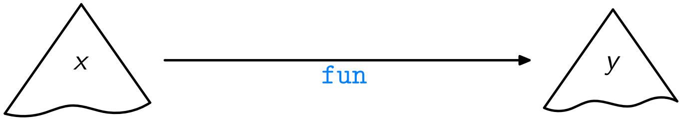
Lightweight Program Inversion

Janis Voigtländer

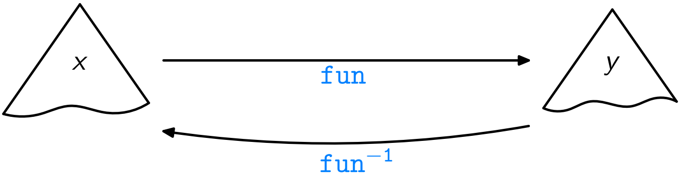
University of Bonn

Dutch HUG Day 2010

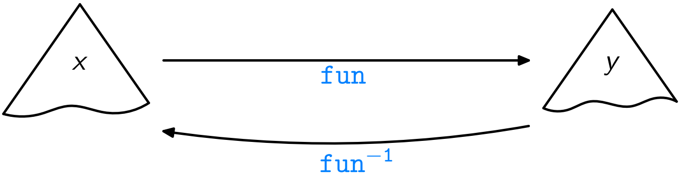
The Problem Statement



The Problem Statement



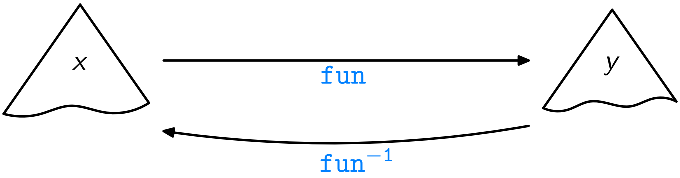
The Problem Statement



Want (given `fun`, find $\texttt{fun}^{-1}$ such that):

- for every x, $\texttt{fun}^{-1}\ (\texttt{fun}\ x) = x$ (1)

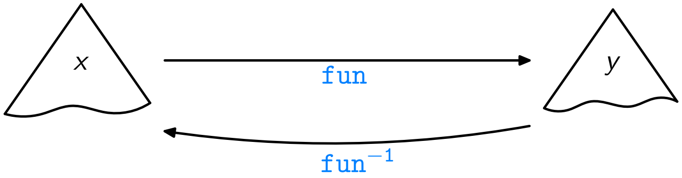
The Problem Statement



Want (given fun, find $\texttt{fun}^{-1}$ such that):

- for every x, $\texttt{fun}^{-1}\,(\texttt{fun}\;x) = x$ (1)
- for every y, $\texttt{fun}\,(\texttt{fun}^{-1}\;y) = y$ (2)

The Problem Statement



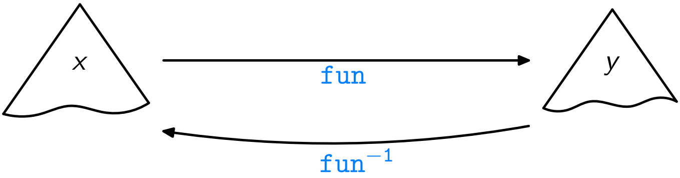
Want (given `fun`, find $\texttt{fun}^{-1}$ such that):

- for every x, $\texttt{fun}^{-1}\,(\texttt{fun}\ x) = x$ (1)
- for every y, $\texttt{fun}\,(\texttt{fun}^{-1}\ y) = y$ (2)

Problems:

- (1) not possible if `fun` is not injective

The Problem Statement



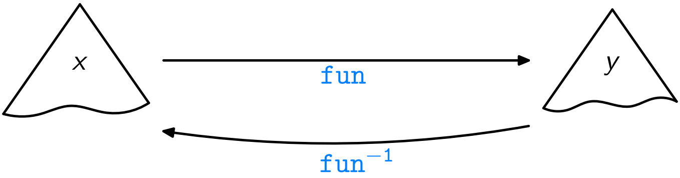
Want (given fun, find fun^{-1} such that):

- for every x, $\texttt{fun}^{-1}\,(\texttt{fun}\ x) = x$ (1)
- for every y, $\texttt{fun}\,(\texttt{fun}^{-1}\ y) = y$ (2)

