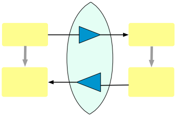
Linguistic Foundations
for
Bidirectional Transformations

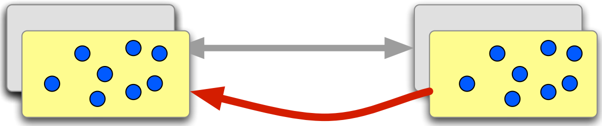
Benjamin Pierce
University of Pennsylvania

Invited tutorial, PODS 2012

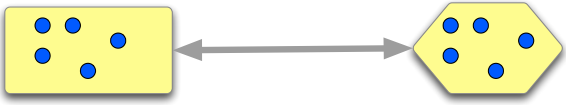


- The world is full of replicated data

- The world is full of replicated data
- ... that is subject to updates



- The world is full of replicated data
- ... that is subject to updates
- ... that then need to be propagated to other replicas



- Things get more interesting when different replicas can have different schemas
  - ... or even different data models

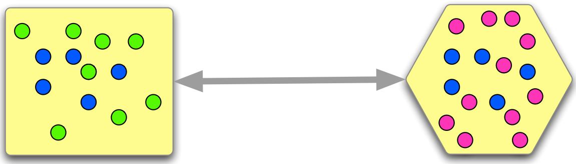
- … and even more interesting when some of the information in one structure is not reflected in the other

- ... and yet more interesting when some information in each structure is not reflected in the other

Connections to PODS

- View update problem [Dayal, Bernstein '82, Bancilhon, Spryatos '81, Gottlob, Paolini, Zicari '88, etc., etc., etc.]
- Inverse mappings in Data Exchange
- Federated databases
- ...

… And Beyond

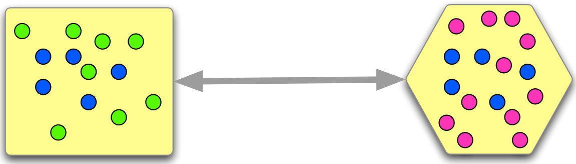


... And Beyond

an in-memory heap structure

its marshalled disk representation

... And Beyond



an in-memory heap structure — its marshalled disk representation

a text pane in a GUI — the scroll bar for this text pane

... And Beyond

an in-memory heap structure — its marshalled disk representation

a text pane in a GUI — the scroll bar for this text pane

a relational schema — an ER diagram of the same schema

… And Beyond

an in-memory heap structure	its marshalled disk representation
a text pane in a GUI	the scroll bar for this text pane
a relational schema	an ER diagram of the same schema
a requirements model of a software system	an implementation model of the same system

… And Beyond

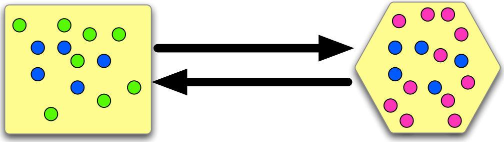
an in-memory heap structure	its marshalled disk representation
a text pane in a GUI	the scroll bar for this text pane
a relational schema	an ER diagram of the same schema
a requirements model of a software system	an implementation model of the same system

So the question is…

What is a good way to program bidirectional transformations?

“Easy” Approach

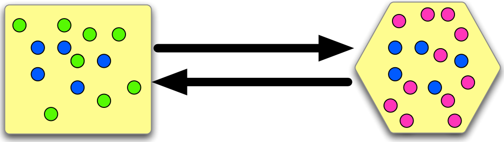
Standard way of building a bidirectional transformation: Write two functions, each propagating updates in one direction.



+ Uses standard technology
+ Works fine for simple transformations

“Easy” Approach

Standard way of building a bidirectional transformation: Write two functions, each propagating updates in one direction.

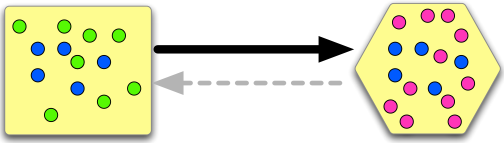


+ Uses standard technology
+ Works fine for simple transformations

- Scales badly
- Maintenance nightmare

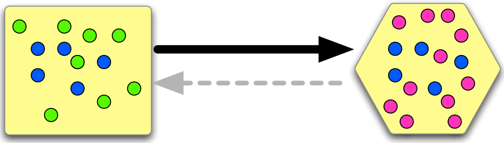
A Better Idea

Write one function in some familiar programming language and infer the other



A Better Idea

Write one function in some familiar programming language and infer the other

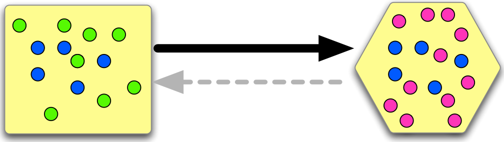


But:

- In general, many possible translations of a given update
- Choosing a "best" one is hard

A Better Idea

Write one function in some familiar programming language and infer the other

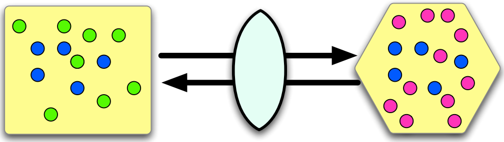


But:

- In general, many possible translations of a given update
- Choosing a "best" one is hard
  - indeed, even NP-hard! [Buneman/Khanna/Tan 2002]
  - ...and can involve fragile heuristics

An Even Better Idea

Design a new language in which every program is naturally bidirectional!



Many instances of this idea...

- ad hoc libraries and tools (marshallers/unmarshallers, parsers/prettyprinters, ...)
- bidirectional variants of standard languages (XQuery, UnQL, relational algebra, ...)
- domain-specific bidirectional languages
  - "coupled grammars" (XSugar, biXid, TGGs, ...)
  - combinator-based (lenses) (Focal, Boomerang, Augeas, ...)

Key Questions

Semantics:

- What is the fundamental shape of a bidirectional transformation?
  - Are the inputs and outputs states or edits?
  - Are extra context inputs needed to restore missing information?
- What laws do we expect it to satisfy?
- What properties follow from the laws? What sorts of generic constructions are possible? (E.g., composition)

Syntax:

- What is a good bidirectional notation for transformations over a specific data model?

Key Questions

Semantics: interesting

- What is the fundamental shape of a bidirectional transformation?
  - Are the inputs and outputs states or edits?
  - Are extra context inputs needed to restore missing information?
- What laws do we expect it to satisfy?
- What properties follow from the laws? What sorts of generic constructions are possible? (E.g., composition)

Syntax:

- What is a good bidirectional notation for transformations over a specific data model?

