

Wie? So!

Es seien

\oplus_1	Zero	One	Two		\oplus_2	Zero	One	Two
Zero	Zero	One	Two	und	Zero	Zero	One	Two
One	One	Two	Two		One	One	One	Two
Two	Two	Two	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: `scanl1` :: $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$
`scanl1` (\oplus) (x : xs) = go x xs
 where go x [] = [x]
 go x (y : ys) = x : go (x \oplus y) ys

`candidate` :: $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$

`data` Three = Zero | One | Two

Theorem: Wenn für jedes `xs` :: [Three] und assoziatives
 (\oplus) :: Three \rightarrow Three \rightarrow Three,

`candidate` (\oplus) xs = `scanl1` (\oplus) xs,

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

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))`

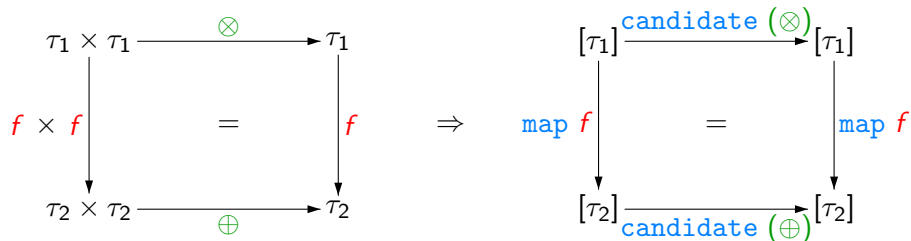
`==> (forall z :: [t1].`

`map f (candidate p z)`

`= candidate q (map f z))`

Anders ausgedrückt

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



Ein Knuth-ähnliches 0-1-2-Prinzip

Gegeben: `scanl1` :: $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$
`scanl1` (\oplus) (x : xs) = go x xs
 where go x [] = [x]
 go x (y : ys) = x : go (x \oplus y) ys

`candidate` :: $(\alpha \rightarrow \alpha \rightarrow \alpha) \rightarrow [\alpha] \rightarrow [\alpha]$

`data` Three = Zero | One | Two

Theorem: Wenn für jedes `xs` :: [Three] und assoziatives
 (\oplus) :: Three \rightarrow Three \rightarrow Three,

`candidate` (\oplus) xs = `scanl1` (\oplus) xs,

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

Dekomponieren des 0-1-2-Prinzips

Aussage 1: Wenn `candidate (\oplus_1)` korrekt ist für jede Liste der Form `[(Zero,)* One (, Zero)* (, Two)*]` und `candidate (\oplus_2)` korrekt ist für jede Liste der Form `[(Zero,)* One, Two (, Zero)*]`, dann gilt für jedes $n \geq 0$,

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

Dekomponieren des 0-1-2-Prinzips

Aussage 1: Wenn `candidate` (\oplus_1) korrekt ist für jede Liste der Form $[(Zero,)^* One (, Zero)^* (, Two)^*]$ und `candidate` (\oplus_2) korrekt ist für jede Liste der Form $[(Zero,)^* One, Two (, Zero)^*]$, dann gilt für jedes $n \geq 0$,

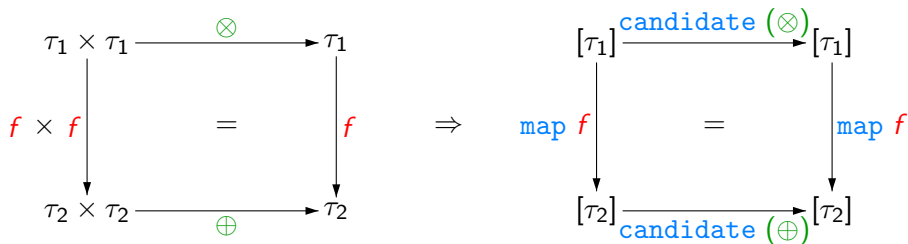
$$\text{candidate } (++) \ [[k \mid k \leftarrow [0..n]] \ = \ [[0..k \mid k \leftarrow [0..n]] \ (*).$$

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 $[(Zero,)^* One (, Zero)^* (, Two)^*]$ und `candidate (\oplus_2)` korrekt ist für jede Liste der Form $[(Zero,)^* One, Two (, Zero)^*]$, dann gilt für jedes $n \geq 0$,
`candidate (++) [[k] | k \leftarrow [0..n]] = [[0..k] | k \leftarrow [0..n]] (*)`.

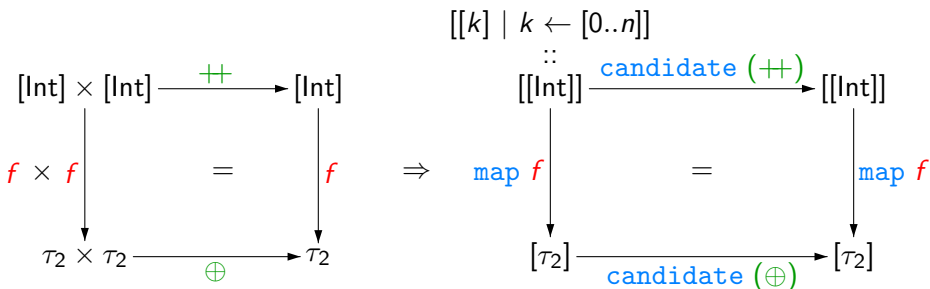
Aussage 2: Wenn (*) für jedes $n \geq 0$ gilt, dann ist `candidate` korrekt für assoziatives \oplus auf beliebigem Typ.



Dekomponieren des 0-1-2-Prinzips

Aussage 1: Wenn `candidate (\oplus_1)` korrekt ist für jede Liste der Form $[(Zero,)^* One (, Zero)^* (, Two)^*]$ und `candidate (\oplus_2)` korrekt ist für jede Liste der Form $[(Zero,)^* One, Two (, Zero)^*]$, 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 $(n + 1)$.

