Linguistic Foundations for
Bidirectional Transformations

Benjamin Pierce
University of Pennsylvania

Invited tutorial, PODS 2012
The world is full of replicated data... that is subject to updates... that then need to be propagated to other replicas.
- 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
a text pane in a GUI

its marshalled disk representation
the scroll bar for this text pane
... And Beyond

an in-memory heap structure
a text pane in a GUI
a relational schema

its marshalled disk representation
the scroll bar for this text pane
an ER diagram of the same schema
... And Beyond

an in-memory heap structure
a text pane in a GUI
a relational schema
a requirements model of a software system

its marshalled disk representation
the scroll bar for this text pane
an ER diagram of the same schema
an implementation model of the same system
... And Beyond

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

---

![Diagram showing a process with bidirectional arrows and data points.](image-url)
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:

- 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
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
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:

\[ R ::= \text{“string”} \quad \text{singleton} \]
\[ R_1 \cdot R_2 \quad \text{concatenation} \]
\[ R^* \quad \text{repetition} \]
\[ R_1 \mid R_2 \quad \text{union} \]

Each regular expression denotes a set of strings.
Examples

Schema with composers first:

\[
(\text{ALPHA} \ . \ " \ [" \ . \ \text{DATE} \ . \ "]\ \text{\backslash n})^* \\
\]

Schema with dates first:

\[
(\text{DATE} \ . \ ": \ " \ . \ \text{ALPHA} \ . \ "]\ \text{\backslash n})^* \\
\]

where...

\[
\text{ALPHA} = ("a"|...|"z"|"A"|...|"Z")^* \\
\text{DATE} = ("0"|...|"9"|"-")^* \\
\]
String Transducers

Starting point for our bijective language:

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

\[ f ::= \begin{align*}
& \text{copy } R & \text{recognize } R \text{ and copy it} \\
& \text{del } R & \text{recognize } R \text{ and emit nothing} \\
& r \Rightarrow s & \text{recognize (singleton) } r \text{ and emit } s \\
& f_1 \cdot f_2 & \text{concatenation} \\
& f_1 | f_2 & \text{union} \\
& f^* & \text{repetition} \\
& f_1 \sim f_2 & \text{swapping concatenation}
\end{align*} \]
String Transducers

\[ f \::= \text{copy } R \]
\[ \text{recognize } R \text{ and copy it} \]
\[ \text{del } R \]
\[ \text{recognize } R \text{ and emit nothing} \]
\[ r \Rightarrow s \]
\[ \text{recognize (singleton) } r \text{ and emit } s \]
\[ f_1 \cdot f_2 \]
\[ \text{concatenation} \]
\[ f_1 \mid f_2 \]
\[ \text{union} \]
\[ f^* \]
\[ \text{repetition} \]
\[ f_1 \sim f_2 \]
\[ \text{swapping concatenation} \]
String Transducers

\[ f ::= \begin{align*}
& \text{copy } R & \text{recognize } R \text{ and copy it} \\
& \text{del } R & \text{recognize } R \text{ and emit nothing} \\
& r \Rightarrow s & \text{recognize (singleton) } r \text{ and emit } s \\
& f_1 \cdot f_2 & \text{concatenation} \\
& f_1 \mid f_2 & \text{union} \\
& f^* & \text{repetition} \\
& f_1 \sim f_2 & \text{swapping concatenation}
\end{align*} \]
String Transducers

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

\[
\text{foo} \Rightarrow \text{"bar"}
\]

foo

"foo" ⇒ "bar"

bar
String Transducers

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

\[
\text{fooXYZ} \quad \text{("foo" \Rightarrow "bar") \cdot (copy ALPHA)} \quad \text{barXYZ}
\]
String Transducers

\[ f ::= \begin{align*}
& \text{copy } R \\
& \text{del } R \\
& r \Rightarrow s \\
& f_1 \cdot f_2 \\
& f_1 \mid f_2 \\
& f^* \\
& f_1 \sim f_2
\end{align*} \]

