Die Komplement-Funktion

$$\textbf{type}\ \mathsf{IntMap}\ \alpha = [(\mathsf{Int}, \alpha)]$$

$$\begin{aligned}
&\texttt{compl} :: [\alpha] \to (\mathsf{Int}, \mathsf{IntMap}\ \alpha)\\
&\texttt{compl}\ s = \textbf{let}\ n = (\texttt{length}\ s) - 1\\
&\qquad\qquad\quad\ t = [0..n]\\
&\qquad\qquad\quad\ g = \texttt{zip}\ t\ s\\
&\qquad\qquad\quad\ g' = \texttt{filter}\ (\lambda(i, _) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g\\
&\qquad\qquad\ \textbf{in}\ \ (n+1, g')
\end{aligned}$$

Zum Beispiel:

$$\begin{aligned}
\texttt{get} = \texttt{tail} \quad &\rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [(0, \text{'a'})])\\
\texttt{get} = \texttt{take}\ 3 \quad &\rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [(3, \text{'d'}), (4, \text{'e'})])\\
\texttt{get} = \texttt{reverse} \quad &\rightsquigarrow \quad \texttt{compl}\ \text{“abcde”} = (5, [\,])
\end{aligned}$$

Ein Inverses zu $\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$

$$
\begin{aligned}
&\texttt{inv} :: ([\alpha], (\mathsf{Int}, \mathsf{IntMap}\ \alpha)) \to [\alpha] \\
&\texttt{inv}\ (v', (n+1, g')) = \textbf{let}\ t\ \ = [0..n] \\
&\qquad\qquad\qquad\qquad\qquad\quad h\ \ = \texttt{assoc}^{\dagger}\ (\texttt{get}\ t)\ v' \\
&\qquad\qquad\qquad\qquad\qquad\quad h' = h +\!\!+ g' \\
&\qquad\qquad\qquad\qquad\quad \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{aligned}
$$

Zum Beispiel:

$$
\begin{aligned}
\texttt{get} = \texttt{tail} \quad &\rightsquigarrow \quad \texttt{inv}\ (\text{“bcde”}, (5, [(0, \text{'a'})])) = \text{“abcde”} \\
\texttt{get} = \texttt{take}\ 3 \quad &\rightsquigarrow \quad \texttt{inv}\ (\text{“xyz”}, (5, [(3, \text{'d'}), (4, \text{'e'})])) = \text{“xyzde”}
\end{aligned}
$$

Formal zu beweisen:

- $\texttt{inv}\ (\texttt{get}\ s, \texttt{compl}\ s) = s$
- wenn $\texttt{inv}\ (v, c)$ definiert, dann $\texttt{get}\ (\texttt{inv}\ (v, c)) = v$
- wenn $\texttt{inv}\ (v, c)$ definiert, dann $\texttt{compl}\ (\texttt{inv}\ (v, c)) = c$

† Für den Moment, kann als `zip` angenommen werden.

Insgesamt:

$$\textbf{type}\ \mathsf{IntMap}\ \alpha = [(\mathsf{Int}, \alpha)]$$

$$\begin{aligned}
&\texttt{compl} :: [\alpha] \to (\mathsf{Int}, \mathsf{IntMap}\ \alpha)\\
&\texttt{compl}\ s = \textbf{let}\ n = (\texttt{length}\ s) - 1\\
&\qquad\qquad\quad\ t = [0..n]\\
&\qquad\qquad\quad\ g = \texttt{zip}\ t\ s\\
&\qquad\qquad\quad\ g' = \texttt{filter}\ (\lambda(i, _) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g\\
&\qquad\qquad\ \textbf{in}\ \ (n+1, g')
\end{aligned}$$

$$\begin{aligned}
&\texttt{inv} :: ([\alpha], (\mathsf{Int}, \mathsf{IntMap}\ \alpha)) \to [\alpha]\\
&\texttt{inv}\ (v', (n+1, g')) = \textbf{let}\ t = [0..n]\\
&\qquad\qquad\qquad\qquad\quad\ h = \texttt{assoc}\ (\texttt{get}\ t)\ v'\\
&\qquad\qquad\qquad\qquad\quad\ h' = h +\!\!+ g'\\
&\qquad\qquad\qquad\qquad\ \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{aligned}$$

$$\begin{aligned}
&\texttt{put} :: [\alpha] \to [\alpha] \to [\alpha]\\
&\texttt{put}\ s\ v' = \texttt{inv}\ (v', \texttt{compl}\ s)
\end{aligned}$$

„Fusion“

Inlining von `compl` und `inv` in `put`:

$$
\begin{aligned}
\texttt{put}\ s\ v' = \textbf{let}\ n &= (\texttt{length}\ s) - 1 \\
t &= [0..n] \\
g &= \texttt{zip}\ t\ s \\
g' &= \texttt{filter}\ (\lambda(i, \_) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g \\
h &= \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
h' &= h +\!\!+ g' \\
\textbf{in}\ &\ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{assoc}\ [\,]\ \ \ [\,]\ \ \ &= [\,] \\
\texttt{assoc}\ (i : is)\ (b : bs) &= \textbf{let}\ m = \texttt{assoc}\ is\ bs \\
&\quad \textbf{in}\ \ \textbf{case}\ \texttt{lookup}\ i\ m\ \textbf{of} \\
&\qquad \text{Nothing} \to (i, b) : m \\
&\qquad \text{Just}\ c \mid b == c \to m
\end{aligned}
$$

