forall \alpha. (\alpha, \alpha) \rightarrow \alpha$ unmöglich ist.

Unerwünschte Ad-Hoc-Polymorphie am Beispiel

- Mit der vorgeschlagenen Definition,
$$\llbracket \forall \alpha. (\alpha, \alpha) \rightarrow \alpha \rrbracket = \{g \mid \forall \tau. g_\tau : \llbracket \tau \rrbracket \times \llbracket \tau \rrbracket \rightarrow \llbracket \tau \rrbracket\}.$$
- Aber das erlaubt auch

$$\begin{aligned}g_{\text{Bool}}(x, y) &= \neg x \\g_{\text{Int}}(x, y) &= y + 1,\end{aligned}$$

was in Haskell beim Typ $\forall \alpha. (\alpha, \alpha) \rightarrow \alpha$ unmöglich ist.

- Um dies zu vermeiden, müssen wir

$$\begin{aligned}g_{\text{Bool}} &: \llbracket \text{Bool} \rrbracket \times \llbracket \text{Bool} \rrbracket \rightarrow \llbracket \text{Bool} \rrbracket \quad \text{und} \\g_{\text{Int}} &: \llbracket \text{Int} \rrbracket \times \llbracket \text{Int} \rrbracket \rightarrow \llbracket \text{Int} \rrbracket\end{aligned}$$

vergleichen und gewährleisten, dass sie sich
„identisch verhalten“.

Unerwünschte Ad-Hoc-Polymorphie am Beispiel

- Mit der vorgeschlagenen Definition,
$$\llbracket \forall \alpha. (\alpha, \alpha) \rightarrow \alpha \rrbracket = \{g \mid \forall \tau. g_\tau : \llbracket \tau \rrbracket \times \llbracket \tau \rrbracket \rightarrow \llbracket \tau \rrbracket\}.$$
- Aber das erlaubt auch

$$\begin{aligned}g_{\text{Bool}}(x, y) &= \neg x \\g_{\text{Int}}(x, y) &= y + 1,\end{aligned}$$

was in Haskell beim Typ $\forall \alpha. (\alpha, \alpha) \rightarrow \alpha$ unmöglich ist.

- Um dies zu vermeiden, müssen wir

$$\begin{aligned}g_{\text{Bool}} &: \llbracket \text{Bool} \rrbracket \times \llbracket \text{Bool} \rrbracket \rightarrow \llbracket \text{Bool} \rrbracket \quad \text{und} \\g_{\text{Int}} &: \llbracket \text{Int} \rrbracket \times \llbracket \text{Int} \rrbracket \rightarrow \llbracket \text{Int} \rrbracket\end{aligned}$$

vergleichen und gewährleisten, dass sie sich
„identisch verhalten“.

Aber wie?

Idee [Reynolds 1983]

Beliebige Relationen benutzen, um Instanzen zu verknüpfen!

Idee [Reynolds 1983]

Beliebige Relationen benutzen, um Instanzen zu verknüpfen!

Im Beispiel ($g :: \forall \alpha. (\alpha, \alpha) \rightarrow \alpha$):

- Wähle eine Relation $\mathcal{R} \subseteq \llbracket \text{Bool} \rrbracket \times \llbracket \text{Int} \rrbracket$.

Idee [Reynolds 1983]

Beliebige Relationen benutzen, um Instanzen zu verknüpfen!

Im Beispiel ($g :: \forall \alpha. (\alpha, \alpha) \rightarrow \alpha$):

- Wähle eine Relation $\mathcal{R} \subseteq \llbracket \text{Bool} \rrbracket \times \llbracket \text{Int} \rrbracket$.
- Nenne $(x_1, x_2) \in \llbracket \text{Bool} \rrbracket \times \llbracket \text{Bool} \rrbracket$ und $(y_1, y_2) \in \llbracket \text{Int} \rrbracket \times \llbracket \text{Int} \rrbracket$ verwandt, wenn $(x_1, y_1) \in \mathcal{R}$ und $(x_2, y_2) \in \mathcal{R}$.

Idee [Reynolds 1983]

Beliebige Relationen benutzen, um Instanzen zu verknüpfen!

Im Beispiel ($g :: \forall \alpha. (\alpha, \alpha) \rightarrow \alpha$):

- Wähle eine Relation $\mathcal{R} \subseteq \llbracket \text{Bool} \rrbracket \times \llbracket \text{Int} \rrbracket$.
- Nenne $(x_1, x_2) \in \llbracket \text{Bool} \rrbracket \times \llbracket \text{Bool} \rrbracket$ und $(y_1, y_2) \in \llbracket \text{Int} \rrbracket \times \llbracket \text{Int} \rrbracket$ verwandt, wenn $(x_1, y_1) \in \mathcal{R}$ und $(x_2, y_2) \in \mathcal{R}$.
- Nenne $f_1 : \llbracket \text{Bool} \rrbracket \times \llbracket \text{Bool} \rrbracket \rightarrow \llbracket \text{Bool} \rrbracket, f_2 : \llbracket \text{Int} \rrbracket \times \llbracket \text{Int} \rrbracket \rightarrow \llbracket \text{Int} \rrbracket$ verwandt, wenn verwandte Eingaben zu verwandten Ausgaben führen.

Idee [Reynolds 1983]

Beliebige Relationen benutzen, um Instanzen zu verknüpfen!

Im Beispiel ($g :: \forall \alpha. (\alpha, \alpha) \rightarrow \alpha$):

- Wähle eine Relation $\mathcal{R} \subseteq \llbracket \text{Bool} \rrbracket \times \llbracket \text{Int} \rrbracket$.
- Nenne $(x_1, x_2) \in \llbracket \text{Bool} \rrbracket \times \llbracket \text{Bool} \rrbracket$ und $(y_1, y_2) \in \llbracket \text{Int} \rrbracket \times \llbracket \text{Int} \rrbracket$ verwandt, wenn $(x_1, y_1) \in \mathcal{R}$ und $(x_2, y_2) \in \mathcal{R}$.
- Nenne $f_1 : \llbracket \text{Bool} \rrbracket \times \llbracket \text{Bool} \rrbracket \rightarrow \llbracket \text{Bool} \rrbracket, f_2 : \llbracket \text{Int} \rrbracket \times \llbracket \text{Int} \rrbracket \rightarrow \llbracket \text{Int} \rrbracket$ verwandt, wenn verwandte Eingaben zu verwandten Ausgaben führen.
- Dann sind g_{Bool} und g_{Int} mit

$$\begin{aligned}g_{\text{Bool}}(x, y) &= \neg x && \text{und} \\g_{\text{Int}}(x, y) &= y + 1\end{aligned}$$

nicht verwandt bei, zum Beispiel, Wahl von $\mathcal{R} = \{(\text{True}, 1)\}$.