Problems:

- (1) not possible if fun is not injective
- (2) not possible if fun is not surjective

The Problem Statement



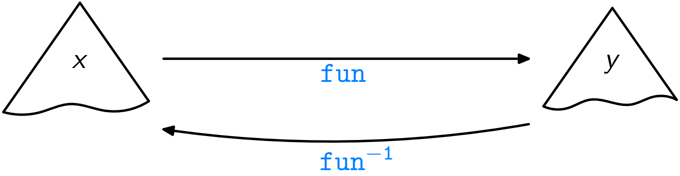
Want (given fun, find fun^{-1} such that):

- for every x, $\texttt{fun}^{-1}\,(\texttt{fun}\ x) = x$ (1)
- for every y, $\texttt{fun}\,(\texttt{fun}^{-1}\ y) = y$ (2)

Problems:

- (1) not possible if fun is not injective
- (2) not possible if fun is not surjective
- even with appropriate side conditions,

The Problem Statement



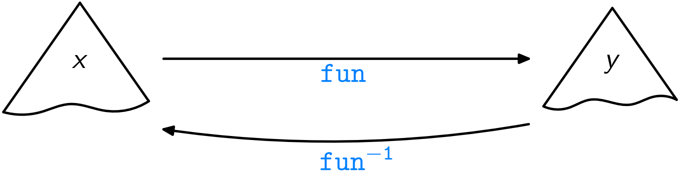
Want (given `fun`, find $\texttt{fun}^{-1}$ such that):

- for every x, $\texttt{fun}^{-1}\,(\texttt{fun}\ x) = x$ (1)
- for every y, $\texttt{fun}\,(\texttt{fun}^{-1}\ y) = y$ (2)

Problems:

- (1) not possible if `fun` is not injective
- (2) not possible if `fun` is not surjective
- even with appropriate side conditions,
  and/or requiring only one of (1) and (2),

The Problem Statement



Want (given fun, find $\texttt{fun}^{-1}$ such that):

- for every x, $\texttt{fun}^{-1}\,(\texttt{fun}\ x) = x$ (1)
- for every y, $\texttt{fun}\,(\texttt{fun}^{-1}\ y) = y$ (2)

Problems:

- (1) not possible if fun is not injective
- (2) not possible if fun is not surjective
- even with appropriate side conditions,
  and/or requiring only one of (1) and (2),
  in general $\texttt{fun}^{-1}$ not effectively computable from fun

A More Realistic Problem

Making the problem simpler:

- restrict to a particular type: `fun` $:: [a] \to [a]$

A More Realistic Problem

Making the problem simpler:

- restrict to a particular type: $\texttt{fun} :: [a] \rightarrow [a]$
- require only (2): $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$

A More Realistic Problem

Making the problem simpler:

- restrict to a particular type: $\texttt{fun} :: [a] \rightarrow [a]$
- require only (2): $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$
- allow $\texttt{fun}^{-1}$ to be partial,

A More Realistic Problem

Making the problem simpler:

- restrict to a particular type: $\texttt{fun} :: [a] \to [a]$
- require only (2): $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$
- allow $\texttt{fun}^{-1}$ to be partial, and
  demand (2) only for *ys* for which $\texttt{fun}^{-1}$ is defined,

A More Realistic Problem

Making the problem simpler:

- restrict to a particular type: $\texttt{fun} :: [a] \to [a]$
- require only (2): $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$
- allow $\texttt{fun}^{-1}$ to be partial, and
  demand (2) only for *ys* for which $\texttt{fun}^{-1}$ is defined, but
  demand $\texttt{fun}^{-1}$ to be defined for all images of $\texttt{fun}$

A More Realistic Problem

Making the problem simpler:

- restrict to a particular type: $\texttt{fun} :: [a] \to [a]$
- require only (2): $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$
- allow $\texttt{fun}^{-1}$ to be partial, and
  demand (2) only for *ys* for which $\texttt{fun}^{-1}$ is defined, but
  demand $\texttt{fun}^{-1}$ to be defined for all images of $\texttt{fun}$

... and simultaneously more complicated:

- prevent any inspection of the definition of $\texttt{fun}$!

