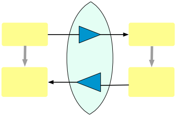
Foundations for Bidirectional Programming

Benjamin Pierce
University of Pennsylvania

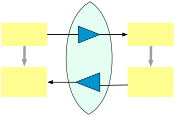
ICMT 2009



~~Foundations for Bidirectional Programming~~

Benjamin Pierce
University of Pennsylvania

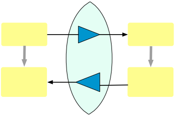
ICMT 2009



How To Build a Bidirectional Programming Language

Benjamin Pierce
University of Pennsylvania

ICMT 2009



Connected Structures

Connected Structures

a database

a materialized view

Connected Structures

a database

an in-memory heap structure

a materialized view

its marshalled disk representation

Connected Structures

a database

an in-memory heap structure

an XML document

a materialized view

its marshalled disk representation

a pretty-printed textual representation

Connected Structures

a database	a materialized view
an in-memory heap structure	its marshalled disk representation
an XML document	a pretty-printed textual representation
a text pane in a GUI	the scroll bar for this text pane

Connected Structures

a database	a materialized view
an in-memory heap structure	its marshalled disk representation
an XML document	a pretty-printed textual representation
a text pane in a GUI	the scroll bar for this text pane
a relational schema	an ER diagram of the same schema

Connected Structures

a database	a materialized view
an in-memory heap structure	its marshalled disk representation
an XML document	a pretty-printed textual 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

Connected Structures

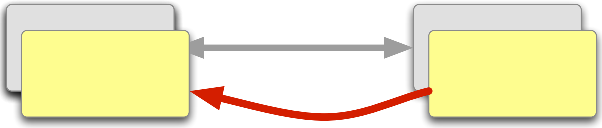
a database	a materialized view
an in-memory heap structure	its marshalled disk representation
an XML document	a pretty-printed textual representation
a text pane in a GUI	the scroll bar for this text pane
a relational schema	an ER diagram of the same schema
a requirements model of a software system	an implementation model of the same system

Unfortunately, nothing stays the same forever…

Connected Structures... and Updates

When one of the structures is changed...

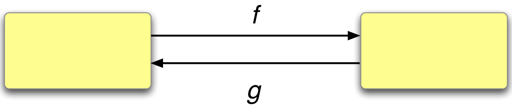
Connected Structures... and Updates



When one of the structures is changed... the other needs to be updated "in the same way"

An “Easy” Solution

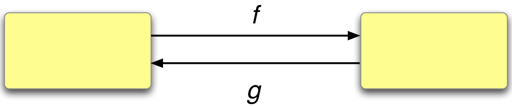
Standard approach: write a pair of functions, each propagating updates in one direction.



- \+ Uses standard technology
- \+ Works fine for simple transformations

An "Easy" Solution

Standard approach: write a pair of functions, each propagating updates in one direction.



- \+ Uses standard technology
- \+ Works fine for simple transformations

- – Scales badly
- – Maintenance nightmare
- – No automatic support for detecting mistakes

A Better Idea

Specify both transformations with a single description!

Many* instances of this idea...

- ad hoc libraries and tools (marshallers/unmarshallers, parsers/prettyprinters, ...)
- bidirectional versions of standard languages (XQuery, UnQL, relational algebra, ...)
- domain-specific bidirectional languages
  - "coupled grammars" (XSugar, biXid, TGGs, ...)
  - combinator-based (this talk)
- "program inversion" / "reversible computation"
- "Bidirectionalization for Free"
- etc.

*dozens, if not hundreds...

Research Challenge

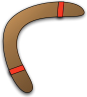
Many solutions exist, but...

1. they tend to be specialized to very particular domains
2. fundamental design principles are not well understood

Harmony

The Harmony project at the University of Pennsylvania has been working in this space for a number of years.

- Focus on strong semantic foundations
- Working prototypes
  - Focal: a bidirectional tree transformation language
  - a bidirectional variant of relational algebra
  - Boomerang: a bidirectional string transformation language
- Applications
  - XML $\leftrightarrow$ ASCII converter for UniProtKB genome DB
  - BibTex, iCal, vCard
  - ...



Goals of the Talk

- Explore fundamental concepts of bidirectional programming in the simplest imaginable setting
  - data = strings
  - types = regular expressions
  - computation = finite state transduction
  - bijective transformations (to start with)

Goals of the Talk

- Explore fundamental concepts of bidirectional programming in the simplest imaginable setting

  no UML, graphs, ...

  - data = strings
  - types = regular expressions
  - computation = finite state transduction
  - bijective transformations (to start with)

Goals of the Talk

- Explore fundamental concepts of bidirectional programming in the simplest imaginable setting
  - data = strings
  - types = regular expressions
  - computation = finite state transduction
  - bijective transformations (to start with)

no UML, graphs, ...

Simple, but not trivial...

- ordered
- lots of implicit structure

Outline

- Bijective lenses
- Non-bijective lenses
- Sketches of additional topics (time permitting)
  - Global alignment
  - Synchronization (handling parallel updates)
  - Data integrity
  - Quotienting away "inessential" information

Please ask questions!