Key Questions

Semantics: interesting

- What is the fundamental shape of a bidirectional transformation?
  - Are the inputs and outputs states or edits?
  - Are extra context inputs needed to restore missing information?
- What laws do we expect it to satisfy?
- What properties follow from the laws? What sorts of generic constructions are possible? (E.g., composition)

Syntax: challenging!

- What is a good bidirectional notation for transformations over a specific data model?
  - Tension between expressiveness and well-behavedness

Outline

- Extended example
  - Bijective transformations over strings
- The non-bijective case
- Alignment and edits
- Other data models
- Challenges

Leaving math in the background...
Feel free to ask questions...

Extended Example: Bijective Transformations on Strings

Example

```
Schubert [1797-1828]
Monteverdi [1567-1643]
```

```
1797-1828: Schubert
1567-1643: Monteverdi
```

Example

```
Schubert [1797-1828]
Monteverdi [1567-1643]
```


```
1797-1828: Schubert
1567-1643: Monteverdi
```


```
1797-1828: Schubert
1567-1643: Monteverdi
1810-1856: Schumann
```

Example

```
Schubert [1797-1828]
Monteverdi [1567-1643]
```

```
1797-1828: Schubert
1567-1643: Monteverdi
```

```
Schubert [1797-1828]
Monteverdi [1567-1643]
Schumann [1810-1856]
```

```
1797-1828: Schubert
1567-1643: Monteverdi
1810-1856: Schumann
```

Example

```
Schubert [1797-1828]
Monteverdi [1567-1643]
```

```
1797-1828: Schubert
1567-1643: Monteverdi
```

```
Schubert [1797-1828]
Monteverdi [1567-1643]
Schumann [1810-1856]
```

```
1797-1828: Schubert
1567-1643: Monteverdi
1810-1856: Schumann
```

```
((copy ALPHA) ~ (" ["<=>"" . copy DATE . "]"<=>": ")
 . copy "\n") *
```

Overview

Semantics:

- Fundamental shape? a pair of total functions
  - inputs and outputs states or edits? states
  - extra context inputs needed? no
- What laws do we expect it to satisfy? bijectivity
- What sorts of generic constructions are possible? e.g., composition

Syntax:

- Data model: strings
- Schemas: regular expressions
- Primitives based on finite-state transducers

Data Model: Strings

Not a lot to say.

Schemas: Regular Expressions

Standard notations:

$$
\begin{array}{rcll}
R & ::= & \text{“}\textit{string}\text{''} & \text{singleton} \\
 & & R_1 \cdot R_2 & \text{concatenation} \\
 & & R* & \text{repetition} \\
 & & R_1 \mid R_2 & \text{union}
\end{array}
$$

Each regular expression denotes a set of strings.

Examples

Schema with composers first:

```
(ALPHA . " [" . DATE . "]\n")*
```

Schema with dates first:

```
(DATE . ": " . ALPHA . "]\n")*
```

where...

```
ALPHA = ("a"|...|"z"|"A"|...|"Z")*
DATE = ("0"|...|"9"|"-")*
```

String Transducers

Starting point for our bijective language:

- simple form of (unidirectional!) string transducers
- reminiscent of finite-state transducers

String Transducers

f	$::=$	$\mathit{copy}\ R$	recognize R and copy it
		$\mathit{del}\ R$	recognize R and emit nothing
		$r \Rightarrow s$	recognize (singleton) r and emit s
		$f_1 \cdot f_2$	concatenation
		$f_1 \mid f_2$	union
		$f*$	repetition
		$f_1 \sim f_2$	swapping concatenation

String Transducers

f	$::=$	$copy\ R$	recognize R and copy it
		$del\ R$	recognize R and emit nothing
		$r \Rightarrow s$	recognize (singleton) r and emit s
		$f_1 \cdot f_2$	concatenation
		$f_1 \mid f_2$	union
		$f*$	repetition
		$f_1 \sim f_2$	swapping concatenation

$$\texttt{Schubert} \xrightarrow{\quad copy\ \texttt{ALPHA}\quad} \texttt{Schubert}$$

String Transducers

f	$::=$	*copy* R	recognize R and copy it
		del R	recognize R and emit nothing
		$r \Rightarrow s$	recognize (singleton) r and emit s
		$f_1 \cdot f_2$	concatenation
		$f_1 \mid f_2$	union
		$f*$	repetition
		$f_1 \sim f_2$	swapping concatenation

$$\boxed{\texttt{Schubert}} \xrightarrow{\textit{del}\ \texttt{ALPHA}} \boxed{}$$

String Transducers

f	$::=$	*copy* R	recognize R and copy it
		del R	recognize R and emit nothing
		$r \Rightarrow s$	recognize (singleton) r and emit s
		$f_1 \cdot f_2$	concatenation
		$f_1 \mid f_2$	union
		$f*$	repetition
		$f_1 \sim f_2$	swapping concatenation

$$\boxed{\texttt{foo}} \xrightarrow{\texttt{"foo"} \Rightarrow \texttt{"bar"}} \boxed{\texttt{bar}}$$

String Transducers

f	$::=$	$copy\ R$	recognize R and copy it
		$del\ R$	recognize R and emit nothing
		$r \Rightarrow s$	recognize (singleton) r and emit s
		$f_1 \cdot f_2$	concatenation
		$f_1 \mid f_2$	union
		$f*$	repetition
		$f_1 \sim f_2$	swapping concatenation

$$\boxed{\texttt{fooXYZ}} \xrightarrow{(\texttt{"foo"} \Rightarrow \texttt{"bar"}) \cdot (copy\ \texttt{ALPHA})} \boxed{\texttt{barXYZ}}$$

String Transducers

f	$::=$	$copy\ R$	recognize R and copy it
		$del\ R$	recognize R and emit nothing
		$r \Rightarrow s$	recognize (singleton) r and emit s
		$f_1 \cdot f_2$	concatenation
		$f_1 \mid f_2$	union
		$f*$	repetition
		$f_1 \sim f_2$	swapping concatenation

$$\texttt{A} \xrightarrow{(\texttt{"A"} \Rightarrow \texttt{"B"}) \mid (\texttt{"B"} \Rightarrow \texttt{"A"})} \texttt{B}$$