Idee [Reynolds 1983]

Beliebige Relationen benutzen, um Instanzen zu verknüpfen!

Im Beispiel ($g :: \forall \alpha. (\alpha, \alpha) \rightarrow \alpha$):

- Wähle eine Relation $\mathcal{R} \subseteq \llbracket \text{Bool} \rrbracket \times \llbracket \text{Int} \rrbracket$.
- Nenne $(x_1, x_2) \in \llbracket \text{Bool} \rrbracket \times \llbracket \text{Bool} \rrbracket$ und $(y_1, y_2) \in \llbracket \text{Int} \rrbracket \times \llbracket \text{Int} \rrbracket$ verwandt, wenn $(x_1, y_1) \in \mathcal{R}$ und $(x_2, y_2) \in \mathcal{R}$.
- Nenne $f_1 : \llbracket \text{Bool} \rrbracket \times \llbracket \text{Bool} \rrbracket \rightarrow \llbracket \text{Bool} \rrbracket, f_2 : \llbracket \text{Int} \rrbracket \times \llbracket \text{Int} \rrbracket \rightarrow \llbracket \text{Int} \rrbracket$ verwandt, wenn verwandte Eingaben zu verwandten Ausgaben führen.
- Dann sind g_{Bool} und g_{Int} mit

$$\begin{aligned}g_{\text{Bool}}(x, y) &= \neg x && \text{und} \\g_{\text{Int}}(x, y) &= y + 1\end{aligned}$$

nicht verwandt bei, zum Beispiel, Wahl von $\mathcal{R} = \{(\text{True}, 1)\}$.

Reynolds: $g \in \llbracket \forall \alpha. \tau \rrbracket$ genau dann wenn für alle τ_1, τ_2 und $\mathcal{R} \subseteq \llbracket \tau_1 \rrbracket \times \llbracket \tau_2 \rrbracket$ gilt, dass g_{τ_1} und g_{τ_2} verwandt per „Fortsetzung“ von \mathcal{R} bezüglich τ sind

Freie Theoreme, allgemein

Zur Interpretation von Typen als Relationen:

1. Ersetze (Quantifizierung über) Typvariablen durch (Quantifizierung über) Relationsvariablen.

Freie Theoreme, allgemein

Zur Interpretation von Typen als Relationen:

1. Ersetze (Quantifizierung über) Typvariablen durch (Quantifizierung über) Relationsvariablen.
2. Ersetze Teiltypen ohne jeglichen Polymorphismus durch Identitätsrelationen.

Freie Theoreme, allgemein

Zur Interpretation von Typen als Relationen:

1. Ersetze (Quantifizierung über) Typvariablen durch (Quantifizierung über) Relationsvariablen.
2. Ersetze Teiltypen ohne jeglichen Polymorphismus durch Identitätsrelationen.
3. Verwende folgende Regeln:

$$(\mathcal{R}, \mathcal{S}) = \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\}$$

Freie Theoreme, allgemein

Zur Interpretation von Typen als Relationen:

1. Ersetze (Quantifizierung über) Typvariablen durch (Quantifizierung über) Relationsvariablen.
2. Ersetze Teiltypen ohne jeglichen Polymorphismus durch Identitätsrelationen.
3. Verwende folgende Regeln:

$$\begin{aligned}(\mathcal{R}, \mathcal{S}) &= \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\} \\ [\mathcal{R}] &= \{([x_1, \dots, x_n], [y_1, \dots, y_n]) \mid n \geq 0, (x_i, y_i) \in \mathcal{R}\}\end{aligned}$$

Freie Theoreme, allgemein

Zur Interpretation von Typen als Relationen:

1. Ersetze (Quantifizierung über) Typvariablen durch (Quantifizierung über) Relationsvariablen.
2. Ersetze Teiltypen ohne jeglichen Polymorphismus durch Identitätsrelationen.
3. Verwende folgende Regeln:

$$(\mathcal{R}, \mathcal{S}) = \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\}$$

$$[\mathcal{R}] = \{([x_1, \dots, x_n], [y_1, \dots, y_n]) \mid n \geq 0, (x_i, y_i) \in \mathcal{R}\}$$

$$\mathcal{R} \rightarrow \mathcal{S} = \{(f_1, f_2) \mid \forall (a_1, a_2) \in \mathcal{R}. (f_1 a_1, f_2 a_2) \in \mathcal{S}\}$$

Freie Theoreme, allgemein

Zur Interpretation von Typen als Relationen:

1. Ersetze (Quantifizierung über) Typvariablen durch (Quantifizierung über) Relationsvariablen.
2. Ersetze Teiltypen ohne jeglichen Polymorphismus durch Identitätsrelationen.
3. Verwende folgende Regeln:

$$(\mathcal{R}, \mathcal{S}) = \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\}$$

$$[\mathcal{R}] = \{([x_1, \dots, x_n], [y_1, \dots, y_n]) \mid n \geq 0, (x_i, y_i) \in \mathcal{R}\}$$

$$\mathcal{R} \rightarrow \mathcal{S} = \{(f_1, f_2) \mid \forall (a_1, a_2) \in \mathcal{R}. (f_1 a_1, f_2 a_2) \in \mathcal{S}\}$$

$$\forall \mathcal{R}. \mathcal{F}(\mathcal{R}) = \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}(\mathcal{R})\}$$

Freie Theoreme, allgemein

Zur Interpretation von Typen als Relationen:

1. Ersetze (Quantifizierung über) Typvariablen durch (Quantifizierung über) Relationsvariablen.
2. Ersetze Teiltypen ohne jeglichen Polymorphismus durch Identitätsrelationen.
3. Verwende folgende Regeln:

$$(\mathcal{R}, \mathcal{S}) = \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\}$$

$$[\mathcal{R}] = \{([x_1, \dots, x_n], [y_1, \dots, y_n]) \mid n \geq 0, (x_i, y_i) \in \mathcal{R}\}$$

$$\mathcal{R} \rightarrow \mathcal{S} = \{(f_1, f_2) \mid \forall (a_1, a_2) \in \mathcal{R}. (f_1 a_1, f_2 a_2) \in \mathcal{S}\}$$

$$\forall \mathcal{R}. \mathcal{F}(\mathcal{R}) = \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}(\mathcal{R})\}$$

Dann gilt für jedes $g :: \tau$, dass das Paar (g, g) in der Interpretation von τ als Relation enthalten ist.