A More Realistic Problem

Making the problem simpler:

- restrict to a particular type: `fun` $:: [a] \rightarrow [a]$
- require only (2): `fun` $($`fun`$^{-1}$ $ys) = ys$
- allow `fun`$^{-1}$ to be partial, and
  demand (2) only for ys for which `fun`$^{-1}$ is defined, but
  demand `fun`$^{-1}$ to be defined for all images of `fun`

. . . and simultaneously more complicated:

- prevent any inspection of the definition of `fun`!

Of course, *some* access to `fun` must be possible:

- can ask for `fun`'s outputs for specific inputs

A More Realistic Problem

Making the problem simpler:

- restrict to a particular type: `fun` :: $[a] \to [a]$
- require only (2): `fun` $($`fun`$^{-1}$ $ys) = ys$
- allow `fun`$^{-1}$ to be partial, and
  demand (2) only for ys for which `fun`$^{-1}$ is defined, but
  demand `fun`$^{-1}$ to be defined for all images of `fun`

. . . and simultaneously more complicated:

- prevent any inspection of the definition of `fun` !

Of course, *some* access to `fun` must be possible:

- can ask for `fun`'s outputs for specific inputs
- anytime, dynamically

Phrased as a Game

Rules:

- Alice implements a function `fun` $:: [a] \to [a]$

Phrased as a Game

Rules:

- Alice implements a function `fun` $:: [a] \to [a]$
- Bob has to try to implement a function $\texttt{fun}^{-1} :: [a] \to [a]$

Phrased as a Game

Rules:

- Alice implements a function `fun` $:: [a] \rightarrow [a]$
- Bob has to try to implement a function $\texttt{fun}^{-1} :: [a] \rightarrow [a]$
  - can call function `fun` on specific inputs

Phrased as a Game

Rules:

- Alice implements a function `fun` $:: [a] \to [a]$
- Bob has to try to implement a function $\texttt{fun}^{-1} :: [a] \to [a]$
  - can call function `fun` on specific inputs
  - can even use it inside the definition of $\texttt{fun}^{-1}$, "linking" against it

Phrased as a Game

Rules:

- Alice implements a function `fun` $:: [a] \to [a]$
- Bob has to try to implement a function $\texttt{fun}^{-1} :: [a] \to [a]$
  - can call function `fun` on specific inputs
  - can even use it inside the definition of $\texttt{fun}^{-1}$, "linking" against it
  - but no dirty tricks, disassembling, ...

Phrased as a Game

Rules:

- Alice implements a function `fun` $:: [a] \to [a]$
- Bob has to try to implement a function $\texttt{fun}^{-1} :: [a] \to [a]$
  - can call function `fun` on specific inputs
  - can even use it inside the definition of $\texttt{fun}^{-1}$,
    "linking" against it
  - but no dirty tricks, disassembling, ...
  - has to guarantee that for every *xs*, $\texttt{fun}^{-1}\ (\texttt{fun}\ xs)$ is defined,

Phrased as a Game

Rules:

- Alice implements a function `fun` $:: [a] \to [a]$
- Bob has to try to implement a function $\texttt{fun}^{-1} :: [a] \to [a]$
  - can call function `fun` on specific inputs
  - can even use it inside the definition of $\texttt{fun}^{-1}$, "linking" against it
  - but no dirty tricks, disassembling, ...
  - has to guarantee that for every xs, $\texttt{fun}^{-1}\ (\texttt{fun}\ xs)$ is defined, and that whenever $\texttt{fun}^{-1}\ ys$ is defined, $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$

Phrased as a Game

Rules:

- Alice implements a function $\texttt{fun} :: [a] \rightarrow [a]$
- Bob has to try to implement a function $\texttt{fun}^{-1} :: [a] \rightarrow [a]$
  - can call function $\texttt{fun}$ on specific inputs
  - can even use it inside the definition of $\texttt{fun}^{-1}$, "linking" against it
  - but no dirty tricks, disassembling, ...
  - has to guarantee that for every xs, $\texttt{fun}^{-1}\ (\texttt{fun}\ xs)$ is defined, and that whenever $\texttt{fun}^{-1}\ ys$ is defined, $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$

Who will win?