Formaler Beweis

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

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

die Vorbedingung von

$$\begin{array}{ccc} [\text{Int}] \times [\text{Int}] & \xrightarrow{++} & [\text{Int}] \\ \downarrow f \times f & = & \downarrow f \\ \tau_2 \times \tau_2 & \xrightarrow{\oplus} & \tau_2 \end{array}$$

$[[k \mid k \leftarrow [0..n]]$

$$\begin{array}{ccc} \begin{array}{c} \vdots \\ [[\text{Int}]] \end{array} & \xrightarrow{\text{candidate } (++)} & [[\text{Int}]] \\ \downarrow \text{map } f & = & \downarrow \text{map } f \\ [\tau_2] & \xrightarrow{\text{candidate } (\oplus)} & [\tau_2] \end{array}$$

Formaler Beweis

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

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

die Vorbedingung von

$$\begin{array}{ccc} [\text{Int}] \times [\text{Int}] & \xrightarrow{++} & [\text{Int}] \\ \downarrow f \times f & & \downarrow f \\ \tau_2 \times \tau_2 & \xrightarrow{\oplus} & \tau_2 \end{array} =$$

\Rightarrow

$$\begin{array}{ccc} [[k \mid k \leftarrow [0..n]] & & \\ \vdots & & \\ [[\text{Int}]] & \xrightarrow{\text{candidate } (++)} & [[\text{Int}]] \\ \downarrow \text{map } f & & \downarrow \text{map } f \\ [\tau_2] & \xrightarrow{\text{candidate } (\oplus)} & [\tau_2] \end{array} =$$

erfüllt, sofern \oplus assoziativ.

Formaler Beweis

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

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

die Vorbedingung von

$$\begin{array}{ccc} [\text{Int}] \times [\text{Int}] & \xrightarrow{++} & [\text{Int}] \\ \downarrow f \times f & & \downarrow f \\ \tau_2 \times \tau_2 & \xrightarrow{\oplus} & \tau_2 \end{array} \quad \Rightarrow \quad \begin{array}{ccc} [[k \mid k \leftarrow [0..n]] & & \\ \vdots & & \\ [[\text{Int}]] & \xrightarrow{\text{candidate } (++)} & [[\text{Int}]] \\ \downarrow \text{map } f & & \downarrow \text{map } f \\ [\tau_2] & \xrightarrow{\text{candidate } (\oplus)} & [\tau_2] \end{array}$$

erfüllt, sofern \oplus assoziativ.

Also dann:

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

Formaler Beweis

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

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

die Vorbedingung von

$$\begin{array}{ccc} \begin{array}{ccc} [\text{Int}] \times [\text{Int}] & \xrightarrow{++} & [\text{Int}] \\ \downarrow f \times f & & \downarrow f \\ \tau_2 \times \tau_2 & \xrightarrow{\oplus} & \tau_2 \end{array} & \Rightarrow & \begin{array}{ccc} [[k \mid k \leftarrow [0..n]] & & \\ \vdots & & \\ [[\text{Int}]] & \xrightarrow{\text{candidate } (++)} & [[\text{Int}]] \\ \downarrow \text{map } f & & \downarrow \text{map } f \\ [\tau_2] & \xrightarrow{\text{candidate } (\oplus)} & [\tau_2] \end{array} \end{array}$$

erfüllt, sofern \oplus assoziativ.

Also dann:

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

Formaler Beweis

Aussage 2: Wenn für jedes $n \geq 0$,

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

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

Formaler Beweis

Aussage 2: Wenn für jedes $n \geq 0$,

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

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

Beweis:

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

Formaler Beweis

Aussage 2: Wenn für jedes $n \geq 0$,

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

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

Beweis:

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

Formaler Beweis

Aussage 2: Wenn für jedes $n \geq 0$,

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

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

Beweis:

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

Formaler Beweis

Aussage 2: Wenn für jedes $n \geq 0$,

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

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

Beweis:

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

Formaler Beweis

Aussage 2: Wenn für jedes $n \geq 0$,

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

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

Beweis:

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

Formaler Beweis

Aussage 2: Wenn für jedes $n \geq 0$,

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

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

Beweis:

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

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 $n \geq 0$,

`candidate` (++) $[[k \mid k \leftarrow [0..n]] = [[0..k \mid k \leftarrow [0..n]]$.

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 $n \geq 0$,
`candidate (++) [[k | k \leftarrow [0..n]]] = [[0..k] | k \leftarrow [0..n]].`

Beweis:

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

```
map (foldl1 ( $\oplus_1$ )  $\circ$  map ([ (Zero, )p One (, Zero)q (, Two)r !!))  
  (candidate (++) [[k | k  $\leftarrow$  [0..(p + q + r)]]])
```

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 $n \geq 0$,
`candidate (++)` `[[k | k \leftarrow [0..n]]` = `[[0..k | k \leftarrow [0..n]]`.

Beweis:

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

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

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 $n \geq 0$,
`candidate (++)` `[[k | k \leftarrow [0..n]]` = `[[0..k | k \leftarrow [0..n]]`.

Beweis:

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

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

und:

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

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 $n \geq 0$,
`candidate (++)` `[[k | k \leftarrow [0..n]]` = `[[0..k | k \leftarrow [0..n]]`.

Beweis:

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

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

und:

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

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 $n \geq 0$,
`candidate (++)` `[[k | k \leftarrow [0..n]]` = `[[0..k | k \leftarrow [0..n]]`.

Beweis:

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

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

und:

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

...

Formaler Beweis

Beweis:

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

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

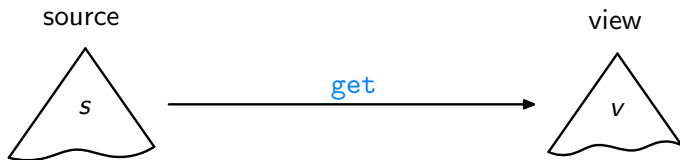
und:

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

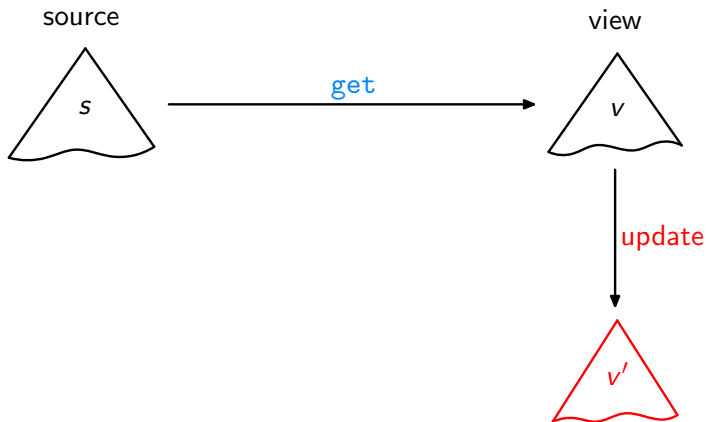
...