Geht es vielleicht etwas formaler?

Strategie:

- zu jedem Typ angeben, welche möglichen Varianten (semantisch, nicht syntaktisch) von Elementen diesen Typs es gibt
- das Ganze möglichst kompositionell über den Aufbau von Typen
- später per Induktion über die Syntax beweisen, dass jede Funktion die semantischen Einschränkungen ihres Typs erfüllt

Nun formal am Beispiel

Gegeben sei $g :: \forall \alpha. (\alpha \rightarrow \text{Bool}) \rightarrow ([\alpha] \rightarrow [\alpha])$.

Nun formal am Beispiel

Gegeben sei $g :: \forall \alpha. (\alpha \rightarrow \text{Bool}) \rightarrow ([\alpha] \rightarrow [\alpha])$.

Dann:

$$(g, g) \in \forall \mathcal{R}. (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}])$$

Nun formal am Beispiel

Gegeben sei $g :: \forall \alpha. (\alpha \rightarrow \text{Bool}) \rightarrow ([\alpha] \rightarrow [\alpha])$.

Dann:

$$(g, g) \in \forall \mathcal{R}. (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}])$$

$$\Leftrightarrow \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. (g_{\tau_1}, g_{\tau_2}) \in (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}])$$

wegen Definition von $\forall \mathcal{R}. \mathcal{F}(\mathcal{R})$

Nun formal am Beispiel

Gegeben sei $g :: \forall \alpha. (\alpha \rightarrow \text{Bool}) \rightarrow ([\alpha] \rightarrow [\alpha])$.

Dann:

$$(g, g) \in \forall \mathcal{R}. (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}])$$

$$\Leftrightarrow \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. (g_{\tau_1}, g_{\tau_2}) \in (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}])$$

$$\Leftrightarrow \forall \mathcal{R}. \forall (a_1, a_2) \in (\mathcal{R} \rightarrow id_{\text{Bool}}). (g_{\tau_1} a_1, g_{\tau_2} a_2) \in ([\mathcal{R}] \rightarrow [\mathcal{R}])$$

wegen Definition von $\mathcal{R} \rightarrow \mathcal{S}$

Nun formal am Beispiel

Gegeben sei $g :: \forall \alpha. (\alpha \rightarrow \text{Bool}) \rightarrow ([\alpha] \rightarrow [\alpha])$.

Dann:

$$(g, g) \in \forall \mathcal{R}. (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}])$$

$$\Leftrightarrow \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. (g_{\tau_1}, g_{\tau_2}) \in (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}])$$

$$\Leftrightarrow \forall \mathcal{R}. \forall (a_1, a_2) \in (\mathcal{R} \rightarrow id_{\text{Bool}}). (g_{\tau_1} a_1, g_{\tau_2} a_2) \in ([\mathcal{R}] \rightarrow [\mathcal{R}])$$

$$\Leftrightarrow \forall \mathcal{R}. \forall (a_1, a_2) \in (\mathcal{R} \rightarrow id_{\text{Bool}}). \forall (l_1, l_2) \in [\mathcal{R}].$$

$$(g_{\tau_1} a_1 l_1, g_{\tau_2} a_2 l_2) \in [\mathcal{R}]$$

wegen Definition von $\mathcal{R} \rightarrow \mathcal{S}$

Nun formal am Beispiel

Gegeben sei $g :: \forall \alpha. (\alpha \rightarrow \text{Bool}) \rightarrow ([\alpha] \rightarrow [\alpha])$.

Dann:

$$\begin{aligned} & (g, g) \in \forall \mathcal{R}. (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. (g_{\tau_1}, g_{\tau_2}) \in (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (a_1, a_2) \in (\mathcal{R} \rightarrow id_{\text{Bool}}). (g_{\tau_1} a_1, g_{\tau_2} a_2) \in ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (a_1, a_2) \in (\mathcal{R} \rightarrow id_{\text{Bool}}). \forall (l_1, l_2) \in [\mathcal{R}]. \\ & \qquad \qquad \qquad (g_{\tau_1} a_1 l_1, g_{\tau_2} a_2 l_2) \in [\mathcal{R}] \\ \Rightarrow & \forall (a_1, a_2) \in (h \rightarrow id_{\text{Bool}}). \forall (l_1, l_2) \in (\text{map } h). \\ & \qquad \qquad \qquad (g_{\tau_1} a_1 l_1, g_{\tau_2} a_2 l_2) \in (\text{map } h) \end{aligned}$$

durch Spezialisierung zu $\mathcal{R} = h$, wobei dann $[\mathcal{R}] = (\text{map } h)$

für jede Funktion $h :: \tau_1 \rightarrow \tau_2$

Nun formal am Beispiel

Gegeben sei $g :: \forall \alpha. (\alpha \rightarrow \text{Bool}) \rightarrow ([\alpha] \rightarrow [\alpha])$.

