Wie? So!

Es seien

$\oplus_1$	Zero	One	Two
Zero	Zero	One	Two
One	One	Two	Two
Two	Two	Two	Two

und

$\oplus_2$	Zero	One	Two
Zero	Zero	One	Two
One	One	One	Two
Two	Two	One	Two

Ist `candidate` $(\oplus_1)$ korrekt auf jeder Liste der Form

$$[(\text{Zero},)^* \ \text{One} \ (, \text{Zero})^* \ (, \text{Two})^*]$$

und `candidate` $(\oplus_2)$ ist korrekt auf jeder Liste der Form

$$[(\text{Zero},)^* \ \text{One}, \text{Two} \ (, \text{Zero})^*]$$

dann ist `candidate` korrekt für assoziatives $\oplus$ auf beliebigem Typ.

Ein Knuth-ähnliches 0-1-2-Prinzip

Gegeben: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned}\texttt{scanl1}\ (\oplus)\ (x : xs) &= go\ x\ xs \\ \textbf{where}\ go\ x\ [\,] &= [x] \\ go\ x\ (y : ys) &= x : go\ (x \oplus y)\ ys\end{aligned}$$

$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

data Three = Zero | One | Two

Theorem: Wenn für jedes *xs* :: [Three] und assoziatives $(\oplus) :: \text{Three} \to \text{Three} \to \text{Three}$,

$$\texttt{candidate}\ (\oplus)\ xs \quad = \quad \texttt{scanl1}\ (\oplus)\ xs\,,$$

dann gilt das Gleiche für jeden Typ τ, *xs* :: $[\tau]$, und assoziatives $(\oplus) :: \tau \to \tau \to \tau$.

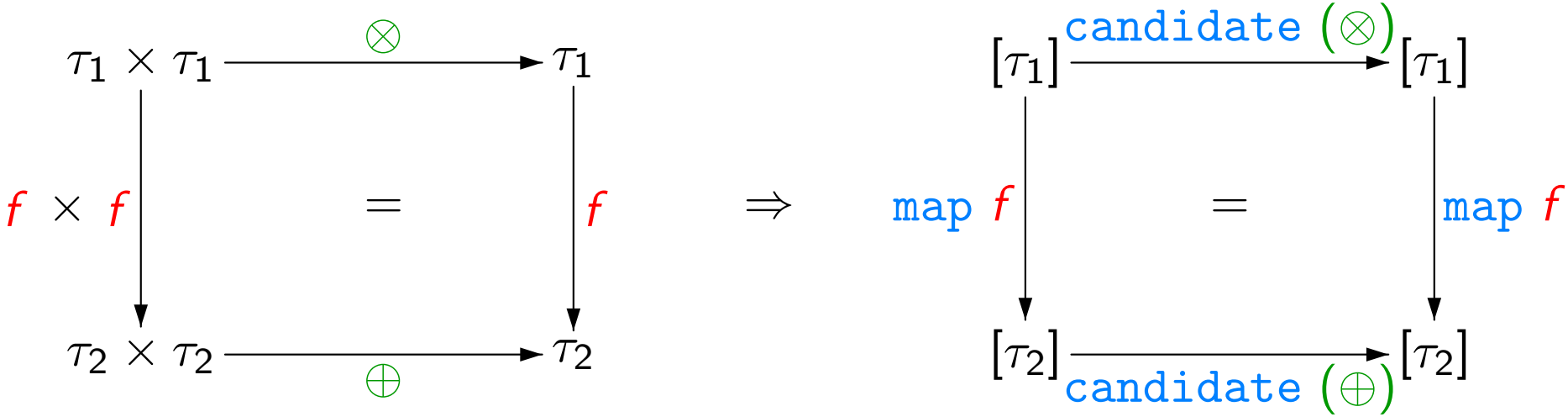
Verwendung des Theorem-Generators

Eingabe:
```
candidate :: (a -> a -> a) -> [a] -> [a]
```

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

Anders ausgedrückt

Für jede Wahl von Typen τ_1, τ_2 und Funktionen $f :: \tau_1 \to \tau_2$, $(\otimes) :: \tau_1 \to \tau_1 \to \tau_1$ und $(\oplus) :: \tau_2 \to \tau_2 \to \tau_2$:



Ein Knuth-ähnliches 0-1-2-Prinzip

Gegeben: $\texttt{scanl1} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

$$\begin{aligned}\texttt{scanl1}\ (\oplus)\ (x : xs) &= go\ x\ xs \\ \textbf{where}\ go\ x\ [\,] &= [x] \\ go\ x\ (y : ys) &= x : go\ (x \oplus y)\ ys\end{aligned}$$

$\texttt{candidate} :: (\alpha \to \alpha \to \alpha) \to [\alpha] \to [\alpha]$

data Three = Zero | One | Two

Theorem: Wenn für jedes *xs* :: [Three] und assoziatives $(\oplus) :: \text{Three} \to \text{Three} \to \text{Three}$,

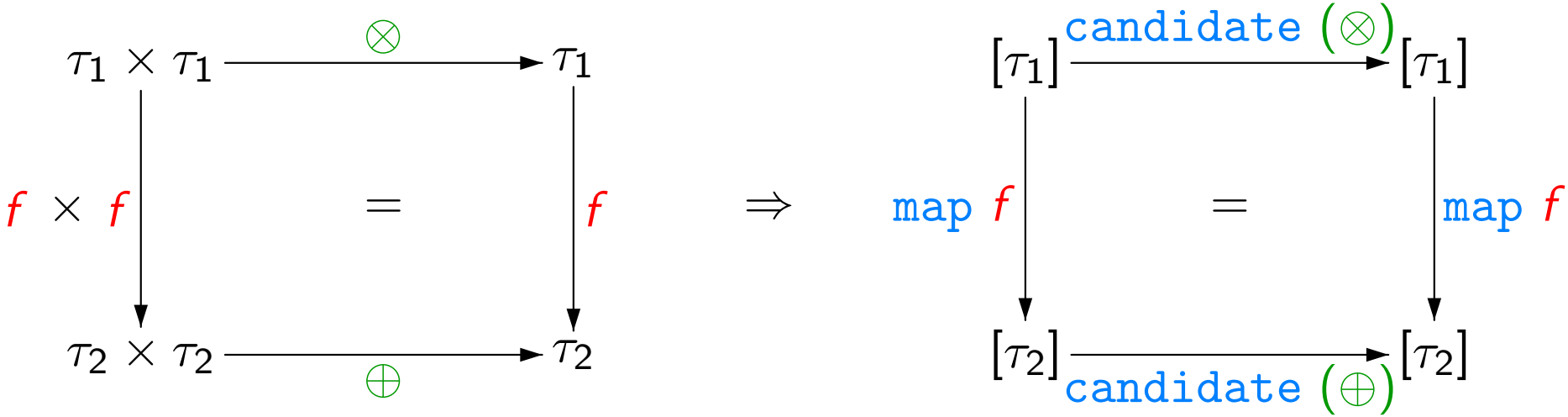
$$\texttt{candidate}\ (\oplus)\ xs \quad = \quad \texttt{scanl1}\ (\oplus)\ xs\,,$$

dann gilt das Gleiche für jeden Typ τ, $xs :: [\tau]$, und assoziatives $(\oplus) :: \tau \to \tau \to \tau$.

