Bidirectional Programming

Nate Foster

CS 4110
We *can* write complicated data transformations in C...
or Java...
or C++...
...or a tool specifically designed for the task!
Domain-specific languages

- Clean semantics
- Natural syntax
- Better tools
S \rightarrow V
Updated
V
update
       
S
 V

Updated
V

update
The View Update Problem

In databases, this is known as the view update problem.

[Bancilhon, Spryatos '81]
The View Update Problem In Practice

It also arises in **data converters and synchronizers**...

![Diagram showing replica and synchronization relationships](image)

[Foster, Greenwald, Pierce, Schmitt JCSS '07]— Harmony
The View Update Problem In Practice

...in picklers and unpicklers...

[Fisher, Gruber '05]— PADS
The View Update Problem In Practice

...in model-driven software development...

[Stevens ’07]— bidirectional model transformations
The View Update Problem In Practice

...in tools for managing operating system configurations...

Configuration file → Abstract tree

Updated file

edit operation

[Lutterkort '08]— Augeas
Problem

How do we write these bidirectional transformations?
Problem: Why is it hard?

We want updates to the view to be translated “exactly”...
Problem: Why is it hard?

We want updates to the view to be translated “exactly”...
Problem: Why is it hard?

...but some updates have *many* corresponding source updates...
Problem: Why is it hard?

...while others have none!
Possible Approaches

Bad: write the two transformations as separate functions.

- tedious to program
- difficult to get right
- a nightmare to maintain
Possible Approaches

**Good:** derive both transformations from the *same program*.

- **Clean semantics:** behavioral laws guide language design
- **Natural syntax:** parsimonious and compositional
- **Better tools:** type system guarantees well-behavedness
This talk: Goal

“Bidirectional programming languages are an effective and elegant means of describing updatable views”
This talk: Outline

1. Lenses
   ▶ Design goals
   ▶ Semantics

2. String Lenses
   ▶ Core operators
   ▶ Type system

3. Boomerang
   ▶ Ordered data
   ▶ Ignorable data
   ▶ Implementation & Applications

4. Ongoing Work
   ▶ Updatable Security Views

5. Future Directions
   ▶ Data provenance
   ▶ Model transformations
Lenses

‘‘Never look back unless you are planning to go that way”
—H D Thoreau
Terminology

put
Terminology

lens
Bidirectional vs. Bijective

Goal #1: lenses should be capable of hiding source data.
Bidirectional vs. Bijective

Goal #1: lenses should be capable of hiding source data.

• In general, get may be non-injective
• and so put needs to take the original source as an argument

(Of course, the purely bijective case is also very interesting.)
Choice of Put Function

Recall that for some view updates there are many corresponding source updates.
**Choice of Put Function**

**Goal #2:** programmers should be able to choose a `put` function that embodies an appropriate policy for propagating updates back to sources.

“Bidirectionalization” appears attractive...

...but does not provide a way to make this choice.
Recall that some view updates do not have any corresponding source updates.
Goal #3: the **put** function should be a *total* function, capable of doing *something* reasonable with every view and source.

Totality ensures that the view is a *robust abstraction*, but forces us to use an *extremely precise* type system.
Well-Behaved Lenses

A lens \( l \) mapping between a set \( S \) of sources and \( V \) of view is a pair of total functions

\[
\begin{align*}
\text{\textit{l.get}} & \in S \rightarrow V \\
\text{\textit{l.put}} & \in V \rightarrow S \rightarrow S
\end{align*}
\]

obeying “round-tripping” laws

\[
\begin{align*}
\text{\textit{l.get}} \ (\text{\textit{l.put}} \ \nu \ s) & = \nu \quad \text{(PutGet)} \\
\text{\textit{l.put}} \ (\text{\textit{l.get}} \ s) & = s \quad \text{(GetPut)}
\end{align*}
\]

for every \( s \in S \) and \( \nu \in V \).
Related Frameworks

Databases: many related ideas
- [Dayal, Bernstein ’82] “exact translation”
- [Bancilhon, Spryatos ’81] “constant complement”
- [Gottlob, Paolini, Zicari ’88] “dynamic views”

