Bidirectional Programming Languages

Nate Foster
Cornell University

CS 2110
24 November 2015
Most programming languages, like C...
Java,
Python,
and C++ are general purpose.
I’m interested in designing languages that are specifically designed for particular tasks
Domain-specific languages

- Clean semantics
- Natural syntax
- Better tools
update data
convert clean exchange summarize
replicate integrate analyze
query modify curate synchronize
redact hide evolve
mashup transform reconcile
maintain
In databases, this is known as the view update problem.
It also arises in data converters and synchronizers...
The View Update Problem In Practice

...in picklers and unpicklers...

Binary file

In-memory representation

Updated binary file

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

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

[Stevens ‘07]— bidirectional model transformations
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
“Bidirectional languages are an effective and elegant means of describing updatable views”
Lenses

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

put

put
Terminology

lens
If \textit{get} is \textit{non-injective}, \textit{put} needs access to the \textit{original} source.

Of course, the purely bijective case is also interesting.
The Bijective Case

For bijective transformations...

...the desired behavior is obvious.
The General Case

But for bidirectional transformations...

... we need to identify conditions that allow us to

- recognize and reject bad (unreasonable) programs
- understand and predict behavior
An Unreasonable Example

project out string component
An Unreasonable Example
An Unreasonable Example

```
return a constant
```

```plaintext
foo 0
blech 5 bar
foo
```
An Unreasonable Example
The PutGet law

Principle:

*Updates should be “translated exactly” — i.e., to a source for which get yields exactly the updated target.*

Formally:

\[
\text{get} \ (\text{put} \ v \ s) \ = \ v
\]
A Debatable Example

project out and duplicate string component

foo 0

foo foo
A Debatable Example
A Debatable Example

propagate "newest" string
A Debatable Example

foo 0

bar 0

foo

bar

≠

foo

bar

≠

bar 0

foo

bar

≠

bar 0

bar
Another Unreasonable Example

`project out string component`

```
foo 5
```

```
foo
```
Another Unreasonable Example
Another Unreasonable Example

- propagate updated string
- always set numeric field to 0
Another Unreasonable Example
Another Unreasonable Example
The GetPut law

Principle: 

*If the view does not change, neither should the source.*

Formally:

\[
\text{put} \ (\text{get} \ s) \ s \ = \ s
\]
Another Debatable Example

project out string component

foo 0 → foo
Another Debatable Example

```
foo 0
```

```
foo
```

```
bar
```
Another Debatable Example

increment numeric component if string component has changed
Another Debatable Example

```
foo 0
bar 1
quux
bar
foo
```
translated updates produce "side effects" on source
Another Debatable Example

```
foo 0
bar 1
foo
bar
foo
```

restore original target
Another Debatable Example

Original source is not restored
The PutPut law

Principle:

*Each update should completely overwrite the effect of the previous one. In particular, the effect of two puts in a row should be the same as just the second.*

Formally:

\[
\text{put } v_2 \left( \text{put } v_1 s \right) = \text{put } v_2 s
\]
The PutPut law

Principle:

Each update should completely overwrite the effect of the previous one. In particular, the effect of two puts in a row should be the same as just the second.

Formally:

\[ \text{put } v_2 \left( \text{put } v_1 \ s \right) = \text{put } v_2 \ s \]

Nice properties:

- Ensures that every update can be “rolled back”
- Implies that \( S \) is isomorphic to \( V \times C \), for some \( C \)

Seems sensible. But do we want to always require it?
Another Example

select green records

```
foo 3
bar 5
baz 8
foo 3
baz 8
```
Another Example

```
select green records
```
delete baz

```
foo 3
bar 5
baz 8
foo 3
baz 8
foo 3
```
Another Example

select green records

delete baz
Another Example

```
foo 3
bar 5
baz 8
foo 3
baz 8
foo 3
bar 5
foo 3
baz 8
foo 3

select green records
delete baz
restore baz
```
Another Example

foo 3  
bar 5  
baz 8  

foo 3  
bar 5  
baz 8  

foo 3  
bar 5  
baz 8  

foo 3  
bar 5  
baz 8  

foo 3  
bar 5  
baz 8  

select green records

delete baz

restore baz

back to original source
Yet Another Example

```
foo  3  
bar  5  
baz  8  
```

select green records and delete numbers

```
foo
baz
```
Yet Another Example

- Select green records and delete numbers
- Delete baz
Yet Another Example

select green records and delete numbers

delete baz
Yet Another Example

select green records and delete numbers
delete baz
restore baz
Yet Another Example