Dann:

$$\begin{aligned} & (g, g) \in \forall \mathcal{R}. (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. (g_{\tau_1}, g_{\tau_2}) \in (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (a_1, a_2) \in (\mathcal{R} \rightarrow id_{\text{Bool}}). (g_{\tau_1} a_1, g_{\tau_2} a_2) \in ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (a_1, a_2) \in (\mathcal{R} \rightarrow id_{\text{Bool}}). \forall (l_1, l_2) \in [\mathcal{R}]. \\ & \qquad \qquad \qquad (g_{\tau_1} a_1 l_1, g_{\tau_2} a_2 l_2) \in [\mathcal{R}] \\ \Rightarrow & \forall (a_1, a_2) \in (h \rightarrow id_{\text{Bool}}). \forall (l_1, l_2) \in (\text{map } h). \\ & \qquad \qquad \qquad (g_{\tau_1} a_1 l_1, g_{\tau_2} a_2 l_2) \in (\text{map } h) \\ \Rightarrow & \forall (l_1, l_2) \in (\text{map } h). (g_{\tau_1} (p \circ h) l_1, g_{\tau_2} p l_2) \in (\text{map } h) \\ & \text{durch Wahl von } (a_1, a_2) = (p \circ h, p) \in (h \rightarrow id_{\text{Bool}}) \end{aligned}$$

für jede Funktion $h :: \tau_1 \rightarrow \tau_2$ und jedes $p :: \tau_2 \rightarrow \text{Bool}$.

Nun formal am Beispiel

Gegeben sei $g :: \forall \alpha. (\alpha \rightarrow \text{Bool}) \rightarrow ([\alpha] \rightarrow [\alpha])$.

Dann:

$$\begin{aligned} & (g, g) \in \forall \mathcal{R}. (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. (g_{\tau_1}, g_{\tau_2}) \in (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (a_1, a_2) \in (\mathcal{R} \rightarrow id_{\text{Bool}}). (g_{\tau_1} a_1, g_{\tau_2} a_2) \in ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (a_1, a_2) \in (\mathcal{R} \rightarrow id_{\text{Bool}}). \forall (l_1, l_2) \in [\mathcal{R}]. \\ & \qquad \qquad \qquad (g_{\tau_1} a_1 l_1, g_{\tau_2} a_2 l_2) \in [\mathcal{R}] \\ \Rightarrow & \forall (a_1, a_2) \in (h \rightarrow id_{\text{Bool}}). \forall (l_1, l_2) \in (\text{map } h). \\ & \qquad \qquad \qquad (g_{\tau_1} a_1 l_1, g_{\tau_2} a_2 l_2) \in (\text{map } h) \\ \Rightarrow & \forall (l_1, l_2) \in (\text{map } h). (g_{\tau_1} (p \circ h) l_1, g_{\tau_2} p l_2) \in (\text{map } h) \\ \Leftrightarrow & \forall l_1 :: [\tau_1]. \text{map } h (g_{\tau_1} (p \circ h) l_1) = g_{\tau_2} p (\text{map } h l_1) \\ & \text{durch einfache Umformung} \end{aligned}$$

für jede Funktion $h :: \tau_1 \rightarrow \tau_2$ und jedes $p :: \tau_2 \rightarrow \text{Bool}$.

Nun formal am Beispiel

Gegeben sei $g :: \forall \alpha. (\alpha \rightarrow \text{Bool}) \rightarrow ([\alpha] \rightarrow [\alpha])$.

Dann:

$$\begin{aligned} & (g, g) \in \forall \mathcal{R}. (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2. (g_{\tau_1}, g_{\tau_2}) \in (\mathcal{R} \rightarrow id_{\text{Bool}}) \rightarrow ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (a_1, a_2) \in (\mathcal{R} \rightarrow id_{\text{Bool}}). (g_{\tau_1} a_1, g_{\tau_2} a_2) \in ([\mathcal{R}] \rightarrow [\mathcal{R}]) \\ \Leftrightarrow & \forall \mathcal{R}. \forall (a_1, a_2) \in (\mathcal{R} \rightarrow id_{\text{Bool}}). \forall (l_1, l_2) \in [\mathcal{R}]. \\ & \qquad \qquad \qquad (g_{\tau_1} a_1 l_1, g_{\tau_2} a_2 l_2) \in [\mathcal{R}] \\ \Rightarrow & \forall (a_1, a_2) \in (h \rightarrow id_{\text{Bool}}). \forall (l_1, l_2) \in (\text{map } h). \\ & \qquad \qquad \qquad (g_{\tau_1} a_1 l_1, g_{\tau_2} a_2 l_2) \in (\text{map } h) \\ \Rightarrow & \forall (l_1, l_2) \in (\text{map } h). (g_{\tau_1} (p \circ h) l_1, g_{\tau_2} p l_2) \in (\text{map } h) \\ \Leftrightarrow & \forall l_1 :: [\tau_1]. \text{map } h (g_{\tau_1} (p \circ h) l_1) = g_{\tau_2} p (\text{map } h l_1) \end{aligned}$$

für jede Funktion $h :: \tau_1 \rightarrow \tau_2$ und jedes $p :: \tau_2 \rightarrow \text{Bool}$.

Das entspricht genau der ursprünglichen Behauptung!

Automatische Erzeugung freier Theoreme

Auf <http://linux.tcs.inf.tu-dresden.de/~voigt/ft:>

This tool allows to generate free theorems for sublanguages of Haskell as described [here](#).

The source code of the underlying library and a shell-based application using it is available [here](#) and [here](#).

Please enter a (polymorphic) type, e.g. "(a -> Bool) -> [a] -> [a]" or simply "filter":

Please choose a sublanguage of Haskell:

- no bottoms (hence no general recursion and no selective strictness)
- general recursion but no selective strictness
- general recursion and selective strictness

Please choose a theorem style (without effect in the sublanguage with no bottoms):

- equational
- inequational

Automatische Erzeugung freier Theoreme

The theorem generated for functions of the type

```
g :: forall a . (a -> Bool) -> [a] -> [a]
```

in the sublanguage of Haskell with no bottoms is:

```
forall t1,t2 in TYPES, R in REL(t1,t2).
forall p :: t1 -> Bool.
forall q :: t2 -> Bool.
  (forall (x, y) in R. p x = q y)
==> (forall (z, v) in lift{[]}(R).
      (g p z, g q v) in lift{[]}(R))
```

The structural lifting occurring therein is defined as follows:

```
lift{[]}(R)
= {[[], []]}
  u {(x : xs, y : ys) |
      ((x, y) in R) && ((xs, ys) in lift{[]}(R))}
```

Reducing all permissible relation variables to functions yields:

```
forall t1,t2 in TYPES, f :: t1 -> t2.
forall p :: t1 -> Bool.
forall q :: t2 -> Bool.
  (forall x :: t1. p x = q (f x))
==> (forall y :: [t1]. map f (g p y) = g q (map f y))
```

Verweis auf Skript...

