# Adventures in Bidirectional Programming

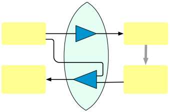

Benjamin Pierce

University of Pennsylvania



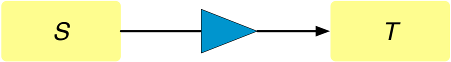
FSTTCS

December 2007

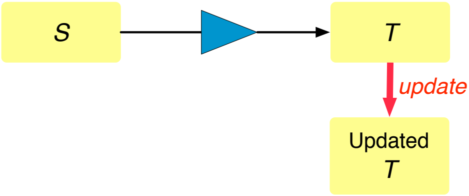
# Bidirectional Mappings

- Most programs work in one direction—from source to target



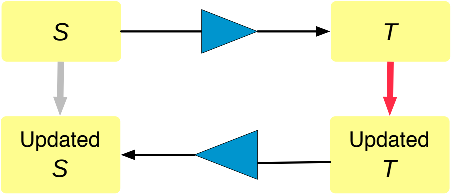
# Bidirectional Mappings

- Most programs work in one direction—from source to target
- But sometimes we want to update the target



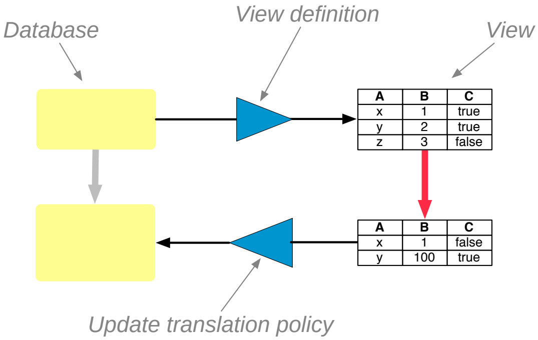
# Bidirectional Mappings

- Most programs work in one direction—from source to target
- But sometimes we want to update the target...
- ...and "translate" this update to obtain an appropriately updated source



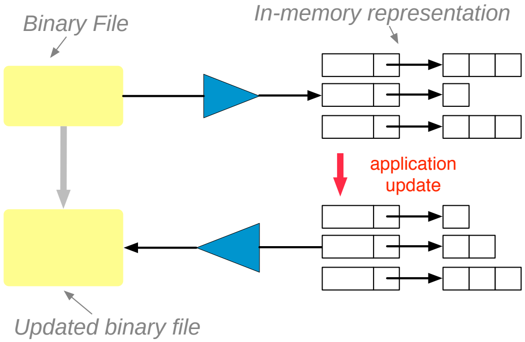
# The View Update Problem

This is called the view update problem in the database literature.



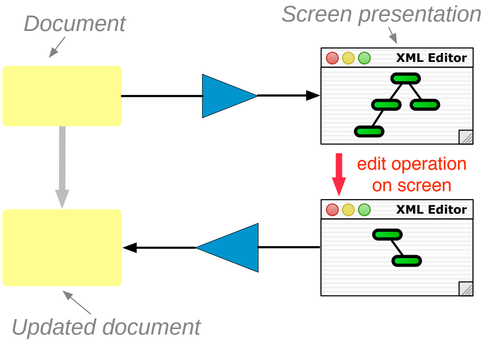
# The View Update Problem In Practice

It also arises with picklers and unpicklers...



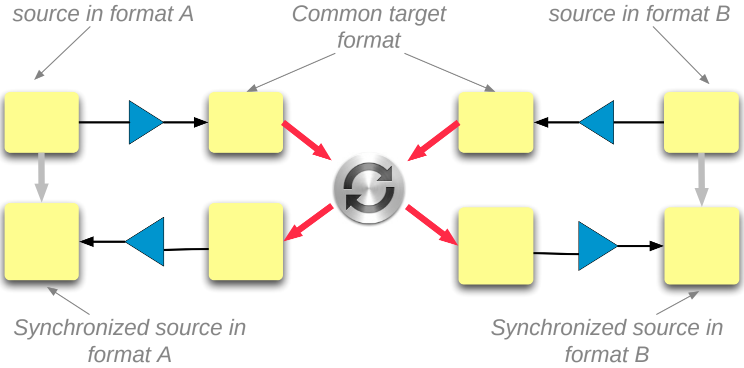
# The View Update Problem In Practice

...in structure editors...