Bijective Programming

Example

```
<composers>
  <name>Schubert</name>
  <dates>1797-1828</dates>
</composers>
```

```
Schubert, 1797-1828
```

Example

```
<composers>
  <name>Schubert</name>
  <dates>1797-1828</dates>
</composers>
```

```
Schubert, 1797-1828
```

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

Example

```
<composers>
  <name>Schubert</name>
  <dates>1797-1828</dates>
</composers>
```

```
Schubert, 1797-1828
```

```
<composers>
  <name>Schubert</name>
  <dates>1797-1828</dates>
  <name>Schumann</name>
  <dates>1810-1856</dates>
</composers>
```

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

Example

```
<composers>
  <name>Schubert</name>
  <dates>1797-1828</dates>
  <name>Schumann</name>
  <dates>1810-1856</dates>
</composers>
```

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

```
composers =
    "<composers>\n" <=> "" .
    ( "<name>" <=> "" .
      copy ALPHA .
      "  </name><dates>" <=> ", " .
      copy ALPHA .
      "  </dates>\n" <=> "")* .
    "</composers>" <=> ""
```

Example

```
<composers>
  <name>Schubert</name>
  <dates>1797-1828</dates>
</
```

```
Schubert, 1797-1828
```

```
<composers>
  <name>Schubert</name>
  <dates>1797-1828</dates>
  <name>Schumann</name>
  <dates>1810-1856</dates>
</composers>
```

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

```
composers =
    "<composers>\n" <=> "" .
    ( "<name>" <=> "" .
      copy ALPHA .
      "  </name><dates>" <=> ", " .
      copy ALPHA .
      "  </dates>\n" <=> "")* .
    "</composers>" <=> ""
```

Now let's break it down...

Basic Structures

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

$$l \in R \rightleftharpoons S$$

comprises two (total) functions

$$\begin{array}{rcl} l^{\rightarrow} & \in & R \longrightarrow S \\ l^{\leftarrow} & \in & S \longrightarrow R \end{array}$$

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

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

Regular Expressions

$$
\begin{array}{lcll}
R & ::= & \{\textit{string}\} & \text{singleton} \\
 & & R_1 \cdot R_2 & \text{concatenation} \\
 & & R_1 \mid R_2 & \text{union} \\
 & & R* & \text{repetition} \\
 & & \emptyset & \text{empty set}
\end{array}
$$

As always, a regular expression denotes a set of strings

Examples

```
ALPHA  =  ( {a}|...|{z}|{A}|...|{Z} )*

composersXML =
   "<composers>\n" .
   ( "<name>" .
     ALPHA .
     "  </name><dates>" .
     ALPHA .
     "  </dates>\n")* .
   "</composers>"

composersASCII = ...similar...
```

Examples

```
ALPHA  =  ( {a}|...|{z}|{A}|...|{Z} )*

composersXML =
   "<composers>\n" .
   ( "<name>" .
     ALPHA .
     "  </name><dates>" .
     ALPHA .
     "  </dates>\n")* .
   "</composers>"

composersASCII = ...similar...
```

Next step...

Finite-State Transducers

```
ALPHA  =  ( {a}|...|{z}|{A}|...|{Z} )*

composersXML =
   "<composers>\n" . => ""
   ( "<name>" . => ""
     copy ALPHA .
     "  </name><dates>" . => ", "
     copy ALPHA .
     "  </dates>\n" => "" )* .
   "</composers>" => ""

composersASCII = ...similar...
```

Finite-State Transducers

=

Regular expressions with outputs

Finite-State Transducers (FSTs)

The simplest possible programming language over strings...

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

Finite-State Transducers (FSTs)

The simplest possible programming language over strings...

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
		$f_1 \ ; f_2$	composition (do f_1 then f_2)
		$f_1 \sim f_2$	swapping concatenation

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

Finite-State Transducers (FSTs)

The simplest possible programming language over strings...

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
		$f_1 \,;\, f_2$	composition (do f_1 then f_2)
		$f_1 \sim f_2$	swapping concatenation

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

Finite-State Transducers (FSTs)

The simplest possible programming language over strings...

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
		$f_1 ; f_2$	composition (do f_1 then f_2)
		$f_1 \sim f_2$	swapping concatenation

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

Finite-State Transducers (FSTs)

The simplest possible programming language over strings...

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
		$f_1 \; ; \; f_2$	composition (do f_1 then f_2)
		$f_1 \sim f_2$	swapping concatenation

$$\texttt{fooXX} \xrightarrow{(\texttt{"foo"} \Rightarrow \texttt{"bar"}) \cdot (\textit{copy}\ \texttt{ALPHA})} \texttt{barXX}$$

Finite-State Transducers (FSTs)

The simplest possible programming language over strings...

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

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

Finite-State Transducers (FSTs)

The simplest possible programming language over strings...

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
		$f_1 \; ; \; f_2$	composition (do f_1 then f_2)
		$f_1 \sim f_2$	swapping concatenation

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

Finite-State Transducers (FSTs)

The simplest possible programming language over strings...

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
		$f_1 \; ; \; f_2$	composition (do f_1 then f_2)
		$f_1 \sim f_2$	swapping concatenation

$$\texttt{AAABA} \xrightarrow[]{\substack{(\texttt{"A"} \Rightarrow \texttt{"B"} \mid \texttt{"B"} \Rightarrow \texttt{"A"})* \\ ;\; (\texttt{"A"} \Rightarrow \texttt{"A"} \mid \texttt{"B"} \Rightarrow \texttt{"C"})*}} \texttt{CCCAC}$$

Finite-State Transducers (FSTs)

The simplest possible programming language over strings...

