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

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 :: $[a] \to [a]$ gibt es:

- eine Funktion $f : \mathbb{N} \to \mathbb{N}$, 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 $h : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, 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

Ein kompositioneller Ansatz

Frage: Welche g haben einen gewissen Typ τ?

Ansatz: Denotationen $[\![\tau]\!]$ von Typen als Mengen angeben.

$$
\begin{array}{lcl}
[\![\mathsf{Bool}]\!] & = & \{\mathsf{True}, \mathsf{False}\} \\
[\![\mathsf{Int}]\!] & = & \{\ldots, -2, -1, 0, 1, 2, \ldots\} \\
[\![(\tau_1, \tau_2)]\!] & = & [\![\tau_1]\!] \times [\![\tau_2]\!] \\
[\![[\tau]]\!] & = & \{[x_1, \ldots, x_n] \mid n \geq 0, x_i \in [\![\tau]\!]\} \\
[\![\tau_1 \to \tau_2]\!] & = & \{f : [\![\tau_1]\!] \to [\![\tau_2]\!]\} \\
[\![\forall \alpha.\tau]\!] & = & ?
\end{array}
$$

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

Unerwünschte Ad-Hoc-Polymorphie am Beispiel

- Mit der vorgeschlagenen Definition,
  $[\![\forall\alpha.\ (\alpha,\alpha)\to\alpha]\!] = \{g \mid \forall\tau.\ g_\tau : [\![\tau]\!]\times[\![\tau]\!]\to[\![\tau]\!]\}$.
- Aber das erlaubt auch

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

  was in Haskell beim Typ $\forall\alpha.\ (\alpha,\alpha)\to\alpha$ unmöglich ist.
- Um dies zu vermeiden, müssen wir

$$\begin{aligned} g_{\mathsf{Bool}} &: [\![\mathsf{Bool}]\!]\times[\![\mathsf{Bool}]\!]\to[\![\mathsf{Bool}]\!] \quad \text{und} \\ g_{\mathsf{Int}} &: [\![\mathsf{Int}]\!]\times[\![\mathsf{Int}]\!]\to[\![\mathsf{Int}]\!] \end{aligned}$$

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

Idee [Reynolds 1983]

Beliebige Relationen benutzen, um Instanzen zu verknüpfen!

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

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

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

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

Reynolds: $g \in [\![\forall\alpha.\tau]\!]$ genau dann wenn für alle τ_1, τ_2 und $\mathcal{R} \subseteq [\![\tau_1]\!] \times [\![\tau_2]\!]$ 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.
2. Ersetze Teiltypen ohne jeglichen Polymorphismus durch Identitätsrelationen.
3. Verwende folgende Regeln:

$$
\begin{aligned}
(\mathcal{R}, \mathcal{S}) &= \{((x_1, x_2), (y_1, y_2)) \mid (x_1, y_1) \in \mathcal{R}, (x_2, y_2) \in \mathcal{S}\} \\
[\mathcal{R}] &= \{([x_1, \ldots, x_n], [y_1, \ldots, y_n]) \mid n \geq 0, (x_i, y_i) \in \mathcal{R}\} \\
\mathcal{R} \to \mathcal{S} &= \{(f_1, f_2) \mid \forall (a_1, a_2) \in \mathcal{R}.\ (f_1\ a_1, f_2\ a_2) \in \mathcal{S}\} \\
\forall \mathcal{R}.\ \mathcal{F}(\mathcal{R}) &= \{(u, v) \mid \forall \tau_1, \tau_2, \mathcal{R} \subseteq \tau_1 \times \tau_2.\ (u_{\tau_1}, v_{\tau_2}) \in \mathcal{F}(\mathcal{R})\}
\end{aligned}
$$

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

Nun formal am Beispiel

Gegeben sei $\mathtt{g} :: \forall\alpha.\ (\alpha \to \mathsf{Bool}) \to ([\alpha] \to [\alpha])$.

Dann:

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

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

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":

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

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

[Generate]