- foo 3
- bar 5
- baz 8

- foo
- baz

- foo 3
- bar 5
- baz

- select green records and delete numbers

- delete baz

- restore baz

- number set to some default
A well-behaved lens \( l \) mapping between a set \( S \) of sources and \( V \) of view is a pair of total functions

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

obeying “round-tripping” laws

\[
\begin{align*}
{l}.\text{get} \ (l.\text{put} \ v \ s) & = v & \text{(PutGet)} \\
{l}.\text{put} \ (l.\text{get} \ s) \ s & = s & \text{(GetPut)}
\end{align*}
\]
A well-behaved lens \( l \) mapping between a set \( S \) of sources and \( V \) of view is a pair of total functions

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

obeying “round-tripping” laws

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

A very well-behaved lens \( l \) also obeys

\[
\text{\text{put}} v_2 (\text{\text{put}} v_1 s) = \text{\text{put}} v_2 s \quad \text{(PutPut)}
\]

A very well-behaved lens \( l \) also obeys
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”
- [Greenberg ‘07] DOM trees

Category Theory:
- [O’Connor ‘10] “co-algebras to monads”
- [Johnson ‘11] “algebras to co-monads”

See [Foster et. al TOPLAS ‘07] for a survey...
Related Languages

Harmony Group @ Penn

- [Hoffman et al. POPL ’10] — symmetric version
- [Foster et al. TOPLAS ’07] — trees
- [Bohannon et al. PODS ’06] — relations
- [Foster et al. JCSS ’07] — data synchronization

Bidirectional languages

- [PSD @ Tokyo] — “bidirectionalization”, structure editors
- [Gibbons, Wang @ Oxford] — Wadler’s views
- [Voïgtlaender ’09] — bidirectionalization “for free”
- [Stevens ’07] — lenses for model transformations
- [GSD @ Waterloo] — synchronizing software models
- [PADS Project @ AT&T] — picklers and unpicklers
- [Braband, Møller, Schwartzbach ’05] — XSugar
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 → 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)
Computation Model

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 (Gimme!)
15:30 PLD (Upson 5126)
*19:00 Workout (Noyes)

08:30 BUSY
15:30 PLD
19:00 BUSY
*08:30 Coffee with Sara (Gimme!)
  15:30 PLD (Upson 5126)
*19:00 Workout (Noyes)
Example: Redacting Lens (Put)

*08:30 Coffee with Sara (Gimme!)
15:30 PLD (Upson 5126)
*19:00 Workout (Noyes)

08:30 BUSY
15:30 PLD
19:00 BUSY

*08:30 Coffee with Sara (Gimme!)
12:15 PLDG (Upson 5126)
*19:00 Workout (Noyes)
21:00 Dinner (Unknown)

08:30 BUSY
15:30 PLDG
19:00 BUSY
21:00 Dinner
Example: Redacting Lens (Definition)

(* regular expressions *)
let TEXT : regexp = ([^\n\s()]|"\\("|"\")|"\\\\")*
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\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*
## String Lens Type System

Based on **regular expression** types...

<table>
<thead>
<tr>
<th>copy $E : [E] \leftrightarrow [E]$</th>
<th>$E \leftrightarrow d : [E] \leftrightarrow {d}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I : S \leftrightarrow V \quad d \in [S]$</td>
<td>$l_1 : S_1 \leftrightarrow V_1 \quad S_1 \cdot\lnot S_2$</td>
</tr>
<tr>
<td>default $I _d : S \leftrightarrow V$</td>
<td>$l_2 : S_2 \leftrightarrow V_2 \quad V_1 \cdot\lnot V_2$</td>
</tr>
<tr>
<td>$(I_1 \cdot I_2) : S_1 \cdot S_2 \leftrightarrow V_1 \cdot V_2$</td>
<td></td>
</tr>
</tbody>
</table>

$I_1 : S_1 \leftrightarrow V_1 \quad S_1 \cap S_2 = \emptyset$

$I_2 : S_2 \leftrightarrow V_2$

$(I_1 \mid I_2) : S_1 \cup S_2 \leftrightarrow V_1 \cup V_2$

$l : S \leftrightarrow V \quad S^{!*} \quad V^{!*}$

$l^* : S^* \leftrightarrow V^*$

$S_1 \cdot\lnot S_2$ (or $S^{!*}$) means that the concatenation (or iteration) is unambiguous.
### String Lens Type System

Based on regular expression types...