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
		$f_1 \ ; \ f_2$	composition (do f_1 then f_2)
		$f_1 \sim f_2$	swapping concatenation

$$\texttt{fooXX} \xrightarrow{(\texttt{"foo"} \Rightarrow \texttt{"bar"}) \sim (copy\ \texttt{ALPHA})} \texttt{XXbar}$$

Finite-State Functions (FSFs)

In general, an FST denotes a relation on strings.

For today, we want to restrict attention to FSTs that denote total functions.

Finite-State Functions (FSFs)

In general, an FST denotes a relation on strings.

For today, we want to restrict attention to FSTs that denote total functions.

Given an FST f, how can we tell whether it is a function?

Finite-State Functions (FSFs)

In general, an FST denotes a relation on strings.

For today, we want to restrict attention to FSTs that denote total functions.

Given an FST f, how can we tell whether it is a function?

One way: With a type system!

...that generalizes nicely for other purposes...

Finite-State Functions: Types

Write $f \in R \longrightarrow S$ to mean "f is a finite-state function from R to S"

- i.e., f relates each string in R to a unique string in S

Now, for each syntactic form, we give a rule that describes when an FST of that form is guaranteed to be a function (and tells us its domain and range)...

Finite-State Functions: Typing Rules

$$\textit{copy}\ R \ \in\ R \rightarrow R$$

Finite-State Functions: Typing Rules

$$\textit{copy } R \;\in\; R \rightarrow R$$

$$\textit{delete } R \;\in\; R \rightarrow \{\texttt{""}\}$$

Finite-State Functions: Typing Rules

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

$$\mathit{delete}\ R \ \in\ R \to \{\texttt{""}\}$$

$$s \Rightarrow t \ \in\ \{s\} \to \{t\}$$

Finite-State Functions: Typing Rules

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

$$\mathit{delete}\ R\ \in\ R \to \{\texttt{""}\}$$

$$s \Rightarrow t\ \in\ \{s\} \to \{t\}$$

$$\frac{f_1\ \in\ R_1 \to S_1 \qquad f_2\ \in\ R_2 \to S_2}{f_1 \cdot f_2\ \in\ R_1 \cdot R_2\ \to\ S_1 \cdot S_2}$$

first try

Finite-State Functions: Typing Rules

$$copy\ R \ \in\ R \to R$$

$$delete\ R \ \in\ R \to \{\texttt{""}\}$$

$$s \Rightarrow t \ \in\ \{s\} \to \{t\}$$

$$\frac{f_1 \ \in\ R_1 \to S_1 \qquad f_2 \ \in\ R_2 \to S_2}{f_1 \cdot f_2 \ \in\ R_1 \cdot R_2 \ \to\ S_1 \cdot S_2}$$

first try

Problem: Concatenation is not always deterministic!

$$\begin{array}{lcl} f & = & (copy\ \texttt{ALPHA}) \cdot (del\ \texttt{ALPHA}) \\ f\ \texttt{"abcd"} & = & ??? \end{array}$$

Finite-State Functions: Typing Rules

$$copy\ R \ \in\ R \rightarrow R$$

$$delete\ R \ \in\ R \rightarrow \{\texttt{""}\}$$

$$s \Rightarrow t \ \in\ \{s\} \rightarrow \{t\}$$

$$\frac{f_1 \ \in\ R_1 \rightarrow S_1 \qquad f_2 \ \in\ R_2 \rightarrow S_2 \qquad R_1 \cdot^{!} R_2}{f_1 \cdot f_2 \ \in\ R_1 \cdot R_2 \ \rightarrow\ S_1 \cdot S_2}$$

Problem: Concatenation is not always deterministic!

$$\begin{array}{lcl} f & = & (copy\ \texttt{ALPHA}) \cdot (del\ \texttt{ALPHA}) \\ f\ \texttt{"abcd"} & = & ??? \end{array}$$

Solution: Require that R_1 and R_2 be "uniquely splittable"

- 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

Finite-State Functions: Typing Rules

$$copy\ R \ \in\ R \rightarrow R$$

$$delete\ R \ \in\ R \rightarrow \{""\}$$

$$s \Rightarrow t \ \in\ \{s\} \rightarrow \{t\}$$

$$\frac{f_1 \ \in\ R_1 \rightarrow S_1 \qquad f_2 \ \in\ R_2 \rightarrow S_2 \qquad R_1 \cdot^! R_2}{f_1 \cdot f_2 \ \in\ R_1 \cdot R_2 \ \rightarrow\ S_1 \cdot S_2}$$

$$\frac{f \ \in\ R \rightarrow S \qquad R^{*!}}{f^* \ \in\ R^* \rightarrow S^*}$$

similarly

Finite-State Functions: Typing Rules

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

$$\mathit{delete}\ R \ \in\ R \rightarrow \{\texttt{""}\}$$

$$s \Rightarrow t \ \in\ \{s\} \rightarrow \{t\}$$

$$\frac{f_1 \ \in\ R_1 \rightarrow S_1 \qquad f_2 \ \in\ R_2 \rightarrow S_2 \qquad R_1 \cdot^! R_2}{f_1 \cdot f_2 \ \in\ R_1 \cdot R_2 \ \rightarrow\ S_1 \cdot S_2}$$

