Type-Based Reasoning and Bidirectional Transformation

Janis Voigtländer

Technische Universität Dresden

February 20th, 2009

Polymorphic Types: An Example in Haskell

A standard function:

$$
\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}
$$

Polymorphic Types: An Example in Haskell

A standard function:

$$
\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}
$$

Some invocations:

$\texttt{map succ}\ [1, 2, 3] \qquad = [2, 3, 4] \qquad\qquad$ — $\alpha, \beta \mapsto \mathsf{Int}, \mathsf{Int}$

Polymorphic Types: An Example in Haskell

A standard function:

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

Some invocations:

$\texttt{map succ}\ [1, 2, 3] = [2, 3, 4]$ — $\alpha, \beta \mapsto \mathsf{Int}, \mathsf{Int}$

$\texttt{map not}\ [\mathsf{True}, \mathsf{False}] = [\mathsf{False}, \mathsf{True}]$ — $\alpha, \beta \mapsto \mathsf{Bool}, \mathsf{Bool}$

Polymorphic Types: An Example in Haskell

A standard function:

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

Some invocations:

$$\begin{array}{lll}
\texttt{map succ}\ [1,2,3] & = [2,3,4] & \text{— } \alpha,\beta \mapsto \mathsf{Int},\mathsf{Int} \\
\texttt{map not}\ [\mathsf{True},\mathsf{False}] & = [\mathsf{False},\mathsf{True}] & \text{— } \alpha,\beta \mapsto \mathsf{Bool},\mathsf{Bool} \\
\texttt{map even}\ [1,2,3] & = [\mathsf{False},\mathsf{True},\mathsf{False}] & \text{— } \alpha,\beta \mapsto \mathsf{Int},\mathsf{Bool}
\end{array}$$

Polymorphic Types: An Example in Haskell

A standard function:

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

Some invocations:

$$\begin{array}{lll}
\texttt{map succ}\ [1,2,3] & = [2,3,4] & \text{— } \alpha,\beta \mapsto \mathsf{Int},\mathsf{Int} \\
\texttt{map not}\ \ [\mathsf{True},\mathsf{False}] & = [\mathsf{False},\mathsf{True}] & \text{— } \alpha,\beta \mapsto \mathsf{Bool},\mathsf{Bool} \\
\texttt{map even}\ [1,2,3] & = [\mathsf{False},\mathsf{True},\mathsf{False}] & \text{— } \alpha,\beta \mapsto \mathsf{Int},\mathsf{Bool} \\
\texttt{map not}\ \ [1,2,3] & &
\end{array}$$

Polymorphic Types: An Example in Haskell

A standard function:

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

Some invocations:

$\texttt{map succ}\ [1, 2, 3] = [2, 3, 4]$ — $\alpha, \beta \mapsto \mathsf{Int}, \mathsf{Int}$

$\texttt{map not}\ [\mathsf{True}, \mathsf{False}] = [\mathsf{False}, \mathsf{True}]$ — $\alpha, \beta \mapsto \mathsf{Bool}, \mathsf{Bool}$

$\texttt{map even}\ [1, 2, 3] = [\mathsf{False}, \mathsf{True}, \mathsf{False}]$ — $\alpha, \beta \mapsto \mathsf{Int}, \mathsf{Bool}$

$\texttt{map not}\ [1, 2, 3]$ $\lightning$ rejected at compile-time

Another Example

$$
\begin{aligned}
&\texttt{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha] \\
&\texttt{takeWhile}\ p\ [\,] = [\,] \\
&\texttt{takeWhile}\ p\ (a : as)\ \mid\ p\ a = a : (\texttt{takeWhile}\ p\ as) \\
&\qquad\qquad\qquad\qquad\quad \mid\ \texttt{otherwise} = [\,]
\end{aligned}
$$

Another Example

$$
\begin{aligned}
&\texttt{takeWhile} :: (\alpha \to \mathrm{Bool}) \to [\alpha] \to [\alpha] \\
&\texttt{takeWhile}\ p\ [\,] = [\,] \\
&\texttt{takeWhile}\ p\ (a : as)\ \mid\ p\ a \qquad\qquad = a : (\texttt{takeWhile}\ p\ as) \\
&\qquad\qquad\qquad\qquad\quad\ \mid\ \texttt{otherwise} = [\,]
\end{aligned}
$$

For every choice of p, f, and l:

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ \ l) \quad = \quad \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ \ l)$$

Provable by induction.

Another Example

$$
\begin{array}{lll}
\texttt{takeWhile} :: (\alpha \rightarrow \mathsf{Bool}) \rightarrow [\alpha] \rightarrow [\alpha] \\
\texttt{takeWhile}\ p\ [\,] & = [\,] \\
\texttt{takeWhile}\ p\ (a : as) & \mid\ p\ a & = a : (\texttt{takeWhile}\ p\ as) \\
 & \mid\ \texttt{otherwise} & = [\,]
\end{array}
$$

For every choice of p, f, and l:

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ \ l) \quad = \quad \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ \ l)$$

Provable by induction.

Or as a "free theorem" [Wadler, FPCA'89].

Another Example

$$\texttt{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

For every choice of p, f, and l:

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ \ l) \quad = \quad \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ \ l)$$

Provable by induction.

Or as a "free theorem" [Wadler, FPCA'89].

Another Example

$$\texttt{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

$$\texttt{filter} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

For every choice of p, f, and l:

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ l)$$

$$\texttt{filter}\ p\ (\texttt{map}\ f\ l) = \texttt{map}\ f\ (\texttt{filter}\ (p \circ f)\ l)$$

Another Example

$$\texttt{takeWhile} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$
$$\texttt{filter} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$
$$\texttt{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$$

For every choice of p, f, and l:

$$\texttt{takeWhile}\ p\ (\texttt{map}\ f\ \ l) \quad = \quad \texttt{map}\ f\ (\texttt{takeWhile}\ (p \circ f)\ \ l)$$
$$\texttt{filter}\ p\ (\texttt{map}\ f\ \ l) \quad = \quad \texttt{map}\ f\ (\texttt{filter}\ (p \circ f)\ \ l)$$
$$\texttt{g}\ p\ (\texttt{map}\ f\ \ l) \quad = \quad \texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ \ l)$$

Why, Intuitively

- $\mathrm{g} :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.

Why, Intuitively

- $g :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.

Why, Intuitively

- $g :: (\alpha \rightarrow \mathsf{Bool}) \rightarrow [\alpha] \rightarrow [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.

Why, Intuitively

- $g :: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.

Why, Intuitively

- `g` :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists (`map` f l) and l always have equal length.

Why, Intuitively

- `g` :: $(\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists (`map` f l) and l always have equal length.
- Applying p to an element of (`map` f l) always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.

Why, Intuitively

- `g` $:: (\alpha \to \mathsf{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists (`map` $f\ l$) and l always have equal length.
- Applying p to an element of (`map` $f\ l$) always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- `g` with p always chooses "the same" elements from (`map` $f\ l$) for output as does `g` with $(p \circ f)$ from l,

Why, Intuitively

- `g` $:: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists (`map` f l) and l always have equal length.
- Applying p to an element of (`map` f l) always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- `g` with p always chooses "the same" elements from (`map` f l) for output as does `g` with $(p \circ f)$ from l, except that it outputs their images under f.