# The View Update Problem In Practice

...and in data synchronizers, such as the Harmony system.



# Approaches to View Update

**Bad:** Write the two transformations as separate functions.

- Hard to write
- Harder to maintain

**Good:** Derive both from a single description.

# Approaches to View Update

**Bad:** Write the two transformations as separate functions.

- Hard to write
- Harder to maintain

**Good:** Derive both from a single description.

**Research challenge:** Find good ways to give such descriptions.

# Fertile Area for Research

- Complex design space, full of surprising constraints
  - Desiderata for particular cases often clear; general case often *un*clear
  - Interesting to see what can be done in a principled way
- Pragmatic state of the art is pretty bad
  - Bidirectional transformations common in software systems
  - Mostly hand-crafted
  - Bugs abound

Clean solutions — even partial solutions — welcome

# This Talk...

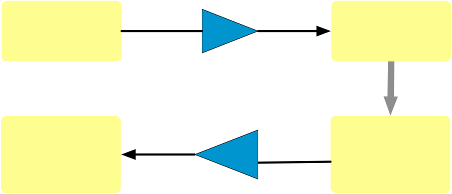
1. Introduces bidirectional programming languages
2. Gives some technical details of what we've achieved so far
3. Describes some current and future challenges

# Bidirectional Programming Languages

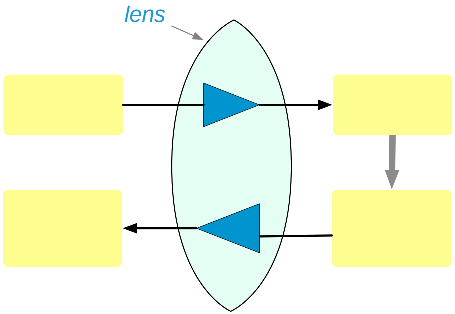
# A Linguistic Approach

- Clean semantic foundation
  - Behavioral laws guide language design
- Compositional syntax
  - Build complex bidirectional transformations out of simpler ones
- Expressive type system
  - Guarantee totality and well-behavedness by local static checks

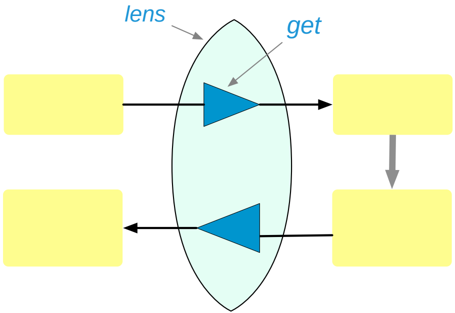
# Terminology



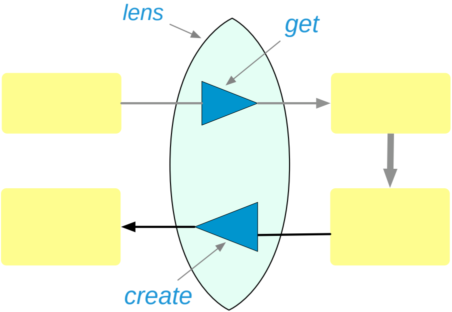
# Terminology



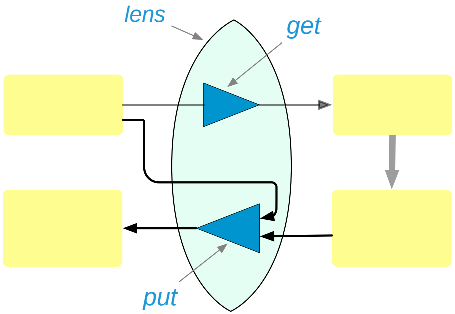
# Terminology



# Terminology



# Terminology



# Semantics

A lens $l$ from $S$ to $T$ is a triple of functions

$$\begin{array}{lcl} l.get & \in & S \to T \\ l.put & \in & T \to S \to S \\ l.create & \in & T \to S \end{array}$$