$$\frac{f \ \in\ R \rightarrow S \qquad R^{*!}}{f^* \ \in\ R^* \rightarrow S^*}$$

$$\frac{f_1 \ \in\ R_1 \rightarrow S_1 \qquad f_2 \ \in\ R_2 \rightarrow S_2}{f_1 \mid f_2 \ \in\ R_1 \mid R_2 \ \rightarrow\ S_1 \mid S_2}$$

first try

Finite-State Functions: Typing Rules

$$\textit{copy}\ R \ \in\ R \to R$$

$$\textit{delete}\ R \ \in\ R \to \{\texttt{""}\}$$

$$s \Rightarrow t \ \in\ \{s\} \to \{t\}$$

$$\frac{f_1 \ \in\ R_1 \to S_1 \qquad f_2 \ \in\ R_2 \to S_2 \qquad R_1 \cdot^! R_2}{f_1 \cdot f_2 \ \in\ R_1 \cdot R_2 \ \to\ S_1 \cdot S_2}$$

$$\frac{f \ \in\ R \to S \qquad R^{*!}}{f^* \ \in\ R^* \to S^*}$$

$$\frac{f_1 \ \in\ R_1 \to S_1 \qquad f_2 \ \in\ R_2 \to S_2 \qquad R_1 \cap R_2 = \emptyset}{f_1 \mid f_2 \ \in\ R_1 \mid R_2 \ \to\ S_1 \mid S_2}$$

But what if R_1 and R_2 overlap? Again, not bijective!

- Need to require that R_1 and R_2 be disjoint

Finite-State Functions: Typing Rules

$$copy\ R \ \in\ R \rightarrow R$$

$$delete\ R \ \in\ R \rightarrow \{""\}$$

$$s \Rightarrow t \ \in\ \{s\} \rightarrow \{t\}$$

$$\frac{f_1 \ \in\ R_1 \rightarrow S_1 \qquad f_2 \ \in\ R_2 \rightarrow S_2 \qquad R_1 \cdot^! R_2}{f_1 \cdot f_2 \ \in\ R_1 \cdot R_2 \ \rightarrow\ S_1 \cdot S_2}$$

$$\frac{f \ \in\ R \rightarrow S \qquad R^{*!}}{f^* \ \in\ R^* \rightarrow S^*}$$

$$\frac{f_1 \ \in\ R_1 \rightarrow S_1 \qquad f_2 \ \in\ R_2 \rightarrow S_2 \qquad R_1 \cap R_2 = \emptyset}{f_1 \mid f_2 \ \in\ R_1 \mid R_2 \ \rightarrow\ S_1 \mid S_2}$$

$$\frac{f_1 \ \in\ R \rightarrow U \qquad f_2 \ \in\ U \rightarrow S}{f_1\ ;\ f_2 \ \in\ R \rightarrow S}$$

Finite-State Functions: Typing Rules

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

$$\mathit{delete}\ R \ \in\ R \to \{""\}$$

$$s \Rightarrow t \ \in\ \{s\} \to \{t\}$$

$$\frac{f_1 \ \in\ R_1 \to S_1 \qquad f_2 \ \in\ R_2 \to S_2 \qquad R_1 \cdot^{!} R_2}{f_1 \cdot f_2 \ \in\ R_1 \cdot R_2 \ \to\ S_1 \cdot S_2}$$

$$\frac{f \ \in\ R \to S \qquad R^{*!}}{f^* \ \in\ R^* \to S^*}$$

$$\frac{f_1 \ \in\ R_1 \to S_1 \qquad f_2 \ \in\ R_2 \to S_2 \qquad R_1 \cap R_2 = \emptyset}{f_1 \mid f_2 \ \in\ R_1 \mid R_2 \ \to\ S_1 \mid S_2}$$

$$\frac{f_1 \ \in\ R \to U \qquad f_2 \ \in\ U \to S}{f_1 \ ; \ f_2 \ \in\ R \to S}$$

$$\frac{f_1 \ \in\ R_1 \to S_1 \qquad f_2 \ \in\ R_2 \to S_2 \qquad R_1 \cdot^{!} R_2}{f_1 \sim f_2 \ \in\ R_1 \cdot R_2 \ \to\ S_2 \cdot S_1}$$

Bidirectionalizing FSFs

Ordinary FSFs		Bidirectional FSFs
$f ::= copy\ R$		$l ::= copy\ R$
$del\ R$		—
$r \Rightarrow s$		$r \Leftrightarrow s$
$f_1 \cdot f_2$	$\Longrightarrow$	$l_1 \cdot l_2$
$f_1 \mid f_2$		$l_1 \mid l_2$
$f*$		$l*$
$f_1 \,;\, f_2$		$l_1 \,;\, l_2$
$f_1 \sim f_2$		$l_1 \sim l_2$

- drop *del* (can't be part of a bijection anyway)
- write $\Rightarrow$ as $\Leftrightarrow$ to emphasize symmetry
- give each syntactic form the natural interpretation as a bijective lens (straightforward details elided)

Example

```
composers =
    "<composers>\n" <=> "" .
    ( "<name>" <=> "" .
      copy ALPHA .
      "  </name><dates>" <=> ", " .
      copy ALPHA .
      "  </dates>\n" <=> "")* .
    "</composers>" <=> ""
```

Example

```
composers =
    "<composers>\n" <=> "" .
    ( "<name>" <=> "" .
      copy ALPHA .
      "  </name><dates>" <=> ", " .
      copy ALPHA .
      "  </dates>\n" <=> "")* .
    "</composers>" <=> ""
```

Next question: How do we know that a given expression in the bijective syntax really denotes a law-abiding (i.e., bijective) lens?