Why, Intuitively

- $\mathtt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists ($\mathtt{map}\ f\ l$) and l always have equal length.
- Applying p to an element of ($\mathtt{map}\ f\ l$) always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- $\mathtt{g}$ with p always chooses "the same" elements from ($\mathtt{map}\ f\ l$) for output as does $\mathtt{g}$ with $(p \circ f)$ from l, except that it outputs their images under f.
- $(\mathtt{g}\ p\ (\mathtt{map}\ f\ l))$ is equivalent to $(\mathtt{map}\ f\ (\mathtt{g}\ (p \circ f)\ l))$.

Why, Intuitively

- $\texttt{g} :: (\alpha \to \text{Bool}) \to [\alpha] \to [\alpha]$ must work uniformly for every instantiation of α.
- The output list can only contain elements from the input list l.
- Which, and in which order/multiplicity, can only be decided based on l and the input predicate p.
- The only means for this decision are to inspect the length of l and to check the outcome of p on its elements.
- The lists $(\texttt{map}\ f\ l)$ and l always have equal length.
- Applying p to an element of $(\texttt{map}\ f\ l)$ always has the same outcome as applying $(p \circ f)$ to the corresponding element of l.
- $\texttt{g}$ with p always chooses "the same" elements from $(\texttt{map}\ f\ l)$ for output as does $\texttt{g}$ with $(p \circ f)$ from l, except that it outputs their images under f.
- $(\texttt{g}\ p\ (\texttt{map}\ f\ l))$ is equivalent to $(\texttt{map}\ f\ (\texttt{g}\ (p \circ f)\ l))$.
- That is what was claimed!

Automatic Generation of Free Theorems

At `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]