Dekomponieren des 0-1-2-Prinzips

Aussage 1: Wenn `candidate` $(\oplus_1)$ korrekt ist für jede Liste der Form $[(\text{Zero},)^* \ \text{One} \ (, \text{Zero})^* \ (, \text{Two})^*]$ und `candidate` $(\oplus_2)$ korrekt ist für jede Liste der Form $[(\text{Zero},)^* \ \text{One}, \text{Two} \ (, \text{Zero})^*]$, dann gilt für jedes $n \geq 0$,
`candidate` $(+\!+)$ $[[k] \mid k \leftarrow [0..n]] \ = \ [[0..k] \mid k \leftarrow [0..n]]$ (*).

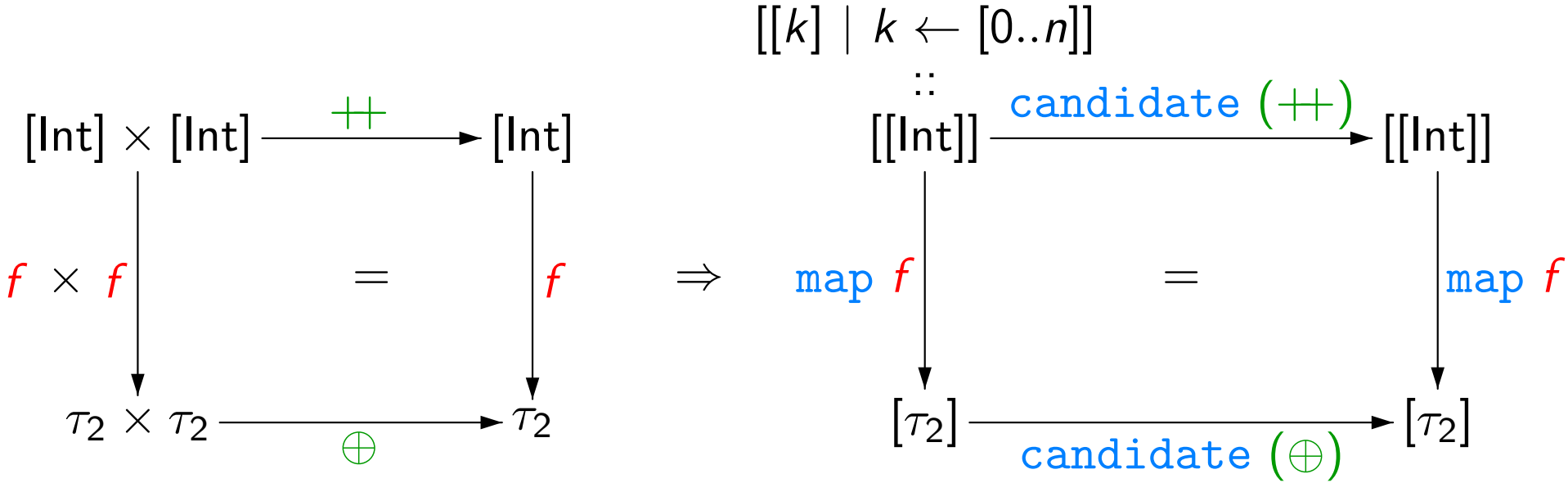
Aussage 2: Wenn (*) für jedes $n \geq 0$ gilt, dann ist `candidate` korrekt für assoziatives $\oplus$ auf beliebigem Typ.



Dekomponieren des 0-1-2-Prinzips

Aussage 1: Wenn `candidate` $(\oplus_1)$ korrekt ist für jede Liste der Form $[(\text{Zero},)^* \ \text{One} \ (, \text{Zero})^* \ (, \text{Two})^*]$ und `candidate` $(\oplus_2)$ korrekt ist für jede Liste der Form $[(\text{Zero},)^* \ \text{One}, \text{Two} \ (, \text{Zero})^*]$, dann gilt für jedes $n \geq 0$,
`candidate` $(+\!\!+)$ $[[k] \mid k \leftarrow [0..n]] \ = \ [[0..k] \mid k \leftarrow [0..n]]$ (*).

Aussage 2: Wenn (*) für jedes $n \geq 0$ gilt, dann ist `candidate` korrekt für assoziatives $\oplus$ auf beliebigem Typ.

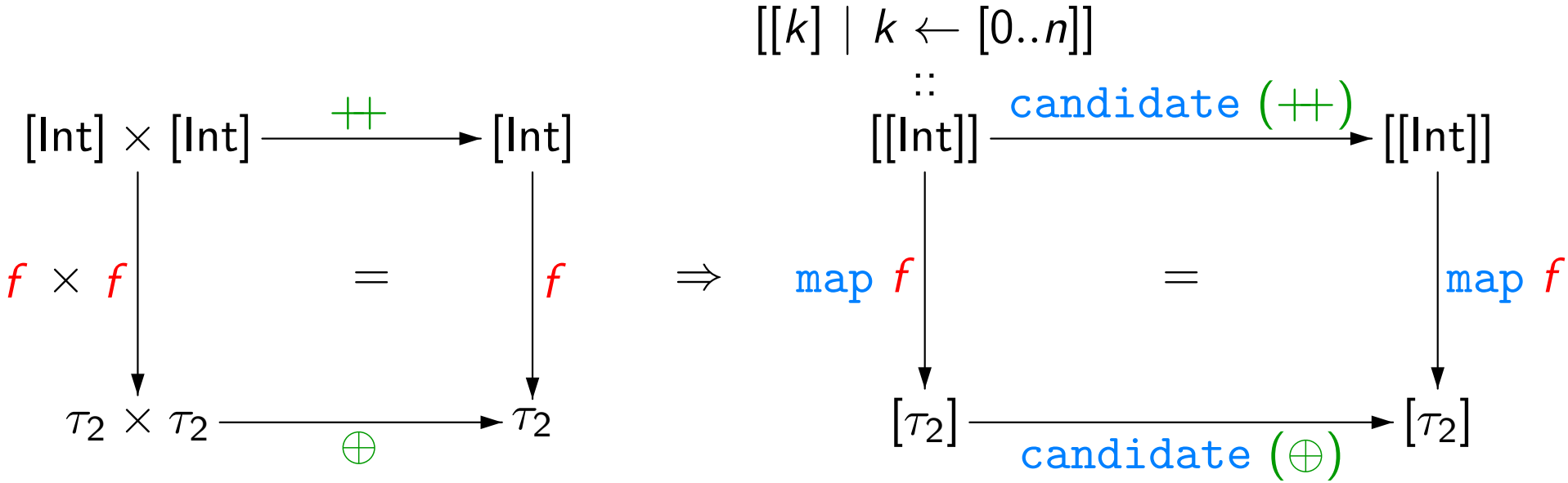


Formaler Beweis

Sei $xs :: [\tau_2]$ mit Länge $(n+1)$. Dann ist für

$$f = \texttt{foldl1}\ (\oplus) \circ \texttt{map}\ (xs\,!!)$$

die Vorbedingung von



erfüllt, sofern $\oplus$ assoziativ.

Also dann:

$$\begin{aligned} & \texttt{map}\ f\ (\texttt{candidate}\ (+\!\!+)\ [[k] \mid k \leftarrow [0..n]]) \\ = \quad & \texttt{candidate}\ (\oplus)\ (\texttt{map}\ f\ [[k] \mid k \leftarrow [0..n]]) \\ = \quad & \texttt{candidate}\ (\oplus)\ xs \end{aligned}$$

Formaler Beweis

Aussage 2: Wenn für jedes $n \geq 0$,

$$\texttt{candidate}\ (+\!\!+)\ [[k] \mid k \leftarrow [0..n]] \;=\; [[0..k] \mid k \leftarrow [0..n]]\,,$$

dann ist `candidate` korrekt für assoziatives $\oplus$ auf beliebigem Typ.