- `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

\[ A ("A" \Rightarrow "B") \mid ("B" \Rightarrow "A") \quad \rightarrow \quad B \]
String Transducers

\[ f ::= \begin{align*}
\text{copy } R & \quad \text{recognize } R \text{ and copy it} \\
\text{del } R & \quad \text{recognize } R \text{ and emit nothing} \\
r \Rightarrow s & \quad \text{recognize (singleton) } r \text{ and emit } s \\
f_1 \cdot f_2 & \quad \text{concatenation} \\
f_1 \mid f_2 & \quad \text{union} \\
f^* & \quad \text{repetition} \\
f_1 \sim f_2 & \quad \text{swapping concatenation}
\end{align*} \]

\[
\begin{array}{c}
\text{AAABA} \\
\text{("A" \Rightarrow "B" | "B" \Rightarrow "A")}^* \\
\text{BBBAB}
\end{array}
\]
String Transducers

\[ f \ ::= \ copy R \quad \text{recognize } R \text{ and copy it} \\
    del R \quad \text{recognize } R \text{ and emit nothing} \\
    r \Rightarrow s \quad \text{recognize (singleton) } r \text{ and emit } s \\
    f_1 \cdot f_2 \quad \text{concatenation} \\
    f_1 | f_2 \quad \text{union} \\
    f^* \quad \text{repetition} \\
    f_1 \sim f_2 \quad \text{swapping concatenation} \]
Next step

Bidirectionalize!
Issue #1: Deletion

Problem:
- Deletion operator throws away information
- Cannot be part of a bijective transformation

Solution:
- Throw it away (it will come back later)

N.b.: “Singleton deletion” is bijective
Issue #2: Union

Problem:

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

$$\text{A \ ("A" \Rightarrow "B") | ("A" \Rightarrow "C") \rightarrow \{B, C\}}$$
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

\[
\begin{align*}
A \quad \text{("A" } \Rightarrow \text{ "B") } & \mid \quad \text{("C" } \Rightarrow \text{ "B")} \\
\text{... in which case it can’t be part of a bijective transformation}
\end{align*}
\]
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

\[
\begin{align*}
l_1 & \in R_1 \Leftrightarrow S_1 \\
R_1 \cap R_2 & = \emptyset \\
l_2 & \in R_2 \Leftrightarrow S_2 \\
S_1 \cap S_2 & = \emptyset \\
l_1 \mid l_2 & \in R_1 \mid R_2 \Leftrightarrow S_1 \mid S_2
\end{align*}
\]
Issue #3: Concatenation

Problem:

- In general, concatenation is not deterministic

\[
\begin{align*}
\text{ABCD} & \quad (\text{copy \ ALPHA}) \cdot (\text{del \ ALPHA}) \\
\text{???} & \quad \text{???}
\end{align*}
\]

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 \( S_2 \))
Issue #3: Concatenation

Problem:

- In general, concatenation is not deterministic

  \[
  \text{ABCD} \quad (\text{copy ALPHA}) \cdot (\text{del ALPHA})
  \]

Solution:

- Schemas again...

  \[
  l_1 \in R_1 \Leftrightarrow S_1 \quad l_2 \in R_2 \Leftrightarrow S_2
  \]

  \[
  R_1 \cdot ! R_2 \quad S_1 \cdot ! S_2
  \]

  \[
  l_1 \cdot l_2 \in R_1 \cdot R_2 \Leftrightarrow S_1 \cdot S_2
  \]

  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 \( S_2 \))
A bijective lens \( l \) between a set \( R \) and a set \( S \), written
\[
l \in R \rightleftharpoons S
\]
comprises two functions
\[
l \rightarrow \in R \rightarrow S \\
l \leftarrow \in S \rightarrow R
\]
where \( l \rightarrow \) and \( l \leftarrow \) are inverses:
\[
l \leftarrow (l \rightarrow r) = r \\
l \rightarrow (l \leftarrow s) = s
\]
Typing Rules

\[\text{copy } R \in R \Rightarrow R\]

\[\text{“s” } \Leftrightarrow \text{“t” } \in \text{“s” } \Rightarrow \text{“t”}\]

\[l_1 \in R_1 \Rightarrow S_1 \quad l_2 \in R_2 \Rightarrow S_2 \quad R_1 \cdot \! R_2 \quad S_1 \cdot \! S_2\]

\[l_1 \cdot l_2 \in R_1 \cdot R_2 \Rightarrow S_1 \cdot S_2\]

\[l \in R \Rightarrow S \quad R^*! \quad S^!\]

\[l^* \in S^* \Rightarrow R^*\]

\[l_1 \in R_1 \Rightarrow S_1 \quad l_2 \in R_2 \Rightarrow S_2 \quad R_1 \cap R_2 = \emptyset \quad S_1 \cap S_2 = \emptyset\]

\[l_1 \mid l_2 \in R_1 \mid R_2 \Rightarrow S_1 \mid S_2\]

\[l_1 \in R_1 \Rightarrow S_1 \quad l_2 \in R_2 \Rightarrow S_2 \quad R_1 \cdot \! R_2 \quad S_1 \cdot \! S_2\]

\[l_1 \Leftrightarrow l_2 \in R_1 \cdot R_2 \Rightarrow S_2 \cdot S_1\]
Type Soundness