Automatic Generation of Free Theorems

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

DFG-Project VO 1512/1-1

Free theorems

- POPL'04
- TCS'07
- I&C'09

DFG-Project VO 1512/1-1

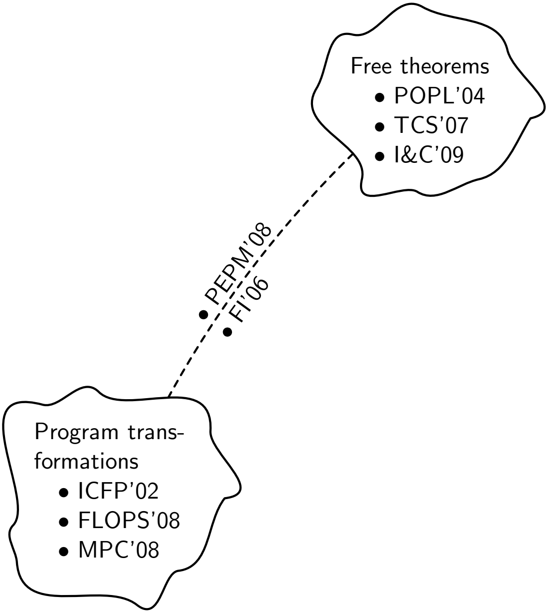
Free theorems

- POPL'04
- TCS'07
- I&C'09

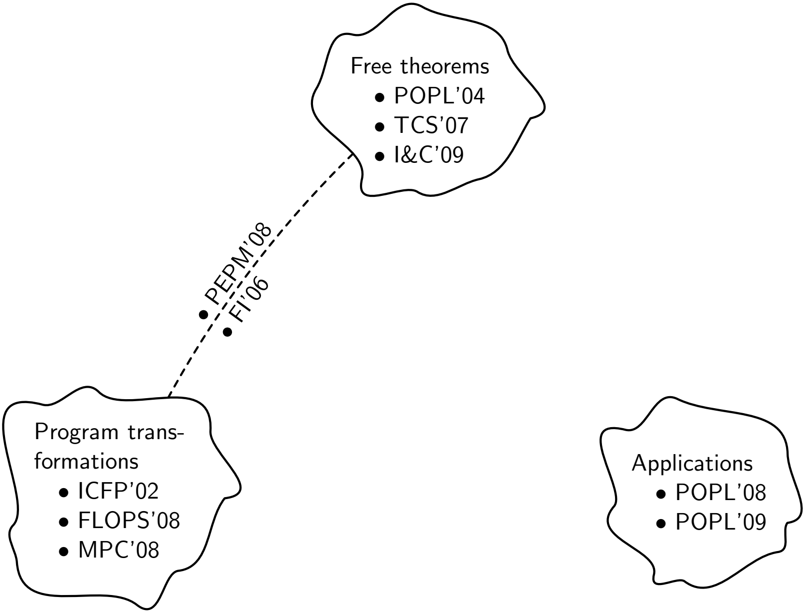
Program transformations

- ICFP'02
- FLOPS'08
- MPC'08

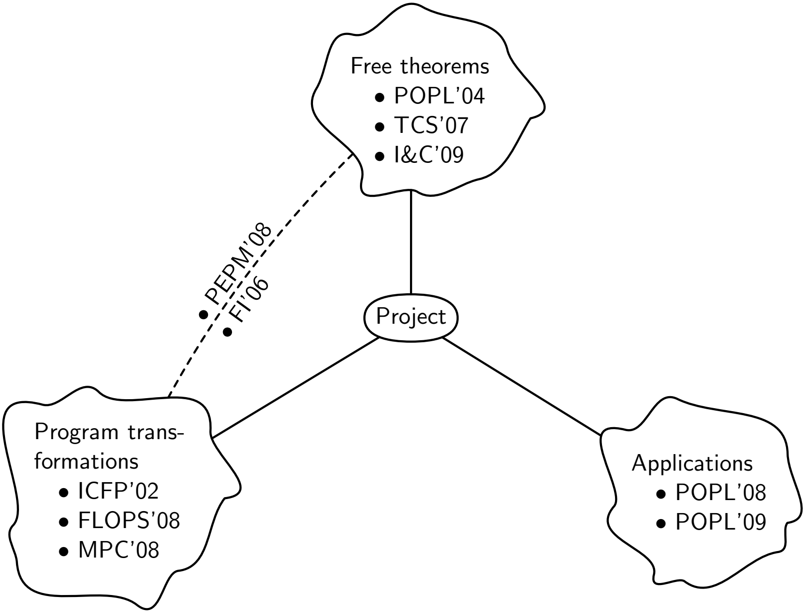
DFG-Project VO 1512/1-1



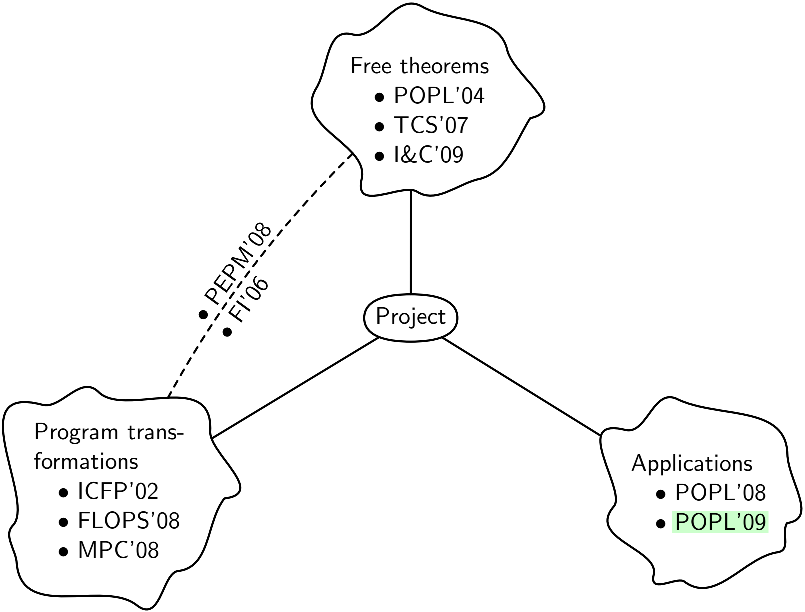
DFG-Project VO 1512/1-1



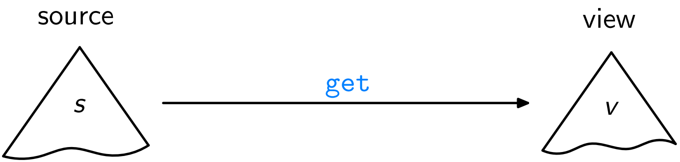
DFG-Project VO 1512/1-1



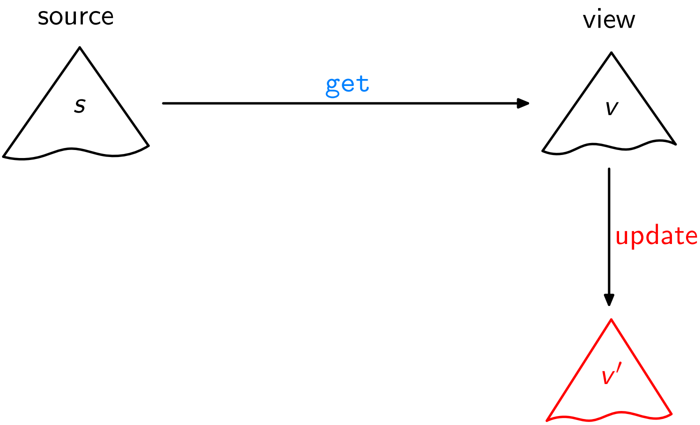
DFG-Project VO 1512/1-1



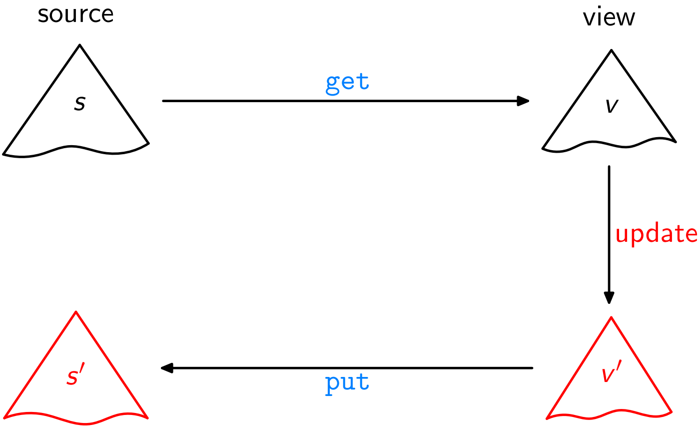
Bidirectional Transformation



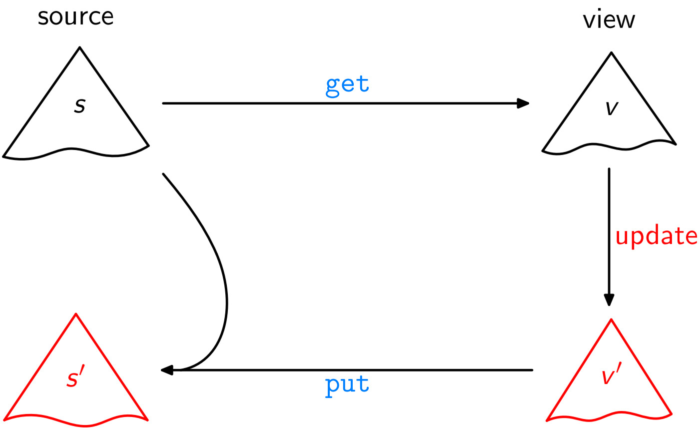
Bidirectional Transformation