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

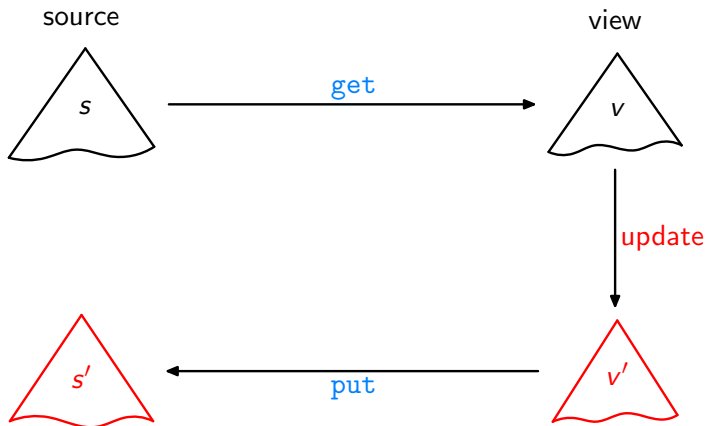
Eine weitere Anwendung: Bidirektionalisierung



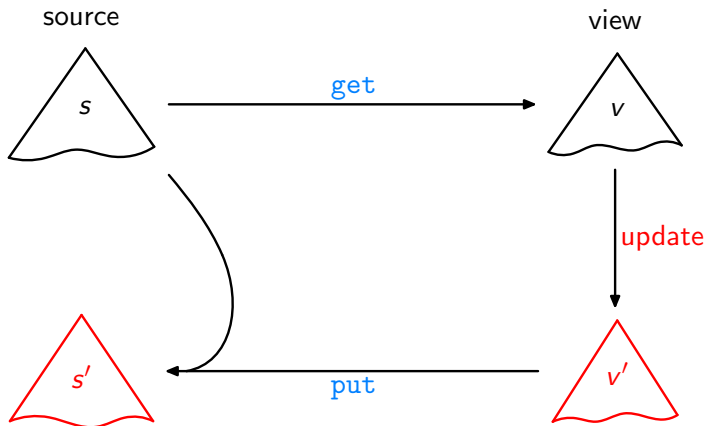
Eine weitere Anwendung: Bidirektionalisierung



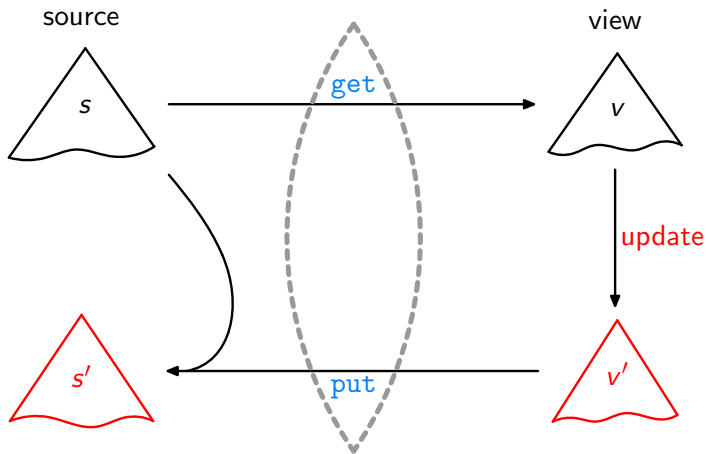
Eine weitere Anwendung: Bidirektionalisierung



Eine weitere Anwendung: Bidirektionalisierung



Eine weitere Anwendung: Bidirektionalisierung



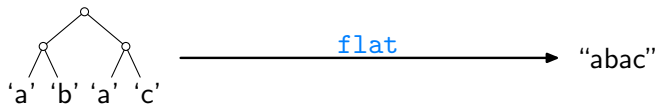
Eine weitere Anwendung: Bidirektionalisierung

Beispiel: `get = flat`

Eine weitere Anwendung: Bidirektionalisierung

Beispiel: `get = flat`

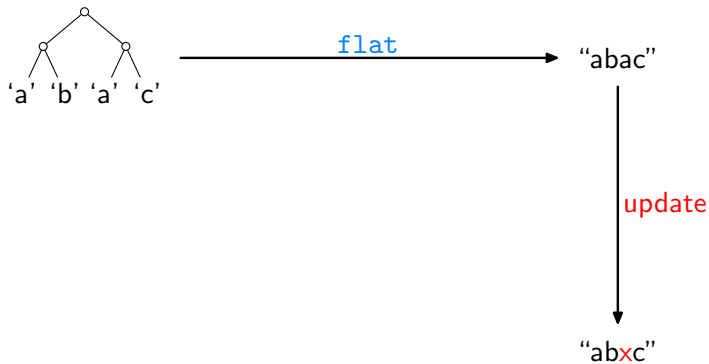
View-Update:



Eine weitere Anwendung: Bidirektionalisierung

Beispiel: `get = flat`

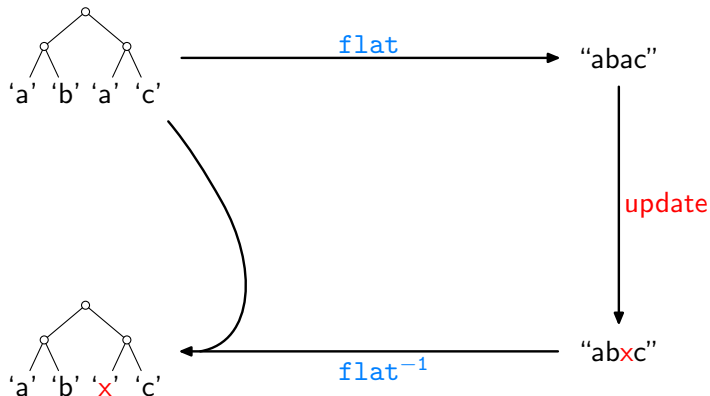
View-Update:



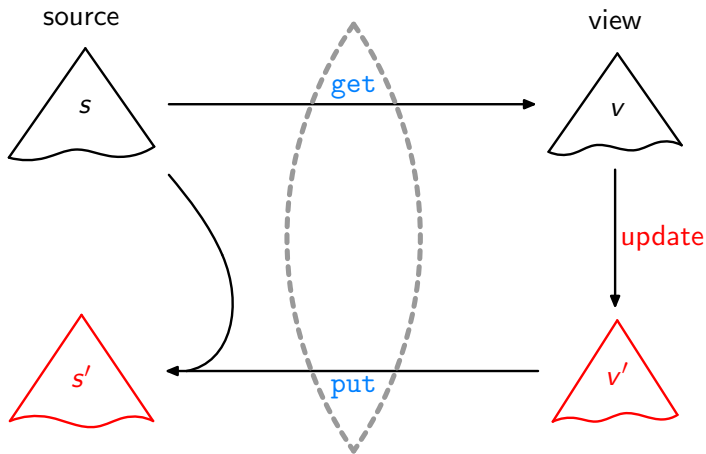
Eine weitere Anwendung: Bidirektionalisierung

Beispiel: `get = flat`

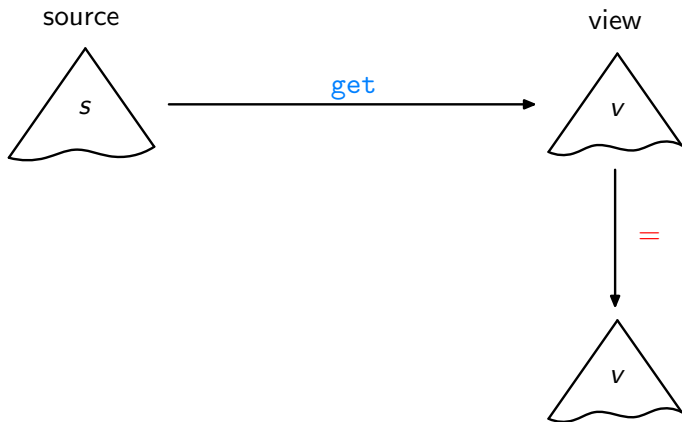
View-Update:



Eine weitere Anwendung: Bidirektionalisierung

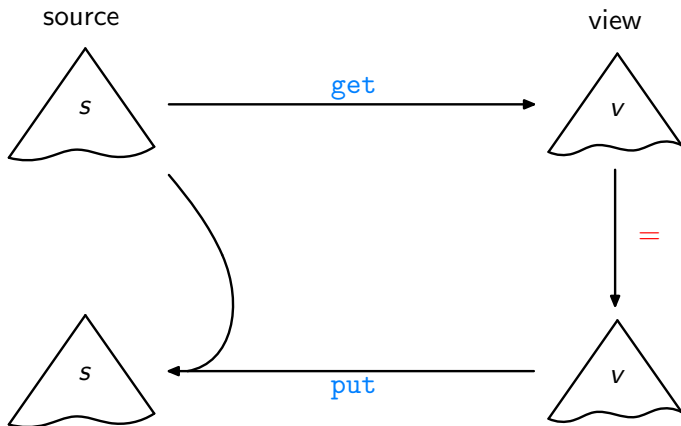


Eine weitere Anwendung: Bidirektionalisierung



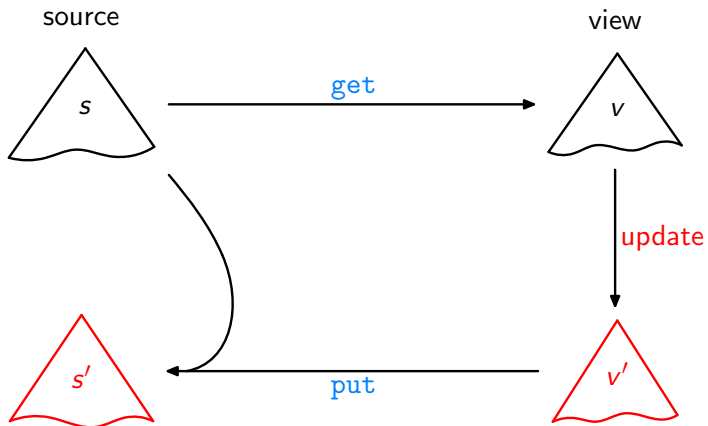
Acceptability / GetPut

Eine weitere Anwendung: Bidirektionalisierung



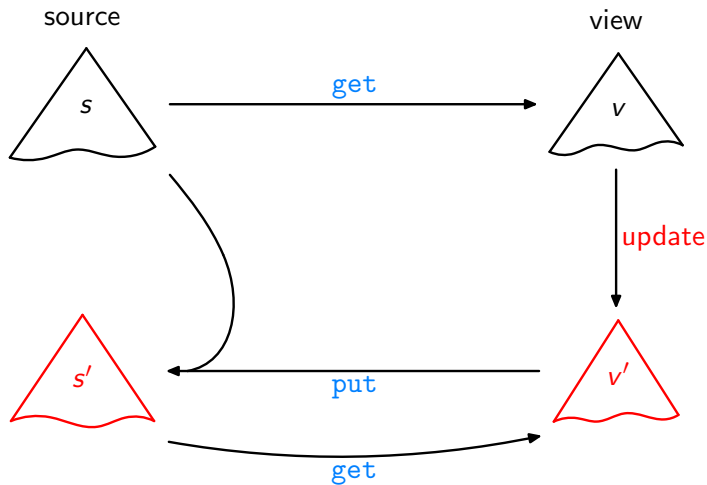
Acceptability / GetPut

Eine weitere Anwendung: Bidirektionalisierung



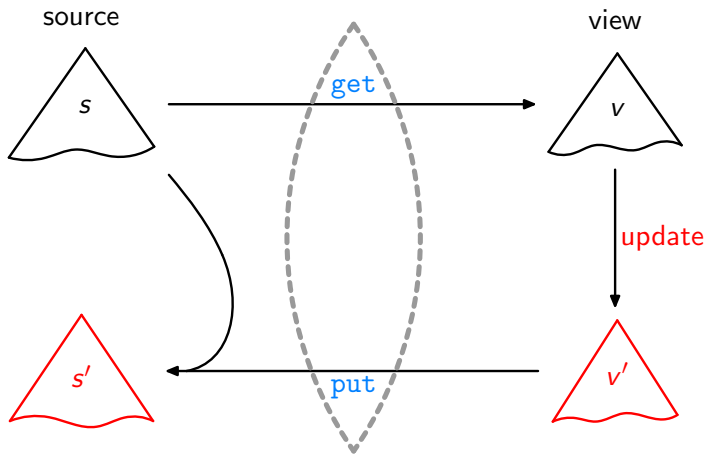
Consistency / PutGet

Eine weitere Anwendung: Bidirektionalisierung



Consistency / PutGet

Eine weitere Anwendung: Bidirektionalisierung



Eine weitere Anwendung: Bidirektionalisierung

Beispiel 1:

`halve` :: $[\alpha] \rightarrow [\alpha]$

`halve as = take (length as `div` 2) as`

Eine weitere Anwendung: Bidirektionalisierung

Beispiel 1:

`halve` :: $[\alpha] \rightarrow [\alpha]$

`halve as = take (length as `div` 2) as`

`halve-1` :: $[\alpha] \rightarrow [\alpha] \rightarrow [\alpha]$

`halve-1 as as' | length as' == n`

`= as' ++ drop n as`

`where n = length as `div` 2`

Eine weitere Anwendung: Bidirektionalisierung

Beispiel 2:

`flat` :: Tree α \rightarrow [α]

`flat` (Leaf a) = [a]

`flat` (Node t_1 t_2) = (`flat` t_1) ++ (`flat` t_2)

Eine weitere Anwendung: Bidirektionalisierung

Beispiel 2:

`flat` :: Tree α \rightarrow [α]

`flat` (Leaf a) = [a]

`flat` (Node t_1 t_2) = (`flat` t_1) ++ (`flat` t_2)

`flat`⁻¹ :: Tree α \rightarrow [α] \rightarrow Tree α

`flat`⁻¹ s v = **case go** s v **of** (t , []) \rightarrow t

where `go` (Leaf a) (b : bs) = (Leaf b , bs)

`go` (Node s_1 s_2) bs = (Node t_1 t_2 , ds)

where (t_1 , cs) = `go` s_1 bs

(t_2 , ds) = `go` s_2 cs

Eine weitere Anwendung: Bidirektionalisierung

Beispiel 3:

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


Eine weitere Anwendung: Bidirektionalisierung

Beispiel 3:

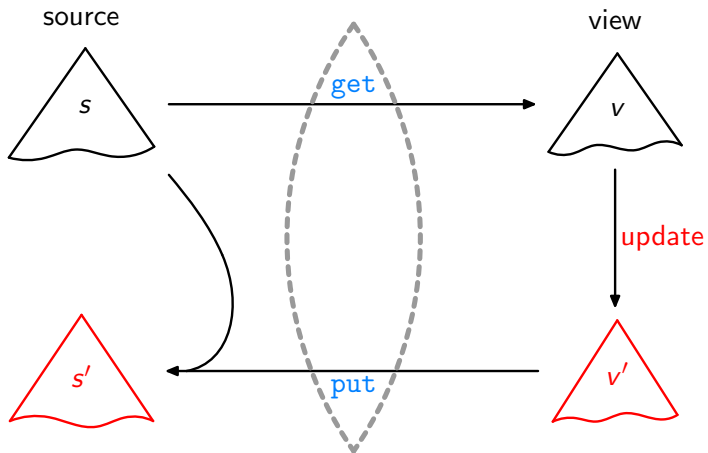
```
rmdups :: Eq  $\alpha$   $\Rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]
```

```
rmdups = List.nub
```

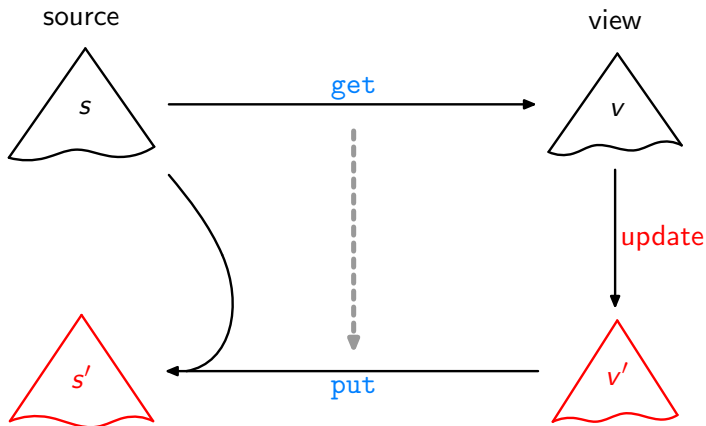
```
rmdups-1 :: Eq  $\alpha$   $\Rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]  $\rightarrow$  [ $\alpha$ ]
```

```
rmdups-1 s v | v == List.nub v && length v == length s'  
              = map (fromJust  $\circ$  flip lookup (zip s' v)) s  
              where s' = List.nub s
```