Beweis:

$$\begin{aligned}
& \texttt{candidate}\ (\oplus)\ xs \\
= \;& \texttt{map}\ f\ (\texttt{candidate}\ (+\!\!+)\ [[k] \mid k \leftarrow [0..n]]) \\
= \;& \texttt{map}\ f\ [[0..k] \mid k \leftarrow [0..n]] \\
= \;& \texttt{map}\ (\texttt{foldl1}\ (\oplus) \circ \texttt{map}\ (xs\,!!))\ [[0..k] \mid k \leftarrow [0..n]] \\
= \;& [\texttt{foldl1}\ (\oplus)\ (\texttt{map}\ (xs\,!!)\ [0..k]) \mid k \leftarrow [0..n]] \\
= \;& [\texttt{foldl1}\ (\oplus)\ (\texttt{take}\ (k+1)\ xs) \mid k \leftarrow [0..n]] \\
= \;& \texttt{scanl1}\ (\oplus)\ xs
\end{aligned}$$

Formaler Beweis

Aussage 1: Wenn `candidate` $(\oplus_1)$ korrekt ist für jede Liste der Form ... und `candidate` $(\oplus_2)$ korrekt ist für jede Liste der Form ..., dann gilt für jedes $n \geq 0$,

$$\texttt{candidate}\ (+\!\!+)\ [[k] \mid k \leftarrow [0..n]] \ = \ [[0..k] \mid k \leftarrow [0..n]].$$

Beweis:

Wir wissen, für alle $p, q, r \geq 0$:

$$\begin{aligned} & \texttt{map}\ (\texttt{foldl1}\ (\oplus_1) \circ \texttt{map}\ ([(\text{Zero},)^p\ \text{One}\ (,\text{Zero})^q\ (,\text{Two})^r]\ !!)) \\ & \ (\texttt{candidate}\ (+\!\!+)\ [[k] \mid k \leftarrow [0..(p+q+r)]]) \\ = \ & [(\text{Zero},)^p\ (\text{One},)^{q+1}\ (\text{Two},)^r] \end{aligned}$$

und:

$$\begin{aligned} & \texttt{map}\ (\texttt{foldl1}\ (\oplus_2) \circ \texttt{map}\ ([(\text{Zero},)^p\ \text{One}, \text{Two}\ (,\text{Zero})^q]\ !!)) \\ & \ (\texttt{candidate}\ (+\!\!+)\ [[k] \mid k \leftarrow [0..(p+q+1)]]) \\ = \ & [(\text{Zero},)^p\ \text{One}\ (,\text{Two})^{q+1}] \end{aligned}$$

...

Formaler Beweis

Beweis:

Wir wissen, für alle $p, q, r \geq 0$:

$$\begin{aligned} & \texttt{map}\ (\texttt{foldl1}\ (\oplus_1) \circ \texttt{map}\ ([(\text{Zero},)^p\ \text{One}\ (, \text{Zero})^q\ (, \text{Two})^r]\ !!)) \\ & \ (\texttt{candidate}\ (+\!\!+)\ [[k] \mid k \leftarrow [0..(p+q+r)]]) \\ = \quad & [(\text{Zero},)^p\ (\text{One},)^{q+1}\ (\text{Two},)^r] \end{aligned}$$

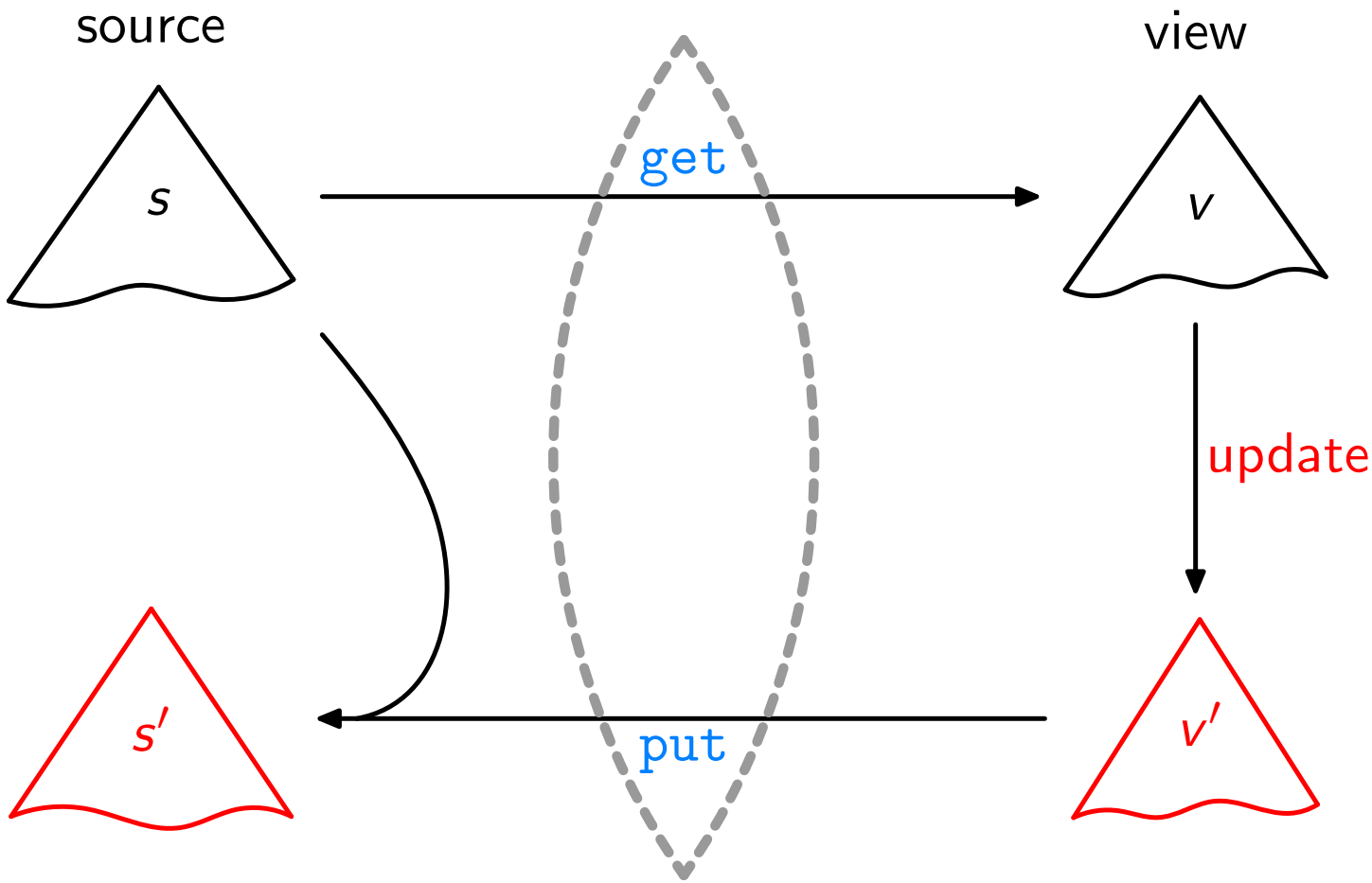
und:

$$\begin{aligned} & \texttt{map}\ (\texttt{foldl1}\ (\oplus_2) \circ \texttt{map}\ ([(\text{Zero},)^p\ \text{One}, \text{Two}\ (, \text{Zero})^q]\ !!)) \\ & \ (\texttt{candidate}\ (+\!\!+)\ [[k] \mid k \leftarrow [0..(p+q+1)]]) \\ = \quad & [(\text{Zero},)^p\ \text{One}\ (, \text{Two})^{q+1}] \end{aligned}$$

...