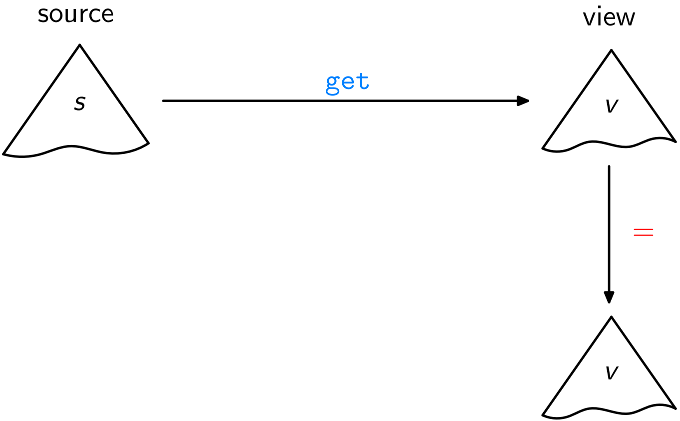
Bidirectional Transformation



Bidirectional Transformation

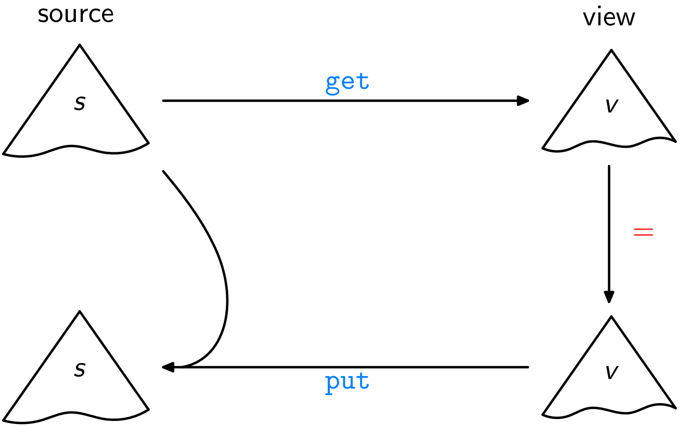


Bidirectional Transformation



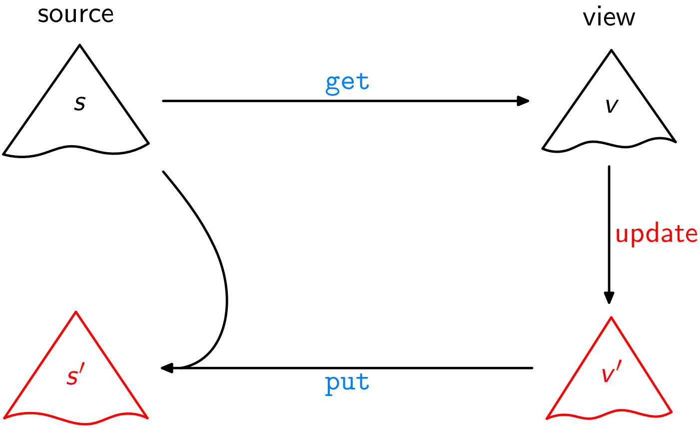
Acceptability / GetPut

Bidirectional Transformation



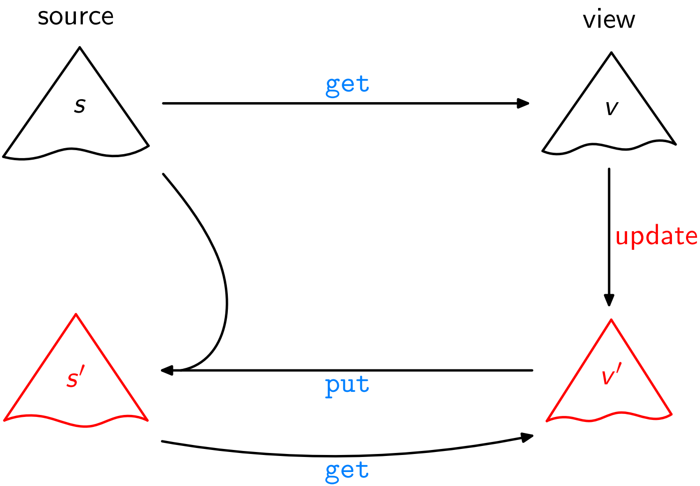
Acceptability / GetPut

Bidirectional Transformation



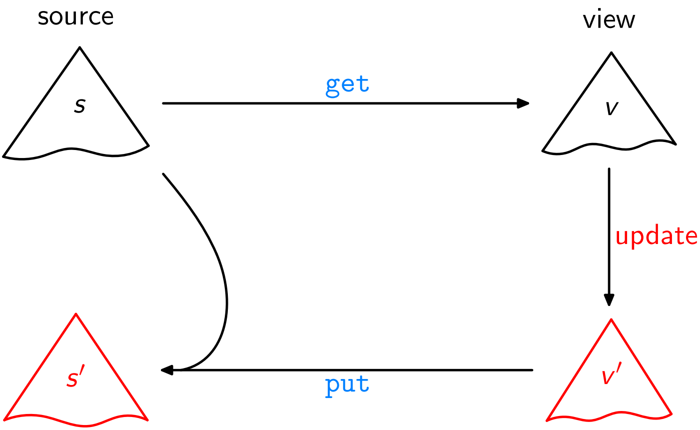
Consistency / PutGet

Bidirectional Transformation

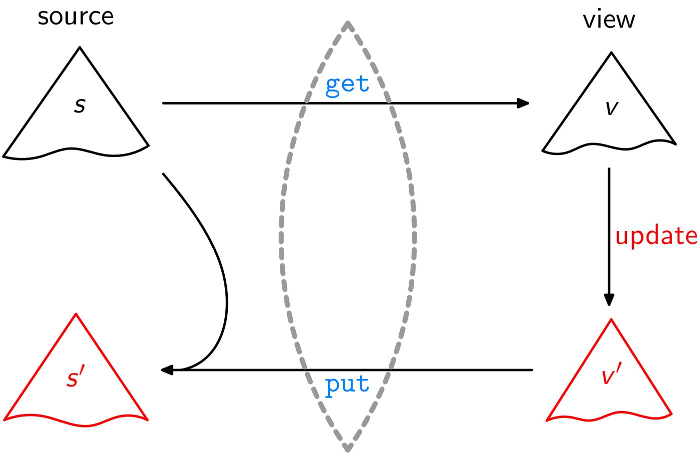


Consistency / PutGet

Bidirectional Transformation



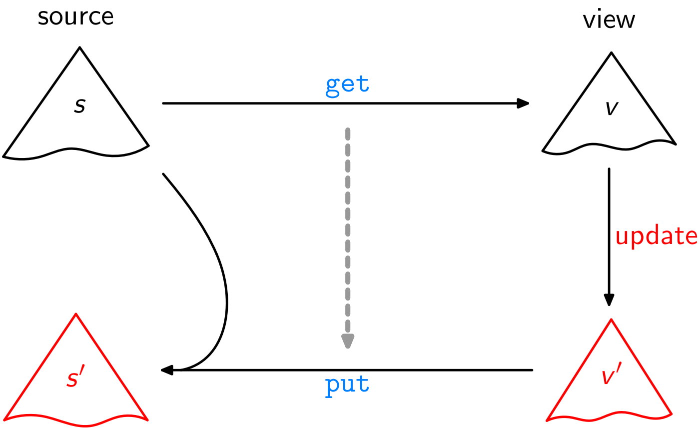
Bidirectional Transformation



Lenses, DSLs

[Foster et al., ACM TOPLAS'07, ...]

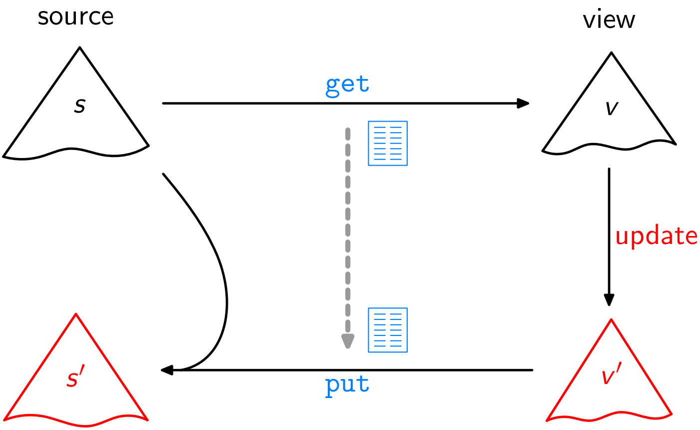
Bidirectional Transformation



Bidirectionalization

[Matsuda et al., ICFP'07]