Theorem: If $l \in R \Leftrightarrow S$ 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, ⇔)
  - 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...
add an extra structure (the "complement") that stores the "private information" from both sides
Intuition

- Schubert, 1797-1828
- Austria
- 1797-1828

- Shumann, 1810-1856
- Germany
- 1810-1856

- Monteverdi, 1567-1643

Schubert, 1797-1828
Schumann, 1810-1856
Schubert, Austria
Shumann, Germany
1797-1828
1810-1856
Austria
Germany

Schubert, Austria
Schumann, Germany
each transformation propagates updates both to the target artifact and to the complement...
Intuition

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

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

- Schubert, Austria
- Germany
- Monteveddi, Italy
Intuition

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

1797-1828
1810-1856
1567-1643

Austria
Germany
Italy

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

$$
l \rightarrow \in R \times C \rightarrow S \times C
$$

$$
l \leftarrow \in S \times C \rightarrow R \times C
$$

where

$$
l \rightarrow (r, c) = (s', c')
$$

$$
l \leftarrow (s', c') = (r, c')
$$

$$
l \leftarrow (s, c) = (r', c')
$$

$$
l \rightarrow (r', c') = (s, c')
$$

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

\[
\begin{align*}
    l &
    \quad \in \quad R \times C \rightarrow S \times C \\
    l &
    \quad \notin \quad S \times C \rightarrow R \times C \\
\end{align*}
\]

where

\[
\begin{align*}
    l \rightarrow (r, c) &= (s', c') \\
    l \leftarrow (s', c') &= (r, c') \\
    l \leftarrow (s, c) &= (r', c') \\
    l \rightarrow (r', c') &= (s, c')
\end{align*}
\]

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 \textit{del} operator back again!

Indeed, since we’re in a symmetric setting, we can have two delete operators

\begin{itemize}
\item \textit{del} \rightarrow ("delete when going left to right")
\item \textit{del} \leftarrow ("delete when going right to left")
\end{itemize}

\begin{align*}
d & \in R \\
del \rightarrow R d & \in R \iff "" \\
\end{align*}

\begin{align*}
d & \in S \\
del \leftarrow S d & \in ""S \iff \\
\end{align*}
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 $l\rightarrow$ 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?
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 \textit{any update}, with respect to \textit{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
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

<table>
<thead>
<tr>
<th>Year (Range)</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>1797-1828</td>
<td>Austria</td>
</tr>
<tr>
<td>1810-1856</td>
<td>Germany</td>
</tr>
<tr>
<td>1567-1643</td>
<td>Italy</td>
</tr>
</tbody>
</table>

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

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

Austria
Germany
Italy
Alignment

<table>
<thead>
<tr>
<th>1797-1828</th>
<th>1810-1856</th>
<th>1567-1643</th>
</tr>
</thead>
<tbody>
<tr>
<td>Schubert,</td>
<td>Schumann,</td>
<td>Monteverdi,</td>
</tr>
<tr>
<td>Austria</td>
<td>Germany</td>
<td>Italy</td>
</tr>
</tbody>
</table>

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

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
Schubert, 1797-1828
Schumann, 1810-1856
Shubert, Austria
Shumann, Germany

(a) initial replicas
(b) a new composer is added to one replica
(c) the lens adds the new composer to the other replica
(d) the curator makes some corrections
(e) the lens transports a small edit
(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)
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...
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

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

http://www.cis.upenn.edu/~bcpierce/
Extra Slides
Creation
In the composers example, the top-level lens has the form

\[
\text{composers} = \text{composer}^* 
\]

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 \( \text{missing} \in C \)
In the composers example, the top-level lens has the form

\[ \text{composers} = \text{composer}^* \]

Since there is no entry in \( C \) for Monteverdi initially, the composers lens needs to call the composer sublens with \textit{just the} \( S \) \textit{argument}.

One simple way to allow this is to assume that each lens specifies a distinguished default element \textit{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

<table>
<thead>
<tr>
<th>Schubert, Austria 1810-1856</th>
<th>Schumann, Germany 1810-1856</th>
</tr>
</thead>
</table>

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

Schubert, Austria
Schumann, Germany
Monteverdi, ?country?

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

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

```plaintext
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
(b) an element is added to one of the partitions
(c) the complement tells how to translate the index