Detailliert in [V. 2008], sowie in einem Theorembeweiser formalisiert [Böhme, `http://afp.sf.net/entries/MuchAdoAboutTwo.shtml`].

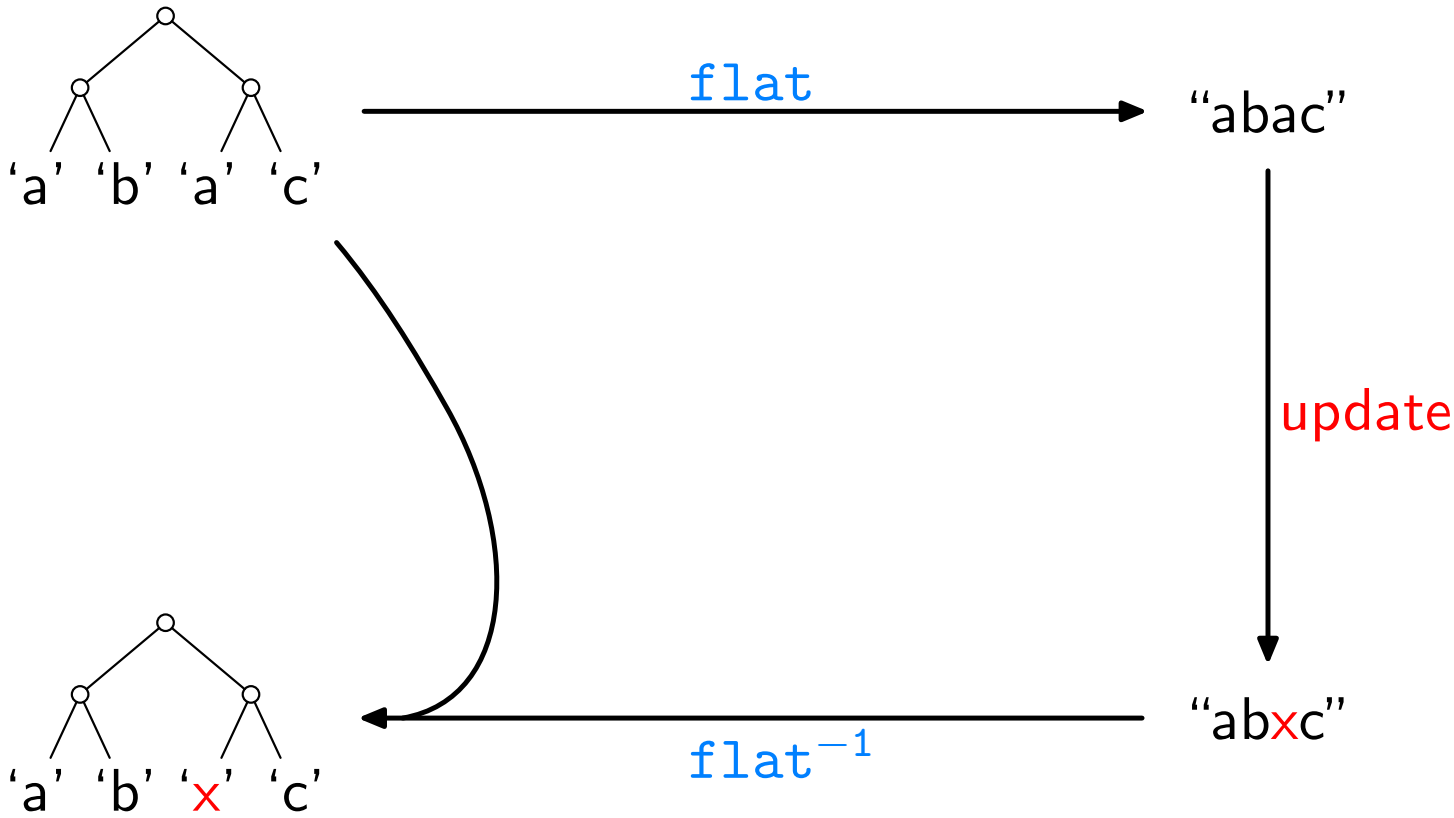
Eine weitere Anwendung: Bidirektionalisierung



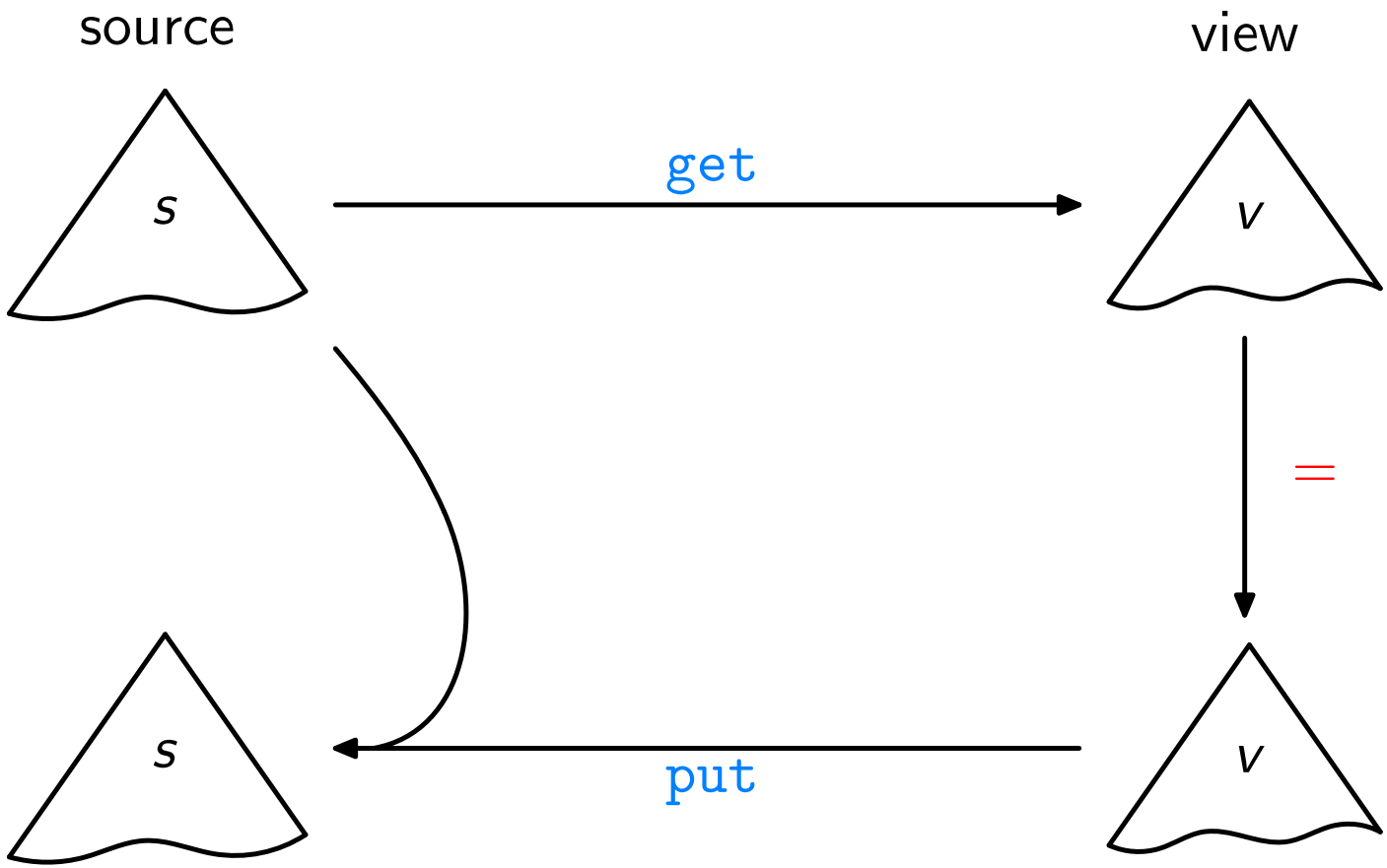
Eine weitere Anwendung: Bidirektionalisierung