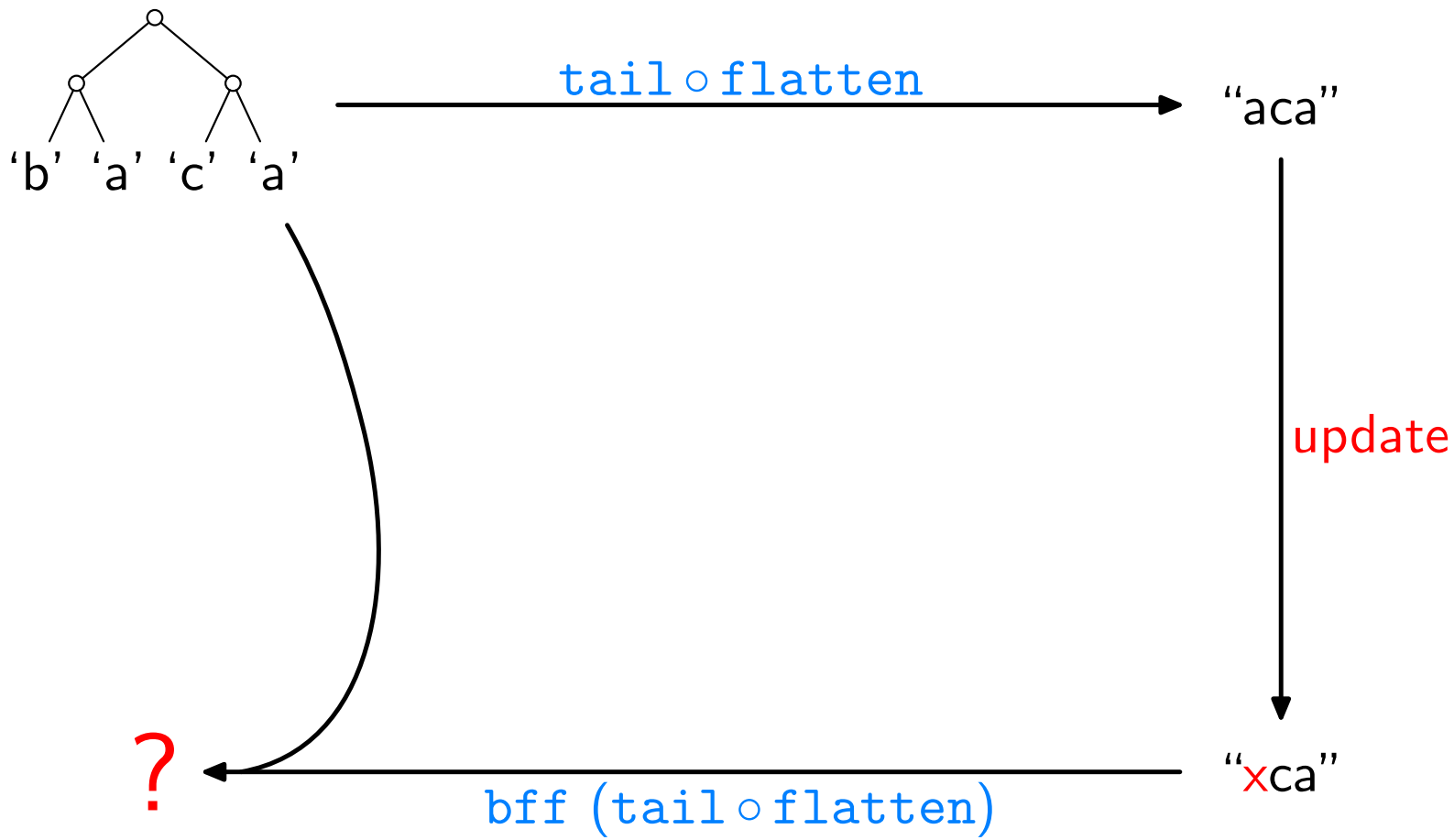
„Fusion“

Inlining von `compl` und `inv` in `put`:

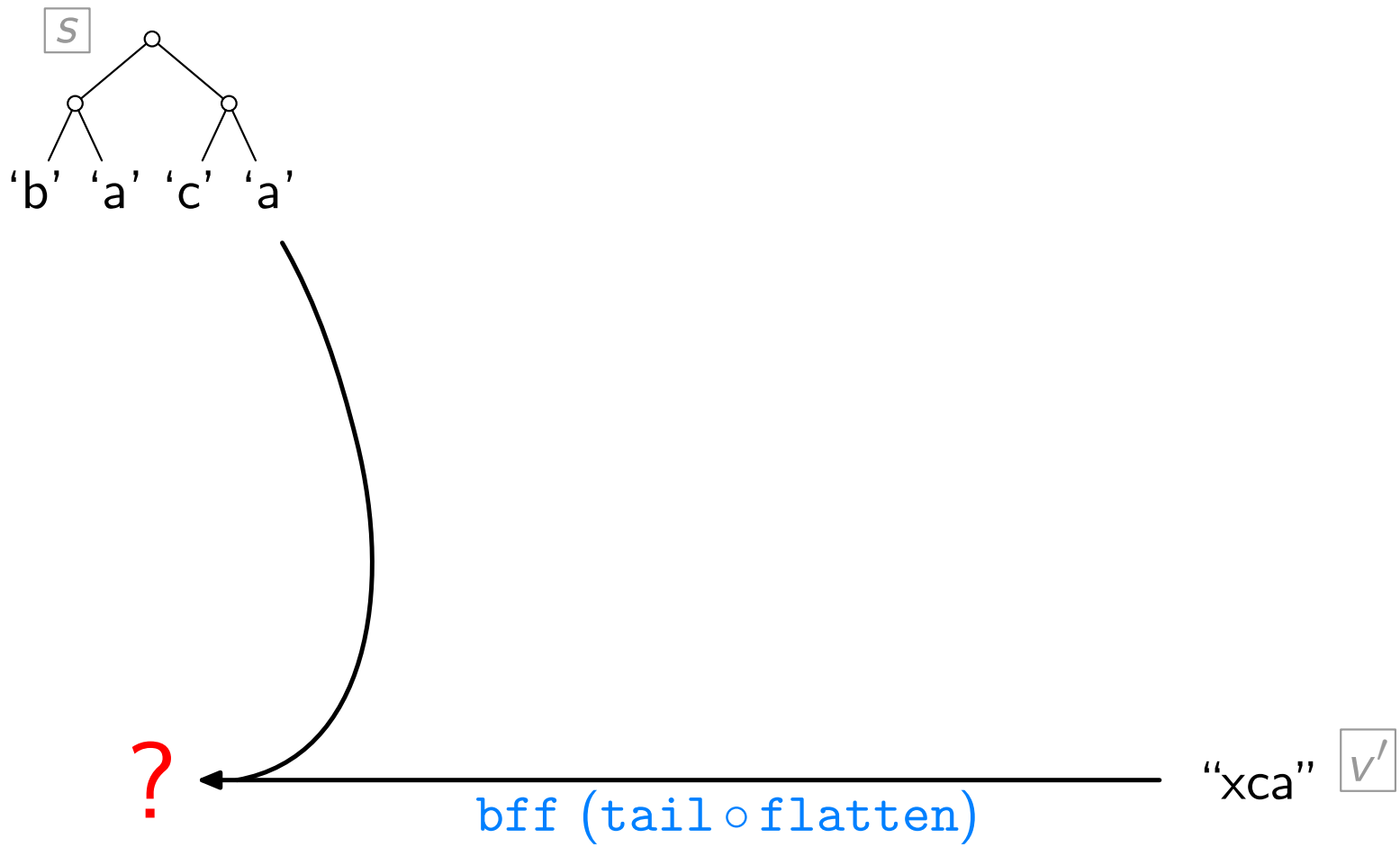
```
bff get s v' = let n  = (length s) - 1
                   t  = [0..n]
                   g  = zip t s
                   g' = filter (λ(i, _) → notElem i (get t)) g
                   h  = assoc (get t) v'
                   h' = h ++ g'
               in  seq h (map (λi → fromJust (lookup i h')) t)

  assoc []       []       = []
  assoc (i : is) (b : bs) = let m = assoc is bs
                            in  case lookup i m of
                                  Nothing          → (i, b) : m
                                  Just c | b == c → m
```