Example

```
composers =
    "<composers>\n" <=> "" .
    ( "<name>" <=> "" .
      copy ALPHA .
      "  </name><dates>" <=> ", " .
      copy ALPHA .
      "  </dates>\n" <=> "")* .
    "</composers>" <=> ""
```

Next question: How do we know that a given expression in the bijective syntax really denotes a law-abiding (i.e., bijective) lens?

Answer: With a type system, naturally! ...

Bijective Lenses: Typing Rules

$$copy\ R \ \in\ R \rightleftharpoons R$$

$$s \Rightarrow t \ \in\ \{s\} \rightleftharpoons \{t\}$$

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

(and similarly for the other syntactic forms)

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

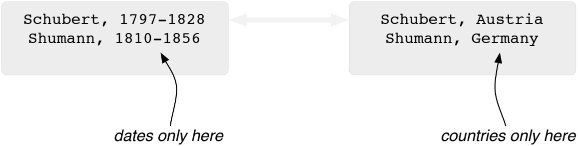
Bidirectional Programming (The Non-Bijective Case)

Symmetric vs. Asymmetric

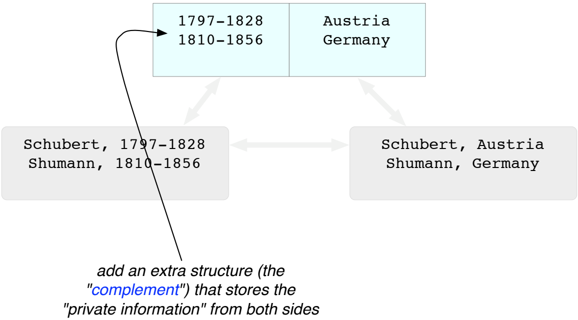
Non-bijective connected structures come in two varieties:

- Symmetric ("many to many")
  - *both* transformations "lose information"
    - formally, they are not injective
  - Example: Two models of different aspects of a software system
- Asymmetric ("many to one")
  - one of the transformations is injective while the other is not
  - Example: A database and a materialized view
- At Penn we've worked mostly on the asymmetric case
  - So, for fun, let's talk about the symmetric case here... :-)

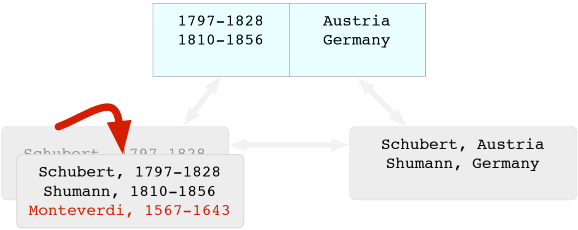
Intuition


Schubert, 1797–1828
Shumann, 1810–1856
Schubert, Austria
Shumann, Germany
dates only here
countries only here

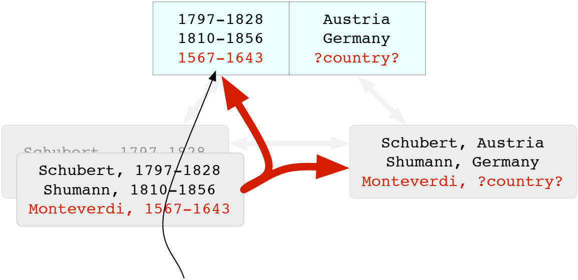
Intuition


1797–1828
1810–1856
Austria
Germany
Schubert, 1797–1828
Shumann, 1810–1856
Schubert, Austria
Shumann, Germany
add an extra structure (the "complement") that stores the "private information" from both sides

Intuition

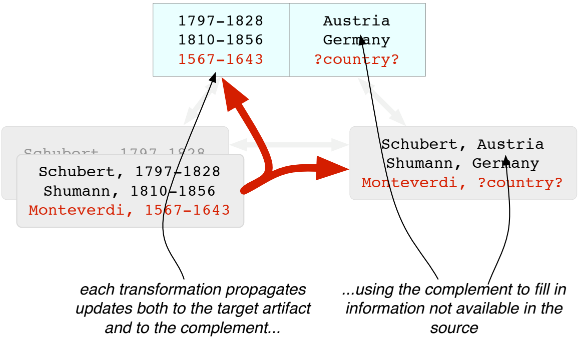

1797–1828
1810–1856
Austria
Germany
Schubert, 1797–1828
Shumann, 1810–1856
Monteverdi, 1567–1643
Schubert, Austria
Shumann, Germany

Intuition



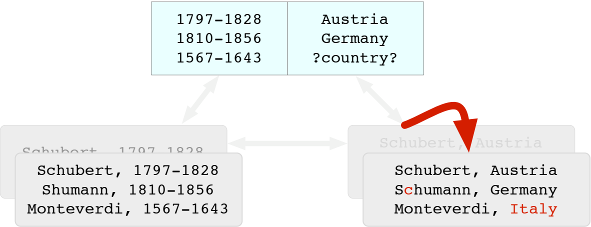
each transformation propagates updates both to the target artifact and to the complement...