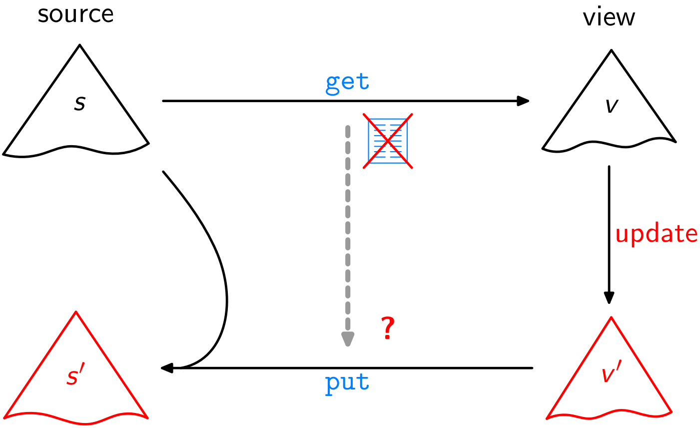
Bidirectional Transformation



Syntactic Bidirectionalization

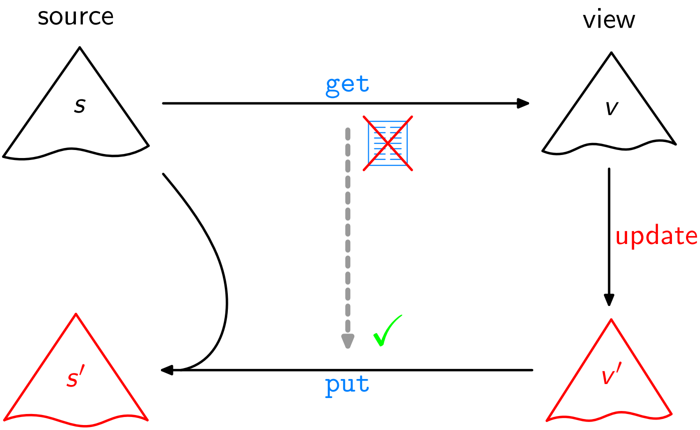
[Matsuda et al., ICFP'07]

Bidirectional Transformation



Semantic Bidirectionalization

Bidirectional Transformation



Semantic Bidirectionalization

[V., POPL'09]

Semantic Bidirectionalization

Aim: Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Semantic Bidirectionalization

Aim: Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:

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

Semantic Bidirectionalization

Aim: Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:

$$
\begin{tikzcd}
\text{“abc”} \arrow[r, "\texttt{tail}"] & \text{“bc”} \arrow[d, "\text{update}"] \\
 & \text{“de”}
\end{tikzcd}
$$

Semantic Bidirectionalization

Aim: Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ...

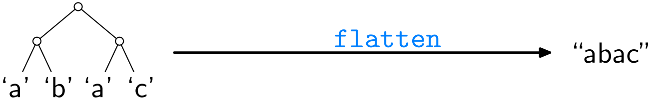
Examples:

$$
\begin{tikzcd}[column sep=huge, row sep=huge]
\text{“abc”} \arrow[r, "\texttt{tail}"] \arrow[d, bend left] & \text{“bc”} \arrow[d, "\text{update}"] \\
\text{“ade”} & \text{“de”} \arrow[l, "\texttt{bff tail}"]
\end{tikzcd}
$$

Semantic Bidirectionalization

Aim: Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ...

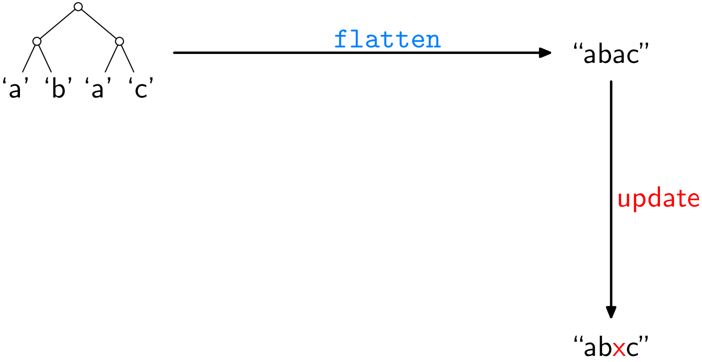
Examples:



Semantic Bidirectionalization

Aim: Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ...

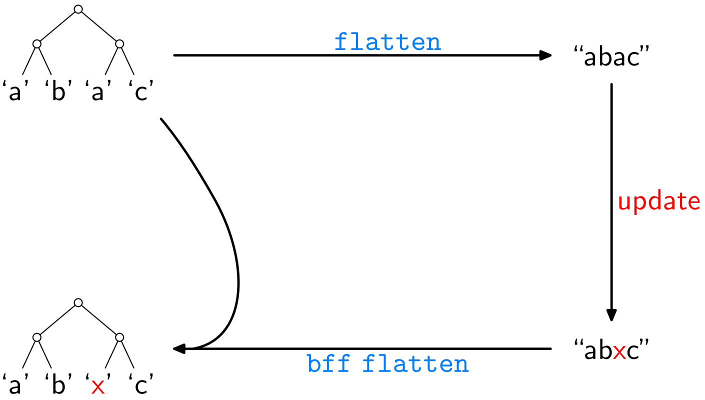
Examples:



Semantic Bidirectionalization

Aim: Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ...

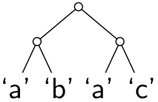
Examples:



Semantic Bidirectionalization

Aim: Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:



$\xrightarrow{\texttt{nub} \circ \texttt{flatten}}$ “abc”

Semantic Bidirectionalization

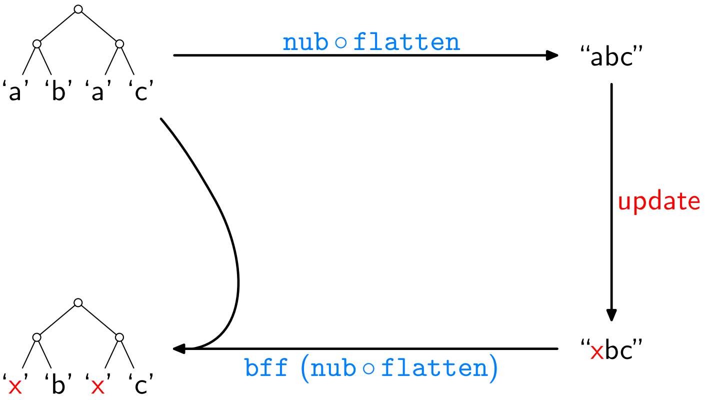
Aim: Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:

Semantic Bidirectionalization

Aim: Write a higher-order function `bff` such that any `get` and `bff get` satisfy GetPut, PutGet, ...

Examples:



Analyzing Specific Instances

Assume we are given some

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

How can we, or `bff`, analyze it without access to its source code?

Analyzing Specific Instances

Assume we are given some

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

How can we, or `bff`, analyze it without access to its source code?

Idea: How about applying `get` to some input?

Analyzing Specific Instances

Assume we are given some

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

How can we, or `bff`, analyze it without access to its source code?

Idea: How about applying `get` to some input?

Like:

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

Analyzing Specific Instances

Assume we are given some

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

How can we, or `bff`, analyze it without access to its source code?

Idea: How about applying `get` to some input?

Like:

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

Then transfer the gained insights to source lists other than $[0..n]$!