Tatsächlicher Code nur geringfügig ausgeklügelter!

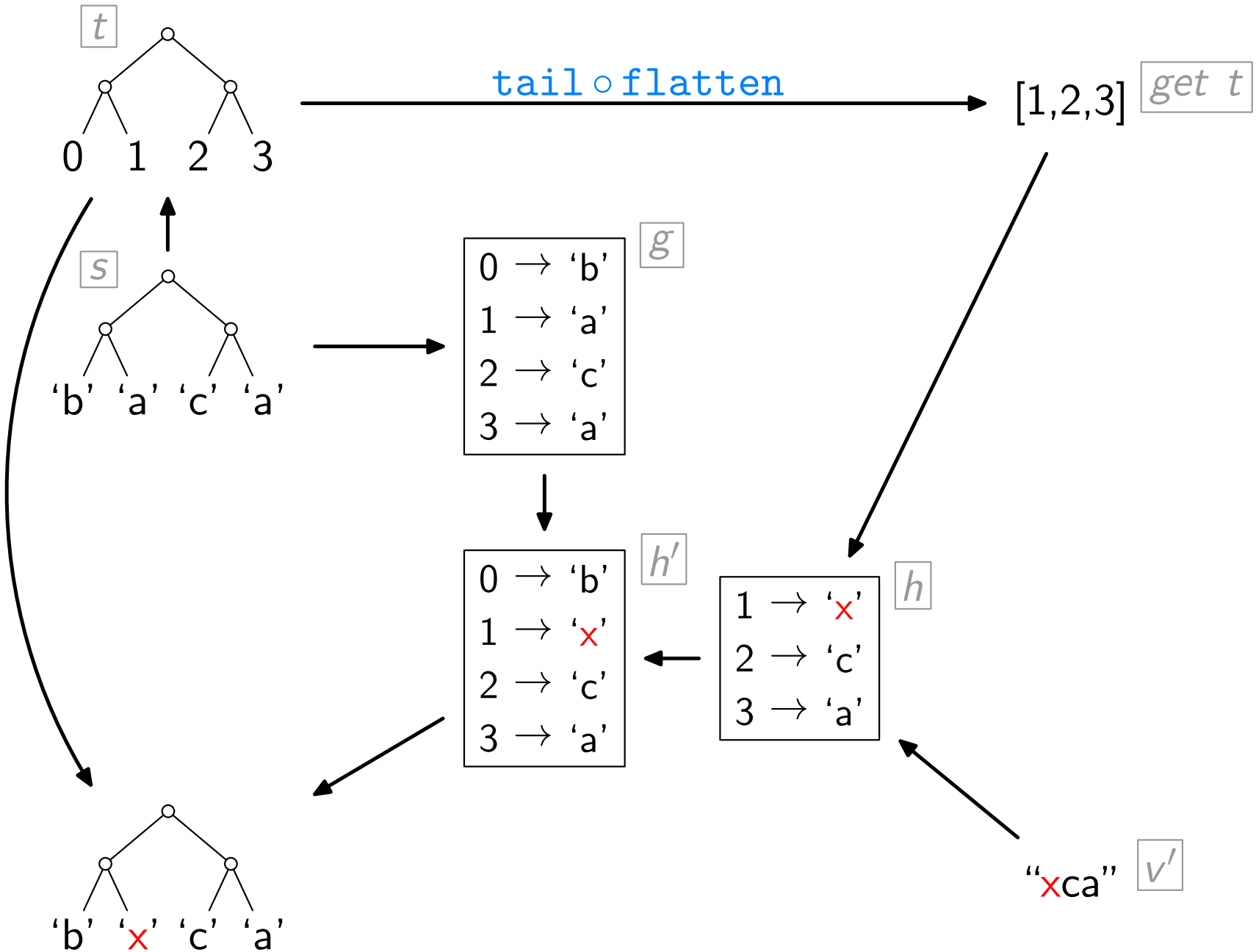
Die resultierende Bidirektionalisierungs-Methode in Aktion