Intuition

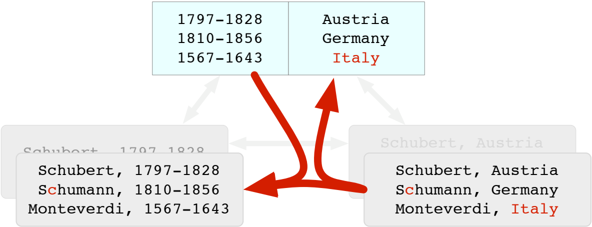


1797-1828
1810-1856
1567-1643
Austria
Germany
?country?
Schubert, 1797-1828
Shumann, 1810-1856
Monteverdi, 1567-1643
Schubert, Austria
Shumann, Germany
Monteverdi, ?country?
each transformation propagates updates both to the target artifact and to the complement...
...using the complement to fill in 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
Schumann, Germany
Monteverdi, Italy

Intuition


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

Symmetric Lenses (First Version)

A symmetric lens l between a set R and a set S with complement C, written $l \in R \rightleftharpoons^{C} S$, comprises two functions

$$\begin{array}{lcl} l^{\Rightarrow} & \in & R \times C \longrightarrow S \times C \\ l^{\Leftarrow} & \in & S \times C \longrightarrow R \times C \end{array}$$

where

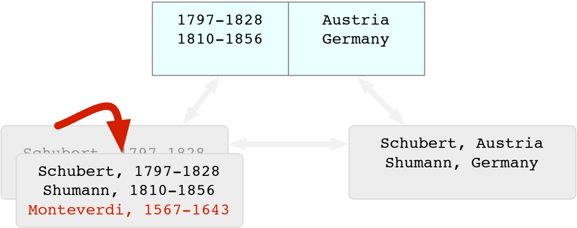
propagating a null update changes nothing

$$\frac{l^{\Rightarrow}(r,c) = (s',c')}{l^{\Leftarrow}(s',c') = (r,c')}$$

ditto

$$\frac{l^{\Leftarrow}(s,c) = (r',c')}{l^{\Rightarrow}(r',c') = (s,c')}$$

Creation

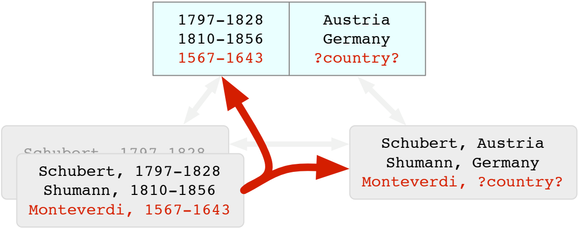


- In the composers example, the top-level lens has the form

  `composers = composer*`

- Since there is no entry in C for `Monteverdi` initially, the `composers` lens needs to call the `composer` sublens with *just an S argument*.
- We need variants of $\texttt{composer}^{\Rightarrow}$ and $\texttt{composer}^{\Leftarrow}$ that create an appropriate C by filling in defaults

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 an S argument*.
- We need variants of `composer`$^{\Rightarrow}$ and `composer`$^{\Leftarrow}$ that create an appropriate C by filling in defaults

Symmetric Lenses (Final Version)

A symmetric lens l between a set R and a set S with complement C, written $l \in R \rightleftharpoons^{C} S$, comprises four functions

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

where

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

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

Building Symmetric Lenses

- We can use all the same syntactic primitives
  - ...generalizing their behavior and typing rules
- And we get to add some interesting new ones...
  - In particular, `del E` now makes sense

See our POPL 08 paper for full details (for the asymmetric case)

The Example, Again

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

Digression: State-based vs.Operation-Based

We've been assuming so far that the main arguments to the $I^{\Rightarrow}$ and $I^{\Leftarrow}$ functions were entire structures. Naturally, there are other choices...

$$I^{\Rightarrow} \quad \in \quad \begin{cases} R \times C \;\longrightarrow\; S \times C & \text{state-based} \\ \Delta R \times C \;\longrightarrow\; S \times C & \text{delta-based} \\ (R \longrightarrow R) \times C \;\longrightarrow\; S \times C & \text{operation-based} \end{cases}$$

Digression: State-based vs.Operation-Based

We've been assuming so far that the main arguments to the $l^{\Rightarrow}$ and $l^{\Leftarrow}$ functions were entire structures. Naturally, there are other choices...

$$l^{\Rightarrow} \quad \in \quad \begin{cases} R \times C \;\longrightarrow\; S \times C & \text{state-based} \\ \Delta R \times C \;\longrightarrow\; S \times C & \text{delta-based} \\ (R \rightarrow R) \times C \;\longrightarrow\; S \times C & \text{operation-based} \end{cases}$$

- state-based: pass both changed and unchanged parts

Digression: State-based vs.Operation-Based

We've been assuming so far that the main arguments to the $I^{\Rightarrow}$ and $I^{\Leftarrow}$ functions were entire structures. Naturally, there are other choices...

$$I^{\Rightarrow} \quad \in \quad \left\{ \begin{array}{rcll} R \times C & \longrightarrow & S \times C & \text{state-based} \\ \Delta R \times C & \longrightarrow & S \times C & \text{delta-based} \\ (R \rightarrow R) \times C & \longrightarrow & S \times C & \text{operation-based} \end{array} \right.$$

intuitively

- state-based: pass both changed and unchanged parts
- delta-based: pass just changed parts