User Interfaces: [Meertens ’98] “constraint maintainers”

See [Foster et. al TOPLAS ’07] for details...
Related Languages

Harmony Group @ Penn
- [Foster et al. TOPLAS ’07] — trees
- [Bohannon, Pierce, Vaughan PODS ’06] — relations
- [Foster et al. JCSS ’07] — data synchronizer

Bijective languages
- [PADS Project @ AT&T] — picklers and unpicklers
- [Hosoya, Kawanaka ’06] — biXid
- [Braband, Møller, Schwartzbach ’05] — XSugar

Bidirectional languages
- [PSD @ Tokyo] — “bidirectionalization”, structure editors
- [Gibbons, Wang @ Oxford] — Wadler’s views
- [Voïgtaender ’09] — bidirectionalization “for free”
- [Stevens ’07] — lenses for model transformations
String Lenses

“The art of progress is to preserve order amid change and to preserve change amid order.”

—A N Whitehead
Why strings?

1. Simple setting $\rightarrow$ exposes fundamental issues
2. There’s a **lot** of string data in the world
3. Programmers are already comfortable with regular operators (union, concatenation, and Kleene star)
Why strings?
1. Simple setting → exposes fundamental issues
2. There’s a lot of string data in the world
3. Programmers are already comfortable with regular operators (union, concatenation, and Kleene star)
Example: Redacting Lens (Get)

*08:30 Coffee with Sara (Starbucks)
  12:15 PLClu (Seminar room)
*15:00 Workout (Gym)

08:30 BUSY
12:15 PLClu
15:00 BUSY
*08:30 Coffee with Sara (Starbucks)
12:15 PLClu (Seminar room)
*15:00 Workout (Gym)

08:30 BUSY
12:15 PLClu
15:00 BUSY
16:00 Meeting
Example: Redacting Lens (Put)

*08:30 Coffee with Sara (Starbucks)
  12:15 PLClub (Seminar room)
*15:00 Workout (Gym)

08:30 BUSY
12:15 PLClub
15:00 BUSY

*08:30 Coffee with Sara (Starbucks)
  12:15 PLClub (Seminar room)
*15:00 Workout (Gym)
  16:00 Meeting (Unknown)

08:30 BUSY
12:15 PLClub
15:00 BUSY
16:00 Meeting
Example: Redacting Lens (Definition)

(* regular expressions *)
let TEXT : regexp = (\[^\n\(\)] | "\" ( | "\") | "\\\\")*
let TIME : regexp = DIGIT{2} . COLON . DIGIT{2} . SPACE
let LOCATION : regexp = SPACE . LPAREN . TEXT . RPAREN

(* helper lenses *)
let public : lens =
  del SPACE .
  copy TIME .
  copy TEXT .
  default (del LOCATION) " (Unknown)"

let private : lens =
  del ASTERISK .
  copy TIME .
  default (TEXT . LOCATION <-> "BUSY") "Unknown (Unknown)"

let event : lens =
  (public | private) .
  copy NL

(* main lens *)
let redact : lens = event*
Example: Redacting Lens (Definition)

(* regular expressions *)
let TEXT : regexp = ([^-\n\(\)] | "\(" | "\)" | "\\\\")*
let TIME : regexp = DIGIT{2} . COLON . DIGIT{2} . SPACE
let LOCATION : regexp = SPACE . LPAREN . TEXT . RPAREN

(* helper lenses *)
let public : lens =
  del SPACE .
  copy TIME .
  copy TEXT .
  default (del LOCATION) " (Unknown)"

let private : lens =
  del ASTERISK .
  copy TIME .
  default (TEXT . LOCATION <-> "BUSY") "Unknown (Unknown)"

let event : lens =
  (public | private) .
  copy NL

(* main lens *)
let redact : lens = event*
Example: Redacting Lens (Definition)