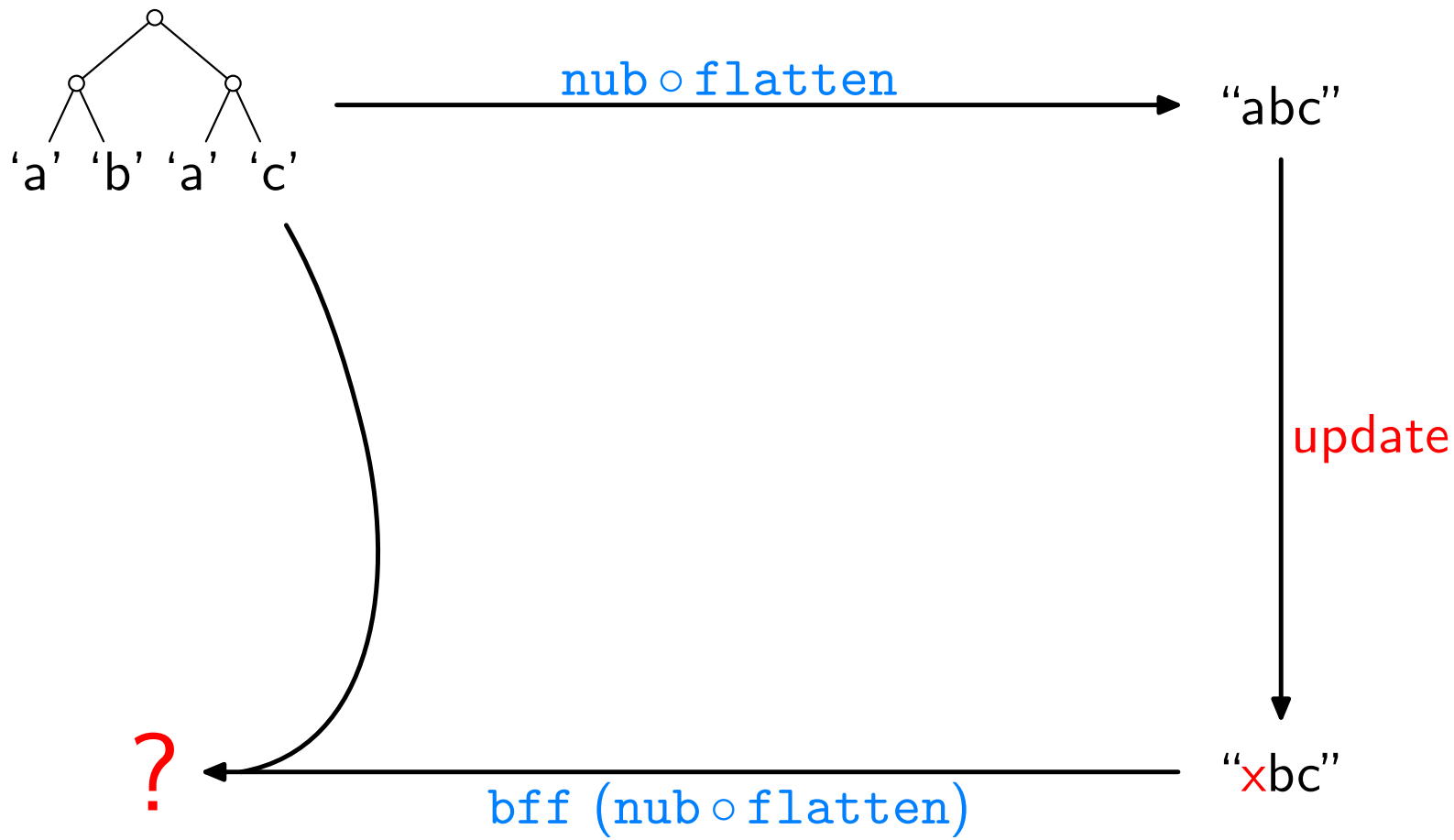
Die resultierende Bidirektionalisierungs-Methode in Aktion