obeying three "round-tripping" laws:

$$l.put\ (l.get\ s)\ s = s \qquad \text{(GetPut)}$$

$$l.get\ (l.put\ t\ s) = t \qquad \text{(PutGet)}$$

$$l.get\ (l.create\ t) = t \qquad \text{(CreateGet)}$$

# String Lenses

# String Lenses

Data model: Strings over a finite alphabet

Computation model: Finite-state string transducers, written using regular-expression-like syntax

Type system: Regular languages (with interesting side conditions)

Why strings? Simple setting $\longrightarrow$ exposes fundamental issues

...and there's a lot of string data in the world...

...and programmers are familiar with finite-state transducers and regular expressions.

# Composer Lens (Get)

Source string:

```
"Benjamin Britten, 1913-1976, English"
```

Target string:

```
"Benjamin Britten, English"
```

# Composer Lens (Put)

Putting new target

```
"Benjamin Britten, British"
```

into original source

```
"Benjamin Britten, 1913-1976, English"
```

yields new source:

```
"Benjamin Britten, 1913-1976, British"
```

# Composer Lens (Definition)

```
let ALPHA : regexp = [A-Za-z ]+
let YEAR  : regexp = [0-9]{4}
let YEARS : regexp = YEAR . "-" . YEAR

let c : lens = cp ALPHA . cp ", "
             . del YEARS . del ", "
             . cp ALPHA
```

```
Benjamin Britten, 1913-1976, English
                  ⇕
       Benjamin Britten, English
```

# Copy

$$cp\ E \in [\![E]\!] \Longleftrightarrow [\![E]\!]$$

$$\begin{aligned} get\ s &= s \\ put\ t\ s &= t \\ create\ t &= t \end{aligned}$$

# Constant

$$\frac{u \in \Sigma^* \qquad v \in [\![E]\!]}{\mathit{const}\ E\ u\ v \in [\![E]\!] \Longleftrightarrow \{u\}}$$

$$\begin{array}{lcl} \mathit{get}\ s & = & u \\ \mathit{put}\ t\ s & = & s \\ \mathit{create}\ t & = & v \end{array}$$

# Constant (and Some Derived Forms)

$$\frac{u \in \Sigma^* \qquad v \in [\![E]\!]}{\mathit{const}\ E\ u\ v \in [\![E]\!] \Longleftrightarrow \{u\}}$$

$$\begin{aligned} \mathit{get}\ s &= u \\ \mathit{put}\ t\ s &= s \\ \mathit{create}\ t &= v \end{aligned}$$

$$E \leftrightarrow u \in [\![E]\!] \Longleftrightarrow \{u\}$$

$$E \leftrightarrow u = \mathit{const}\ E\ u\ (\mathit{choose}(E))$$

$$\mathit{del}\ E \in [\![E]\!] \Longleftrightarrow \{\epsilon\}$$

$$\mathit{del}\ E = E \leftrightarrow \epsilon$$

$$\mathit{ins}\ u \in \{\epsilon\} \Longleftrightarrow \{u\}$$

$$\mathit{ins}\ u = \epsilon \leftrightarrow u$$

# Concatenation

$$\frac{S_1 \cdot^! S_2 \qquad T_1 \cdot^! T_2 \qquad l_1 \in S_1 \Longleftrightarrow T_1 \qquad l_2 \in S_2 \Longleftrightarrow T_2}{l_1 \cdot l_2 \in S_1 \cdot S_2 \Longleftrightarrow T_1 \cdot T_2}$$

$$\begin{array}{lcl}
get\ (s_1 \cdot s_2) & = & (l_1.get\ s_1) \cdot (l_2.get\ s_2) \\
put\ (t_1 \cdot t_2)\ (s_1 \cdot s_2) & = & (l_1.put\ t_1\ s_1) \cdot (l_2.put\ t_2\ s_2) \\
create\ (t_1 \cdot t_2) & = & (l_1.create\ t_1) \cdot (l_2.create\ t_2)
\end{array}$$