Eine weitere Anwendung: Bidirektionalisierung

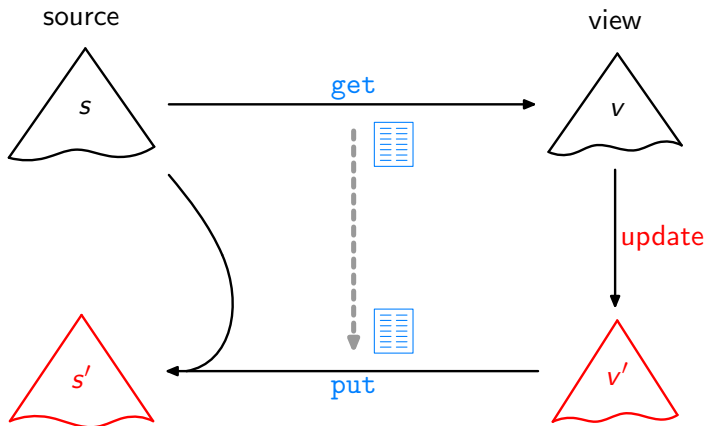


Eine weitere Anwendung: Bidirektionalisierung



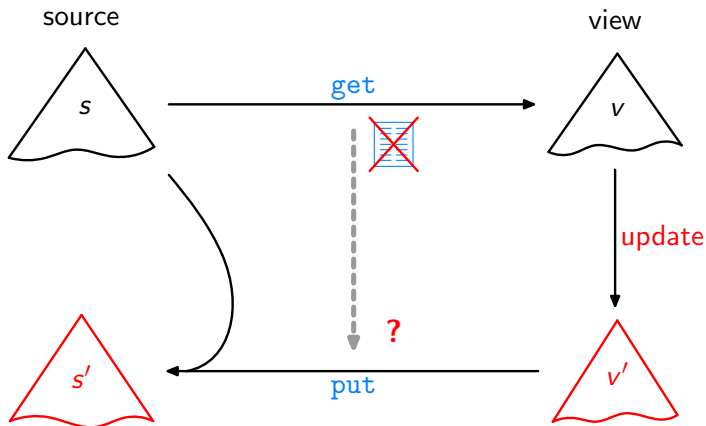
Bidirektionalisierung

Eine weitere Anwendung: Bidirektionalisierung



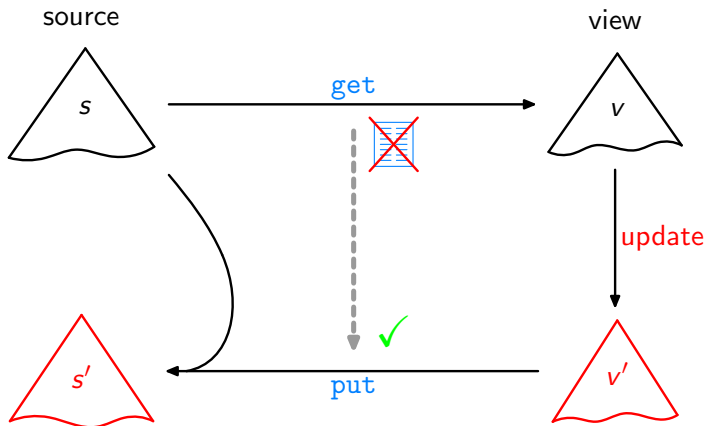
Syntaktische Bidirektionalisierung

Eine weitere Anwendung: Bidirektionalisierung



Semantische 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`, ...

Semantische Bidirektionalisierung

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

[†] "Bidirectionalization for free!"

Semantische Bidirektionalisierung

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

Beispiele:

“abc” $\xrightarrow{\text{tail}}$ “bc”

[†] “Bidirectionalization for free!”

Semantische Bidirektionalisierung

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

Beispiele:

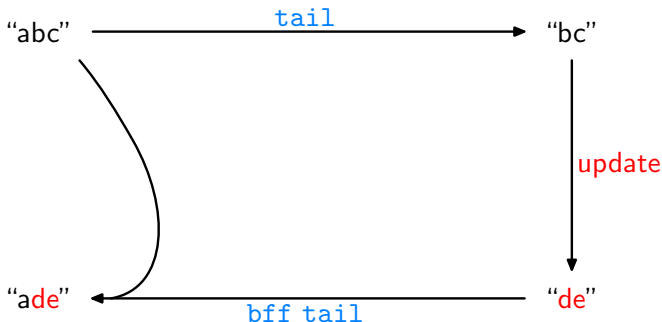


[†] "Bidirectionalization for free!"

Semantische Bidirektionalisierung

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

Beispiele:

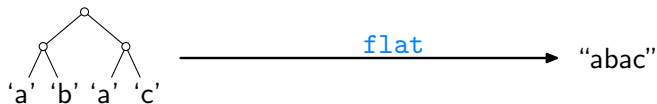


[†] "Bidirectionalization for free!"

Semantische Bidirektionalisierung

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

Beispiele:

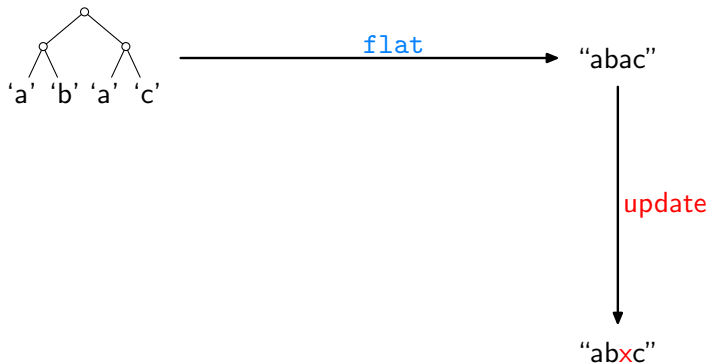


[†] "Bidirectionalization for free!"

Semantische Bidirektionalisierung

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

Beispiele:

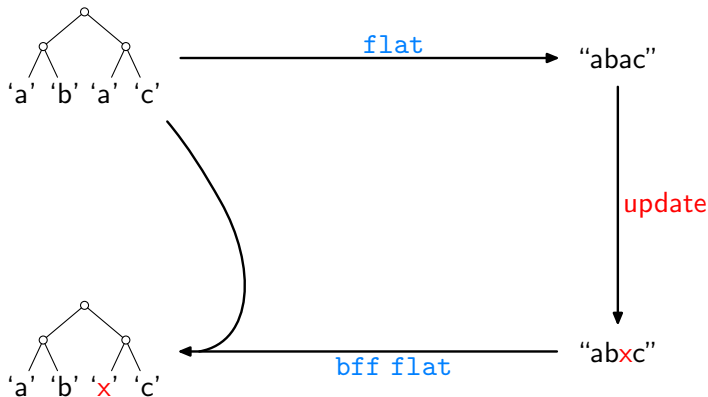


[†] "Bidirectionalization for free!"

Semantische Bidirektionalisierung

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

Beispiele:

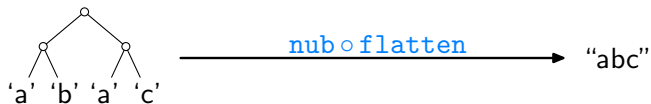


[†] "Bidirectionalization for free!"