(* regular expressions *)
let TEXT : regexp = ([^\n\(\)] | "\(" | "\)" | "\\\")*
let TIME : regexp = DIGIT{2} . COLON . DIGIT{2} . SPACE
let LOCATION : regexp = SPACE . LPAREN . TEXT . RPAREN

(* helper lenses *)
let public : lens =
    del SPACE .
    copy TIME .
    copy TEXT .
    default (del LOCATION) " (Unknown)"

let private : lens =
    del ASTERISK .
    copy TIME .
    default (TEXT . LOCATION <-> "BUSY") "Unknown (Unknown)"

let event : lens =
    (public | private) .
    copy NL

(* main lens *)
let redact : lens = event*
Example: Redacting Lens (Definition)

(* regular expressions *)
let TEXT : regexp = (\[^\n\(\)\] | "\"" | "\"" | "\\\")*
let TIME : regexp = DIGIT{2} . COLON . DIGIT{2} . SPACE
let LOCATION : regexp = SPACE . LPAREN . TEXT . RPAREN

(* helper lenses *)
let public : lens =
  del SPACE .
  copy TIME .
  copy TEXT .
  default (del LOCATION) " (Unknown)"

let private : lens =
  del ASTERISK .
  copy TIME .
  default (TEXT . LOCATION <-> "BUSY") "Unknown (Unknown)"

let event : lens =
  (public | private) .
  copy NL

(* main lens *)
let redact : lens = event*
Example: Redacting Lens (Definition)

(* regular expressions *)
let TEXT : regexp = ([^-\n\"] | "\"(" | "\")" | "\\\\\\")*
let TIME : regexp = DIGIT{2} . COLON . DIGIT{2} . SPACE
let LOCATION : regexp = SPACE . LPAREN . TEXT . RPAREN

(* helper lenses *)
let public : lens =
  del SPACE .
  copy TIME .
  copy TEXT .
  default (del LOCATION) " (Unknown)"

let private : lens =
  del ASTERISK .
  copy TIME .
  default (TEXT . LOCATION <-> "BUSY") "Unknown (Unknown)"

let event : lens =
  (public | private) .
  copy NL

(* main lens *)
let redact : lens = event*
\[ E \leftrightarrow d \] (Get)
$E \leftrightarrow d$  (Put)
Type system ensures that choice is deterministic.
Type system ensures that choice is deterministic.
/* (Get) */
/*  (Get) */
Type system ensures that strings are split the same way.
Type system ensures that strings are split the same way.

/* (Put) */
Type system ensures that strings are split the same way.
String Lens Type System

Based on regular expression types...
# String Lens Type System

Based on regular expression types...

<table>
<thead>
<tr>
<th>Expression</th>
<th>Rules</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy $E$ ∈ $\langle E \rangle$ ⇔ $\langle E \rangle$</td>
<td>$E$ ⇔ $d$ ∈ $\langle E \rangle$ ⇔ ${d}$</td>
</tr>
<tr>
<td>$l$ ∈ $S$ ⇔ $V$</td>
<td>$l_1$ ∈ $S_1$ ⇔ $V_1$</td>
</tr>
<tr>
<td>$d$ ∈ $\langle S \rangle$</td>
<td>$l_2$ ∈ $S_2$ ⇔ $V_2$</td>
</tr>
<tr>
<td>default $l$, $d$ ∈ $S$ ⇔ $V$</td>
<td>$(l_1 \cdot l_2)$ ∈ $S_1 \cdot S_2$ ⇔ $V_1 \cdot V_2$</td>
</tr>
<tr>
<td>$l_1$ ∈ $S_1$ ⇔ $V_1$</td>
<td>$l$ ∈ $S$ ⇔ $V$</td>
</tr>
<tr>
<td>$S_1 \cap S_2 = \emptyset$</td>
<td>$S_1 \cdot^! S_2$</td>
</tr>
<tr>
<td>$l_2$ ∈ $S_2$ ⇔ $V_2$</td>
<td>$l_1 \cdot l_2$ ∈ $S_1 \cup S_2$ ⇔ $V_1 \cup V_2$</td>
</tr>
<tr>
<td>$(l_1 \mid l_2)$ ∈ $S_1 \cup S_2$ ⇔ $V_1 \cup V_2$</td>
<td>$l \in S$ ⇔ $V$</td>
</tr>
</tbody>
</table>
| $S_1 \cdot^! S_2$ (or $S_1^! \cdot^! S_2$) means that the concatenation (or iteration) is unambiguous. | $S_1^! \cdot^! S_2$ (or $S_1^! \cdot^! S_2$) means that the concatenation (or iteration) is unambiguous.