String Transducers

$$
\begin{array}{lll}
f & ::= & \textit{copy } R & \text{recognize } R \text{ and copy it} \\
 & & \textit{del } R & \text{recognize } R \text{ and emit nothing} \\
 & & r \Rightarrow s & \text{recognize (singleton) } r \text{ and emit } s \\
 & & f_1 \cdot f_2 & \text{concatenation} \\
 & & f_1 \mid f_2 & \text{union} \\
 & & f* & \text{repetition} \\
 & & f_1 \sim f_2 & \text{swapping concatenation}
\end{array}
$$

$$\boxed{\texttt{AAABA}} \xrightarrow{(\texttt{"A"} \Rightarrow \texttt{"B"} \mid \texttt{"B"} \Rightarrow \texttt{"A"})*} \boxed{\texttt{BBBAB}}$$

String Transducers

f	$::=$	$copy\ R$	recognize R and copy it
		$del\ R$	recognize R and emit nothing
		$r \Rightarrow s$	recognize (singleton) r and emit s
		$f_1 \cdot f_2$	concatenation
		$f_1 \mid f_2$	union
		$f*$	repetition
		$f_1 \sim f_2$	swapping concatenation

$$\boxed{\texttt{fooXYZ}} \xrightarrow{(\texttt{"foo"} \Rightarrow \texttt{"bar"}) \sim (copy\ \texttt{ALPHA})} \boxed{\texttt{XYZbar}}$$

Next step

Bidirectionalize!

Issue #1: Deletion

Problem:

- Deletion operator throws away information

$$\boxed{\texttt{Schubert}} \xrightarrow{\textit{del}\ \texttt{ALPHA}} \boxed{}$$

- Cannot be part of a bijective transformation

Solution:

- Throw it away (it will come back later)

N.b.: “Singleton deletion” is bijective

$$\boxed{\texttt{foo}} \xrightarrow{\texttt{"foo"} \Rightarrow \texttt{""}} \boxed{}$$

Issue #2: Union

Problem:

- in general, union of two string transducers defines a relation, not a function

$$\boxed{\texttt{A}} \xrightarrow{(\texttt{"A"} \Rightarrow \texttt{"B"}) \mid (\texttt{"A"} \Rightarrow \texttt{"C"})} \boxed{\{\texttt{B},\texttt{C}\}}$$

Issue #2: Union

Problems:

- in general, union of two string transducers defines a relation, not a function
- indeed, even when the union of two string transducers is a function, it may not be injective

  $$\boxed{\texttt{A}} \xrightarrow{(\texttt{"A"} \Rightarrow \texttt{"B"}) \mid (\texttt{"C"} \Rightarrow \texttt{"B"})} \boxed{\texttt{B}}$$

  ... in which case it can't be part of a bijective transformation

Issue #2: Union

Problems:

- in general, union of two string transducers defines a relation, not a function
- indeed, even when the union of two string transducers is a function, it may not be injective

Solution:

- Use schemas to ensure that domains and ranges are disjoint

$$\frac{l_1 \in R_1 \rightleftharpoons S_1 \qquad l_2 \in R_2 \rightleftharpoons S_2 \qquad R_1 \cap R_2 = \emptyset \qquad S_1 \cap S_2 = \emptyset}{l_1 \mid l_2 \in R_1 \mid R_2 \rightleftharpoons S_1 \mid S_2}$$

Issue #3: Concatenation

Problem:

- In general, concatenation is not deterministic

$$\boxed{\texttt{ABCD}} \xrightarrow{(\textit{copy}\ \texttt{ALPHA}) \cdot (\textit{del}\ \texttt{ALPHA})} \boxed{\texttt{???}}$$

Issue #3: Concatenation

Problem:

- In general, concatenation is not deterministic

$$\boxed{\texttt{ABCD}} \xrightarrow{\quad (\mathit{copy}\ \texttt{ALPHA}) \cdot (\mathit{del}\ \texttt{ALPHA}) \quad} \boxed{\texttt{???}}$$

Solution:

- Schemas again...

$$\frac{l_1 \in R_1 \rightleftharpoons S_1 \qquad l_2 \in R_2 \rightleftharpoons S_2 \qquad R_1 \cdot^{!} R_2 \qquad S_1 \cdot^{!} S_2}{l_1 \cdot l_2 \in R_1 \cdot R_2 \rightleftharpoons S_1 \cdot S_2}$$

i.e., every element of $R_1 \cdot R_2$ can be formed in exactly one way by concatenating an element of R_1 and an element of R_2 (and similarly for S_1 and S_2)

Formally...

A bijective lens l between a set R and a set S, written

$$l \ \in \ R \rightleftharpoons S$$

comprises two functions

$$\begin{array}{lcl} l^{\rightarrow} & \in & R \to S \\ l^{\leftarrow} & \in & S \to R \end{array}$$

where $l^{\rightarrow}$ and $l^{\leftarrow}$ are inverses:

$$\begin{array}{lcl} l^{\leftarrow}\ (l^{\rightarrow}\ r) & = & r \\ l^{\rightarrow}\ (l^{\leftarrow}\ s) & = & s \end{array}$$

Typing Rules

$$\mathit{copy}\ R \ \in\ R \rightleftharpoons R$$

$$\text{“}s\text{”} \Leftrightarrow \text{“}t\text{”} \ \in\ \text{“}s\text{”} \rightleftharpoons \text{“}t\text{”}$$

$$\frac{l_1 \ \in\ R_1 \rightleftharpoons S_1 \qquad l_2 \ \in\ R_2 \rightleftharpoons S_2 \qquad R_1 \cdot^! R_2 \qquad S_1 \cdot^! S_2}{l_1 \cdot l_2 \ \in\ R_1 \cdot R_2 \rightleftharpoons S_1 \cdot S_2}$$

$$\frac{l \ \in\ R \rightleftharpoons S \qquad R^{*!} \qquad S^{*!}}{l^* \ \in\ S^* \rightleftharpoons R^*}$$

$$\frac{l_1 \ \in\ R_1 \rightleftharpoons S_1 \quad l_2 \ \in\ R_2 \rightleftharpoons S_2 \qquad R_1 \cap R_2 = \emptyset \quad S_1 \cap S_2 = \emptyset}{l_1 \mid l_2 \ \in\ R_1 \mid R_2 \rightleftharpoons S_1 \mid S_2}$$

$$\frac{l_1 \ \in\ R_1 \rightleftharpoons S_1 \qquad l_2 \ \in\ R_2 \rightleftharpoons S_2 \qquad R_1 \cdot^! R_2 \qquad S_1 \cdot^! S_2}{l_1 \Leftrightarrow l_2 \ \in\ R_1 \cdot R_2 \rightleftharpoons S_2 \cdot S_1}$$

Type Soundness

Theorem: If $l \in R \rightleftharpoons S$ according to the typing rules, then l is a bijective lens between R and S.