Beispiel: `get` = `flat`

View-Update:

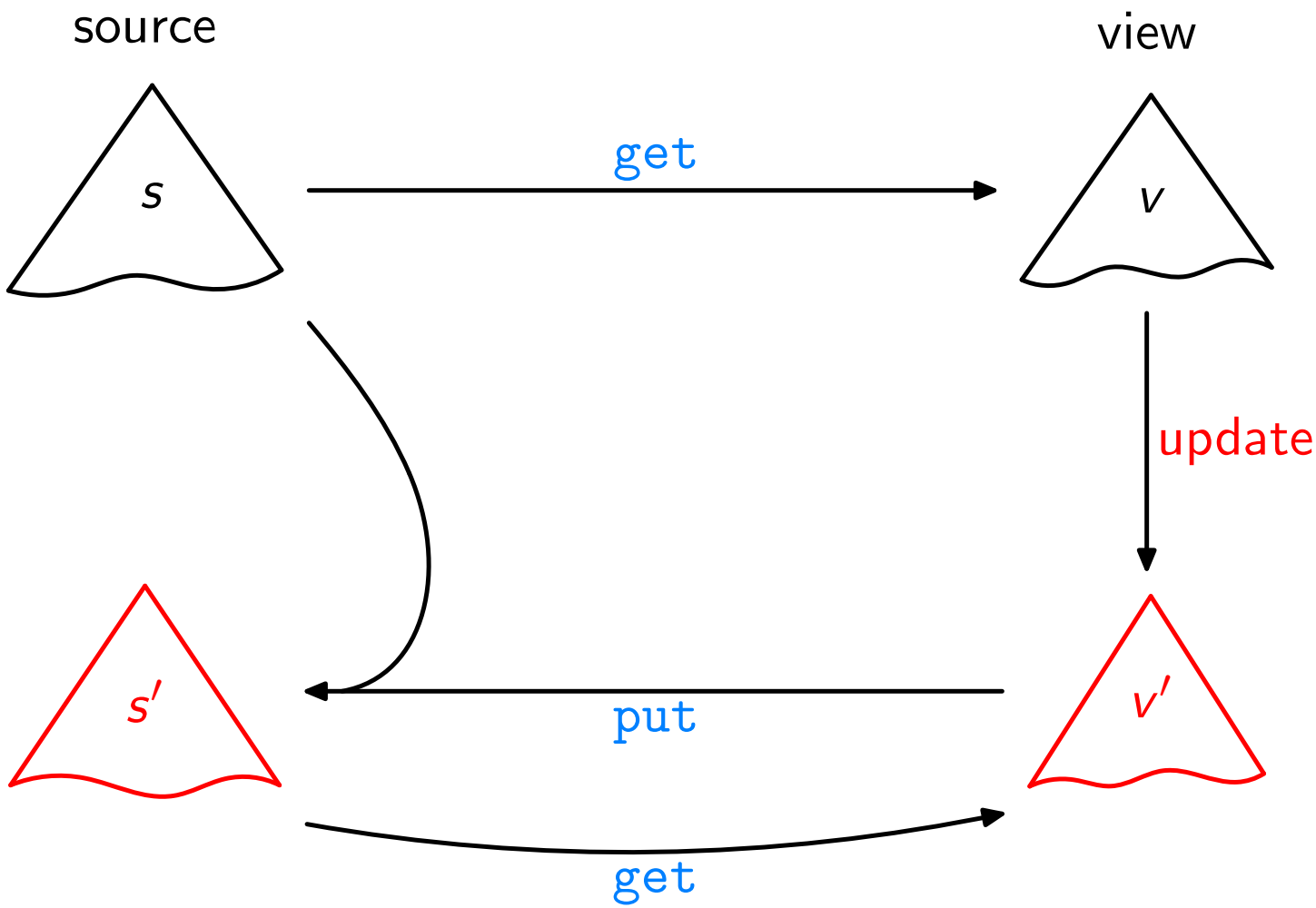


Eine weitere Anwendung: Bidirektionalisierung



Acceptability / GetPut

Eine weitere Anwendung: Bidirektionalisierung



Consistency / PutGet

Eine weitere Anwendung: Bidirektionalisierung

Beispiel 1:

```
halve :: [α] → [α]
halve as = take (length as `div` 2) as

halve⁻¹ :: [α] → [α] → [α]
halve⁻¹ as as' | length as' == n
                 = as' ++ drop n as
                   where n = length as `div` 2
```

Eine weitere Anwendung: Bidirektionalisierung

Beispiel 2:

$$
\begin{aligned}
&\texttt{flat} :: \text{Tree } \alpha \to [\alpha] \\
&\texttt{flat } (\text{Leaf } a) \quad\quad = [a] \\
&\texttt{flat } (\text{Node } t_1\ t_2) = (\texttt{flat } t_1) +\!\!+ (\texttt{flat } t_2)
\end{aligned}
$$