Phrased as a Game

Rules:

- Alice implements a function `fun` $:: [a] \to [a]$
- Bob has to try to implement a function $\texttt{fun}^{-1} :: [a] \to [a]$
  - can call function `fun` on specific inputs
  - can even use it inside the definition of $\texttt{fun}^{-1}$, "linking" against it
  - but no dirty tricks, disassembling, ...
  - has to guarantee that for every xs, $\texttt{fun}^{-1}\ (\texttt{fun}\ xs)$ is defined, and that whenever $\texttt{fun}^{-1}\ ys$ is defined, $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$

Who will win?

Observation: If Bob has a winning strategy, he must be able to do without asking Alice for specific inputs up front, instead only provide a single definition of $\texttt{fun}^{-1}$ that works for all `fun`.

Decomposing the Problem

A possible approach for Bob:

- He is only obliged to define $\texttt{fun}^{-1}$ for images of `fun`.

Decomposing the Problem

A possible approach for Bob:

- He is only obliged to define `fun`$^{-1}$ for images of `fun`.
- Given a *ys*, he could try to find *any* *xs* with `fun` $xs = ys$.

Decomposing the Problem

A possible approach for Bob:

- He is only obliged to define $\texttt{fun}^{-1}$ for images of `fun`.
- Given a *ys*, he could try to find *any xs* with `fun` $xs = ys$.
- It would be a good start if at least the length of *xs* could be guessed somehow.

Decomposing the Problem

A possible approach for Bob:

- He is only obliged to define $\texttt{fun}^{-1}$ for images of `fun`.
- Given a *ys*, he could try to find *any* *xs* with `fun` $xs = ys$.
- It would be a good start if at least the length of *xs* could be guessed somehow. So let's assume there is some function

  $$\texttt{lengthInv} :: [a] \rightarrow \mathsf{Int}$$

  such that for every *ys* in the image of `fun`, `lengthInv` *ys* gives an *n* such that there is an *xs* of length *n* with `fun` $xs = ys$.

Decomposing the Problem

A possible approach for Bob:

- He is only obliged to define $\texttt{fun}^{-1}$ for images of `fun`.
- Given a *ys*, he could try to find *any* *xs* with $\texttt{fun}\ xs = ys$.
- It would be a good start if at least the length of *xs* could be guessed somehow. So let's assume there is some function

$$\texttt{lengthInv} :: [a] \to \mathsf{Int}$$

  such that for every *ys* in the image of `fun`, $\texttt{lengthInv}\ ys$ gives an *n* such that there is an *xs* of length *n* with $\texttt{fun}\ xs = ys$.
- Then, an appropriate *xs* could be identified via:

$$\begin{aligned}
&\texttt{fun}^{-1} :: [a] \to [a] \\
&\texttt{fun}^{-1}\ ys = \textbf{let}\ n = \texttt{lengthInv}\ ys \\
&\qquad\qquad\qquad\ \ t\ = [1\,..\,n] \\
&\qquad\qquad\qquad\ \ h = \texttt{zip}\ (\texttt{fun}\ t)\ ys \\
&\qquad\qquad\quad \textbf{in}\ \ \texttt{map}\ (\texttt{fromJust} \circ \texttt{flip}\ \texttt{lookup}\ h)\ t
\end{aligned}$$

Decomposing the Problem Further

How to implement `lengthInv`:

- If Bob had a function

  $$\texttt{checkLength} :: \mathsf{Int} \rightarrow [a] \rightarrow \mathsf{Bool}$$

  such that `checkLength` $n\ ys$ checks whether there is an xs of length n with `fun` $xs = ys$, then he could write:

  $$\begin{aligned}&\texttt{lengthInv} :: [a] \rightarrow \mathsf{Int}\\ &\texttt{lengthInv}\ ys = \texttt{head}\ [n \mid n \leftarrow [0\,..], \texttt{checkLength}\ n\ ys]\end{aligned}$$

Decomposing the Problem Further

How to implement `lengthInv`:

- If Bob had a function

  $$\texttt{checkLength} :: \mathsf{Int} \rightarrow [a] \rightarrow \mathsf{Bool}$$

  such that `checkLength` $n\ ys$ checks whether there is an xs of length n with `fun` $xs = ys$, then he could write:

  $$\texttt{lengthInv} :: [a] \rightarrow \mathsf{Int}$$
  $$\texttt{lengthInv}\ ys = \texttt{head}\ [n \mid n \leftarrow [0\,..],\texttt{checkLength}\ n\ ys]$$