Example (Recap)

```
Schubert [1797-1828]
Monteverdi [1567-1643]
```

```
1797-1828: Schubert
1567-1643: Monteverdi
```

```
((copy ALPHA) ~ (" ["<=>"" . copy DATE . "]"<=>": ")
 . copy "\n") *
```

Example (Recap)

```
Schubert [1797-1828]
Monteverdi [1567-1643]
```


```
1797-1828: Schubert
1567-1643: Monteverdi
```


```
1797-1828: Schubert
1567-1643: Monteverdi
1810-1856: Schumann
```

```
((copy ALPHA) ~ (" ["<=>"" . copy DATE . "]"<=>": ")
 . copy "\n") *
```

Example (Recap)

```
Schubert [1797-1828]
Monteverdi [1567-1643]
```

```
1797-1828: Schubert
1567-1643: Monteverdi
```

```
Schubert [1797-1828]
Monteverdi [1567-1643]
Schumann [1810-1856]
```

```
1797-1828: Schubert
1567-1643: Monteverdi
1810-1856: Schumann
```

```
((copy ALPHA) ~ (" ["<=>"" . copy DATE . "]"<=>": ")
 . copy "\n") *
```

Recap

We've defined a small but complete bidirectional language...

- standard data model
- standard schema language (with a couple of unusual operations)
- bidirectional combinators
  - each atomic form denotes a bijective pair of functions: a "bijective lens" (*copy*, $\Leftrightarrow$)
  - each combining form maps lenses to lenses (concat, union, kleene star, swapping concat)
- some "expected" combinators don't make sense (delete)
- schemas used to restrict to well-behaved cases
  - type soundness theorem

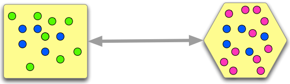
The Non-Bijective Case

Asymmetric vs. Symmetric

- Asymmetric

  - One direction loses information
  - Example: A database and a materialized view
  - Classic view update problem

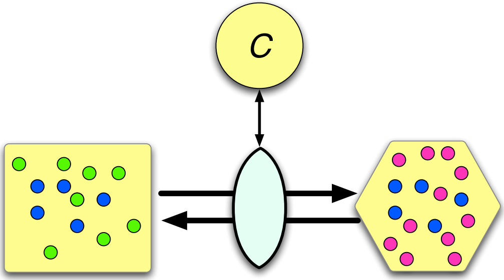
- Symmetric



  - Both directions lose information
  - Example: Two models of different aspects of a software system

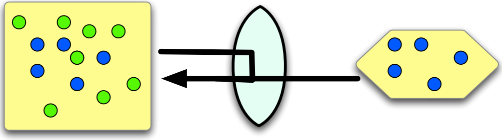
Complements

If information is lost in one direction, it must be restored from someplace in the other direction!

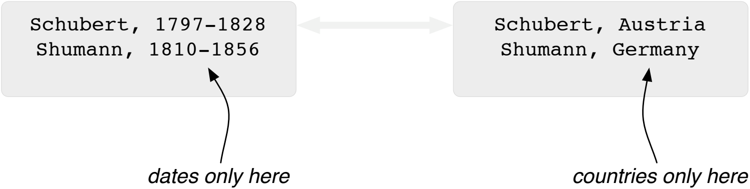


Complements

In the asymmetric case, the larger structure can also serve as the complement

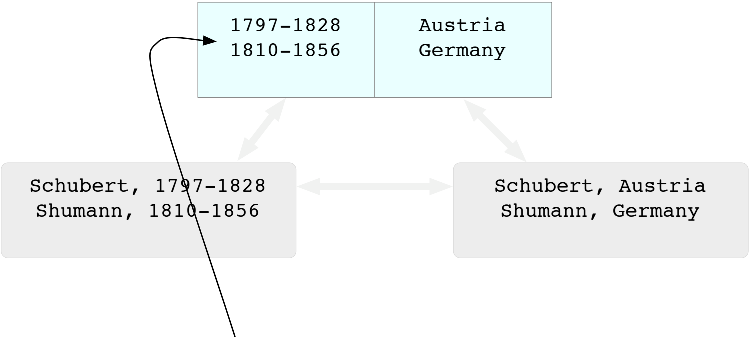


Intuition



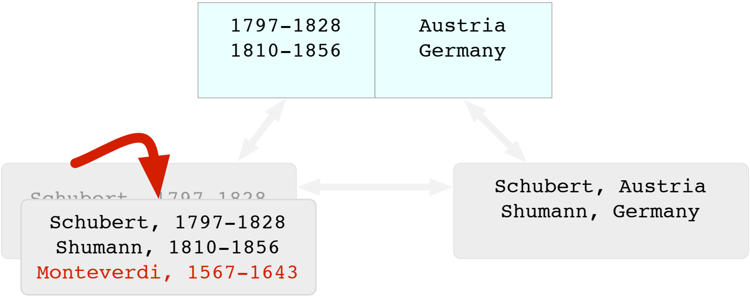
Let's consider the symmetric case...

Intuition

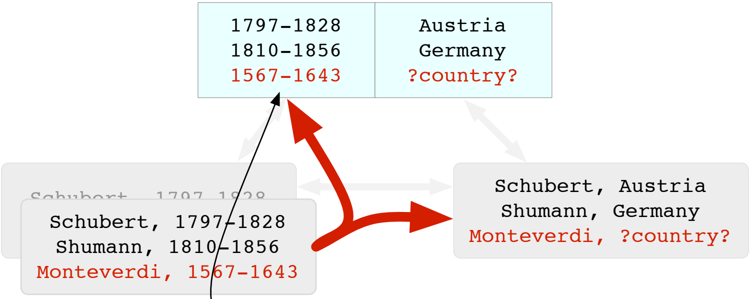


add an extra structure (the "complement") that stores the "private information" from both sides