$S_1 \cdot^! S_2$ means "the concatenation of $S_1$ and $S_2$ is uniquely splittable"

# Kleene-*

$$\frac{l \in S \Longleftrightarrow T \qquad S^{!*} \qquad T^{!*}}{l^* \in S^* \Longleftrightarrow T^*}$$

$$\begin{aligned}
get\ (s_1 \cdots s_n) &= (l.get\ s_1) \cdots (l.get\ s_n) \\
put\ (t_1 \cdots t_n)\ (s_1 \cdots s_m) &= (l.put\ t_1\ s_1) \cdots (l.put\ t_m\ s_m) \cdot \\
&\quad (l.create\ t_{m+1}) \cdots (l.create\ t_n) \\
create\ (t_1 \cdots t_n) &= (l.create\ t_1) \cdots (l.create\ t_n)
\end{aligned}$$

# Union

$$\frac{S_1 \cap S_2 = \emptyset \qquad l_1 \in S_1 \Longleftrightarrow T_1 \qquad l_2 \in S_2 \Longleftrightarrow T_2}{l_1 \mid l_2 \in S_1 \cup S_2 \Longleftrightarrow T_1 \cup T_2}$$

$$\begin{aligned}
get\ s &= \begin{cases} l_1.get\ s & \text{if } s \in S_1 \\ l_2.get\ s & \text{if } s \in S_2 \end{cases} \\
put\ t\ s &= \begin{cases} l_i.put\ t\ s & \text{if } s \in S_i \wedge t \in T_i \\ l_j.create\ t & \text{if } s \in S_i \wedge t \in T_j \setminus T_i \end{cases} \\
create\ a &= \begin{cases} l_1.create\ t & \text{if } t \in T_1 \\ l_2.create\ t & \text{if } t \in T_2 \setminus T_1 \end{cases}
\end{aligned}$$

# The Role of Types

The typing rules for these combinators are designed so that the target structure can be updated with no knowledge of the source structure or the transformation between them.

- *l.put* can be applied to any target structure belonging to the codomain of *l*
- every such *put* is guaranteed to succeed (i.e., *put* is a total function on the specified types)
- round-tripping laws are guaranteed to hold

I.e., the target is a closed view in the sense of Hegner.

# The Role of Types

The typing rules for these combinators are designed so that the target structure can be updated with no knowledge of the source structure or the transformation between them.

- *l.put* can be applied to any target structure belonging to the codomain of *l*
- every such *put* is guaranteed to succeed (i.e., *put* is a total function on the specified types)
- round-tripping laws are guaranteed to hold

I.e., the target is a closed view in the sense of Hegner.

Strong requirement, suitable for off-line applications.

- In on-line situations, weaker guarantees may be acceptable.

# The Role of Types

The requirement that well-typedness should guarantee totality forces us to use an extremely precise type system.

Pros:

- Types capture detailed structural constraints on source/target formats
- Typechecking exposes many programming errors that would be invisible to a coarser type system

Cons:

- Programmers sometimes have to work hard to make the typechecker happy
- Building an efficient typechecker is challenging...

# Typechecker Engineering

Using regular expressions as types demands serious care in the implementation:

- typechecking involves *many* automata-theoretic operations and tests
- algorithms for checking "unambiguous splittability" conditions are rarely implemented and computationally expensive
- "expensive" operations like union, difference, and interleaving are used heavily by programmers
  $\longrightarrow$ naive "determinization" of NFAs will lead to *huge* DFAs

# Typechecker Engineering

Short-term approach:

- NFA representation
- aggressive memoization of automata-theoretic operations, results of emptiness tests, etc. (see our [PLANX 07] paper)
- Good enough for our prototype

Longer-term approach:

- Compile regular expressions to DFAs using derivatives [Brzozowski 1964].
  - Challenge: splittability checking

# Boomerang

# Combinators $\longrightarrow$ Programming Language

Writing large programs using just these combinators would not be much fun!