<table>
<thead>
<tr>
<th>copy $E : [E] \iff [E]$</th>
<th>$E \leftrightarrow d : [E] \iff {d}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l : S \iff V$</td>
<td>$d \in [S]$</td>
</tr>
<tr>
<td>default $l \cdot d : S \iff V$</td>
<td></td>
</tr>
<tr>
<td>$l_1 : S_1 \iff V_1$</td>
<td>$S_1 \cdot \dagger S_2$</td>
</tr>
<tr>
<td>$l_2 : S_2 \iff V_2$</td>
<td>$V_1 \cdot \dagger V_2$</td>
</tr>
<tr>
<td>$(l_1 \cdot l_2) : S_1 \cdot S_2 \iff V_1 \cdot V_2$</td>
<td></td>
</tr>
<tr>
<td>$l_1 : S_1 \iff V_1$</td>
<td>$S_1 \cap S_2 = \emptyset$</td>
</tr>
<tr>
<td>$l_2 : S_2 \iff V_2$</td>
<td></td>
</tr>
<tr>
<td>$(l_1 \mid l_2) : S_1 \cup S_2 \iff V_1 \cup V_2$</td>
<td></td>
</tr>
<tr>
<td>$S_1 \cdot \dagger S_2$ (or $S^i \cdot \dagger$) means that the concatenation (or iteration) is unambiguous.</td>
<td></td>
</tr>
</tbody>
</table>

### Theorem

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

let escape_codes s =
  (* dictionary lens helpers *)
  let uid () = exec "cat /dev/urandom|uuencode -m -|tail +2|tr -dc 'a-zA-Z0-9'|head -c25"
  let now () = exec "date +%H%M%S | tr -d '\n'"
  let today () = exec "date +%Y%m%d | tr -d '\n'"
  let codes = [('\', '\\'); ('(', '\('); (')', '\)')]
  let date = "[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]T[0-9][0-9][0-9][0-9][0-9][0-9][0-9][0-9]Z"
  let text = "[\^\n\n]*"
  let begin_event = rx "BEGIN:VEVENT"

(* regexps *)

let exec s =
  try
    let p = U.open_process_in s in
    let buf = B.create 17 in
    ignore (U.close_process_in p);
    with End_of_file -> ()
  with Error -> error "Couldn't find \"s\" in \"a\" (\"b\")\n"

let unescape codes s =
  let rec lookup k l =
    let m,n = String.length s1, String.length s2 in
    L.fold_left (aux None l') (aux None l) in
    for i=1 to m do
      for j=0 to n do
        if String.get s1 (pred i) = String.get s2 (pred j) then 0 else 1
      done;
    done;
  in
  aux_len + line_len in
  let line len = let n = B.length line_buf in if n=0 then n else succ n in
  let aux_len = B.length aux_buf in
  aux_len

if k = fst h -> L.rev_append acc t
[ ] -> L.rev_append acc [ ]
Some di,ki,ci,(dj,kj,cj)::rest when dj < di -> aux (Some(dj,kj,cj)) rest
Some(_,ki,ci),[] -> Some(ki,ci)
None,[] -> None

let put a c =
  line "SEQUENCE" "0";
  line "STATUS" "TENTATIVE";
  line "DESCRIPTION" "";
  line "TRANSP" "OPAQUE";
  line "UID" (uid ());
  line "LAST-MODIFIED" (today () ^ "T" ^ now () ^ "Z");
  line "CREATED" (today () ^ "T" ^ now () ^ "Z");
  line "DTSTAMP" (today () ^ "T" ^ now () ^ "Z");
  line "DTEND" (today () ^ "T" ^ t ^ "00Z");
  line "DTSTART" (today () ^ "T" ^ t ^ "00Z");