Digression: State-based vs.Operation-Based

We've been assuming so far that the main arguments to the $l^{\Rightarrow}$ and $l^{\Leftarrow}$ functions were entire structures. Naturally, there are other choices...

$$l^{\Rightarrow} \quad \in \quad \left\{ \begin{array}{rcll} R \times C & \longrightarrow & S \times C & \text{state-based} \\ \Delta R \times C & \longrightarrow & S \times C & \text{delta-based} \\ (R \rightarrow R) \times C & \longrightarrow & S \times C & \text{operation-based} \end{array} \right.$$

intuitively

- state-based: pass both changed and unchanged parts
- delta-based: pass just changed parts
- operation-based: pass the edit operation itself

Digression: State-based vs.Operation-Based

We've been assuming so far that the main arguments to the $I^{\Rightarrow}$ and $I^{\Leftarrow}$ functions were entire structures. Naturally, there are other choices...

$$
I^{\Rightarrow} \quad \in \quad \left\{ \begin{array}{rcl} R \times C & \longrightarrow & S \times C \\ \Delta R \times C & \longrightarrow & S \times C \\ (R \to R) \times C & \longrightarrow & S \times C \end{array} \right. \quad \begin{array}{l} \text{state-based} \\ \text{delta-based} \\ \text{operation-based} \end{array}
$$

intuitively

- state-based: pass both changed and unchanged parts
- delta-based: pass just changed parts
- operation-based: pass the edit operation itself

State-based and delta-based are fundamentally similar, while operation-based is a rather different animal.

Digression: Totality

The assumption that $l^{\Rightarrow}$ and $l^{\Leftarrow}$ are total functions is pretty strong:

- It means that our update translators must be able to handle *any update whatsoever*

Can we relax this restriction?

Digression: Totality

The assumption that $l^{\Rightarrow}$ and $l^{\Leftarrow}$ are total functions is pretty strong:

- It means that our update translators must be able to handle *any update whatsoever*

Can we relax this restriction?

Depends on the application!

- If our lenses are being used in an on-line setting, where edits are propagated immediately, totality is not critical
- However, in an off-line setting, arbitrary changes can accumulate before we get a chance to propagate them
  - Here, totality is really important

More Extensions...

Alignment

(The hard part...)

Alignment

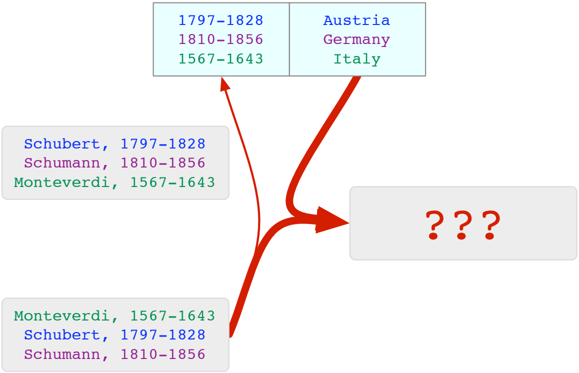
1797-1828	Austria
1810-1856	Germany
1567-1643	Italy

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

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

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

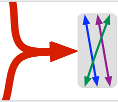
Alignment


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

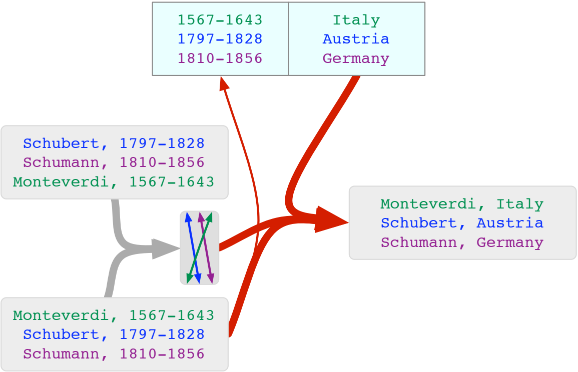
Schubert, 1797-1828
Schumann, 1810-1856
Monteverdi, 1567-1643



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

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.

Aligning Lenses (Sketch)

An aligning lens $l \in R \rightleftharpoons^{C} S$ comprises four functions

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

where...

(...same laws as before, adjusted to take alignment into account, plus some new ones describing how alignments are used...)

Status

Our POPL '08 paper shows how to handle the bushy and positional cases

- We are currently working on generalizing this framework to handle other kinds of alignment

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

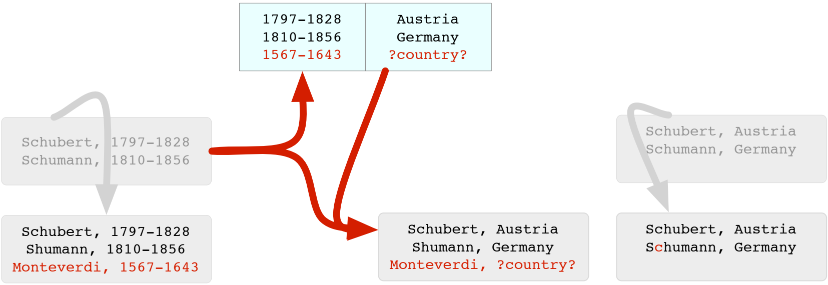
```
Schubert, 1797–1828
Schumann, 1810–1856
```