- Need abstraction facilities, so we can build reusable libraries of parameterized lenses

# Combinators → Programming Language

Writing large programs using just these combinators would not be much fun!

- Need abstraction facilities, so we can build reusable libraries of parameterized lenses

Idea: Embed the combinators in a functional programming language...

# Boomerang

- Boomerang is a simply typed functional language over the base types `string`, `regexp`, `lens`, ...
- ...with primitives:

```
get : lens -> string -> string
put : lens -> string -> string -> string
create : lens -> string -> string

union : lens -> lens -> lens
concat : lens -> lens -> lens
star : lens -> lens
```

etc.

# Two-stage typechecking

- Typecheck initial functional program using these “rough types”
- Execute it
- During execution, lens-constructing operators like `union` and `star` perform precise typechecking according to the rules above

Similar to hybrid checking [Flanagan, POPL 06].

# Ordered Data

# Composers (Get)

Suppose we want to extend the lens to handle ordered lists of composers — i.e., so that

```
"Aaron Copland, 1910-1990, American
 Benjamin Britten, 1913-1976, English"
```

maps to

```
"Aaron Copland, American
 Benjamin Britten, English"
```

and vice versa.

# Composers (Lens)

```
let ALPHA : regexp = [A-Za-z ]+
let YEAR  : regexp = [0-9]{4}
let YEARS : regexp = YEAR . "-" . YEAR

let c : lens = cp ALPHA . cp ", "
             . del YEARS . del ", "
             . cp ALPHA


let cs : lens = cp "" | c . (cp "\n" . c)*
```

# Example (Put)

Putting new target

```
"Aaron Copland, American
 Benjamin Britten, British
 Alexandre Tansman, Polish"
```

into original source

```
"Aaron Copland, 1910-1990, American
 Benjamin Britten, 1913-1976, English"
```

yields new source:

```
"Aaron Copland, 1910-1990, American
 Benjamin Britten, 1913-1976, British
 Alexandre Tansman, 0000-0000, Polish"
```

# Kleene-* and Alignment

Unfortunately, there is a serious problem lurking here.

The *put* component of $l*$ splits its $T$ and $S$ inputs into sequences of elements

$$t = t_1 \;\; . \;\; t_2 \;\; . \;\; t_3 \ldots$$
$$s = s_1 \;\; . \;\; s_2 \;\; . \;\; s_3 \ldots$$

then invokes the *put* of $l$ on $t_1$ and $s_1$, on $t_2$ and $s_2$, etc., and then forms a list of the results.

A *put* function that works by position does not always give us what we want!

For example...

# A Bad Put

Putting

```
"Benjamin Britten, British
 Aaron Copland, American"
```

into the same input as above...

```
"Aaron Copland, 1910-1990, American
 Benjamin Britten, 1913-1976, English"
```

...yields a mangled result:

```
"Benjamin Britten, 1910-1990, British
 Aaron Copland, 1913-1976, American"
```

# A Serious Problem

It arises whenever lenses are used with ordered data and where updates can add, delete, and rearrange elements.

Our experience writing lenses for a variety of real-world data formats shows that it arises frequently in applications.

Neither our basic lenses nor any other variant of lenses in the literature gets it right.

# Dictionary Lenses

# A Way Forward

In the composers lens, we want the *put* function to match up lines with identical name components. It should never pass

```
"Benjamin Britten, British"
```

and

```
"Aaron Copland, 1910-1990, American"
```

to the same *put*!

To achieve this, the lens needs to identify:

- where are the re-orderable chunks in source and target;
- how to compute a key for each chunk.