Intuition

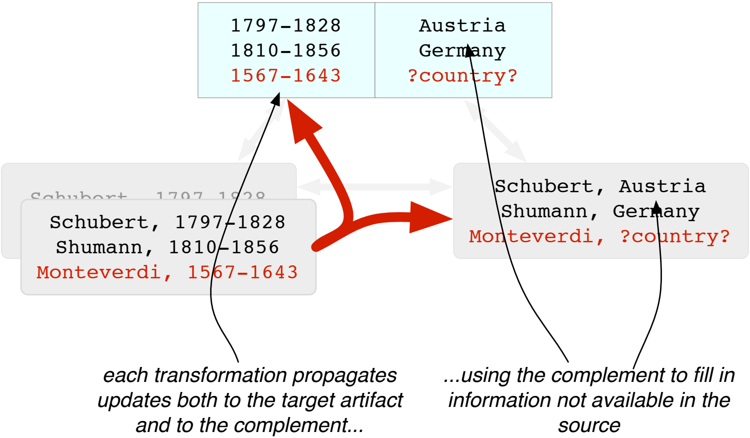

1797–1828
1810–1856
Austria
Germany
Schubert, 1797–1828
Shumann, 1810–1856
Monteverdi, 1567–1643
Schubert, Austria
Shumann, Germany

Intuition

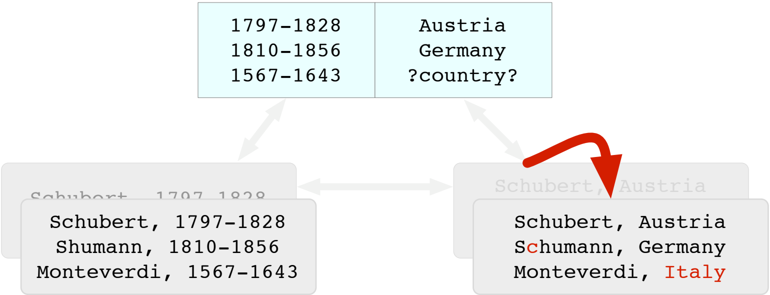


each transformation propagates updates both to the target artifact and to the complement...

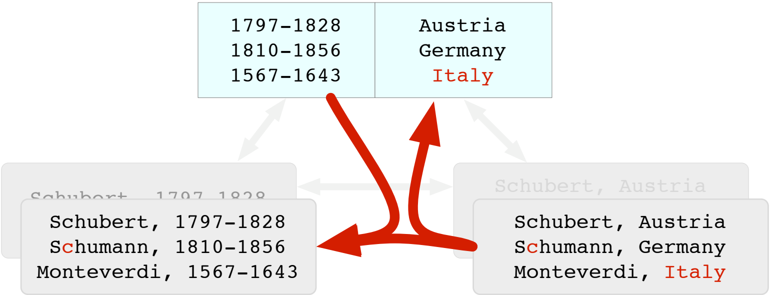
Intuition


1797–1828
1810–1856
1567–1643
Austria
Germany
?country?
Schubert, 1797–1828
Shumann, 1810–1856
Monteverdi, 1567–1643
Schubert, Austria
Shumann, Germany
Monteverdi, ?country?
each transformation propagates updates both to the target artifact and to the complement...
...using the complement to fill in information not available in the source

Intuition


1797–1828
1810–1856
1567–1643
Austria
Germany
?country?
Schubert, 1797–1828
Shumann, 1810–1856
Monteverdi, 1567–1643
Schubert, Austria
Schubert, Austria
Schumann, Germany
Monteverdi, Italy

Intuition


1797–1828
1810–1856
1567–1643
Austria
Germany
Italy
Schubert, Austria
Schubert, 1797–1828
Schumann, 1810–1856
Monteverdi, 1567–1643
Schubert, Austria
Schumann, Germany
Monteverdi, Italy

Formally...

A symmetric lens l between a set R and a set S comprises a complement set C with a distinguished element *missing*, together with two functions

$$\begin{array}{lll} l^{\rightarrow} & \in & R \times C \rightarrow S \times C \\ l^{\leftarrow} & \in & S \times C \rightarrow R \times C \end{array}$$

where

$$\frac{l^{\rightarrow}(r, c) = (s', c')}{l^{\leftarrow}(s', c') = (r, c')}$$

$$\frac{l^{\leftarrow}(s, c) = (r', c')}{l^{\rightarrow}(r', c') = (s, c')}$$

N.b.: Other laws can be considered — e.g., a “Put-Put” law

Formally...

A symmetric lens l between a set R and a set S comprises a complement set C with a distinguished element *missing*, together with two functions

$$
\begin{array}{lcl}
l^{\rightarrow} & \in & R \times C \rightarrow S \times C \\
l^{\leftarrow} & \in & S \times C \rightarrow R \times C
\end{array}
$$

where

propagating a null update changes nothing

$$\frac{l^{\rightarrow}(r, c) = (s', c')}{l^{\leftarrow}(s', c') = (r, c')}$$

ditto

$$\frac{l^{\leftarrow}(s, c) = (r', c')}{l^{\rightarrow}(r', c') = (s, c')}$$

N.b.: Other laws can be considered — e.g., a “Put-Put” law

Deletion

- We've dropped the requirement that transformations be injective
- ... so we can have the *del* operator back again!
- Indeed, since we're in a symmetric setting, we can have two delete operators
  - $del^{\rightarrow}$ ("delete when going left to right")
  - $del^{\leftarrow}$ ("delete when going right to left")

$$\frac{d \in R}{del^{\rightarrow}\ R\ d \ \in \ R \rightleftharpoons ""}$$

$$\frac{d \in S}{del^{\leftarrow}\ S\ d \ \in \ ""S \rightleftharpoons}$$

Example (Recap)

```
composers =
    ( copy ALPHA .
      ", " <=> ", " .
      // delete dates in -> direction
      del-> ALPHA "?dates?" .
      // delete country in <- direction
      del<- ALPHA "?country?" .
      "\n" <=> "\n" )*
```

Key Issue: Totality

The assumption that $I^{\rightarrow}$ and $I^{\leftarrow}$ are total functions is quite strong:

- It means that our update translators must be able to handle *any update*, with respect to *any complement*

Can we relax this restriction?

Key Issue: Totality

The assumption that $l^{\rightarrow}$ and $l^{\leftarrow}$ are total functions is quite strong:

- It means that our update translators must be able to handle *any update*, with respect to *any complement*

Can we relax this restriction?

Depends on the application!

Totality

- If our lenses are being used in an on-line setting, where edits are propagated immediately, totality can be dropped
  - ... but dropping it leads to theories where it's hard to predict which edits will succeed
- In an off-line setting, arbitrary changes can accumulate before we get a chance to propagate them
  - ... so totality is critical

Duplication

A particular case in point:

- It would be useful to have a *duplicate* combinator that (in one direction) makes two copies of its input on its output
- But how should the other direction behave?
  - If one of the copies is changed and the other is not, what should happen??