Bis hierher: Abschnitte 1, 2, 2.1, 2.2 (bis auf den letzten Teil, ab „Functor“), 2.3, 3.1, 8.2

Welche Möglichkeiten gibt es für $g :: [a] \rightarrow [a]$?

Formal beschrieben:

Für jedes $g :: [a] \rightarrow [a]$ gibt es:

- eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$, so dass für jedes $n \in \mathbb{N}$ aus einer Liste der Länge n mit g stets eine Liste der Länge $f(n)$ wird;
- eine Funktion $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, so dass für jede Liste einer Länge $n \in \mathbb{N}$ und für jedes $0 \leq m < f(n)$, das m -te Element der Ausgabeliste das $h(n, m)$ -te Element der Eingabeliste ist.

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Gegeben:

`reverse` :: $[a] \rightarrow [a]$

`reverse` [] = []

`reverse` (a : as) = (`reverse` as) ++ [a]

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Gegeben:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

```
reverse (reverse l) = l
```

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Gegeben:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{reverse} (\text{reverse } l) = l$$

Induktionsschritt:

2. $\text{reverse} (\text{reverse } (a : as)) = a : as$?

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Gegeben:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{reverse} (\text{reverse } l) = l$$

Induktionsschritt:

2. $\text{reverse} (\text{reverse} (a : as)) = a : as$?

$$\text{reverse} (\text{reverse} (a : as))$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Gegeben:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{reverse (reverse } l) = l$$

Induktionsschritt:

2. $\text{reverse (reverse (} a : as)) = a : as$?

$$\begin{aligned} & \text{reverse (reverse (} a : as)) \\ = & \text{reverse ((reverse } as) ++ [a]) \end{aligned}$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Gegeben:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{reverse (reverse } l) = l$$

Induktionsschritt:

2. $\text{reverse (reverse (} a : as)) = a : as ?$

$$\begin{aligned} & \text{reverse (reverse (} a : as)) \\ = & \text{reverse ((reverse } as) ++ [a]) \\ = & (\text{reverse } [a]) ++ (\text{reverse (reverse } as)) \end{aligned}$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Gegeben:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{reverse (reverse l)} = l$$

Induktionsschritt:

2. $\text{reverse (reverse (a : as))} = a : as$?

$$\begin{aligned} & \text{reverse (reverse (a : as))} \\ = & \text{reverse ((reverse as) ++ [a])} \\ = & (\text{reverse [a]}) ++ (\text{reverse (reverse as)}) \\ = & [a] ++ as \end{aligned}$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Gegeben:

```
reverse :: [a] → [a]
reverse []      = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{reverse (reverse l)} = l$$

Induktionsschritt:

2. $\text{reverse (reverse (a : as))} = a : as$?

$$\begin{aligned} & \text{reverse (reverse (a : as))} \\ = & \text{reverse ((reverse as) ++ [a])} \\ = & (\text{reverse [a]}) ++ (\text{reverse (reverse as)}) \\ = & [a] ++ as \\ = & a : as \end{aligned}$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Gegeben:

```
reverse :: [a] → [a]
reverse [] = []
reverse (a : as) = (reverse as) ++ [a]
```

Zu beweisen:

$$\text{reverse (reverse } l) = l$$

Induktionsschritt:

2. $\text{reverse (reverse (a : as))} = a : as$? OK
(mit Hilfe: $\text{reverse (} l_1 ++ l_2) = (\text{reverse } l_2) ++ (\text{reverse } l_1)$)

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Oder aber:

`reverse` :: $[a] \rightarrow [a]$

`reverse` / = `reverse'` / []

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$

`reverse'` [] $bs = bs$

`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Oder aber:

`reverse` :: $[a] \rightarrow [a]$

`reverse` / = `reverse'` / []

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$

`reverse'` [] $bs = bs$

`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Induktionsschritt:

2. `reverse` (`reverse` (a : as)) = a : as ?

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Oder aber:

`reverse` :: $[a] \rightarrow [a]$
`reverse` / = `reverse'` / []

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$
`reverse'` [] $bs = bs$
`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Induktionsschritt:

2. `reverse` (`reverse` (a : as)) = a : as ?

`reverse` (`reverse` (a : as))

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Oder aber:

```
reverse :: [a] → [a]
reverse l = reverse' l []
```

```
reverse' :: [a] → [a] → [a]
reverse' []      bs = bs
reverse' (a : as) bs = reverse' as (a : bs)
```

Induktionsschritt:

2. `reverse (reverse (a : as)) = a : as` ?