let remove k =
  let rec loop acc =
    if k = fst h then (h,t) in
    let rec loop (h:t) =
      let rec loop acc =
        if k = fst h then (h,t) in
        let rec loop (h:t) =
          if k = fst h then (h,t) in
          let rec loop (h:t) =
            if k = fst h then (h,t) in
            if m mod 2 = 0 then do_it "\n" line_buf;
          done;
        line "SEQUENCE" "0";
        line "STATUS" "TENTATIVE";
        line "DESCRIPTION" "";
        line "TRANSP" "OPAQUE";
        line "UID" (uid ());
        line "LAST-MODIFIED" (today () ^ "T" ^ now () ^ "Z");
        line "CREATED" (today () ^ "T" ^ now () ^ "Z");
        line "DTSTAMP" (today () ^ "T" ^ now () ^ "Z");
        line "DTEND" (today () ^ "T" ^ t ^ "00Z");
        line "DTSTART" (today () ^ "T" ^ t ^ "00Z");
        if m mod 2 = 0 then do_it "\n" line_buf;
      done;
    loop (h:t) in
      loop (h:t) in
        loop (h:t) in
          loop (h:t) in
            loop (h:t) in
              loop (h:t) in
                loop (h:t) in
                  loop (h:t) in
                    loop (h:t) in
                      loop (h:t) in
                        loop (h:t) in
                          loop (h:t) in
                            loop (h:t) in
                              loop (h:t) in
                                loop (h:t) in
                                  loop (h:t) in
                                    loop (h:t) in
                                      loop (h:t) in
                                        loop (h:t) in
                                          loop (h:t) in
                                            loop (h:t) in
                                              loop (h:t) in
                                                loop (h:t) in
                                                  loop (h:t) in
                                                    loop (h:t) in
                                                      loop (h:t) in
                                                        loop (h:t) in
                                                          loop (h:t) in
                                                            loop (h:t) in
                                                              loop (h:t) in
                                                                loop (h:t) in
                                                                  loop (h:t) in
                                                                    loop (h:t) in
                                                                      loop (h:t) in
 Adam

Forest
Lenses for Filestores

A **filestore** is a collection of directories, files, and symbolic links organized into a coherent data set.
**Filestores are Pervasive**

**Monitoring:**
- CoralCDN
- *Many* applications at AT&T

**Science:**
- Physics simulations
- Environmental data

**Ad hoc data:**
- Princeton student records
- Linux standard base
- Websites, repositories, etc.
Filestores vs. Databases

Database Challenges
- Up-front costs
- Loading overhead
- Loss of control

Filestore Challenges
- No documentation
- Must “roll own” tools
- No way to check for errors
- Difficult to maintain
- Large scale
Idea: lenses for filestores!

File system

In-memory representation

Updated file system

application update
Idea: lenses for filestores!

Generated Artifacts

- Representation type
- Metadata type
- Load function
- Store function
- Class instances for generic programming

```haskell
{forest}
type Top = [ s :: Site | s <- matches site_regexp ]
type Site = [ d :: Log  | d <- matches time_regexp ]
type Log = Directory
  { web is "coralwebsrv.log.gz" :: Gzip (File Coral),
    dns is "coraldnssrv.log.gz" :: Maybe (Gzip (File Ptext)),
    prb is "probed.log.gz" :: Maybe (Gzip (File Ptext))
    dmn is "corald.log.gz" :: Maybe (Gzip (File Ptext)) } |
```
Example: Universal Description

```
[forest]
  type Universal_d = Directory
  { ascii_files is [ f :: Text
      f <- matches (GL "*"),
      <| get_kind f_att == AsciiK |> ],
  binary_files is [ b :: Binary
      b <- matches (GL "*"),
      <| get_kind b_att == Binary |> ]
  directories is [ d :: Universal_d
      d <- matches (GL "*"),
      <| get_kind d_att == DirectoryK |> ]
  symLinks is [ s :: SymLink
      s <- matches (GL "*"),
      <| get_isSym s_att == True |> ] }
```
Theorem 1 (LoadStore)
\[
\begin{align*}
\mathcal{E}; r; s &\vdash \text{load } F \triangleright (v, d) \\
&\land \mathcal{E}; r; s \vdash \text{store } (F, v, d) \triangleright (F', \phi) \\
\end{align*}
\]
\Rightarrow (F = F') \land \phi(F')

Theorem 2 (StoreLoad)
\[
\begin{align*}
\mathcal{E}; r; s &\vdash \text{store } (F, v, d) \triangleright (F', \phi') \\
&\land \phi'(F') \\
&\land \mathcal{E}; r; s \vdash \text{load } F' \triangleright (v', d') \\
\end{align*}
\]
\Rightarrow (v', d') = (v, d).
Ongoing Work

**Lenses:** state-based → operation-based
- More efficient
- Increased precision
- Applications to audit
- Support for concurrent viewers

**Frenetic:** network programming language
- Automatic code partitioning
- Consistency abstractions
- Isolation/virtualization
- End-host integration

Want to play? Boomerang and Forest are available:

- Source code (under an open source license)
- Examples
- Research papers

http://www.cs.cornell.edu/~jnfoster/
Weakening the PutGet law

If we want to allow such behavior, we need to weaken PutGet. Here is one possibility:

\[
\begin{align*}
\text{put } v \ s &= s' & \text{get } s' &= v' \\
\text{put } v' \ s &= s'
\end{align*}
\]

Intuition:

Propagating an update may have “side-effects”, but only on the initial round-trip.

Similar idea in databases:

Propagating an update must have “minimal side-effects”.