$S_1 \cdot^! S_2$ (or $S_1^! \cdot^! S_2$) means that the concatenation (or iteration) is unambiguous.
String Lens Type System

Based on regular expression types...

\[
\begin{align*}
\text{copy } E & \in [E] \iff [E] \\
I & \in S \iff V \\
E & \leftrightarrow d \in [E] \iff \{d\} \\
E & \leftrightarrow d \in [E] \iff \{d\} \\
I_1 & \in S_1 \iff V_1 \\
S_1 \cap S_2 &= \emptyset \\
S_1 \cdot^! S_2 \\
I_2 & \in S_2 \iff V_2 \\
V_1 \cdot^! V_2 \\
(l_1 \cdot l_2) & \in S_1 \cdot S_2 \iff V_1 \cdot V_2 \\
(l_1 \mid l_2) & \in S_1 \cup S_2 \iff V_1 \cup V_2 \\
I & \in S \iff V \\
S_1 \cdot^* S_2 \quad V_1 \cdot^* V_2 \\
I^* & \in S^* \iff V^* \\
\end{align*}
\]

\(S_1 \cdot^! S_2\) (or \(S_1 \cdot^!\)) means that the concatenation (or iteration) is unambiguous.
## String Lens Type System

Based on regular expression types...

<table>
<thead>
<tr>
<th>Copy</th>
<th>$E \in [E] \iff [E]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I \in S \iff V$</td>
<td>$d \in [S] \iff$</td>
</tr>
<tr>
<td>Default</td>
<td>$I, d \in S \iff V$</td>
</tr>
</tbody>
</table>

| $l_1 \in S_1 \iff V_1$ | $S_1 \cdot ! S_2$ |
| $l_2 \in S_2 \iff V_2$ | $V_1 \cdot ! V_2$ |
| $(l_1 \cdot l_2) \in S_1 \cdot S_2 \iff V_1 \cdot V_2$ |

| $l_1 \in S_1 \iff V_1$ | $S_1 \cdot ! S_2$ |
| $l_2 \in S_2 \iff V_2$ | $V_1 \cdot ! V_2$ |
| $(l_1 \cdot l_2) \in S_1 \cdot S_2 \iff V_1 \cdot V_2$ |

$S_1 \cdot ! S_2$ (or $S_1 \cdot ! S_2$) means that the concatenation (or iteration) is unambiguous.

### Theorem

If $l \in S \iff V$ then $l$ is a well-behaved lens.
Comparison: Separate Functions

```ocaml
module B = Buffer
module R = Str
module L = List
module U = Unix
module R = Str

let prog = (
    let rec loop (h::t) =
        if h = [] then loop t
        else loop (h::t)
    in
    loop (
        let rec do_it s b =
            let buf, line_buf, aux_buf = B.create b_len, B.create 75, B.create 75 in
            B.reset b in
            B.add_buffer buf b in
            if B.length line_buf <> 0 then B.add_string buf s in
            loop [] l
        in
        let rec loop acc =
            function
            | h::t when k = fst h -> L.rev_append acc t
            | h::t -> loop (h::acc) t
            | [] -> L.rev_append acc []
            | Some(di,ki,ci),(dj,kj,cj)::rest when dj < di -> aux acc rest
            | Some(_,ki,ci),[] -> Some(ki,ci)
            | None,[] -> None
            in
        loop acc
    in
    loop (unwrap a) in
    let rec aux acc l =
        match acc, l with
        | Some(di,ki,ci),(dj,kj,cj)::rest when dj < di -> aux acc rest
        | Some(_,ki,ci),[] -> Some(ki,ci)
        | None,[] -> None
        | x -> aux (Some(x,ki,ci)) l
    in
    loop acc
```
Comparison: String Lens

Helpers

Source to View and View to Source
Boomerang

“Good men must not obey the laws too well”
—R W Emerson
Challenge: Ignorable Data

Many real-world data formats contain inessential data.