- It would be tempting to implement `checkLength` as follows:

  $$\texttt{checkLength} :: \mathsf{Int} \rightarrow [a] \rightarrow \mathsf{Bool}$$
  $$\texttt{checkLength}\ n\ ys = \texttt{length}\ (\texttt{fun}\ [1\,..\,n]) == \texttt{length}\ ys$$

Decomposing the Problem Further

How to implement `lengthInv`:

- If Bob had a function

$$\texttt{checkLength} :: \mathsf{Int} \to [a] \to \mathsf{Bool}$$

  such that $\texttt{checkLength}\ n\ ys$ checks whether there is an xs of length n with $\texttt{fun}\ xs = ys$, then he could write:

$$\texttt{lengthInv} :: [a] \to \mathsf{Int}$$
$$\texttt{lengthInv}\ ys = \texttt{head}\ [n \mid n \leftarrow [0\,..], \texttt{checkLength}\ n\ ys]$$

- It would be tempting to implement `checkLength` as follows:

$$\texttt{checkLength} :: \mathsf{Int} \to [a] \to \mathsf{Bool}$$
$$\texttt{checkLength}\ n\ ys = \texttt{length}\ (\texttt{fun}\ [1\,..\,n]) == \texttt{length}\ ys$$

- But that would be wrong!

Decomposing the Problem Further

How to implement `lengthInv`:

- If Bob had a function

  $$\texttt{checkLength} :: \mathsf{Int} \to [a] \to \mathsf{Bool}$$

  such that `checkLength` $n\ ys$ checks whether there is an xs of length n with `fun` $xs = ys$, then he could write:

  $$\begin{aligned}&\texttt{lengthInv} :: [a] \to \mathsf{Int}\\ &\texttt{lengthInv}\ ys = \texttt{head}\ [n \mid n \leftarrow [0\,..], \texttt{checkLength}\ n\ ys]\end{aligned}$$

- It would be tempting to implement `checkLength` as follows:

  $$\begin{aligned}&\texttt{checkLength} :: \mathsf{Int} \to [a] \to \mathsf{Bool}\\ &\texttt{checkLength}\ n\ ys = \texttt{length}\ (\texttt{fun}\ [1\,..\,n]) == \texttt{length}\ ys\end{aligned}$$

- But that would be wrong! Why?

A Problematic Case

Let

$$\texttt{fun} :: [a] \to [a]$$
$$\texttt{fun}\ xs = xs +\!\!+ xs$$

A Problematic Case

Let

$$\texttt{fun} :: [a] \rightarrow [a]$$
$$\texttt{fun}\ xs = xs +\!\!+ xs$$

Then, with the current definitions of `checkLength`, `lengthInv`, and $\texttt{fun}^{-1}$:

$$\texttt{fun}^{-1}\ \texttt{"abcdef"} = \texttt{"abc"}$$

A Problematic Case

Let

$$\texttt{fun} :: [a] \to [a]$$
$$\texttt{fun}\ xs = xs +\!\!+\ xs$$

Then, with the current definitions of `checkLength`, `lengthInv`, and $\texttt{fun}^{-1}$:

$$\texttt{fun}^{-1}\ \texttt{"abcdef"} = \texttt{"abc"}$$

but:

$$\texttt{fun}\ \texttt{"abc"} = \texttt{"abcabc"}$$

A Problematic Case

Let

$$\texttt{fun} :: [a] \to [a]$$
$$\texttt{fun}\ xs = xs +\!\!+\ xs$$

Then, with the current definitions of `checkLength`, `lengthInv`, and $\texttt{fun}^{-1}$:

$$\texttt{fun}^{-1}\ \texttt{"abcdef"} = \texttt{"abc"}$$

but:

$$\texttt{fun}\ \texttt{"abc"} = \texttt{"abcabc"}$$

which violates the requirement that whenever $\texttt{fun}^{-1}\ ys$ is defined, $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$.

A Problematic Case

Let

$$\texttt{fun} :: [a] \to [a]$$
$$\texttt{fun}\ xs = xs +\!\!+ xs$$

Then, with the current definitions of `checkLength`, `lengthInv`, and $\texttt{fun}^{-1}$:

$$\texttt{fun}^{-1}\ \texttt{"abcdef"} = \texttt{"abc"}$$

but:

$$\texttt{fun}\ \texttt{"abc"} = \texttt{"abcabc"}$$

which violates the requirement that whenever $\texttt{fun}^{-1}\ ys$ is defined, $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$.