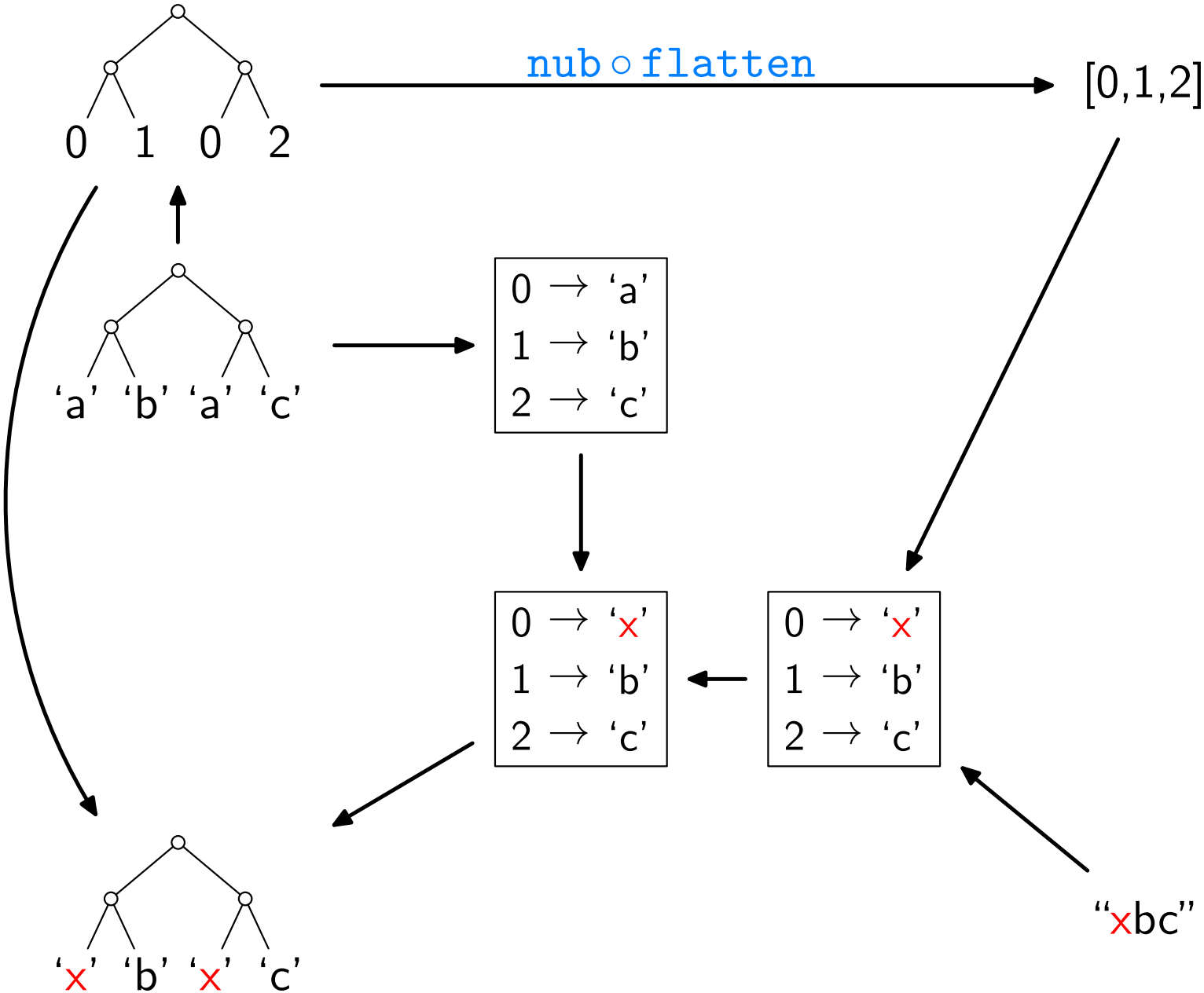
Die resultierende Bidirektionalisierungs-Methode in Aktion



Ein weiteres interessantes Beispiel (mit Typklasse Eq)



Ein weiteres interessantes Beispiel (mit Typklasse Eq)



Andere Erweiterung der Methode

Ein Hauptproblem:

- Shape-verändernde Updates führen zu Fehlschlag.
- Zum Beispiel, `bff tail` “abcde” “xyz” . . .

Weil:

- Unser Ansatz zur „Injektivierung“ von

$$\lambda s \to (\texttt{get}\ s, \texttt{compl}\ s)$$

  war, mittels `compl` folgende Informationen festzuhalten:

  1. Länge der ursprünglichen Liste
  2. verworfene Listenelemente

- Derart konservativ zu sein, ist oft nicht „nicht-injektiv“ genug.
- Zum Beispiel:
  `get = tail` $\rightsquigarrow$ `put` “abcde” “xyz” schlägt fehl genau weil `compl` “abcde” $= (5, [(0, \text{'a'})])$

Annahme von Shape-Injektivität

Angenommen, es gäbe eine Funktion

$$\texttt{shapeInv} :: \mathsf{Int} \to \mathsf{Int}$$

so dass, für jede ursprüngliche Liste s,

$$\texttt{length}\ s = \texttt{shapeInv}\ (\texttt{length}\ (\texttt{get}\ s))$$

Dann:

$$\begin{aligned}
&\texttt{compl} :: [\alpha] \to && \mathsf{IntMap}\ \alpha \\
&\texttt{compl}\ s = \mathbf{let}\ n &&= (\texttt{length}\ s) - 1 \\
&\qquad\qquad\ \ t &&= [0..n] \\
&\qquad\qquad\ \ g &&= \texttt{zip}\ t\ s \\
&\qquad\qquad\ \ g' &&= \texttt{filter}\ (\lambda(i, _) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g \\
&\qquad\ \mathbf{in} && \quad g'
\end{aligned}$$

Annahme von Shape-Injektivität

$$
\begin{aligned}
&\texttt{inv} :: ([\alpha], \quad \text{IntMap } \alpha\ ) \to [\alpha] \\
&\texttt{inv}\ (v', \quad g'\ ) = \textbf{let}\ n\ = (\texttt{shapeInv}\ (\texttt{length}\ v')) - 1 \\
&\qquad\qquad\qquad\qquad\quad t\ = [0..n] \\
&\qquad\qquad\qquad\qquad\quad h\ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
&\qquad\qquad\qquad\qquad\quad h' = h +\!\!+ g' \\
&\qquad\qquad\qquad\quad \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{aligned}
$$

Aber wie `shapeInv` erhalten???

Eine Möglichkeit: durch Nutzer angeben lassen.

Eine andere Möglichkeit: statisch ermitteln (durch Typsystem?).

Nur zum Experimentieren:

$$
\begin{aligned}
&\texttt{shapeInv} :: \text{Int} \to \text{Int} \\
&\texttt{shapeInv}\ l = \texttt{head}\ [n+1 \mid n \leftarrow [0..],\ (\texttt{length}\ (\texttt{get}\ [0..n])) == l]
\end{aligned}
$$

Noch nicht ganz geschafft

Funktioniert ganz gut in einigen Fällen:

`get` = `tail` $\rightsquigarrow$ `put` "abcde" "xyz" = "axyz", unter Verwendung von `compl` "abcde" = [(0, 'a')]

Aber nicht so gut in anderen Fällen:

`get` = `init` $\rightsquigarrow$ `put` "abcde" "xyz" schlägt fehl, weil `compl` "abcde" = [(4, 'e')]

Das Problem: durch Vorhalten der Indizes ist `compl` noch nicht „nicht-injektiv" genug.