# A Better Composers Lens

Similar to previous version but with a `key` annotation and a new combinator (`<c>`) that identifies the pieces of source and target that may be reordered.

```
let c = key ALPHA . cp ", "
      . del YEARS . del ", "
      . cp ALPHA
let cs = cp "" | <c> . (cp "\n" . <c>)*
```

The *put* function operates on a dictionary structure where source chunks are accessed by key.

# Semantics of Dictionary Lenses

A dictionary lens $l \in S \overset{R,D}{\Longleftrightarrow} T$ is a tuple of functions

$$
\begin{aligned}
l.get &\in S \rightarrow T \\
l.parse &\in S \rightarrow R \times D \\
l.key &\in T \rightarrow K \\
l.put &\in T \rightarrow R \times D \rightarrow S \times D \\
l.create &\in T \rightarrow D \rightarrow S \times D
\end{aligned}
$$

A dictionary lens can be coerced to a basic lens $\hat{l} \in S \Longleftrightarrow T$:

$$
\begin{aligned}
\hat{l}.get\ s &= l.get\ s \\
\hat{l}.put\ t\ s &= \pi_1(l.put\ t\ (l.parse\ s)) \\
\hat{l}.create\ t &= \pi_1(l.create\ t\ \{\})
\end{aligned}
$$

# The Essential Dictionary Lens

$$\frac{l \in S \overset{R,D}{\Longleftrightarrow} T}{<l> \in S \overset{\{\square\},D'}{\Longleftrightarrow} T}$$

$$
\begin{array}{lcl}
<l>.get\ s & = & l.get\ s \\
<l>.put\ t\ (\square, d) & = & \pi_1(l.put\ t\ (r, d'')), d' \\
 & & \text{where } (r, d''), d' = lookup\ (l.key\ t)\ d \\
<l>.parse\ s & = & \square, \{(l.key\ (l.get\ s)) \mapsto [s]\}
\end{array}
$$

# Some Applications of Dictionary Lenses

# Address Books (vCard Source)

```
BEGIN:VCARD
VERSION:3.0
N:Sanjiva Prasad;;;;
FN:Sanjiva Prasad
TEL;type=WORK;type=pref:+91 11 2659 1294
X-ABUID:827704A0-38A3-4034-84BF-BADFB87EB1E2 ABPerson
NOTE:FSTTCS
END:VCARD
BEGIN:VCARD
VERSION:3.0
N:Pierce;Benjamin C.;;;
FN:Benjamin C. Pierce
TEL;type=WORK:215 898-6222
TEL;type=HOME:215 732-4684
X-ABUID:87B85E7E-AB0F-4819-8647-0BD532019144 ABPerson
END:VCARD
```

# Address Books (vCard Source)

```
BEGIN:VCARD
VERSION:3.0
N:Sanjiva Prasad;;;;
FN:Sanjiva Prasad
TEL;type=WORK;type=pref:+91 11 2659 1294
X-ABUID:827704A0-38A3-4034-84BF-BADFB87EB1E2 ABPerson
NOTE:FSTTCS
END:VCARD

BEGIN:VCARD
VERSION:3.0
N:Pierce;Benjamin C.;;;
FN:Benjamin C. Pierce
TEL;type=WORK:215 898-6222
TEL;type=HOME:215 732-4684
X-ABUID:87B85E7E-AB0F-4819-8647-0BD532019144 ABPerson
END:VCARD
```

# Address Books (vCard Source)

```
BEGIN:VCARD
VERSION:3.0
N:Sanjiva Prasad;;;;
FN:Sanjiva Prasad
TEL;type=WORK;type=pref:+91 11 2659 1294
X-ABUID:827704A0-38A3-4034-84BF-BADFB87EB1E2 ABPerson
NOTE:FSTTCS
END:VCARD

BEGIN:VCARD
VERSION:3.0
N:Pierce;Benjamin C.;;;
FN:Benjamin C. Pierce
TEL;type=WORK:215 898-6222
TEL;type=HOME:215 732-4684
X-ABUID:87B85E7E-AB0F-4819-8647-0BD532019144 ABPerson
END:VCARD
```