Intuition: In `checkLength`, should check that not only has $\texttt{fun}\ [1\mathinner{\ldotp\ldotp} n]$ the same length as ys, but also if two elements at positions i and j in $\texttt{fun}\ [1\mathinner{\ldotp\ldotp} n]$ are equal, then for the corresponding positions in ys, $y_i = y_j$.

Some (Necessary) Restrictions

- Need to assume that elements in *ys* can be compared. Then:

$$
\begin{array}{l}
\texttt{checkLength} :: \mathsf{Eq}\ a \Rightarrow \mathsf{Int} \to [a] \to \mathsf{Bool} \\
\texttt{checkLength}\ n\ ys = \\
\quad \mathbf{let}\ t' = \texttt{fun}\ [1\,..\,n] \\
\quad \mathbf{in}\ \ \texttt{length}\ t' == \texttt{length}\ ys\ \wedge \\
\qquad\ \ \texttt{and}\ [(i == j) \leqslant (y == z) \mid \mathbf{let}\ zs = \texttt{zip}\ t'\ ys, \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (i, y) \leftarrow zs, (j, z) \leftarrow zs]
\end{array}
$$

Some (Necessary) Restrictions

- Need to assume that elements in *ys* can be compared. Then:

$$
\begin{aligned}
&\texttt{checkLength} :: \mathsf{Eq}\ a \Rightarrow \mathsf{Int} \rightarrow [a] \rightarrow \mathsf{Bool} \\
&\texttt{checkLength}\ n\ ys = \\
&\quad \mathbf{let}\ t' = \texttt{fun}\ [1\,..\,n] \\
&\quad \mathbf{in}\ \ \texttt{length}\ t' == \texttt{length}\ ys\ \wedge \\
&\qquad\ \ \texttt{and}\ [(i == j) \leqslant (y == z) \mid \mathbf{let}\ zs = \texttt{zip}\ t'\ ys, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (i, y) \leftarrow zs, (j, z) \leftarrow zs]
\end{aligned}
$$

- $\texttt{lengthInv} :: \mathsf{Eq}\ a \Rightarrow [a] \rightarrow \mathsf{Int}$, $\texttt{fun}^{-1} :: \mathsf{Eq}\ a \Rightarrow [a] \rightarrow [a]$

Some (Necessary) Restrictions

- Need to assume that elements in *ys* can be compared. Then:

$$
\begin{aligned}
&\texttt{checkLength} :: \mathsf{Eq}\ a \Rightarrow \mathsf{Int} \rightarrow [a] \rightarrow \mathsf{Bool} \\
&\texttt{checkLength}\ n\ ys = \\
&\quad \textbf{let}\ t' = \texttt{fun}\ [1\,..\,n] \\
&\quad \textbf{in}\ \ \texttt{length}\ t' == \texttt{length}\ ys\ \wedge \\
&\qquad\ \ \texttt{and}\ [(i == j) \leqslant (y == z) \mid \textbf{let}\ zs = \texttt{zip}\ t'\ ys, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad (i, y) \leftarrow zs, (j, z) \leftarrow zs]
\end{aligned}
$$

- $\texttt{lengthInv} :: \mathsf{Eq}\ a \Rightarrow [a] \rightarrow \mathsf{Int}$, $\texttt{fun}^{-1} :: \mathsf{Eq}\ a \Rightarrow [a] \rightarrow [a]$

- Instead of $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$, get only $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) == ys$ (whenever $\texttt{fun}^{-1}\ ys$ is defined).