Beachte: selbst ohne diese Indizes wäre $\lambda s \rightarrow$ (`get` s, `compl` s) injektiv.

Elimination von Indizes

$$
\begin{aligned}
&\texttt{compl} :: [\alpha] \to [\ \ \alpha\ ] \\
&\texttt{compl}\ s = \textbf{let}\ n\ = (\texttt{length}\ s) - 1 \\
&\qquad\qquad\quad\ t\ = [0..n] \\
&\qquad\qquad\quad\ g\ = \texttt{zip}\ t\ s \\
&\qquad\qquad\quad\ g' = \texttt{filter}\ (\lambda(i, \_) \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ g \\
&\qquad\qquad\ \textbf{in}\ \ \texttt{map snd}\ g'
\end{aligned}
$$

$$
\begin{aligned}
&\texttt{inv} :: ([\alpha], [\ \ \alpha\ ]) \to [\alpha] \\
&\texttt{inv}\ (v', c\ ) = \textbf{let}\ n\ = (\texttt{shapeInv}\ (\texttt{length}\ v')) - 1 \\
&\qquad\qquad\qquad\ t\ = [0..n] \\
&\qquad\qquad\qquad\ h\ = \texttt{assoc}\ (\texttt{get}\ t)\ v' \\
&\qquad\qquad\qquad\ g' = \texttt{zip}\ (\texttt{filter}\ (\lambda i \to \texttt{notElem}\ i\ (\texttt{get}\ t))\ t)\ c \\
&\qquad\qquad\qquad\ h' = h +\!\!+ g' \\
&\qquad\qquad\quad\ \textbf{in}\ \ \texttt{seq}\ h\ (\texttt{map}\ (\lambda i \to \texttt{fromJust}\ (\texttt{lookup}\ i\ h'))\ t)
\end{aligned}
$$

Nun:

$$\texttt{get} = \texttt{init} \quad \rightsquigarrow \quad \texttt{put}\ \text{“abcde”}\ \text{“xyz”} = \text{“xyze”}$$

Weitere Beispiele

Sei $\texttt{get} = \texttt{sieve}$ mit:

$$\begin{aligned}\texttt{sieve} &:: [\alpha] \to [\alpha] \\ \texttt{sieve}\ (a : b : cs) &= b : (\texttt{sieve}\ cs) \\ \texttt{sieve}\ \_ &= []\end{aligned}$$

Dann:

$$\begin{aligned}\texttt{put}\ [1..8]\ [2,-4,6,8] &= [1,2,3,-4,5,6,7,8] \\ \texttt{put}\ [1..8]\ [2,-4,6] &= [1,2,3,-4,5,6] \\ \texttt{put}\ [1..8]\ [2,-4,6,8,10,12] &= [1,2,3,-4,5,6,7,8,\bot,10,\bot,12]\end{aligned}$$

Allerdings:

$$\texttt{put}\ [1..8]\ [0,2,-4,6,8] = [1,0,3,2,5,-4,7,6,\bot,8]$$

Wobei wir vielleicht lieber gehabt hätten:

$$\texttt{put}\ [1..8]\ [0,2,-4,6,8] = [\bot,0,1,2,3,-4,5,6,7,8]$$

Freie Theoreme und Anwendungen

Wie wir gesehen haben,

- schränken Typen das Verhalten von Programmen ein,
- führen so zu interessanten Erkenntnissen über Programme,
- gut kombinierbar mit algebraischen Techniken und gleichungsbasiertem Schließen.

Anwendungsgebiete umfassen:

- Programmoptimierungen
- spezifischere Domains

Aber:

- Wie ist das Ganze formal garantiert?
- Decken wir wirklich die ganze Sprache Haskell ab?
- Ist noch mehr drin?

Freie Theoreme [Wadler, 1989]

Ursprüngliches Beispiel:

$$\texttt{filter} :: (a \to \text{Bool}) \to [a] \to [a]$$

$$\texttt{takeWhile} :: (a \to \text{Bool}) \to [a] \to [a]$$

$$\texttt{g} :: (a \to \text{Bool}) \to [a] \to [a]$$

Für jede Wahl von p, h und l gilt:

$$\begin{aligned}
\texttt{filter}\ p\ (\texttt{map}\ h\ l) &= \texttt{map}\ h\ (\texttt{filter}\ (p \circ h)\ l) \\
\texttt{takeWhile}\ p\ (\texttt{map}\ h\ l) &= \texttt{map}\ h\ (\texttt{takeWhile}\ (p \circ h)\ l) \\
\texttt{g}\ p\ (\texttt{map}\ h\ l) &= \texttt{map}\ h\ (\texttt{g}\ (p \circ h)\ l)
\end{aligned}$$

Aber wie soll man so etwas beweisen können?

Versuch einer Argumentation

- `g` :: $(a \to \text{Bool}) \to [a] \to [a]$ muss für jede mögliche Instanziierung von a einheitlich arbeiten.
- Die Ausgabeliste kann nur Elemente der Eingabe l enthalten.
- Welche, und in welcher Reihenfolge/Vielfachheit, kann lediglich von l und dem Eingabeprädikat p abhängen.
- Die einzig möglichen Grundlagen zur Entscheidung sind die Länge von l und die Ergebnisse von p auf Elementen von l.
- Aber, die Listen (`map` h l) und l haben stets die selbe Länge.
- Und Anwendung von p auf ein Element von (`map` h l) hat stets das selbe Ergebnis wie Anwendung von $(p \circ h)$ auf das entsprechende Element von l.
- Also wählt `g` mit p stets „die selben“ Elemente aus (`map` h l) wie es `g` mit $(p \circ h)$ aus l tut, außer dass im ersten Fall die entsprechenden Abbilder unter h ausgegeben werden.
- Also ist (`g` p (`map` h l)) gleich (`map` h (`g` $(p \circ h)$ l)).
- Genau das wollten wir beweisen!