# Address Books (vCard Source)

```
BEGIN:VCARD
VERSION:3.0
N:Sanjiva Prasad;;;;
FN:Sanjiva Prasad
TEL;type=WORK;type=pref:+91 11 2659 1294
X-ABUID:827704A0-38A3-4034-84BF-BADFB87EB1E2 ABPerson
NOTE:FSTTCS
END:VCARD

BEGIN:VCARD
VERSION:3.0
N:Pierce;Benjamin C.;;;
FN:Benjamin C. Pierce
TEL;type=WORK:215 898-6222
TEL;type=HOME:215 732-4684
X-ABUID:87B85E7E-AB0F-4819-8647-0BD532019144 ABPerson
END:VCARD
```

# Address Books (Target)

```
Sanjiva Prasad, +91 11 2659 1294 (w), FSTTCS (note)
Pierce, Benjamin C., 215 898-6222 (w), 215 732-4684 (h)
```

# Address Books (Lens)

```
let chunk : ? <-> AbsAddr =
   del "BEGIN:VCARD" . del NL .
   del "VERSION:3.0" . del NL .
   (name; key (atype name)) . del NL .
   (remove_item_numbers;
    filterwith Field_unnumbered entry) .
   del "END:VCARD" . del NL

let vcards = (<chunk> . ins NL) * . ws
```

# Bibliographic Data (BibTeX Source)

```
@inproceedings{utts07,
   author = {J. Nathan Foster
             and Benjamin C. Pierce
             and Alan Schmitt},
   title = {A {L}ogic {Y}our {T}ypechecker {C}an {C}ount {O}n:
           {U}nordered {T}ree {T}ypes in {P}ractice},
   booktitle = {PLAN-X},
   year = 2007,
   month = jan,
   pages = {80--90},
   jnf = "yes",
   plclub = "yes",
}
```

# Bibliographic Data (RIS Target)

```
TY - CONF
ID - utts07
AU - Foster, J. Nathan
AU - Pierce, Benjamin C.
AU - Schmitt, Alan
T1 - A Logic Your Typechecker Can Count On:
     Unordered Tree Types in Practice
T2 - PLAN-X
PY - 2007/01//
SP - 80
EP - 90
M1 - jnf: yes
M1 - plclub: yes
ER -
```

# Bibliographic Data (Lens)

```
let fields : lens =
  let non_author_fields =
   ( do_field (tag "T1") del "title" canonize_title canonize_title bare_value nl
   | do_dates
   | do_field "" del "pages" page_value page_value none nl
   | do_std_field (tag "T2") "booktitle" nl
   | do_std_field (tag "JO") "journal" nl
   | do_std_field (tag "VL") "volume" nl
   | do_std_field (tag "IS") "number" nl
   | do_std_field (tag "N1") "note" nl
   | do_std_field (tag "AD") "address" nl
   | do_std_field (tag "UR") "url" nl
   | do_std_field (tag "L1") "pdf" nl
   | do_std_field (tag "SN") "issn" nl
   | do_std_field (tag "PB") "publisher" nl
   | do_std_field (tag "N2") "abstract" nl
   | do_std_field (tag "T3") "series" nl
   | do_field (tag "M1")
      (fun (r:regexp) -> r . ins ": ")
      ([a-zA-Z]+ - ("author" | "title" | "booktitle" | "journal" | "volume" | "number"
                   | "note" | "pages" | "year" | "month" | "address" | "url" | "pdf"
                   | "issn" | "publisher" | "abstract" | "series" ))
      braced_value quoted_value bare_value nl)* in
  let author_field = do_field "" del "author" authors authors none nl in
  author_field . non_author_fields
```

# Genomic Data (SwissProt Source)

```
CC -!- INTERACTION: Self;
   NbExp=1; IntAct=EBI-1043398, EBI-1043398;
   Q8NBH6:-;
   NbExp=1;
   IntAct=EBI-1043398, EBI-1050185;
   P21266:GSTM3;
   NbExp=1;
   IntAct=EBI-1043398, EBI-350350;
```