This operator is the source of a deep split among different bidirectional transformation frameworks

- Some choose *duplicate*
- Some choose totality
- (Can't have both)

Alignment and Edits

Alignment

Depending on the data model, representing the alignment of structures before and after updates can raise additional challenges...

- unordered data (sets, tables, etc.): straightforward (align by keys)
- ordered data (lists, documents): more problematic

Alignment

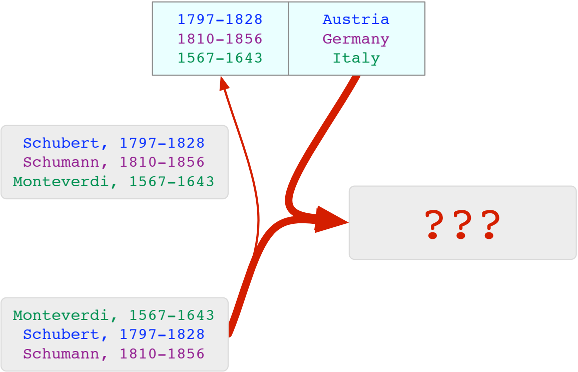
1797-1828 1810-1856 1567-1643	Austria Germany Italy

```
Schubert, 1797-1828
Schumann, 1810-1856
Monteverdi, 1567-1643
```

```
Schubert, Austria
Schumann, Germany
Monteverdi, Italy
```

```
Monteverdi, 1567-1643
Schubert, 1797-1828
Schumann, 1810-1856
```

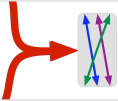
Alignment


1797-1828
1810-1856
1567-1643
Austria
Germany
Italy
Schubert, 1797-1828
Schumann, 1810-1856
Monteverdi, 1567-1643
???
Monteverdi, 1567-1643
Schubert, 1797-1828
Schumann, 1810-1856

Alignment

1797-1828	Austria
1810-1856	Germany
1567-1643	Italy

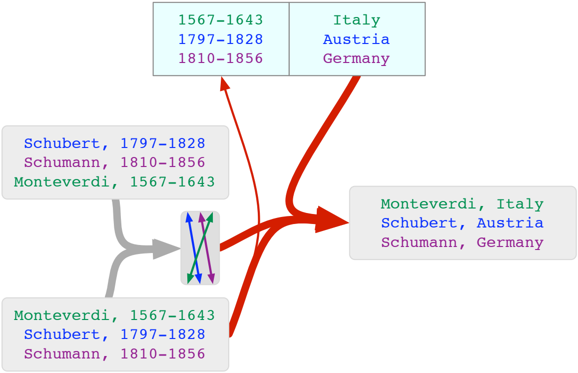
```
Schubert, 1797-1828
Schumann, 1810-1856
Monteverdi, 1567-1643
```



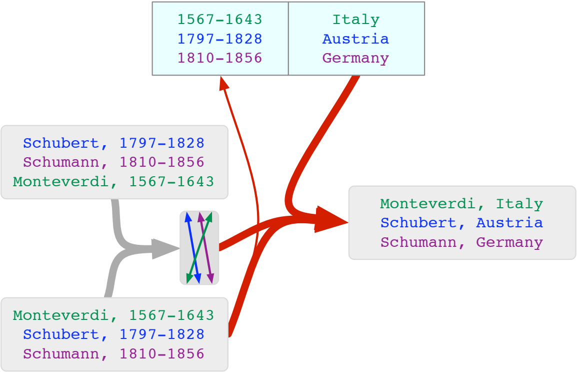
```
Monteverdi, 1567-1643
Schubert, 1797-1828
Schumann, 1810-1856
```

```
Schubert, Austria
Schumann, Germany
Monteverdi, Italy
```

Alignment


1567–1643
1797–1828
1810–1856
Italy
Austria
Germany
Schubert, 1797–1828
Schumann, 1810–1856
Monteverdi, 1567–1643
Monteverdi, 1567–1643
Schubert, 1797–1828
Schumann, 1810–1856
Monteverdi, Italy
Schubert, Austria
Schumann, Germany

Alignment



How to represent this alignment information?

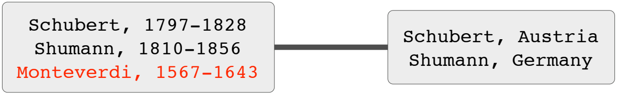
1. As a separate structure, passed to the lens along with the current state ("matching lenses")
2. As an edit on the current state

Edit Lenses

(a) initial replicas

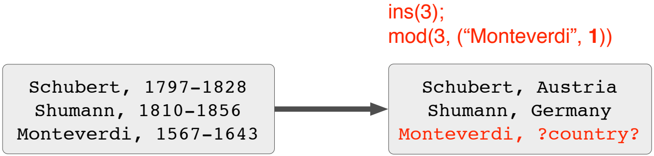
Edit Lenses

ins(3);
mod(3, (“Monteverdi”, “1567-1643”))



(b) a new composer is added to one replica

Edit Lenses



(c) the lens adds the new composer to the other replica

Edit Lenses

mod(3, (**1**, "Italy"));
mod(2, ("Schumann", **1**))

```
Schubert, 1797-1828
Shumann, 1810-1856
Monteverdi, 1567-1643
```

```
Schubert, Austria
Schumann, Germany
Monteverdi, Italy
```

(d) the curator makes some corrections

Edit Lenses

1;
mod(2, ("Schumann", **1**))

```
Schubert, 1797-1828
Schumann, 1810-1856
Monteverdi, 1567-1643
```

```
Schubert, Austria
Schumann, Germany
Monteverdi, Italy
```

(e) the lens transports a small edit

Edit Lenses

del(3); ins(1);
mod(1, (“Monteverdi”, “1567-1643”))

```
Monteverdi, 1567-1643
 Schubert, 1797-1828
 Schumann, 1810-1856
```

```
Monteverdi, ?country?
  Schubert, Austria
  Schumann, Germany
```

del(3); ins(1);
mod(1, (“Monteverdi”, **1**))

```
Monteverdi, 1567-1643
 Schubert, 1797-1828
 Schumann, 1810-1856
```

reorder(3,1,2)

```
Monteverdi, Italy
Schubert, Austria
Schumann, Germany
```

reorder(3,1,2)

(f) two different edits with the same effect on the left

Formally...

Roughly:

- Endpoints of a lens are modules (not just sets):
  - A set of states
  - A monoid of edits
  - An action of edits on states
- A lens between R and S is a pair of module homomorphisms

See POPL 2012 paper for details...