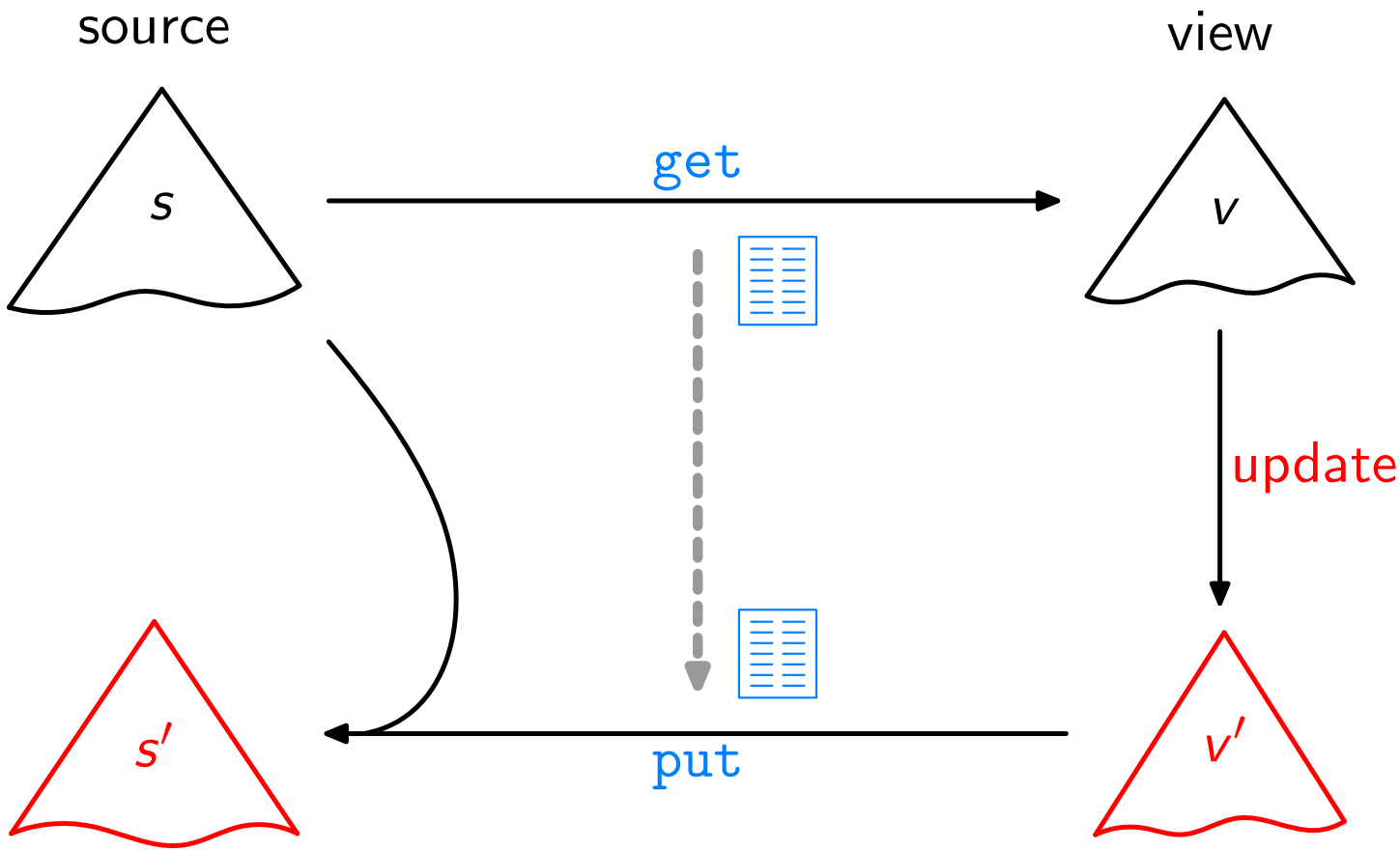
$$
\begin{aligned}
&\texttt{flat}^{-1} :: \text{Tree } \alpha \to [\alpha] \to \text{Tree } \alpha \\
&\texttt{flat}^{-1}\ s\ v = \textbf{case } go\ s\ v \textbf{ of } (t, []) \to t \\
&\quad \textbf{where } go\ (\text{Leaf } a) \quad\quad (b : bs) = (\text{Leaf } b, bs) \\
&\qquad\qquad go\ (\text{Node } s_1\ s_2)\ bs \quad\quad = (\text{Node } t_1\ t_2, ds) \\
&\qquad\qquad\quad \textbf{where } (t_1, cs) = go\ s_1\ bs \\
&\qquad\qquad\qquad\quad\ (t_2, ds) = go\ s_2\ cs
\end{aligned}
$$

Eine weitere Anwendung: Bidirektionalisierung

Beispiel 3:

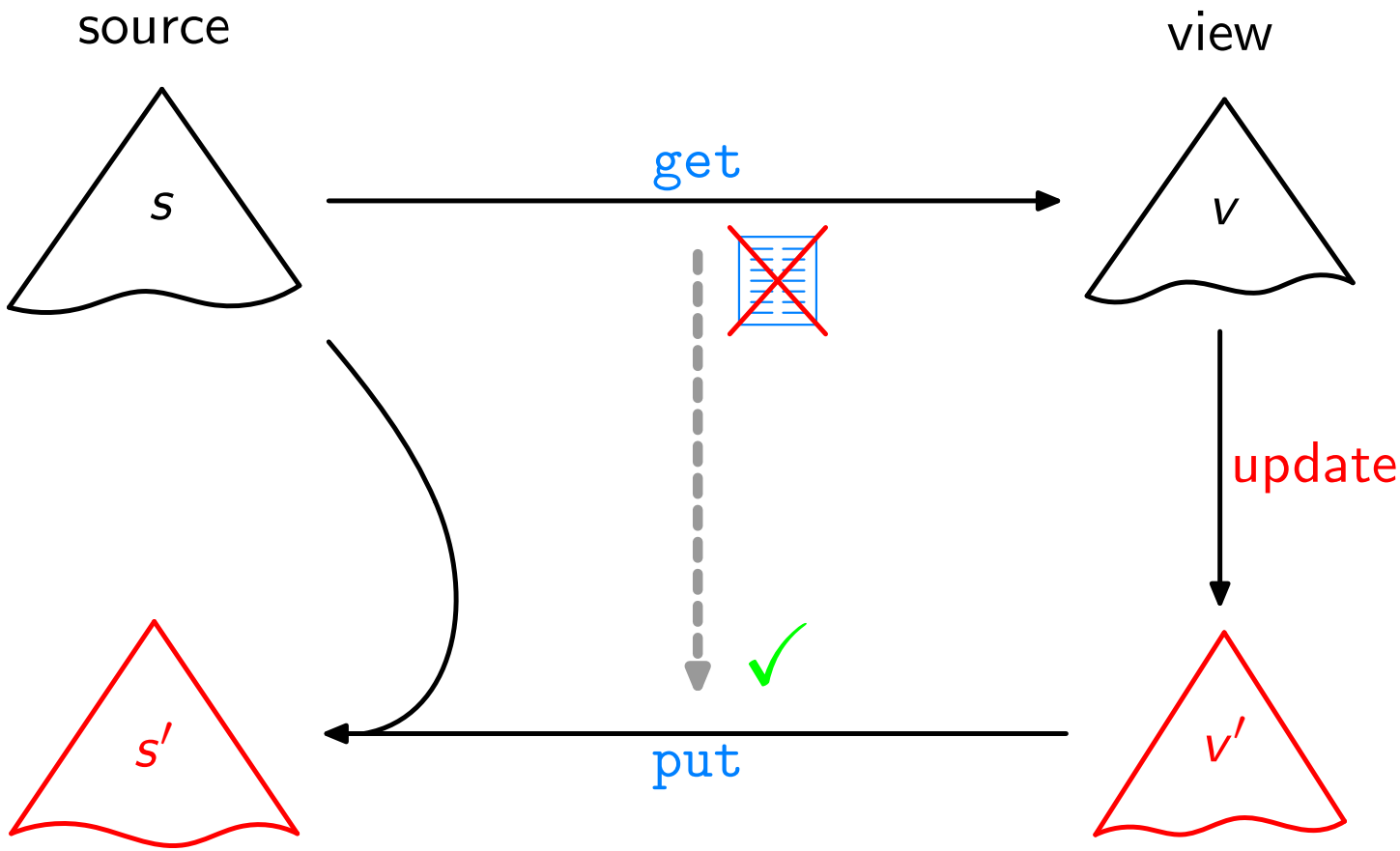
```
rmdups :: Eq α ⇒ [α] → [α]
rmdups = List.nub

rmdups⁻¹ :: Eq α ⇒ [α] → [α] → [α]
rmdups⁻¹ s v | v == List.nub v && length v == length s'
             = map (fromJust ∘ flip lookup (zip s' v)) s
               where s' = List.nub s
```