```
reverse (reverse (a : as))
= reverse' (reverse' (a : as) []) []
```

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Oder aber:

`reverse` :: $[a] \rightarrow [a]$
`reverse` / = `reverse'` / []

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$
`reverse'` [] $bs = bs$
`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Induktionsschritt:

2. `reverse` (`reverse` (a : as)) = a : as ?

`reverse` (`reverse` (a : as))
= `reverse'` (`reverse'` (a : as) []) []
= `reverse'` (`reverse'` as [a]) []

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Oder aber:

`reverse` :: $[a] \rightarrow [a]$
`reverse` / = `reverse'` / []

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$
`reverse'` [] $bs = bs$
`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Induktionsschritt:

2. `reverse` (`reverse` (a : as)) = a : as ?

`reverse` (`reverse` (a : as))
= `reverse'` (`reverse'` (a : as) []) []
= `reverse'` (`reverse'` as [a]) []
?

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Oder aber:

`reverse` :: $[a] \rightarrow [a]$
`reverse` $l = \text{reverse}' l []$

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$
`reverse'` [] $bs = bs$
`reverse'` (a : as) $bs = \text{reverse}' as (a : bs)$

Induktionsschritt:

2. `reverse` (`reverse` (a : as)) = a : as ?

`reverse` (`reverse` (a : as))
= `reverse'` (`reverse'` (a : as) []) []
= `reverse'` (`reverse'` as [a]) []
?
a : as

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Oder aber:

`reverse` :: $[a] \rightarrow [a]$
`reverse` / = `reverse'` / []

`reverse'` :: $[a] \rightarrow [a] \rightarrow [a]$
`reverse'` [] $bs = bs$
`reverse'` (a : as) bs = `reverse'` as (a : bs)

Induktionsschritt:

2. `reverse` (`reverse` (a : as)) = a : as ?

`reverse` (`reverse` (a : as))
= `reverse'` (`reverse'` (a : as) []) []
= `reverse'` (`reverse'` as [a]) []
?
= a : as

Könnte ein freies Theorem helfen?

Wir haben:

$$\text{map } h (\text{reverse } l) = \text{reverse } (\text{map } h l)$$

Könnte ein freies Theorem helfen?

Wir haben:

$$\text{map } h (\text{reverse } l) = \text{reverse } (\text{map } h l)$$

Leider hilft das für

$$\text{reverse } (\text{reverse } l) = l$$

nicht wirklich weiter.

Könnte ein freies Theorem helfen?

Wir haben:

$$\text{map } h (\text{reverse } l) = \text{reverse } (\text{map } h l)$$

Leider hilft das für

$$\text{reverse } (\text{reverse } l) = l$$

nicht wirklich weiter.

Und sollte es auch nicht, denn

$$g (g l) = l$$

gilt ganz sicher **nicht** für jedes $g :: [a] \rightarrow [a]!$

Dennoch besteht Hoffnung...

Wir wissen:

Für jedes $g :: [a] \rightarrow [a]$ gibt es:

- eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$, so dass für jedes $n \in \mathbb{N}$ aus einer Liste der Länge n mit g stets eine Liste der Länge $f(n)$ wird;
- eine Funktion $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, so dass für jede Liste einer Länge $n \in \mathbb{N}$ und für jedes $0 \leq m < f(n)$, das m -te Element der Ausgabeliste das $h(n, m)$ -te Element der Eingabeliste ist.

Dennoch besteht Hoffnung...

Wir wissen:

Für jedes $g :: [a] \rightarrow [a]$ gibt es:

- eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$, so dass für jedes $n \in \mathbb{N}$ aus einer Liste der Länge n mit g stets eine Liste der Länge $f(n)$ wird;
- eine Funktion $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, so dass für jede Liste einer Länge $n \in \mathbb{N}$ und für jedes $0 \leq m < f(n)$, das m -te Element der Ausgabeliste das $h(n, m)$ -te Element der Eingabeliste ist.

Also:

Für jede Eingabeliste l :

$$g \ l = \mathbf{let} \ n = \mathbf{length} \ l \ \mathbf{in} \ [l \ !! \ h(n, m) \ | \ m \leftarrow [0..f(n) - 1]]$$

Dennoch besteht Hoffnung...

Wir wissen:

Für jedes $g :: [a] \rightarrow [a]$ gibt es:

- eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$, so dass für jedes $n \in \mathbb{N}$ aus einer Liste der Länge n mit g stets eine Liste der Länge $f(n)$ wird;
- eine Funktion $h : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$, so dass für jede Liste einer Länge $n \in \mathbb{N}$ und für jedes $0 \leq m < f(n)$, das m -te Element der Ausgabeliste das $h(n, m)$ -te Element der Eingabeliste ist.

Also:

Für jede Eingabeliste l :

$$g \ l = \mathbf{let} \ n = \mathbf{length} \ l \ \mathbf{in} \ [l !! h(n, m) \mid m \leftarrow [0..f(n) - 1]]$$

Im Fall von $g = \mathbf{reverse}$:

$$\begin{aligned} f(n) &= n \\ h(n, m) &= n - 1 - m \end{aligned}$$

Dennoch besteht Hoffnung...

Somit:

Für jede Eingabeliste l :

```
reverse l = let n = length l in [l!!(n-1-m) | m ← [0..n-1]]
```

Dennoch besteht Hoffnung...

Somit:

Für jede Eingabeliste l :

```
reverse l = let n = length l in [l!!(n-1-m) | m ← [0..n-1]]
```

Beweisversuch mit dieser Darstellung:

```
reverse (reverse l)
```

Dennoch besteht Hoffnung...

Somit:

Für jede Eingabeliste l :

`reverse l = let n = length l in [l!!(n-1-m) | m ← [0..n-1]]`

Beweisversuch mit dieser Darstellung:

`reverse (reverse l)`
`= reverse (let n = length l in [l!!(n-1-m) | m ← [0..n-1]])`

Dennoch besteht Hoffnung...

Somit:

Für jede Eingabeliste l :

`reverse l = let n = length l in [l!!(n-1-m) | m ← [0..n-1]]`

Beweisversuch mit dieser Darstellung:

```
reverse (reverse l)
= reverse (let n = length l in [l!!(n-1-m) | m ← [0..n-1]])
= let n = length l in reverse [l!!(n-1-m) | m ← [0..n-1]]
```

Dennoch besteht Hoffnung...

Somit:

Für jede Eingabeliste l :

`reverse l = let n = length l in [l!!(n-1-m) | m ← [0..n-1]]`

Beweisversuch mit dieser Darstellung:

```
reverse (reverse l)
= reverse (let n = length l in [l!!(n-1-m) | m ← [0..n-1]])
= let n = length l in reverse [l!!(n-1-m) | m ← [0..n-1]]
= let n = length l
    l' = [l!!(n-1-m) | m ← [0..n-1]]
    in reverse l'
```


Dennoch besteht Hoffnung...

Somit:

Für jede Eingabeliste l :

`reverse l = let n = length l in [l!!(n-1-m) | m ← [0..n-1]]`

Beweisversuch mit dieser Darstellung:

```
reverse (reverse l)
= reverse (let n = length l in [l!!(n-1-m) | m ← [0..n-1]])
= let n = length l in reverse [l!!(n-1-m) | m ← [0..n-1]]
= let n = length l
    l' = [l!!(n-1-m) | m ← [0..n-1]]
    in reverse l'
= let n = length l
    l' = [l!!(n-1-m) | m ← [0..n-1]]
    n' = length l'
    in [l'!!(n'-1-m') | m' ← [0..n'-1]]
```

Dennoch besteht Hoffnung...

Beweisversuch mit dieser Darstellung:

```
reverse (reverse l)
= reverse (let n = length l in [l!!(n - 1 - m) | m ← [0..n - 1]])
= let n = length l in reverse [l!!(n - 1 - m) | m ← [0..n - 1]]
= let n = length l
    l' = [l!!(n - 1 - m) | m ← [0..n - 1]]
    in reverse l'
= let n = length l
    l' = [l!!(n - 1 - m) | m ← [0..n - 1]]
    n' = length l'
    in [l'!!(n' - 1 - m') | m' ← [0..n' - 1]]
```

Dennoch besteht Hoffnung...

Beweisversuch mit dieser Darstellung:

```
reverse (reverse l)
= reverse (let n = length l in [l!!(n - 1 - m) | m ← [0..n - 1]])
= let n = length l in reverse [l!!(n - 1 - m) | m ← [0..n - 1]]
= let n = length l
    l' = [l!!(n - 1 - m) | m ← [0..n - 1]]
    in reverse l'
= let n = length l
    l' = [l!!(n - 1 - m) | m ← [0..n - 1]]
    n' = length l'
    in [l'!!(n' - 1 - m') | m' ← [0..n' - 1]]
= let n = length l
    n' = n
    in [[l!!(n - 1 - m) | m ← [0..n - 1]] !!(n' - 1 - m')
        | m' ← [0..n' - 1]]
```

Dennoch besteht Hoffnung...

Beweisversuch mit dieser Darstellung:

```
reverse (reverse l)
= reverse (let n = length l in [l!!(n - 1 - m) | m ← [0..n - 1]])
= let n = length l in reverse [l!!(n - 1 - m) | m ← [0..n - 1]]
= let n = length l
    l' = [l!!(n - 1 - m) | m ← [0..n - 1]]
    in reverse l'
= let n = length l
    l' = [l!!(n - 1 - m) | m ← [0..n - 1]]
    n' = length l'
    in [l'!!(n' - 1 - m') | m' ← [0..n' - 1]]
= let n = length l
    n' = n
    in [[l!!(n - 1 - m) | m ← [0..n - 1]]!!(n' - 1 - m')
        | m' ← [0..n' - 1]]
= let n = length l in [l!!(n - 1 - (n - 1 - m')) | m' ← [0..n - 1]]
```

Dennoch besteht Hoffnung...

Beweisversuch mit dieser Darstellung:

```
reverse (reverse l)
= reverse (let n = length l in [l!!(n - 1 - m) | m ← [0..n - 1]])
= let n = length l in reverse [l!!(n - 1 - m) | m ← [0..n - 1]]
= let n = length l
    l' = [l!!(n - 1 - m) | m ← [0..n - 1]]
  in reverse l'
= let n = length l
    l' = [l!!(n - 1 - m) | m ← [0..n - 1]]
    n' = length l'
  in [l'!!(n' - 1 - m') | m' ← [0..n' - 1]]
= let n = length l
    n' = n
  in [[l!!(n - 1 - m) | m ← [0..n - 1]]!!(n' - 1 - m')
      | m' ← [0..n' - 1]]
= let n = length l in [l!!(n - 1 - (n - 1 - m')) | m' ← [0..n - 1]]
= let n = length l in [l!! m' | m' ← [0..n - 1]]
```

Dennoch besteht Hoffnung...

Beweisversuch mit dieser Darstellung:

```
reverse (reverse l)
= reverse (let n = length l in [l!!(n - 1 - m) | m ← [0..n - 1]])
= let n = length l in reverse [l!!(n - 1 - m) | m ← [0..n - 1]]
= let n = length l
    l' = [l!!(n - 1 - m) | m ← [0..n - 1]]
  in reverse l'
= let n = length l
    l' = [l!!(n - 1 - m) | m ← [0..n - 1]]
    n' = length l'
  in [l'!!(n' - 1 - m') | m' ← [0..n' - 1]]
= let n = length l
    n' = n
  in [[l!!(n - 1 - m) | m ← [0..n - 1]]!!(n' - 1 - m')
      | m' ← [0..n' - 1]]
= let n = length l in [l!!(n - 1 - (n - 1 - m')) | m' ← [0..n - 1]]
= let n = length l in [l!! m' | m' ← [0..n - 1]] = l
```

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Um sinnvoll mit dieser Darstellung von Funktionen umgehen zu können, Perspektivwechsel auch für Datenstruktur nötig!

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Um sinnvoll mit dieser Darstellung von Funktionen umgehen zu können, Perspektivwechsel auch für Datenstruktur nötig!

Liste über Typ τ wird dargestellt durch:

- Länge n
- Funktion $k :: \text{Int} \rightarrow \tau$ für Zuordnung Listenposition zu Inhalt

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Um sinnvoll mit dieser Darstellung von Funktionen umgehen zu können, Perspektivwechsel auch für Datenstruktur nötig!

Liste über Typ τ wird dargestellt durch:

- Länge n
- Funktion $k :: \text{Int} \rightarrow \tau$ für Zuordnung Listenposition zu Inhalt

Dann, aus

$$g \ / = \mathbf{let} \ n = \mathbf{length} \ / \ \mathbf{in} \ [\ / !! h(n, m) \ | \ m \leftarrow [0..f(n) - 1]]$$

wird

$$g \ (n, k) = (f(n), \lambda m \rightarrow k \ (h(n, m)))$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Um sinnvoll mit dieser Darstellung von Funktionen umgehen zu können, Perspektivwechsel auch für Datenstruktur nötig!

Liste über Typ τ wird dargestellt durch:

- Länge n
- Funktion $k :: \text{Int} \rightarrow \tau$ für Zuordnung Listenposition zu Inhalt

Dann, aus

$$g \ l = \text{let } n = \text{length } l \text{ in } [l !! h(n, m) \mid m \leftarrow [0..f(n) - 1]]$$

wird

$$g \ (n, k) = (f(n), \lambda m \rightarrow k \ (h(n, m)))$$

Zum Beispiel:

$$\text{reverse} \ (n, k) = (n, \lambda m \rightarrow k \ (n - 1 - m))$$

$$\text{tail} \ (n + 1, k) = (n, \lambda m \rightarrow k \ (m + 1))$$

$$\text{init} \ (n + 1, k) = (n, k)$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