Using a Free Theorem

For every

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

we have

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

for arbitrary f and l, where

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

Using a Free Theorem

For every

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

we have

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

for arbitrary f and l, where

$$\begin{array}{l}
\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{array}$$

Given an arbitrary list s of length $n+1$, set $f = (s\,!!)$, $l = [0..n]$, leading to:

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

Using a Free Theorem

For every

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

we have

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

for arbitrary f and l, where

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

Given an arbitrary list s of length $n+1$, set $f = (s\,!!)$, $l = [0..n]$, leading to:

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

Using a Free Theorem

For every

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

we have

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

for arbitrary f and l, where

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

Given an arbitrary list s of length $n+1$,

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

$$= \quad \texttt{get}\ s$$

Using a Free Theorem

For every

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

we have

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

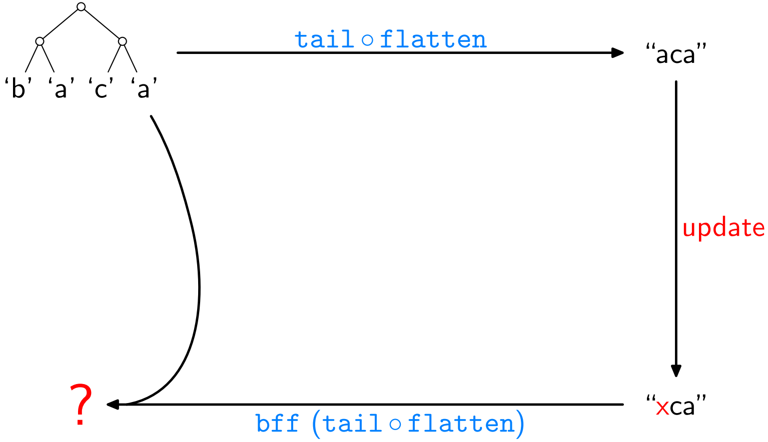
for arbitrary f and l, where

$$\begin{aligned}
&\texttt{map} :: (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ f\ [\,] \qquad\quad = [\,] \\
&\texttt{map}\ f\ (a : as) = (f\ a) : (\texttt{map}\ f\ as)
\end{aligned}$$

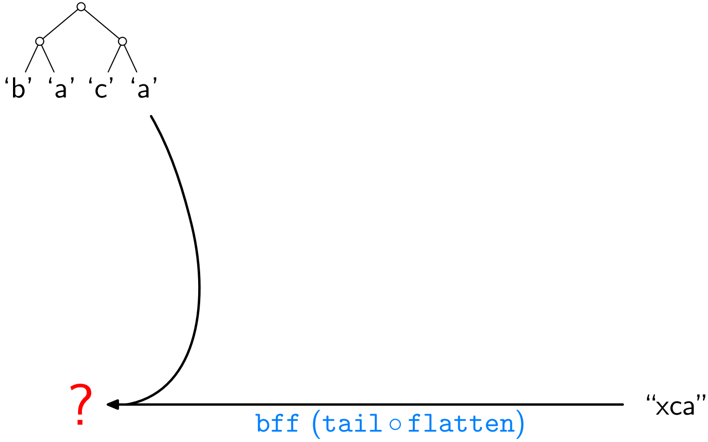
Given an arbitrary list s of length $n+1$,

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

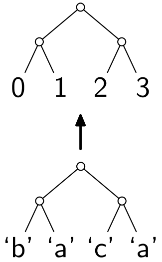
The Resulting Bidirectionalization Scheme by Example



The Resulting Bidirectionalization Scheme by Example

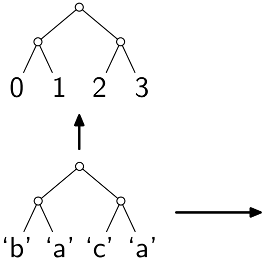


The Resulting Bidirectionalization Scheme by Example



"xca"

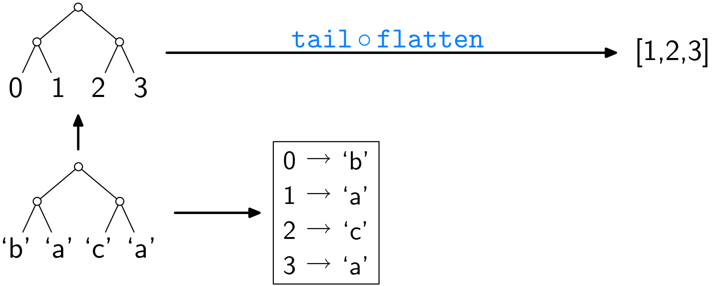
The Resulting Bidirectionalization Scheme by Example



0	$\rightarrow$ 'b'
1	$\rightarrow$ 'a'
2	$\rightarrow$ 'c'
3	$\rightarrow$ 'a'

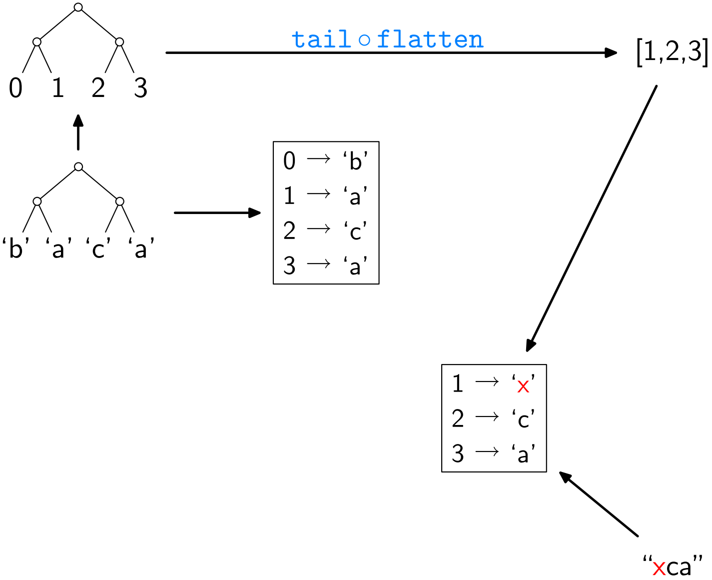
"xca"

The Resulting Bidirectionalization Scheme by Example

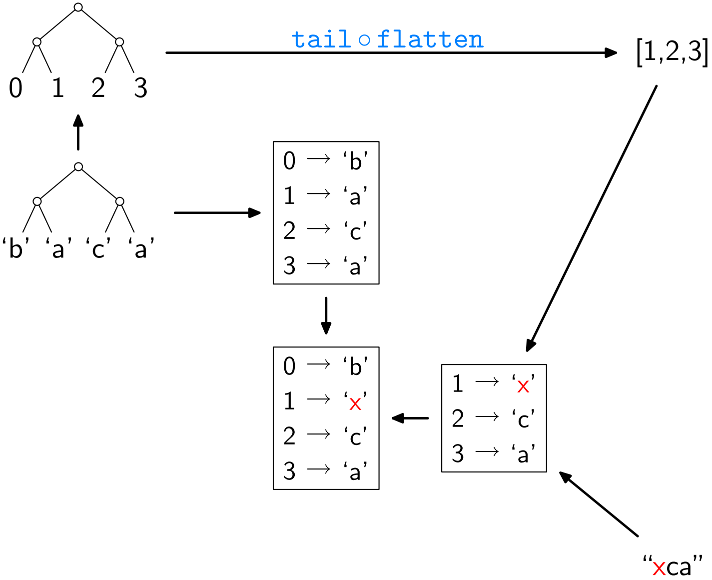


“xca”