Eine weitere Anwendung: Bidirektionalisierung



Syntaktische Bidirektionalisierung

Eine weitere Anwendung: Bidirektionalisierung

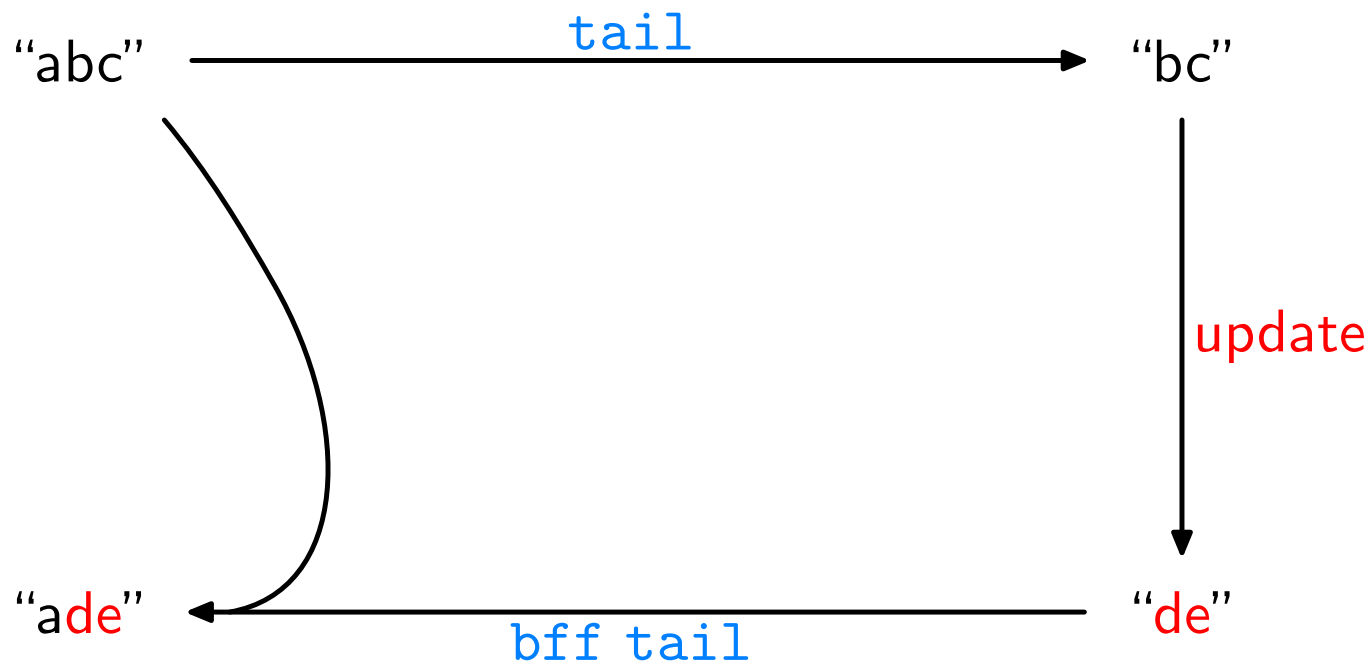


Semantische Bidirektionalisierung

Semantische Bidirektionalisierung

Ziel: Schreibe eine Higher-Order Funktion `bff`† so dass jeweils für `get` und `bff get` gilt: GetPut, PutGet, ...

Beispiele:

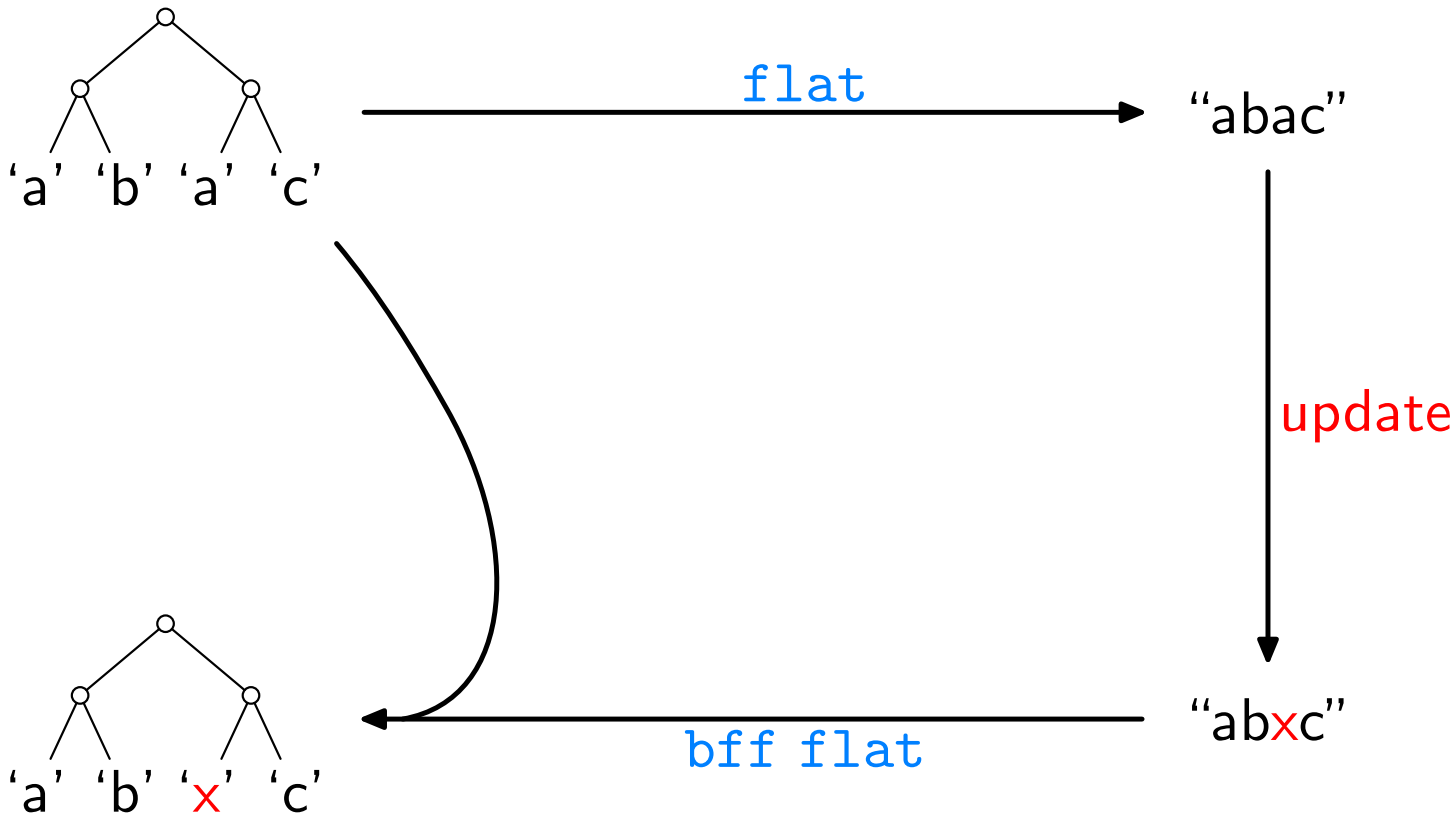


† “Bidirectionalization for free!”

Semantische Bidirektionalisierung

Ziel: Schreibe eine Higher-Order Funktion `bff`† so dass jeweils für `get` und `bff get` gilt: GetPut, PutGet, ...