```
reverse (reverse (n, k))
```

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse} (\text{reverse} (n, k)) \\ = & \text{reverse} (n, \lambda m \rightarrow k (n - 1 - m)) \end{aligned}$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse} (\text{reverse} (n, k)) \\ = & \text{reverse} (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \end{aligned}$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse} (\text{reverse} (n, k)) \\ = & \text{reverse} (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \end{aligned}$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse} (\text{reverse} (n, k)) \\ = & \text{reverse} (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \\ = & (n, \lambda m \rightarrow k m) \end{aligned}$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse} (\text{reverse} (n, k)) \\ = & \text{reverse} (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \\ = & (n, \lambda m \rightarrow k m) \\ = & (n, k) \end{aligned}$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse} (\text{reverse} (n, k)) \\ = & \text{reverse} (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \\ = & (n, \lambda m \rightarrow k m) \\ = & (n, k) \end{aligned}$$

Oder auch:

$$\text{reverse} (\text{tail} (n + 1, k))$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse} (\text{reverse} (n, k)) \\ = & \text{reverse} (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \\ = & (n, \lambda m \rightarrow k m) \\ = & (n, k) \end{aligned}$$

Oder auch:

$$\text{reverse} (\text{tail} (n + 1, k))$$

$$\text{init} (\text{reverse} (n + 1, k))$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse} (\text{reverse} (n, k)) \\ = & \text{reverse} (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \\ = & (n, \lambda m \rightarrow k m) \\ = & (n, k) \end{aligned}$$

Oder auch:

$$\begin{aligned} & \text{reverse} (\text{tail} (n + 1, k)) \\ = & \text{reverse} (n, \lambda m \rightarrow k (m + 1)) \end{aligned}$$

$$\text{init} (\text{reverse} (n + 1, k))$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse } (\text{reverse } (n, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \\ = & (n, \lambda m \rightarrow k m) \\ = & (n, k) \end{aligned}$$

Oder auch:

$$\begin{aligned} & \text{reverse } (\text{tail } (n + 1, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (m + 1)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (m + 1)) (n - 1 - m)) \end{aligned}$$

$$\text{init } (\text{reverse } (n + 1, k))$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse } (\text{reverse } (n, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \\ = & (n, \lambda m \rightarrow k m) \\ = & (n, k) \end{aligned}$$

Oder auch:

$$\begin{aligned} & \text{reverse } (\text{tail } (n + 1, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (m + 1)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (m + 1)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k ((n - 1 - m) + 1)) \end{aligned}$$

$$\text{init } (\text{reverse } (n + 1, k))$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse } (\text{reverse } (n, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \\ = & (n, \lambda m \rightarrow k m) \\ = & (n, k) \end{aligned}$$

Oder auch:

$$\begin{aligned} & \text{reverse } (\text{tail } (n + 1, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (m + 1)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (m + 1)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k ((n - 1 - m) + 1)) \\ = & (n, \lambda m \rightarrow k (n - m)) \end{aligned}$$

$$\text{init } (\text{reverse } (n + 1, k))$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse } (\text{reverse } (n, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \\ = & (n, \lambda m \rightarrow k m) \\ = & (n, k) \end{aligned}$$

Oder auch:

$$\begin{aligned} & \text{reverse } (\text{tail } (n + 1, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (m + 1)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (m + 1)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k ((n - 1 - m) + 1)) \\ = & (n, \lambda m \rightarrow k (n - m)) \\ \\ & \text{init } (n + 1, \lambda m \rightarrow k (n + 1 - 1 - m)) \\ = & \text{init } (\text{reverse } (n + 1, k)) \end{aligned}$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse } (\text{reverse } (n, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \\ = & (n, \lambda m \rightarrow k m) \\ = & (n, k) \end{aligned}$$

Oder auch:

$$\begin{aligned} & \text{reverse } (\text{tail } (n + 1, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (m + 1)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (m + 1)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k ((n - 1 - m) + 1)) \\ = & (n, \lambda m \rightarrow k (n - m)) \\ & (n, \lambda m \rightarrow k (n + 1 - 1 - m)) \\ = & \text{init } (n + 1, \lambda m \rightarrow k (n + 1 - 1 - m)) \\ = & \text{init } (\text{reverse } (n + 1, k)) \end{aligned}$$

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$\begin{aligned} & \text{reverse } (\text{reverse } (n, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (n - 1 - m)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (n - 1 - m)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k (n - 1 - (n - 1 - m))) \\ = & (n, \lambda m \rightarrow k m) \\ = & (n, k) \end{aligned}$$

Oder auch:

$$\begin{aligned} & \text{reverse } (\text{tail } (n + 1, k)) \\ = & \text{reverse } (n, \lambda m \rightarrow k (m + 1)) \\ = & (n, \lambda m \rightarrow (\lambda m \rightarrow k (m + 1)) (n - 1 - m)) \\ = & (n, \lambda m \rightarrow k ((n - 1 - m) + 1)) \\ = & (n, \lambda m \rightarrow k (n - m)) \\ = & (n, \lambda m \rightarrow k (n + 1 - 1 - m)) \\ = & \text{init } (n + 1, \lambda m \rightarrow k (n + 1 - 1 - m)) \\ = & \text{init } (\text{reverse } (n + 1, k)) \end{aligned}$$

Verweis auf Skript...

Bis vorhin: Abschnitte 1, 2, 2.1, 2.2 (bis auf den letzten Teil, ab „Functor“), 2.3, 3.1, 8.2

Verweis auf Skript...

Bis vorhin: Abschnitte 1, 2, 2.1, 2.2 (bis auf den letzten Teil, ab „Functor“), 2.3, 3.1, 8.2

Der Rest von heute nicht im Skript.

Aber Referenzen: [Bundy & Richardson 1999], [Prince et al. 2008]