Other Data Models

Other Data Models

- In practice, lenses over strings have been the most used so far
  - e.g., Augeas configuration management tool from RedHat

- But there has been substantial work on other data models...

Algebraic Data Structures

- Our bidirectional language over strings is essentially a way of describing (all in the same expression!)
  - two string parsers / unparsers that map strings to algebraic structures built up from singletons, products, disjoint unions, and sequences
  - ... plus a bidirectional transform on the underlying algebraic structures
- Alternative: Ignore the string concrete syntax and just define transformations between algebraic structures
  - Basis for study of lenses from a category-theoretic viewpoint (details in POPL 2011 paper)
  - Common example: in-memory data structures (as in lens libraries for Haskell and Scala)

Relations

- A language of asymmetric bidirectional transformations can be built using familiar relational operators as models for the primitives
  - dropping some relational operators
  - refining others
- Precise schemas help restrict to cases where bidirectionality makes sense
  - "Tree-shaped functional dependencies"

See PODS 2006 paper for more details; also see related work on Prism, Prima, Guava, ...

Trees

- Well-studied area
  - Many proposals
  - Serious examples
- Current state of the art not fully satisfactory
  - Some proposals invent ad-hoc combinators, making it hard to gauge expressiveness
  - Others are based on standard tree-transformation languages, but either give up totality or weaken lens laws
- Challenge: Find a total, law-abiding bidirectional variant of some standard notation for tree transduction

Graphs

- Biggest current challenge (IMHO): Syntax for bidirectional graph transformation
- Small amount of work in this direction so far
  - Bidirectional variant of UnQL (dropping totality, and not fully dealing with node identity)
  - Recent proposal based on triple-graph-grammars (TGGs)
- A good solution would be very useful, e.g., for bidirectional transformation of software models

Wrapping Up...

Take-Away Thoughts

- Bidirectionalizing programs in existing languages is hard
  - alternative is to build new languages that are naturally bidirectional
- Behavioral laws and totality are strong constraints
  - fundamental tension between expressiveness and well-behavedness
  - precise schemas help balance this tension
- Semantic frameworks are interesting
- … but the hardest problems are syntactic
  - i.e., designing bidirectional combinators over specific data models and schema languages

Resources

- Terwilliger, Cleve, and Curino, *How Clean Is Your Sandbox? Towards a Unified Theoretical Framework for Incremental Bidirectional Transformations* (Invited talk/paper at ICMT 2012)
- International Workshop on Bidirectional Transformations (BX 2012 and 2013)
- Report from Dagstuhl workshop on Bidirectional Transformations (Dagstuhl Seminar 11031, 2011)
- Bidirectional Transformations: A Cross-Discipline Perspective (Report from GRACE Workshop, 2008)

Thank You!

Aaron Bohannon, Davi Barbosa, Ravi Chugh, Julien Cretin, Malo Denielou, Nate Foster, Michael Greenberg, Michael Greenwald, Martin Hofmann, Owen Gunden, Sanjeev Khanna, Keshav Kunal, Stéphane Lescuyer, Jon Moore, Benjamin C. Pierce, Alexandre Pilkiewicz, Alan Schmitt, Jeff Vaughan, Daniel Wagner, Zhe Yang

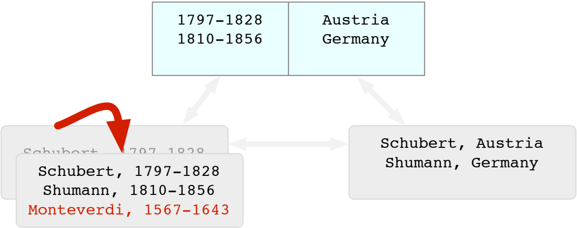
HARMONY

http://www.cis.upenn.edu/~bcpierce/

Extra Slides

Creation

Creation

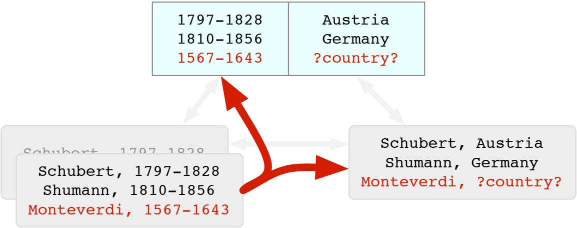


- In the composers example, the top-level lens has the form

  `composers = composer*`

- Since there is no entry in C for `Monteverdi` initially, the `composers` lens needs to call the `composer` sublens with *just the S argument*.
- One simple way to allow this is to assume that each lens specifies a distinguished default element $missing \in C$

Creation



- In the composers example, the top-level lens has the form

  `composers = composer*`

- Since there is no entry in C for `Monteverdi` initially, the `composers` lens needs to call the `composer` sublens with *just the S argument*.
- One simple way to allow this is to assume that each lens specifies a distinguished default element *missing* $\in C$

Synchronization

Synchronization

So far, we've assumed that only one structure at a time can be modified

To handle the case where *both* structures can be edited between propagating updates, we need to add synchronization to our story...

Synchronization

1797–1828 1810–1856	Austria Germany

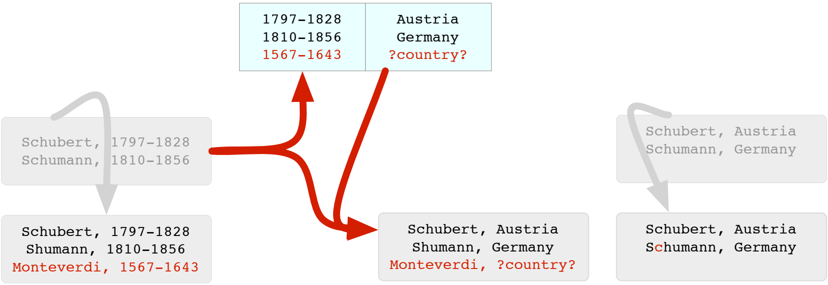
```
Schubert, 1797–1828
Schumann, 1810–1856
```

```
Schubert, 1797–1828
Shumann, 1810–1856
Monteverdi, 1567–1643
```

```
Schubert, Austria
Schumann, Germany
```