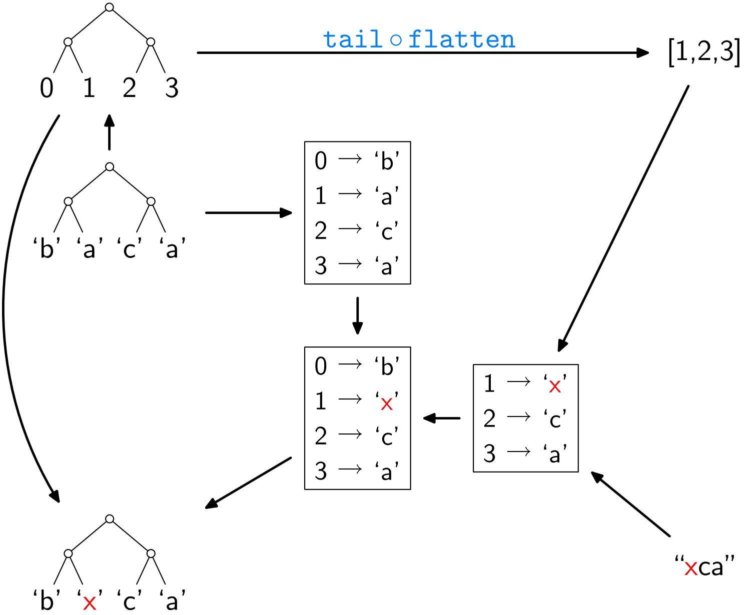
The Resulting Bidirectionalization Scheme by Example



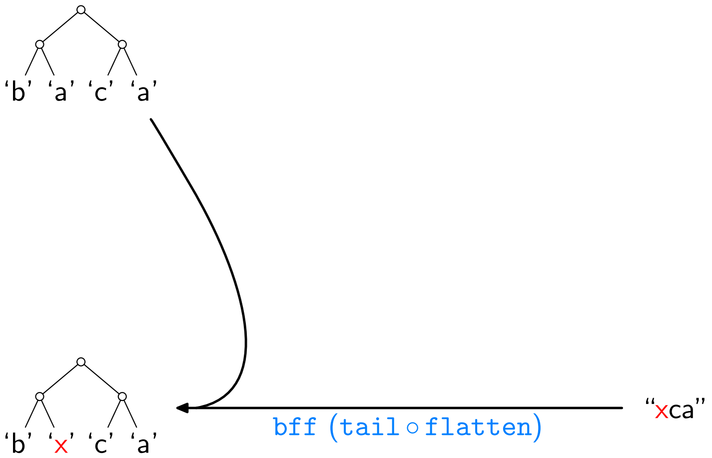
The Resulting Bidirectionalization Scheme by Example



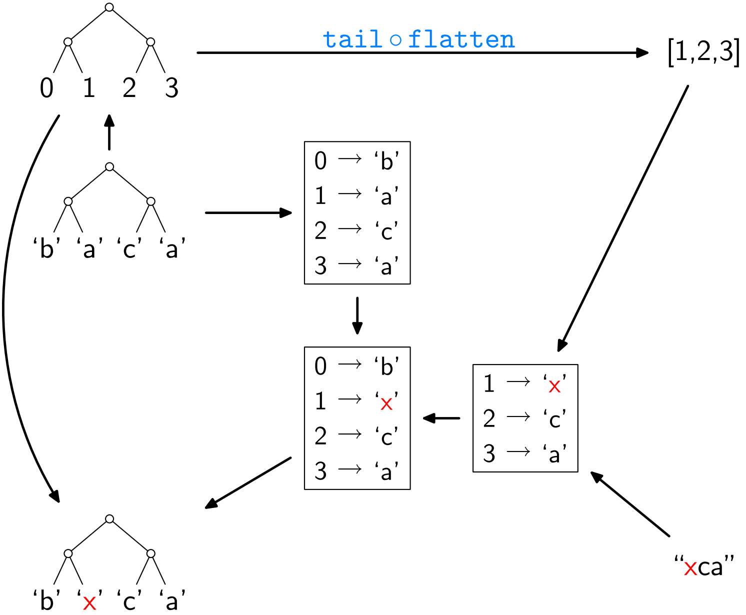
The Resulting Bidirectionalization Scheme by Example



The Resulting Bidirectionalization Scheme by Example



The Resulting Bidirectionalization Scheme by Example



The Implementation (here: lists only, inefficient version)

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

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

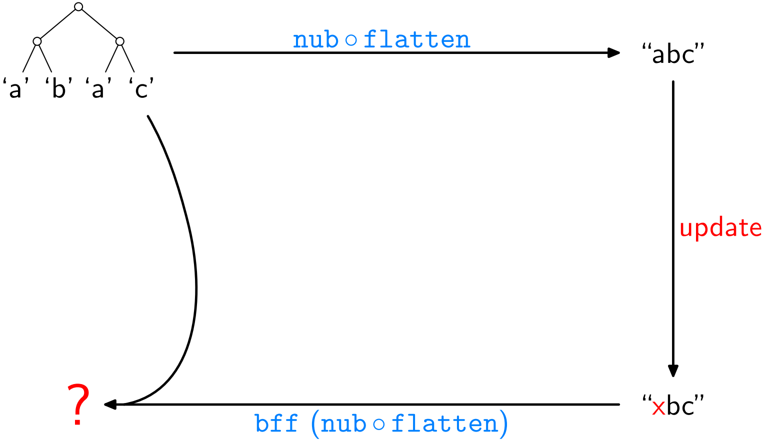
The Implementation (here: lists only, inefficient version)

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

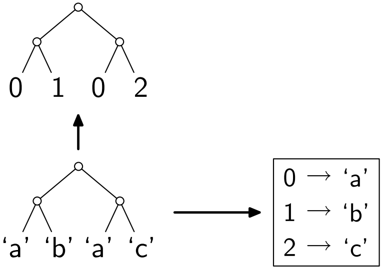
$$\begin{aligned}
&\texttt{assoc}\ [\,]\ \ \ \ \ \ \ \ \ [\,] \ \ \ \ \ \ \ \ \ = [\,] \\
&\texttt{assoc}\ (i : is)\ (b : bs) = \textbf{let}\ m = \texttt{assoc}\ is\ bs \\
&\qquad\qquad\qquad\qquad\quad\ \textbf{in}\ \ \textbf{case}\ \texttt{lookup}\ i\ m\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\quad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

- actual code only slightly more elaborate
- online: `http://linux.tcs.inf.tu-dresden.de/~bff`