- whitespace, wrapping of long lines of text
- order of fields in record-structured data
- escaping of special characters
- aggregate values, timestamps, etc.

In practice, to handle these details, we need lenses that are well behaved modulo equivalence relations on the source and view.

\[
\begin{align*}
  l.\text{get} (l.\text{put} \ v \ s) & \sim_V \ v \\
  l.\text{put} (l.\text{get} \ s) & \sim_S \ s
\end{align*}
\]

(PutGet) (GetPut)
Quotient Lenses
Quotient Lenses

[Diagram of quotient lenses]

*
Challenge: Ordered Data

The lenses we have seen so far align data by position.

But, in practice, we often need to align data according to different criteria—e.g., by key.
Challenge: Ordered Data

The lenses we have seen so far align data by position.

But, in practice, we often need to align data according to different criteria—e.g., by key.
A Better Redact Lens

Similar to previous version but with a key annotations and a combinator (<l>) that identifies “chunks”

(* helper lenses *)
let location : lens = default (del LOCATION) " (Unknown)"

let public : lens =
  del SPACE .
  key TIME .
  copy TEXT .
  default (del LOCATION) " (Unknown)"

let private : lens =
  del ASTERISK .
  key TIME .
  default (TEXT . LOCATION <-> "BUSY") "Unknown (Unknown)" .

let event : lens =
  (public | private) .
  copy NL

(* main lens *)
let redact : lens = <~ event>*
A Better Redact Lens

Similar to previous version but with a key annotations and a combinator (<1>) that identifies “chunks”

(* helper lenses *)
let location : lens = default (del LOCATION) " (Unknown)"

let public : lens =
  del SPACE .
  key TIME .
  copy TEXT .
  default (del LOCATION) " (Unknown)"

let private : lens =
  del ASTERISK .
  key TIME .
  default (TEXT . LOCATION <-> "BUSY") "Unknown (Unknown)" .

let event : lens =
  (public | private) .
  copy NL

(* main lens *)
let redact : lens = <event>*
The **put** function works on a **dictionary** structure where chunks are organized by **key**.
Challenge: Language Design

Writing big programs only using combinators would not be fun!

Boomerang is a full-blown functional language over the base types string, regexp, lens, ...

Functional Programming Language

Lens primitives
Additional Features

Boomerang has many other lens primitives

• partition
• filter
• permute
• sort
• duplicate
• merge

• sequentially compose
• columnize
• normalize
• clobber
• probe
• etc.

and an extremely rich type system

• regular expression types
• dependent types
• refinement types

• polymorphism
• user-defined datatypes
• modules

implemented in hybrid style [Flanagan ’06][Findler, Wadler ’09]
Challenge: Typechecker Engineering

Typechecking uses many automata-theoretic operations.

- “Expensive” operations like intersection, difference, and interleaving are used often in practice
- Algorithms for checking ambiguity are computationally expensive rarely implemented

Implementation strategy:

- Compile compact automata [Brzozowski ’64]
- Aggressive memoization [Foster et al. PLAN-X ’07]
The Boomerang System

Lenses
- Bibliographies (BibTeX, RIS)
- Address Books (vCard, XML, ASCII)
- Calendars (iCal, XML, ASCII)
- Scientific Data (SwissProt, UniProtKB)
- Documents (MediaWiki, literate source code)
- Apple Preference Lists (e.g., iTunes)
- CSV

Libraries
- Escaping
- Sorting
- Lists
- XML

System
- Stable prototype complete
- Available under LGPL

Unison Integration
- On the way...
Boomerang in Industry

“A configuration API.”