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!

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.

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

Deskriptive Programmierung

SS 2009

3.3 Theoretische Grundlagen der funktionalen Programmierung

Lambda-Kalkül

Der Lambda-Kalkül besteht aus zwei Bausteinen:

- Funktionsabstraktion

  λ x .A definiert eine (anonyme) Funktion, die ein x als Parameter bekommt und einen Ausdruck A als Funktionskörper hat (in dem wiederum x vorkommen kann aber nicht muss).

- Funktionsapplikation

  F A bedeutet, dass die Funktion F auf den Ausdruck A angewandt wird

Bemerkung: Die folgende Darstellung des Lambda-Klaküls orientiert sich an der ‚online'-Einführung von Fabian Nilius.

Beispiele für Funktionen (1)

- Identität:

  $\lambda\ x\ .\ x$

- Eine Funktion, die jedes Argument auf die Identitätsfunktion abbildet:

  $\lambda\ y\ .\ (\lambda\ x\ .\ x)$

- Die Identitätsfunktion angewendet auf sich selbst:

  $(\lambda\ x\ .\ x)\ (\lambda\ y\ .\ y) \quad \rightarrow \quad (\lambda\ y\ .\ y)$

Beispiele für Funktionen (3)

- Eine komplexere Funktion:

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

- Ein damit konstruierter Ausdruck:

$$\lambda\ f\ .\ (\lambda\ x\ .\ f\ (f\ x)))\ u\ v$$

$$\equiv\quad \lambda\ f\ .\ (\lambda\ x\ .\ f\ (f\ x)))\ u\ v$$

$$\rightarrow\quad (\lambda\ x\ .\ u\ (u\ x)))\ v$$

$$\rightarrow\quad u\ (u\ v)$$

Die Variable f wird auch in den inneren Ausdrücken ersetzt.

Diese Funktion wendet also eine Funktion zweimal auf ein Argument an!

Ableitungsregeln (1)

- Es gibt zwei grundlegende Ableitungsregeln in diesem Kalkül:
  - **α-Konversion** (gestattet den Austausch von Variablennamen)

$$(\lambda\, x\, .\, A) \quad \leftrightarrow \quad (\lambda\, y\, .\, A')\,,$$

wobei in A' alle Vorkommnisse von x durch y ersetzt wurden

  - **β-Konversion** (Applikation einer Funktion)

$$(\lambda\, x\, .\, A)\,(B) \quad \leftrightarrow \quad A'\,,$$

wobei in A' alle Vorkommnisse von x durch B ersetzt wurden

- Die β-Konversion wurde bereits bei der Anwendung der Identitätsfunktion auf sich selbst verwendet:

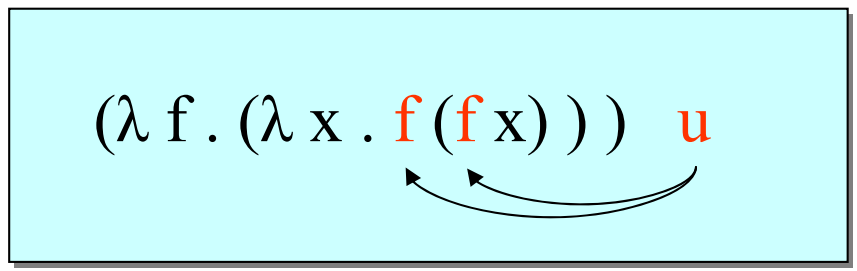
$$(\lambda\, x\, .\, x)\,(\lambda\, y\, .\, y) \quad \overset{\beta}{\leftrightarrow} \quad (\lambda\, y\, .\, y)$$

Ableitungsregeln (2)

- Wir haben bisher nicht definiert, wie bei einer Funktionsanwendung die Variablen ersetzt werden sollen.

- **Sichtbarkeit**: Schachteln wir F-Definitionen sind die weiter außen definierten Variablennamen innen sichtbar.

- Die β-Konversion drückt das dadurch aus, dass die Argumente für alle Vorkommen der Variable ersetzt werden:



- Allerdings sind nicht alle Variablen beim Ersetzungsprozess frei, da sie durch eine weitere F-Abstraktion gebunden sein können (hier z.B. das x).

Ableitungsregeln (3)

- Eine Variable x ist in einem Ausdruck frei bzw. gebunden:

x ist frei in x		(aber nicht in einer anderen Variable)
x ist frei in E F	$\Leftrightarrow$	x frei in E oder x frei in F
x ist frei in λ y. E	$\Leftrightarrow$	$x \neq y$ und x frei in E

- Beim Ersetzen von Variablen werden nur die freien Vorkommen herangezogen:

$$(\lambda f . f (\lambda f . f)) A$$

Richtig:

$\rightarrow$ A (λ f . f)

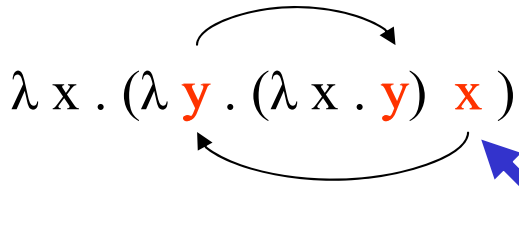
Falsch:

$\rightarrow$ A (λ f . A)

$\rightarrow$ A (λ A . A)

Ableitungsregeln (4)

- Wenn Variablennamen mehrfach und sowohl frei als auch gebunden auftreten, können weitere Namenskonflikte auftreten:



Setzen wir einfach dieses x für das y ein, so bezieht sich das x plötzlich auf die innerste Funktionsdefinition!

- Um diese Form von Namenskonflikten zu vermeiden, verwenden wir die α-Konversion zur Umbenennung (hier x zu x‘):

$$\lambda x . (\lambda y . (\lambda x' . y)\ x)$$

$$\rightarrow \quad \lambda x . (\lambda x' . x)$$

Vordefinierte Funktionen

- Der Kern des λ-Kalküls bietet keine Konstanten (wie 1, "foo", True) oder vordefinierte Funktionen (wie +, × , if).