Beispiele:

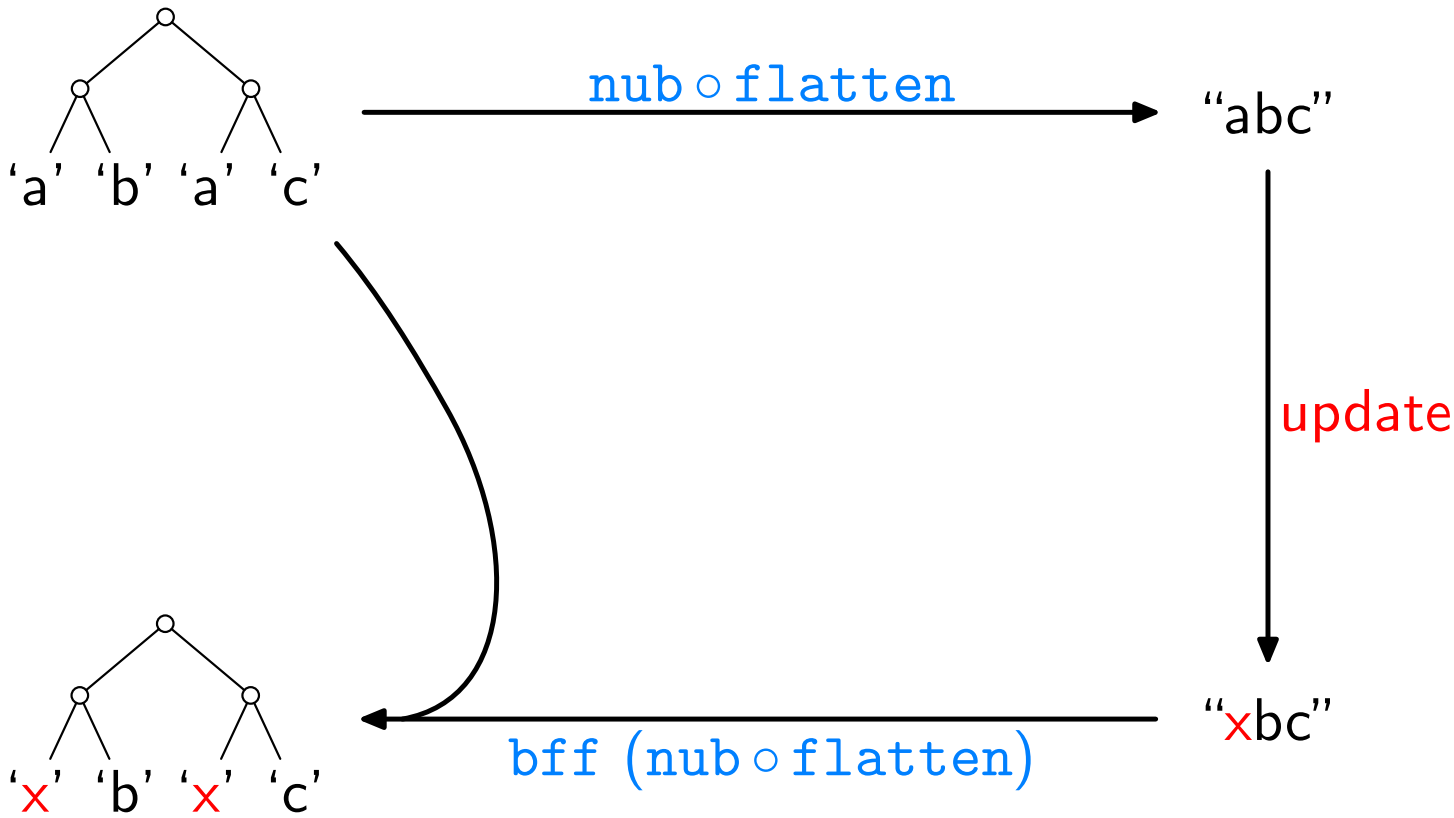


† “Bidirectionalization for free!”

Semantische Bidirektionalisierung

Ziel: Schreibe eine Higher-Order Funktion `bff`† so dass jeweils für `get` und `bff get` gilt: GetPut, PutGet, ...

Beispiele:



† "Bidirectionalization for free!"

Analyse spezifischer Aufrufe

Angenommen, es sei gegeben:

$$\texttt{get} :: [\alpha] \to [\alpha]$$

Wie können wir, oder `bff`, die Funktion analysieren ohne Zugriff auf ihren Quellcode?

Idee: Wie wäre es damit, `get` für „irgendeine“ Eingabe aufzurufen?

Etwa:

$$\texttt{get}\ [0..n] = \begin{cases} [1..n] & \text{if } \texttt{get} = \texttt{tail} \\ [n..0] & \text{if } \texttt{get} = \texttt{reverse} \\ [0..(\texttt{min}\ 4\ n)] & \text{if } \texttt{get} = \texttt{take}\ 5 \\ \vdots & \end{cases}$$

Dann, Übertragung der Erkenntnisse auf andere Listen als $[0..n]$!

Verwendung eines freien Theorems

Für jedes

$$\texttt{g} :: [\alpha] \to [\alpha]$$

gilt:

$$\texttt{map}\ f\ (\texttt{g}\ l) \quad = \quad \texttt{g}\ (\texttt{map}\ f\ l)$$

für beliebige f und l.

Für eine beliebige Liste s der Länge $n+1$, setze $\texttt{g} = \texttt{get}$, $l = [0..n]$, $f = (s\,!!)$, woraus sich ergibt:

$$\begin{aligned} \texttt{map}\ (s\,!!)\ (\texttt{get}\ [0..n]) &= \texttt{get}\ (\underbrace{\texttt{map}\ (s\,!!)\ [0..n]}_{s}) \\ &= \texttt{get}\ s \end{aligned}$$

Verwendung eines freien Theorems

Für jedes

$$\texttt{g} :: [\alpha] \to [\alpha]$$

gilt:

$$\texttt{map}\ f\ (\texttt{g}\ l) \quad = \quad \texttt{g}\ (\texttt{map}\ f\ l)$$

für beliebige f und l.

Für eine beliebige Liste s der Länge $n+1$,

$$\texttt{get}\ s \quad = \quad \texttt{map}\ (s\,!!)\ (\texttt{get}\ [0..n])$$

für jedes $\texttt{get} :: [\alpha] \to [\alpha]$.

Der „Constant-Complement“ Ansatz [Bancilhon & Spyratos 1981]

Allgemein, für

$$\texttt{get} :: S \to V$$

definiere ein V^C und

$$\texttt{compl} :: S \to V^C$$

so dass

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

injektiv ist und ein Inverses hat:

$$\texttt{inv} :: (V, V^C) \to S$$

Dann:

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

Wichtig: `compl` sollte so „nicht-injektiv“ wie möglich sein.

Der „Constant-Complement“ Ansatz

Für unseren Fall,

$$\texttt{get} :: [\alpha] \to [\alpha]\,,$$

was sollten wir wählen für V^C und

$$\texttt{compl} :: [\alpha] \to V^C \quad ???$$

Um

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

injektiv zu kriegen, müssen Informationen festgehalten werden, welche `get` verwirft.

Kandidaten:

1. Länge der ursprünglichen Liste
2. verworfene Listenelemente

Für den Augenblick, seien wir möglichst konservativ.

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.