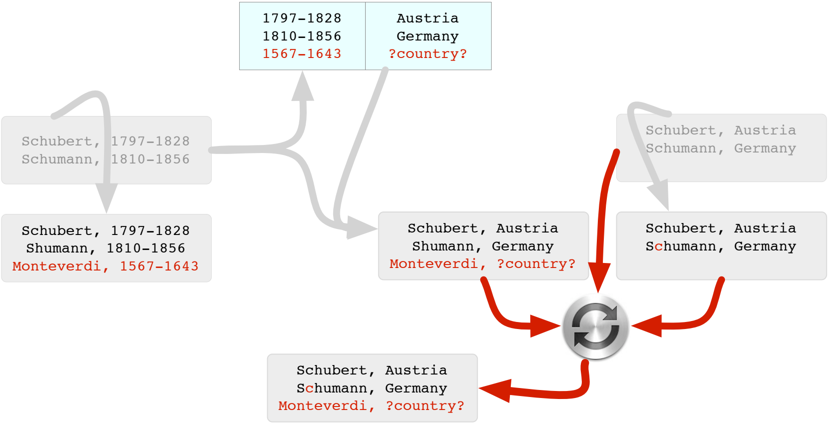
```
Schubert, Austria
Schumann, Germany
```

Synchronization



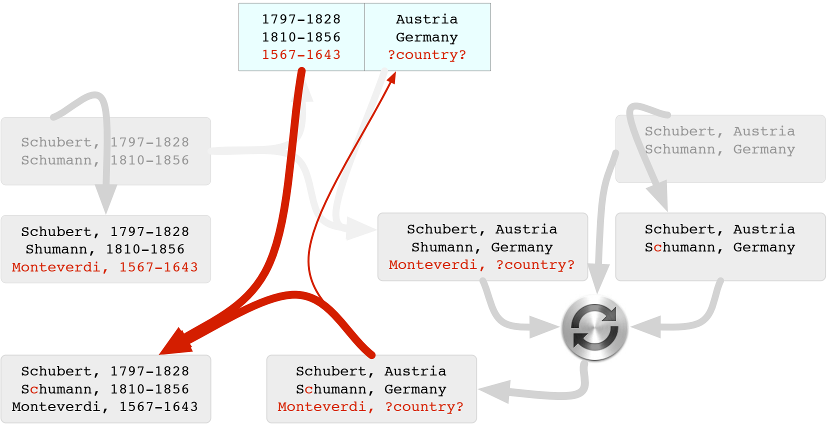
Step 1: Propagate edit from left to right with respect to existing complement (i.e., using the private information from the original right-hand structure)

Synchronization



Step 2: Combine ("synchronize") result with edited right-hand structure to obtain new right-hand structure

Synchronization



Step 3: Propagate new right-hand structure to left; everything is now up to date

Inessential Information

Dealing With "Inessential Information"

- The round-tripping laws we've imposed are attractive for both language designers and programmers
- However, writing lenses in practice, one quickly discovers that they are a bit too strong
  - Most real-world structures include "inessential information" that should be preserved when possible but that can be changed if necessary
    - whitespace, diagram layout, order of rows in tables, etc.
  - Need to loosen the lens laws just a little so that they hold "up to changes in inessential information"
- An "obvious" idea, but takes some work to carry through
- Essential in practice

Our ICFP 2008 paper develops a semantic theory and syntactic constructs for "quotient lenses" that embody this idea.

Controlling Alignment Heuristics

Chunks and Keys

We also need to enrich the syntax a little so the programmer can tell the aligner

1. where are the alignable chunks
2. what are their keys

Chunks and Keys

We also need to enrich the syntax a little so the programmer can tell the aligner

1. where are the alignable chunks
2. what are their keys

```
composers =
    ( copy ALPHA .
      ", " <=> ", " .
      del-> ALPHA "?dates?" .
      del<- ALPHA "?country?" .
      "\n" <=> "\n" )*
```

Chunks and Keys

We also need to enrich the syntax a little so the programmer can tell the aligner

1. where are the alignable chunks
2. what are their keys

```
composers =
    < key ALPHA .
      ", " <=> ", " .
      del-> ALPHA "?dates?" .
      del<- ALPHA "?country?" .
      "\n" <=> "\n" >*
```

Separation of Concerns

1. Alignment is a global matter
2. Alignment algorithms are complicated and messy
   - Often heuristic
   - Different kinds of alignment are useful for different data
     - "bushy" (for "table-like" structures with keys)
     - "diffy" (for "document-like" structures without keys)
     - positional
     - etc.?

To keep the theory (and implementation) clean, separate finding the alignment from using the alignment to translate updates.

Splittability

Footnote: Unique Splittability

The unique splittability conditions ($\cdot^!$ and $^!*$) are strong!

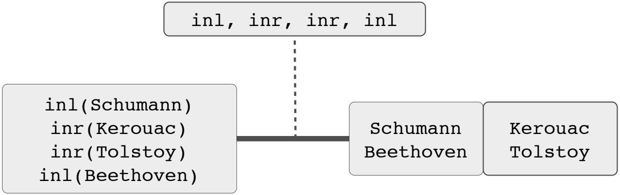
- Not easy to check efficiently, even for regular expressions
- Can be annoying for programmers

But they are fundamental:

- We want to know that $l_1 \cdot l_2$ is a bijective lens
- We're using a type system (i.e., a compositional static analysis) to check this automatically
- So we need to be able to prove that $l_1 \cdot l_2$ is a bijective lens, knowing only that l_1 and l_2 are
- This simply isn't true without the unique splittability restriction

Edit Lenses With Complements

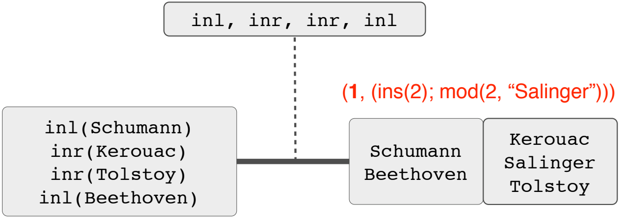
Edit Lenses (With Complements)



(a) initial replicas:

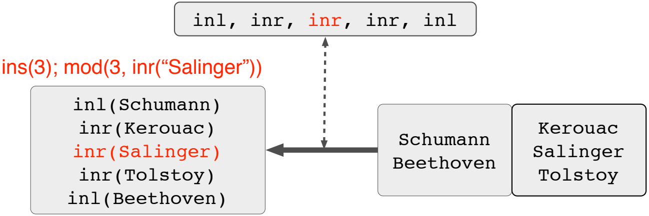
- a tagged list of composers and authors on the left
- a pair of lists on the right
- a complement storing just the tags

Edit Lenses (With Complements)



(b) an element is added to one of the partitions

Edit Lenses (With Complements)



(c) the complement tells how to translate the index