- Im folgenden werden wir einen um Konstanten und vordef. Funktionen erweiterten Kalkül verwenden (legitimiert durch die vorgestellten Funktionssimulationen).

β-Konversion einiger vordefinierter Funktionen:

$$
\begin{array}{lcl}
(+\ x\ y) & \rightarrow & x \oplus y \\
(\times\ x\ y) & \rightarrow & x \otimes y \\
(\text{if True } e\ f) & \rightarrow & e \\
(\text{if False } e\ f) & \rightarrow & f \\
(\text{and False } e) & \rightarrow & \text{False} \\
(\text{and True } e) & \rightarrow & e
\end{array}
$$

(Operationen in ○ direkt auf der Zielmaschine ausführbar, wenn die Argumente x und y zuvor bis zur Normalform reduziert wurden.)

Beispiele für das erweiterte λ-Kalkül (1)

- Funktion, die ihr Argument inkrementiert:

$$\lambda\, x\, .\, (+\, x\, 1)$$

- Maximumsfunktion:

$$\lambda x\, .\, (\lambda y.\, (\text{if}\, (<\, x\, y)\, y\, x)\,)$$

- β-Konversion mit freien und gebundenen Variablen:

$$\begin{aligned} (\lambda x\, .\, (\lambda x.+\, (\times\, x\, 2))\;\; x\, 3)\;\; 9 \;\; &\xrightarrow[\beta]{} \;\; (\lambda\, x\, .+\, (\times\, x\, 2))\, 9\, 3 \\ &\xrightarrow[\beta]{} \;\; +\, (\times\, 9\, 2)\, 3 \\ &\rightarrow \;\; +\, (9 \otimes 2)\, 3 \rightarrow +\, 18\, 3 \rightarrow \ldots \end{aligned}$$

Einfach typisierter λ-Kalkül:

- bisherige Syntax bleibt erhalten
- nur noch typisierbare Terme:
  - Typen: $\tau := bool \mid nat \mid \ldots \mid \alpha \mid \beta \mid \ldots \mid \tau \rightarrow \tau$
  - Typisierungskontext: $\Gamma: V \rightarrow \tau$ (partielle Funktion über Variablenmenge V)
  - Typisierungsregeln:

$$\frac{\Gamma(x) \text{ ist definiert}}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma \vdash E_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash E_2 : \tau_1}{\Gamma \vdash (E_1\ E_2) : \tau_2}$$

$$\frac{\Gamma[x:\tau] \vdash E : \tau'}{\Gamma \vdash (\lambda\, x.E) : \tau \rightarrow \tau'}$$

  - Terme z.B. der Form (E E) sind nicht typisierbar!

Polymorpher Lambda-Kalkül [Girard 1972, Reynolds 1974]

Typen: $\tau := \alpha \mid \tau \to \tau \mid \forall\alpha.\tau$

Terme: $t := x \mid \lambda x : \tau.t \mid t\ t \mid \Lambda\alpha.t \mid t\ \tau$

$$\Gamma, x : \tau \vdash x : \tau \qquad [\![x]\!]_{\theta,\sigma} = \sigma(x)$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.t) : \tau_1 \to \tau_2} \qquad [\![\lambda x : \tau_1.t]\!]_{\theta,\sigma}\ a = [\![t]\!]_{\theta,\sigma[x \mapsto a]}$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau_2 \qquad \Gamma \vdash u : \tau_1}{\Gamma \vdash (t\ u) : \tau_2} \qquad [\![t\ u]\!]_{\theta,\sigma} = [\![t]\!]_{\theta,\sigma}\ [\![u]\!]_{\theta,\sigma}$$

$$\frac{\alpha, \Gamma \vdash t : \tau}{\Gamma \vdash (\Lambda\alpha.t) : \forall\alpha.\tau} \qquad [\![\Lambda\alpha.t]\!]_{\theta,\sigma}\ S = [\![t]\!]_{\theta[\alpha \mapsto S],\sigma}$$

$$\frac{\Gamma \vdash t : \forall\alpha.\tau}{\Gamma \vdash (t\ \tau') : \tau[\tau'/\alpha]} \qquad [\![t\ \tau']\!]_{\theta,\sigma} = [\![t]\!]_{\theta,\sigma}\ [\![\tau']\!]_{\theta}$$

Das Parametrizitäts-Theorem [Reynolds 1983, Wadler 1989]

Gegeben τ und Environments θ_1, θ_2, ρ mit $\rho(\alpha) \subseteq \theta_1(\alpha) \times \theta_2(\alpha)$, definiere $\Delta_{\tau,\rho} \subseteq [\![\tau]\!]_{\theta_1} \times [\![\tau]\!]_{\theta_2}$ wie folgt:

$$
\begin{array}{lcl}
\Delta_{\alpha,\rho} & = & \rho(\alpha) \\
\Delta_{\tau_1 \to \tau_2,\rho} & = & \{(f_1, f_2) \mid \forall (a_1, a_2) \in \Delta_{\tau_1,\rho}.\ (f_1\ a_1, f_2\ a_2) \in \Delta_{\tau_2,\rho}\} \\
\Delta_{\forall\alpha.\tau,\rho} & = & \{(g_1, g_2) \mid \forall \mathcal{R} \subseteq S_1 \times S_2.\ (g_1\ S_1, g_2\ S_2) \in \Delta_{\tau,\rho[\alpha \mapsto \mathcal{R}]}\}
\end{array}
$$

Dann gilt für jeden geschlossenen Term t geschlossenen Typs τ:

$$([\![t]\!]_{\emptyset,\emptyset}, [\![t]\!]_{\emptyset,\emptyset}) \in \Delta_{\tau,\emptyset}.$$