Some (Necessary) Restrictions

- Need to assume that elements in ys can be compared. Then:

$$
\begin{array}{l}
\texttt{checkLength} :: \mathsf{Eq}\ a \Rightarrow \mathsf{Int} \rightarrow [a] \rightarrow \mathsf{Bool} \\
\texttt{checkLength}\ n\ ys = \\
\quad \mathbf{let}\ t' = \texttt{fun}\ [1\mathinner{..}n] \\
\quad \mathbf{in}\ \ \texttt{length}\ t' == \texttt{length}\ ys \wedge \\
\qquad\ \ \texttt{and}\ [(i == j) \leqslant (y == z) \mid \mathbf{let}\ zs = \texttt{zip}\ t'\ ys, \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad (i, y) \leftarrow zs, (j, z) \leftarrow zs]
\end{array}
$$

- $\texttt{lengthInv} :: \mathsf{Eq}\ a \Rightarrow [a] \rightarrow \mathsf{Int}$, $\texttt{fun}^{-1} :: \mathsf{Eq}\ a \Rightarrow [a] \rightarrow [a]$

- Instead of $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) = ys$, get only $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) == ys$ (whenever $\texttt{fun}^{-1}\ ys$ is defined).

- Need to assume that instances of Eq used are reflexive, transitive, and symmetric.

Some Examples

- Let $\texttt{fun} = \texttt{reverse}$.
  Then, $\texttt{fun}^{-1}$ "abcdef" = "fedcba".

Some Examples

- Let $\texttt{fun} = \texttt{reverse}$.
  Then, $\texttt{fun}^{-1}$ $\texttt{"abcdef"} = \texttt{"fedcba"}$.

- Let $\texttt{fun} = \texttt{take}\ 5$.
  Then, $\texttt{fun}^{-1}$ $\texttt{"abcde"} = \texttt{"abcde"}$ and $\texttt{fun}^{-1}$ $\texttt{"abcdef"} = \bot$.

Some Examples

- Let $\texttt{fun} = \texttt{reverse}$.
  Then, $\texttt{fun}^{-1}$ $\texttt{"abcdef"} = \texttt{"fedcba"}$.
- Let $\texttt{fun} = \texttt{take}\ 5$.
  Then, $\texttt{fun}^{-1}$ $\texttt{"abcde"} = \texttt{"abcde"}$ and $\texttt{fun}^{-1}$ $\texttt{"abcdef"} = \bot$.
- Let $\texttt{fun} = \texttt{drop}\ 5$.
  Then, $\texttt{fun}^{-1}$ $\texttt{"abcde"} = \bot : \bot : \bot : \bot : \bot : \texttt{"abcde"}$.

Some Examples

- Let $\texttt{fun} = \texttt{reverse}$.
  Then, $\texttt{fun}^{-1}$ $\texttt{"abcdef"} = \texttt{"fedcba"}$.

- Let $\texttt{fun} = \texttt{take}\ 5$.
  Then, $\texttt{fun}^{-1}$ $\texttt{"abcde"} = \texttt{"abcde"}$ and $\texttt{fun}^{-1}$ $\texttt{"abcdef"} = \bot$.

- Let $\texttt{fun} = \texttt{drop}\ 5$.
  Then, $\texttt{fun}^{-1}$ $\texttt{"abcde"} = \bot : \bot : \bot : \bot : \bot : \texttt{"abcde"}$.

- Let $\texttt{fun} = \lambda xs \rightarrow xs +\!\!+\ xs$.
  Then, $\texttt{fun}^{-1}$ $\texttt{"abcabc"} = \texttt{"abc"}$ and $\texttt{fun}^{-1}$ $\texttt{"abcdef"} = \bot$.

Some Examples

- Let $\texttt{fun} = \texttt{reverse}$.
  Then, $\texttt{fun}^{-1}$ "abcdef" = "fedcba".

- Let $\texttt{fun} = \texttt{take}\ 5$.
  Then, $\texttt{fun}^{-1}$ "abcde" = "abcde" and $\texttt{fun}^{-1}$ "abcdef" = $\bot$.

- Let $\texttt{fun} = \texttt{drop}\ 5$.
  Then, $\texttt{fun}^{-1}$ "abcde" = $\bot : \bot : \bot : \bot : \bot :$ "abcde".

- Let $\texttt{fun} = \lambda xs \rightarrow xs +\!\!+\ xs$.
  Then, $\texttt{fun}^{-1}$ "abcabc" = "abc" and $\texttt{fun}^{-1}$ "abcdef" = $\bot$.

- ...

Exercises

1. Give an example of `fun` and *ys* where $\texttt{fun}^{-1}\ ys$ is defined, but $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) \neq ys$.

Exercises

1. Give an example of $\texttt{fun}$ and ys where $\texttt{fun}^{-1}\ ys$ is defined, but $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) \neq ys$.