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:

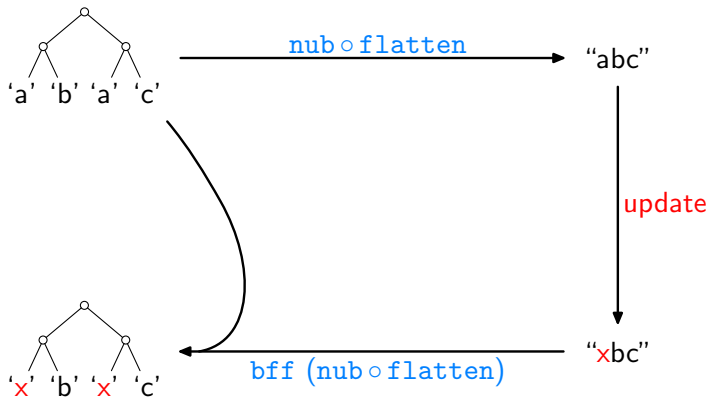


[†] "Bidirectionalization for free!"

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:

`get` :: $[\alpha] \rightarrow [\alpha]$

Analyse spezifischer Aufrufe

Angenommen, es sei gegeben:

$$\text{get} :: [\alpha] \rightarrow [\alpha]$$

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

Analyse spezifischer Aufrufe

Angenommen, es sei gegeben:

$$\text{get} :: [\alpha] \rightarrow [\alpha]$$

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

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

Analyse spezifischer Aufrufe

Angenommen, es sei gegeben:

$$\text{get} :: [\alpha] \rightarrow [\alpha]$$

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

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

Etwa:

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

Analyse spezifischer Aufrufe

Angenommen, es sei gegeben:

$$\text{get} :: [\alpha] \rightarrow [\alpha]$$

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

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

Etwa:

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

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

Verwendung eines freien Theorems

Für jedes

$$g :: [\alpha] \rightarrow [\alpha]$$

gilt:

$$\text{map } f (g \ l) = g (\text{map } f \ l)$$

für beliebige f und l .

Verwendung eines freien Theorems

Für jedes

$$g :: [\alpha] \rightarrow [\alpha]$$

gilt:

$$\text{map } f (g l) = g (\text{map } f l)$$

für beliebige f und l .

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

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

Verwendung eines freien Theorems

Für jedes

$$g :: [\alpha] \rightarrow [\alpha]$$

gilt:

$$\text{map } f (g l) = g (\text{map } f l)$$

für beliebige f und l .

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

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

Verwendung eines freien Theorems

Für jedes

$$g :: [\alpha] \rightarrow [\alpha]$$

gilt:

$$\text{map } f (g l) = g (\text{map } f l)$$

für beliebige f und l .

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

$$\begin{aligned} \text{map } (s !!) (\text{get } [0..n]) \\ = \text{get } s \end{aligned}$$

Verwendung eines freien Theorems

Für jedes

$$g :: [\alpha] \rightarrow [\alpha]$$

gilt:

$$\text{map } f (g l) = g (\text{map } f l)$$

für beliebige f und l .

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

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

für jedes $\text{get} :: [\alpha] \rightarrow [\alpha]$.

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

Allgemein, für

`get` :: $S \rightarrow V$

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

Allgemein, für

$$\text{get} :: S \rightarrow V$$

definiere ein V^C und

$$\text{compl} :: S \rightarrow V^C$$

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

Allgemein, für

$$\text{get} :: S \rightarrow V$$

definiere ein V^C und

$$\text{compl} :: S \rightarrow V^C$$

so dass

$$\lambda s \rightarrow (\text{get } s, \text{compl } s)$$

injektiv ist

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

Allgemein, für

$$\text{get} :: S \rightarrow V$$

definiere ein V^C und

$$\text{compl} :: S \rightarrow V^C$$

so dass

$$\lambda s \rightarrow (\text{get } s, \text{compl } s)$$

injektiv ist und ein Inverses hat:

$$\text{inv} :: (V, V^C) \rightarrow S$$

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

Allgemein, für

$$\text{get} :: S \rightarrow V$$

definiere ein V^C und

$$\text{compl} :: S \rightarrow V^C$$

so dass

$$\lambda s \rightarrow (\text{get } s, \text{compl } s)$$

injektiv ist und ein Inverses hat:

$$\text{inv} :: (V, V^C) \rightarrow S$$

Dann:

$$\text{put} :: S \rightarrow V \rightarrow S$$

$$\text{put } s \ v' = \text{inv } (v', \text{compl } s)$$

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

Allgemein, für

$$\text{get} :: S \rightarrow V$$

definiere ein V^C und

$$\text{compl} :: S \rightarrow V^C$$

so dass

$$\lambda s \rightarrow (\text{get } s, \text{compl } s)$$

injektiv ist und ein Inverses hat:

$$\text{inv} :: (V, V^C) \rightarrow S$$

Dann:

$$\text{put} :: S \rightarrow V \rightarrow S$$

$$\text{put } s v' = \text{inv } (v', \text{compl } s)$$

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

Der „Constant-Complement“ Ansatz

Für unseren Fall,

$$\text{get} :: [\alpha] \rightarrow [\alpha],$$

was sollten wir wählen für V^C und

$$\text{compl} :: [\alpha] \rightarrow V^C \quad ???$$

Der „Constant-Complement“ Ansatz

Für unseren Fall,

$$\text{get} :: [\alpha] \rightarrow [\alpha],$$

was sollten wir wählen für V^C und

$$\text{compl} :: [\alpha] \rightarrow V^C \quad ???$$

Um

$$\lambda s \rightarrow (\text{get } s, \text{compl } s)$$

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

Der „Constant-Complement“ Ansatz

Für unseren Fall,

$$\text{get} :: [\alpha] \rightarrow [\alpha],$$

was sollten wir wählen für V^C und

$$\text{compl} :: [\alpha] \rightarrow V^C \quad ???$$

Um

$$\lambda s \rightarrow (\text{get } s, \text{compl } s)$$

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

Kandidaten:

1. Länge der ursprünglichen Liste

Der „Constant-Complement“ Ansatz

Für unseren Fall,

$$\text{get} :: [\alpha] \rightarrow [\alpha],$$

was sollten wir wählen für V^C und

$$\text{compl} :: [\alpha] \rightarrow V^C \quad ???$$

Um

$$\lambda s \rightarrow (\text{get } s, \text{compl } s)$$

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

Kandidaten:

1. Länge der ursprünglichen Liste
2. verworfene Listenelemente

Der „Constant-Complement“ Ansatz

Für unseren Fall,

$$\text{get} :: [\alpha] \rightarrow [\alpha],$$

was sollten wir wählen für V^C und

$$\text{compl} :: [\alpha] \rightarrow V^C \quad ???$$

Um

$$\lambda s \rightarrow (\text{get } s, \text{compl } s)$$

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

```
type IntMap  $\alpha$  = [(Int,  $\alpha$ )]
```

```
compl :: [ $\alpha$ ]  $\rightarrow$  (Int, IntMap  $\alpha$ )
```

```
compl s = let n = (length s) - 1
```

```
    t = [0..n]
```

```
    g = zip t s
```

```
    g' = filter ( $\lambda(i, \_) \rightarrow$  notElem i (get t)) g
```

```
in (n + 1, g')
```

Die Komplement-Funktion

```
type IntMap  $\alpha$  = [(Int,  $\alpha$ )]
```

```
compl :: [ $\alpha$ ]  $\rightarrow$  (Int, IntMap  $\alpha$ )
```

```
compl s = let n = (length s) - 1
```

```
    t = [0..n]
```

```
    g = zip t s
```

```
    g' = filter ( $\lambda(i, \_) \rightarrow$  notElem i (get t)) g
```

```
  in (n + 1, g')
```

Zum Beispiel:

```
get = tail       $\rightsquigarrow$  compl "abcde" = (5, [(0, 'a')])
```


Die Komplement-Funktion

```
type IntMap  $\alpha$  = [(Int,  $\alpha$ )]
```

```
compl :: [ $\alpha$ ]  $\rightarrow$  (Int, IntMap  $\alpha$ )
```

```
compl s = let n = (length s) - 1
```

```
    t = [0..n]
```

```
    g = zip t s
```

```
    g' = filter ( $\lambda(i, \_)$   $\rightarrow$  notElem i (get t)) g
```

```
  in (n + 1, g')
```

Zum Beispiel:

```
get = tail       $\rightsquigarrow$  compl "abcde" = (5, [(0, 'a')])
```

```
get = take 3     $\rightsquigarrow$  compl "abcde" = (5, [(3, 'd'), (4, 'e')])
```

Die Komplement-Funktion

```
type IntMap  $\alpha$  = [(Int,  $\alpha$ )]
```

```
compl :: [ $\alpha$ ]  $\rightarrow$  (Int, IntMap  $\alpha$ )
```

```
compl s = let n = (length s) - 1
```

```
    t = [0..n]
```

```
    g = zip t s
```

```
    g' = filter ( $\lambda(i, \_) \rightarrow$  notElem i (get t)) g
```

```
  in (n + 1, g')
```

Zum Beispiel:

```
get = tail       $\rightsquigarrow$  compl "abcde" = (5, [(0, 'a')])
```

```
get = take 3     $\rightsquigarrow$  compl "abcde" = (5, [(3, 'd'), (4, 'e')])
```

```
get = reverse   $\rightsquigarrow$  compl "abcde" = (5, [])
```

Ein Inverses zu $\lambda s \rightarrow (\text{get } s, \text{compl } s)$

```
inv :: ([α], (Int, IntMap α)) → [α]
```

```
inv (v', (n + 1, g')) = let t = [0..n]
```

```
    h = assoc (get t) v'
```

```
    h' = h ++ g'
```

```
  in seq h (map (\i → fromJust (lookup i h')) t)
```

Ein Inverses zu $\lambda s \rightarrow (\text{get } s, \text{compl } s)$

`inv` :: $([\alpha], (\text{Int}, \text{IntMap } \alpha)) \rightarrow [\alpha]$

```
inv (v', (n + 1, g')) = let t = [0..n]
                        h = assoc† (get t) v'
                        h' = h ++ g'
    in seq h (map (\i → fromJust (lookup i h')) t)
```

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

Ein Inverses zu $\lambda s \rightarrow (\text{get } s, \text{compl } s)$

`inv` :: $([\alpha], (\text{Int}, \text{IntMap } \alpha)) \rightarrow [\alpha]$

```
inv (v', (n + 1, g')) = let t = [0..n]
                        h = assoc† (get t) v'
                        h' = h ++ g'
                    in seq h (map (\i → fromJust (lookup i h')) t)
```

Zum Beispiel:

```
get = tail    ~> inv ("bcde", (5, [(0, 'a')])) = "abcde"
```

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

Ein Inverses zu $\lambda s \rightarrow (\text{get } s, \text{compl } s)$

`inv` :: $([\alpha], (\text{Int}, \text{IntMap } \alpha)) \rightarrow [\alpha]$

```
inv (v', (n + 1, g')) = let t = [0..n]
                        h = assoc† (get t) v'
                        h' = h ++ g'
                    in seq h (map (\i → fromJust (lookup i h')) t)
```

Zum Beispiel:

`get = tail` \rightsquigarrow `inv ("bcde", (5, [(0, 'a')])) = "abcde"`

`get = take 3` \rightsquigarrow `inv ("xyz", (5, [(3, 'd'), (4, 'e')])) = "xyzde"`

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

Ein Inverses zu $\lambda s \rightarrow (\text{get } s, \text{compl } s)$

`inv` :: $([\alpha], (\text{Int}, \text{IntMap } \alpha)) \rightarrow [\alpha]$

```
inv (v', (n + 1, g')) = let t = [0..n]
                          h = assoc† (get t) v'
                          h' = h ++ g'
                      in seq h (map (\i → fromJust (lookup i h')) t)
```

Zum Beispiel:

`get = tail` \rightsquigarrow `inv ("bcde", (5, [(0, 'a')])) = "abcde"`

`get = take 3` \rightsquigarrow `inv ("xyz", (5, [(3, 'd'), (4, 'e')])) = "xyzde"`

Formal zu beweisen:

- `inv (get s, compl s) = s`
- wenn `inv (v, c)` definiert, dann `get (inv (v, c)) = v`
- wenn `inv (v, c)` definiert, dann `compl (inv (v, c)) = c`

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