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))
```

Export as PDF Show type instantiations Enter a new type Help page

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

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Gegeben:

$$\begin{aligned}
&\texttt{reverse} :: [a] \to [a] \\
&\texttt{reverse}\ [\,] \quad\ = [\,] \\
&\texttt{reverse}\ (a : as) = (\texttt{reverse}\ as) +\!\!+ [a]
\end{aligned}$$

Zu beweisen:

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

Induktionsschritt:

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

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

Ein Beweisprinzip für polymorphe Funktionen auf Listen?

Oder aber:

$$
\begin{aligned}
&\texttt{reverse} :: [a] \to [a] \\
&\texttt{reverse}\ l = \texttt{reverse}'\ l\ [\,] \\
\\
&\texttt{reverse}' :: [a] \to [a] \to [a] \\
&\texttt{reverse}'\ [\,]\ \ \ \ \ \ \ \ bs = bs \\
&\texttt{reverse}'\ (a : as)\ bs = \texttt{reverse}'\ as\ (a : bs)
\end{aligned}
$$

Induktionsschritt:

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

$$
\begin{aligned}
& \texttt{reverse}\ (\texttt{reverse}\ (a : as)) \\
= \ & \texttt{reverse}'\ (\texttt{reverse}'\ (a : as)\ [\,])\ [\,] \\
= \ & \texttt{reverse}'\ (\texttt{reverse}'\ as\ [a])\ [\,] \\
? \\
= \ & a : as
\end{aligned}
$$

Könnte ein freies Theorem helfen?

Wir haben:

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

Leider hilft das für

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

nicht wirklich weiter.

Und sollte es auch nicht, denn

$$\mathrm{g}\ (\mathrm{g}\ l)\ =\ l$$

gilt ganz sicher nicht für jedes $\mathrm{g} :: [a] \to [a]$!

Dennoch besteht Hoffnung. . .

Wir wissen:

Für jedes `g` :: $[a] \to [a]$ gibt es:

- eine Funktion $f : \mathbb{N} \to \mathbb{N}$, 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 $h : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, 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:

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

Im Fall von `g` = `reverse`:

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

Dennoch besteht Hoffnung...

Somit:

Für jede Eingabeliste l:

$$\texttt{reverse}\ l\ =\ \textbf{let}\ n = \texttt{length}\ l\ \textbf{in}\ [l\,!!(n-1-m)\mid m \leftarrow [0..n-1]]$$

Beweisversuch mit dieser Darstellung:

$$\begin{aligned}
& \texttt{reverse}\ (\texttt{reverse}\ l) \\
= \quad & \texttt{reverse}\ (\textbf{let}\ n = \texttt{length}\ l\ \textbf{in}\ [l\,!!(n-1-m)\mid m \leftarrow [0..n-1]]) \\
= \quad & \textbf{let}\ n = \texttt{length}\ l\ \textbf{in}\ \texttt{reverse}\ [l\,!!(n-1-m)\mid m \leftarrow [0..n-1]] \\
= \quad & \textbf{let}\ n = \texttt{length}\ l \\
& \qquad l' = [l\,!!(n-1-m)\mid m \leftarrow [0..n-1]] \\
& \textbf{in}\ \texttt{reverse}\ l' \\
= \quad & \textbf{let}\ n\ = \texttt{length}\ l \\
& \qquad l'\ = [l\,!!(n-1-m)\mid m \leftarrow [0..n-1]] \\
& \qquad n' = \texttt{length}\ l' \\
& \textbf{in}\ [l'\,!!(n'-1-m')\mid m' \leftarrow [0..n'-1]]
\end{aligned}$$

Dennoch besteht Hoffnung...

Beweisversuch mit dieser Darstellung:

$$
\begin{aligned}
& \texttt{reverse}\ (\texttt{reverse}\ l) \\
= \quad & \texttt{reverse}\ (\textbf{let}\ n = \texttt{length}\ l\ \textbf{in}\ [l\,!!(n-1-m) \mid m \leftarrow [0..n-1]]) \\
= \quad & \textbf{let}\ n = \texttt{length}\ l\ \textbf{in}\ \texttt{reverse}\ [l\,!!(n-1-m) \mid m \leftarrow [0..n-1]] \\
= \quad & \textbf{let}\ n = \texttt{length}\ l \\
& \qquad l' = [l\,!!(n-1-m) \mid m \leftarrow [0..n-1]] \\
& \textbf{in}\ \texttt{reverse}\ l' \\
= \quad & \textbf{let}\ n\ = \texttt{length}\ l \\
& \qquad l'\ = [l\,!!(n-1-m) \mid m \leftarrow [0..n-1]] \\
& \qquad n' = \texttt{length}\ l' \\
& \textbf{in}\ [l'\,!!(n'-1-m') \mid m' \leftarrow [0..n'-1]] \\
= \quad & \textbf{let}\ n\ = \texttt{length}\ l \\
& \qquad n' = n \\
& \textbf{in}\ [[l\,!!(n-1-m) \mid m \leftarrow [0..n-1]]\,!!(n'-1-m') \\
& \qquad \mid m' \leftarrow [0..n'-1]] \\
= \quad & \textbf{let}\ n = \texttt{length}\ l\ \textbf{in}\ [l\,!!(n-1-(n-1-m')) \mid m' \leftarrow [0..n-1]] \\
= \quad & \textbf{let}\ n = \texttt{length}\ l\ \textbf{in}\ [l\,!!\,m' \mid m' \leftarrow [0..n-1]] \quad = \quad l
\end{aligned}
$$

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 $k :: \mathsf{Int} \to \tau$ für Zuordnung Listenposition zu Inhalt

Dann, aus

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

wird

$$\mathtt{g}\ (n,k)\ =\ (f(n), \lambda m \to k\ (h(n,m)))$$

Zum Beispiel:

$$\mathtt{reverse}\ (n,k)\ =\ (n, \lambda m \to k\ (n-1-m))$$

$$\mathtt{tail}\ (n+1,k)\ =\ (n, \lambda m \to k\ (m+1))$$

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

Ein Beweisprinzip für polymorphe Funktionen auf Listen!

Jetzt Beweis einfach:

$$
\begin{aligned}
& \texttt{reverse}\ (\texttt{reverse}\ (n, k)) \\
= \quad & \texttt{reverse}\ (n, \lambda m \to k\ (n-1-m)) \\
= \quad & (n, \lambda m \to (\lambda m \to k\ (n-1-m))\ (n-1-m)) \\
= \quad & (n, \lambda m \to k\ (n-1-(n-1-m))) \\
= \quad & (n, \lambda m \to k\ m) \\
= \quad & (n, k)
\end{aligned}
$$

Oder auch:

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

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

Der Rest von heute nicht im Skript.
Aber Referenzen: [Bundy & Richardson 1999], [Prince et al. 2008]