Configuration file

Abstract tree

Updated file

edit operation
Boomerang in Industry

“a configuration API.”

Also used in

- **Puppet** – declarative configuration management tool
- **Show** – SQL-like queries on the filesystem
- **Netcf** – a network configuration library
Ongoing Work
Security Views

1. Confidentiality: get does not leak secret data
2. Integrity: put does not taint endorsed data
1. Confidentiality: get does not leak secret data
2. Integrity: put does not taint endorsed data

[Foster, Pierce, Zdancewic CSF ’09]
Requirements for Updatable Security Views

1. Confidentiality: get does not leak secret data
2. Integrity: put does not taint endorsed data

[Foster, Pierce, Zdancewic CSF ’09]
Non-interference

Requirements can be formulated as non-interference properties.
Non-interference

Requirements can be formulated as non-interference properties.
Secure Lenses

To distinguish high and low-security data we use equivalences

- $\sim_k$ — “agree on $k$-public data”
- $\approx_k$ — “agree on $k$-endorsed data”
Secure Lenses

To distinguish high and low-security data we use equivalences

- \( \sim_k \) — “agree on \( k \)-public data”
- \( \approx_k \) — “agree on \( k \)-endorsed data”

described using annotated regular expressions.

\[
R ::= \emptyset \mid u \mid R \cdot R \mid R|R \mid R^* \mid R : k
\]
Secure Lenses

To distinguish high and low-security data we use equivalences

- $\sim_k$ — “agree on $k$-public data”
- $\approx_k$ — “agree on $k$-endorsed data”

described using annotated regular expressions.

$$R ::= \emptyset \mid u \mid R \cdot R \mid R | R \mid R^* \mid R : k$$

A secure lens obeys refined laws:

\[
\frac{s \sim_k s'}{l \cdot \text{get} \; s \sim_k l \cdot \text{get} \; s'} \quad \text{(GetNoLeak)}
\]

\[
\frac{v \approx_k (l \cdot \text{get} \; s)}{l \cdot \text{put} \; v \; s \approx_k s} \quad \text{(GetPut)}
\]

(See paper for a dynamic approach to integrity tracking.)
Future Directions
Provenance is metadata that describes the origin and causal history of pieces of data.

In the context of lenses, provenance is useful:

• for fine-grained tracking of confidentiality and integrity
  [Foster, Green, Tannen PODS '08]
• for incremental view maintenance
• as an additional input to the put function
Provenance is metadata that describes the origin and causal history of pieces of data.
Data Provenance

Provenance is metadata that describes the origin and causal history of pieces of data.

In the context of lenses, provenance is useful

- for fine-grained tracking of confidentiality and integrity
  [Foster, Green, Tannen PODS ’08]
- for incremental view maintenance
- as an additional input to the put function
Much interest in the software engineering community in using lenses for bidirectional model transformations [Stevens ’07] [Czarnecki, Foster, Hu, Lämmel, Schürr, Terwilliger ICMT ’09]

Requires lenses for richer structures — e.g., graphs.
“Bidirectional programming languages are an effective and elegant means of describing updatable views”

Lenses

- Semantic space of well-behaved bidirectional transformations
- Provides foundation for bidirectional languages

Boomerang

- Language for lenses on strings
- Natural syntax based on regular operators
- Extensions to handle ordered and ignorable data
- Type system guarantees well-behavedness and totality

Implementation & Applications

- Lenses for a number of real-world formats
- Adoption in Augeas
- Updatable security views
Thank You!

Collaborators: Benjamin Pierce, Alexandre Pilkiewicz, Aaron Bohannon, Michael Greenberg, and Alan Schmitt.

Want to play? Boomerang is available for download.

- Source code (LGPL)
- Precompiled binaries for Linux, OS X, Windows
- Research papers
- Tutorial and demos

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