Another Interesting Example

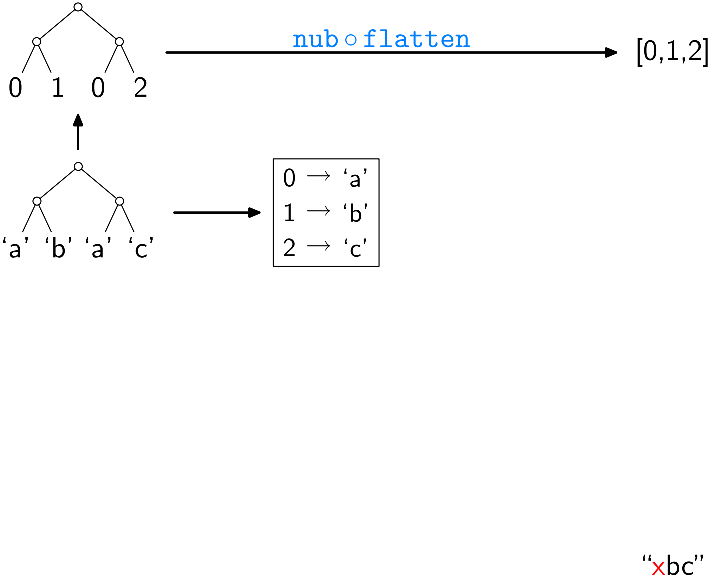


Another Interesting Example

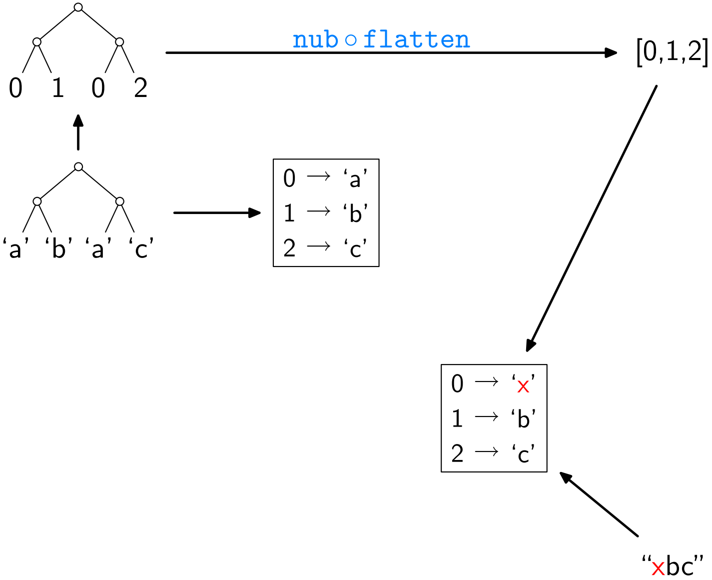


"xbc"

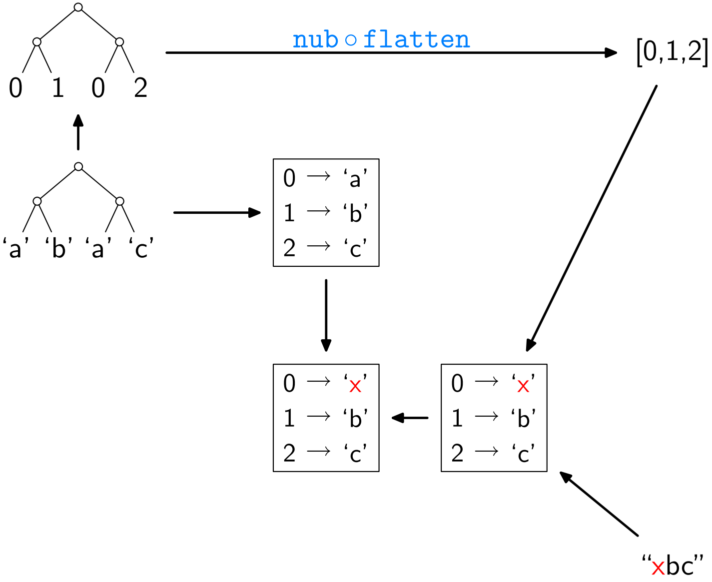
Another Interesting Example



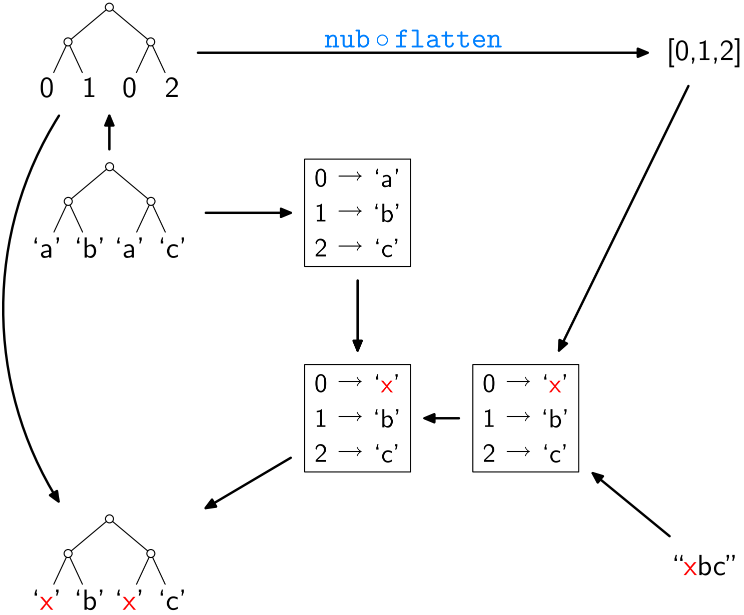
Another Interesting Example



Another Interesting Example



Another Interesting Example



Summary and Outlook

Types:

- constrain the behavior of programs

Summary and Outlook

Types:

- constrain the behavior of programs
- thus lead to interesting theorems about programs

Summary and Outlook

Types:

- constrain the behavior of programs
- thus lead to interesting theorems about programs
- combine well with algebraic techniques, equational reasoning

Summary and Outlook

Types:

- constrain the behavior of programs
- thus lead to interesting theorems about programs
- combine well with algebraic techniques, equational reasoning

On the programming language side:

- push towards full programming languages

Summary and Outlook

Types:

- constrain the behavior of programs
- thus lead to interesting theorems about programs
- combine well with algebraic techniques, equational reasoning

On the programming language side:

- push towards full programming languages
- strife for more expressive type systems

Summary and Outlook

Types:

- constrain the behavior of programs
- thus lead to interesting theorems about programs
- combine well with algebraic techniques, equational reasoning

On the programming language side:

- push towards full programming languages
- strife for more expressive type systems

On the practical side:

- efficiency-improving program transformations

Summary and Outlook

Types:

- constrain the behavior of programs
- thus lead to interesting theorems about programs
- combine well with algebraic techniques, equational reasoning

On the programming language side:

- push towards full programming languages
- strife for more expressive type systems

On the practical side:

- efficiency-improving program transformations
- applications in specific domains

References I

F. Bancilhon and N. Spyratos.
Update semantics of relational views.
ACM Transactions on Database Systems, 6(3):557–575, 1981.

J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt.
Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem.
ACM Transactions on Programming Languages and Systems, 29(3):17, 2007.

P. Hudak, R.J.M. Hughes, S.L. Peyton Jones, and P. Wadler.
A history of Haskell: Being lazy with class.
In *History of Programming Languages, Proceedings*, pages 12-1–12-55. ACM Press, 2007.

References II

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi.
Bidirectionalization transformation based on automatic derivation of view complement functions.
In *International Conference on Functional Programming, Proceedings*, pages 47–58. ACM Press, 2007.

J. Voigtländer.
Bidirectionalization for free!
In *Principles of Programming Languages, Proceedings*, pages 165–176. ACM Press, 2009.

P. Wadler.
Theorems for free!
In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.