2. Give an example of $\texttt{fun}$ and xs where for $ys = \texttt{fun}\ xs$, $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) \neq ys$.

Exercises

1. Give an example of `fun` and *ys* where $\texttt{fun}^{-1}\ ys$ is defined, but $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) \neq ys$.
2. Give an example of `fun` and *xs* where for $ys = \texttt{fun}\ xs$, $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) \neq ys$.
3. Give an example of `fun` and *xs* where $\texttt{fun}^{-1}\ (\texttt{fun}\ xs) \neq xs$.

Exercises

1. Give an example of `fun` and *ys* where $\texttt{fun}^{-1}\ ys$ is defined, but $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) \neq ys$.

2. Give an example of `fun` and *xs* where for $ys = \texttt{fun}\ xs$, $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) \neq ys$.

3. Give an example of `fun` and *xs* where $\texttt{fun}^{-1}\ (\texttt{fun}\ xs) \neq xs$.

4. Actually prove that for every *xs*, $\texttt{fun}^{-1}\ (\texttt{fun}\ xs)$ is defined, and that whenever $\texttt{fun}^{-1}\ ys$ is defined, $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) == ys$.

Exercises

1. Give an example of `fun` and *ys* where $\texttt{fun}^{-1}\ ys$ is defined, but $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) \neq ys$.

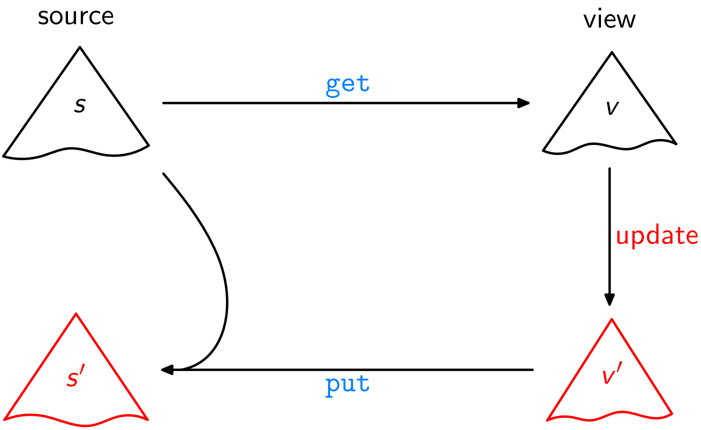
2. Give an example of `fun` and *xs* where for $ys = \texttt{fun}\ xs$, $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) \neq ys$.

3. Give an example of `fun` and *xs* where $\texttt{fun}^{-1}\ (\texttt{fun}\ xs) \neq xs$.

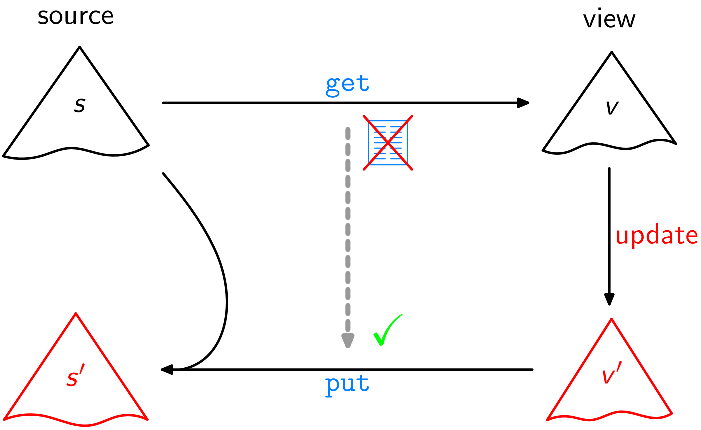
4. Actually prove that for every *xs*, $\texttt{fun}^{-1}\ (\texttt{fun}\ xs)$ is defined, and that whenever $\texttt{fun}^{-1}\ ys$ is defined, $\texttt{fun}\ (\texttt{fun}^{-1}\ ys) == ys$.

5. Generalize from lists to other data types.

Bidirectional Transformation



Bidirectional Transformation



Bidirectionalization for Free!

[V., POPL'09]

HCAR

Don't forget to submit entries about your projects to the upcoming Haskell Communities and Activities Report!