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

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

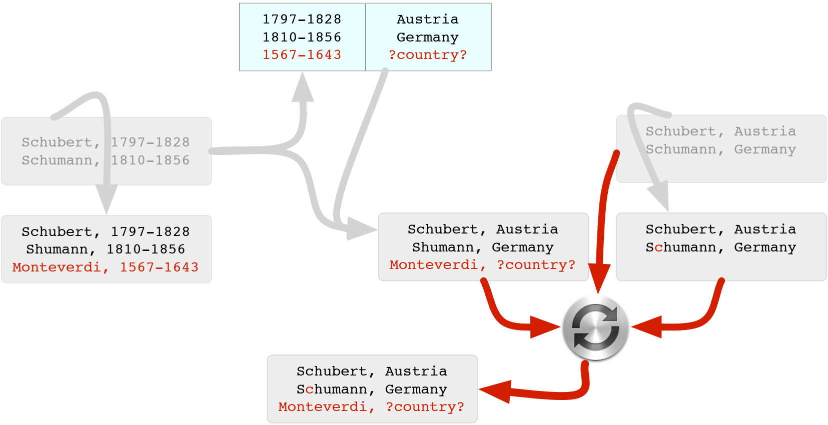
```
Schubert, Austria
Schumann, Germany
```

Synchronization



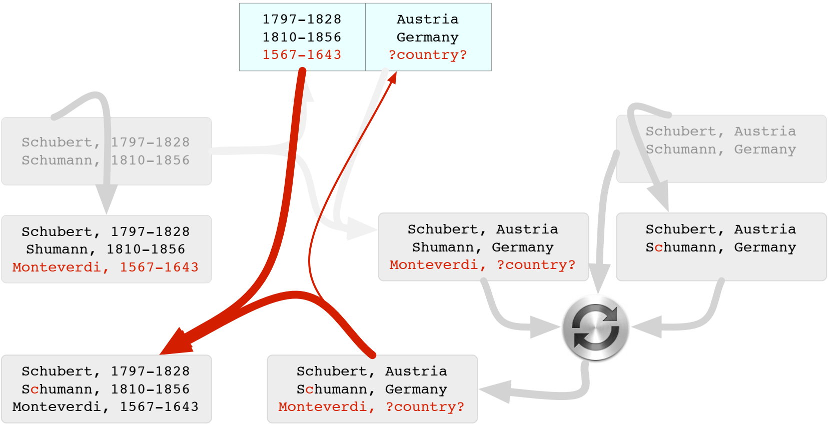
Step 1: Propagate edit from left to right with respect to existing complement (i.e., using the private information from the original right-hand structure)

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

Integrity

The Integrity Issue

- Propagating updates can cause changes in private data in the target structure!
- This can be prevented by adding another law requiring that updates always be propagated in an “undoable” way
  - or, equivalently, by requiring that translating updates not change the complement (cf. “constant complement approach to view update” from the database literature)
- However, this condition is very strong!
  - Imposing it in both directions means that the complement cannot ever be changed — i.e., it takes us back to bijective lenses
  - Even imposing it in just one direction prevents writing many useful transformations

Integrity Annotations

A more refined approach:

- Enrich the schemas of the two structures with integrity annotations specifying "levels of trustedness" of different parts of the data
- Impose new laws requiring that, during update translation, high-integrity data in the target structure be changed only as a result of edits to high-integrity regions of the source
- Refine the typing rules to track information flow; prove that the refined rules guarantee the new lens laws
- Correct handling of confidential information can be treated using the same mechanism

See our CSF 2009 paper for details.

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.

Wrapping Up...

How To Build a Bidirectional Programming Language

1. Think first about semantics
   - What are the inputs and outputs of update translation?
   - What laws capture our intuition of "well-behaved translations"?
2. Design bidirectional syntax
3. Define a static analysis (e.g., a typing relation) to check whether a given program satisfies the behavioral laws
4. Prove that the static analysis is correct
5. Implement
6. Test on practical examples
7. Repeat from (1) :-)

Simple structures, clean theory, real examples!

Deploying the Technology

How would these ideas be used in practice?

1. As a separate, domain-specific language
   - E.g., RedHat's Augeas tool is based directly on Boomerang
2. As an embedded language
   - A library of lenses and lens constructors
   - *lens* is an abstract type provided by the library
   - Each syntactic form becomes an operation in the API
     - Each *lens* object stores its domain and range types
     - Typing constraints are verified when lenses are constructed
   - Predefined constructors can be mixed with *ad hoc* (programmer-provided) lenses performing special / domain-specific transformations

Related Work

... Way too much even to summarize here

- See GRACE Workshop Report for extensive citations and discussion

Want to Play?

Our prototype Boomerang implementation is available for download...

- Source code (GPL)
- Binaries for Windows, OSX, Linux
- Tutorial and demos

A major new release is planned for this summer

Thank You!

Boomerang team: Aaron Bohannon, Davi Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, Benjamin Pierce, Alexandre Pilkiewicz, Alan Schmitt

Past contributors to the Harmony project: Ravi Chugh, Malo Denielou, Michael Greenwald, Owen Gunden, Martin Hofmann, Sanjeev Khanna, Keshav Kunal, Stéphane Lescuyer, Jon Moore, Jeff Vaughan, Zhe Yang

Resources: Papers, slides, sources, binaries, and demos:

http://www.seas.upenn.edu/~harmony/