# Genomic Data (UniProtKB Target)


```
<comment type="interaction">
  <interactant intactId="EBI-1043398"/>
  <interactant intactId="EBI-1043398"/>
  <organismsDiffer>false</organismsDiffer>
  <experiments>1</experiments>
</comment>
<comment type="interaction">
  <interactant intactId="EBI-1043398"/>
  <interactant intactId="EBI-1050185">
   <id>Q8NBH6</id>
  </interactant>
  <organismsDiffer>false</organismsDiffer>
  <experiments>1</experiments>
</comment>
<comment type="interaction">
  <interactant intactId="EBI-1043398"/>
  <interactant intactId="EBI-350350">
   <id>P21266</id>
   <label>GSTM3</label>
  </interactant>
  <organismsDiffer>false</organismsDiffer>
  <experiments>1</experiments>
</comment>
```

# Genomic Data (Lens)

```
let interaction =
  let id = escape_name [\n:;] in
  let esc = xml_esc [\n;()] in
  let esc_no_dash = xml_esc [\n; ()] in
  let esc_space = xml_esc [\n; ()] in
  let label = (esc_space . esc* . esc_space) | esc_no_dash in
  let prot = escape_name [\n,;""] in
  let inter =
   xml_opening_tag_with_att " " "comment" (xml_simple_attribute "type" "interaction") .
   (((( xml_gt . xml_nl .
    xml_tag " " "id" id . del ":" .
    (xml_tag " " "label" label | del "-") .
    xml_closing_tag " " "interactant".
    xml_tag " " "organismsDiffer"
       ("" <-> "false"
       |" (xeno)" <-> "true" ))
   | xml_sgt . xml_nl . xml_tag " " "organismsDiffer" ("Self" <-> "false")).
   del "; " .
   xml_tag " " "experiments" (del "NbExp=" . [0-9]+ .del "; "))
   (del "IntAct=" . xml_tag_with_single_att " " "interactant" "intactId" prot .
    del ", " .
    xml_begin_open_tag " " "interactant" . xml_space .
    xml_attribute "intactId" prot "" )). del ";" .
    xml_closing_tag " " "comment" in
  del_beg_line "CC" . del "-!- INTERACTION:" . del_spaces_bis .
  (inter . del_spaces_bis)* . inter . del \n
```

# Another Challenge: Aligning Documents

# Dealing with Documents

A key issue in view update is aligning the parts of source and target.

- Basic string lenses align by absolute position
- Dictionary lenses align chunks using keys

But for many interesting forms of data, it is difficult to identify "chunks" or "keys"

- raw text
- structured documents
- source code
- etc., etc.

# Example (Get)

Source document:

```
Benjamin Britten (1913-1976) wrote operas.
Aaron Copland (1910-1990) wrote orchestral
works.
```

Target document:

```
Benjamin Britten wrote operas.  Aaron Copland
wrote orchestral works.
```

# Example (Put)

Original source:

```
Benjamin Britten (1913-1976) wrote operas.
Aaron Copland (1910-1990) wrote orchestral
works.
```

Updated target:

```
Aaron Copland is best known for his
orchestral works, while Benjamin Britten
wrote operas of great power and beauty.
```

Updated source:

```
Aaron Copland (1910-1990) is best known for
his orchestral works, while Benjamin Britten
(1913-1976) wrote operas of great power and
beauty.
```

# Example (Put)

Original source:

```
Benjamin Britten (1913-1976) wrote operas.
Aaron Copland (1910-1990) wrote orchestral
works.
```

Updated target:

```
Aaron Copland is best known for his
orchestral works, while Benjamin Britten
wrote operas of great power and beauty.
```

Updated source:

```
Aaron Copland (1910-1990) is best known for
his orchestral works, while Benjamin Britten
(1913-1976) wrote operas of great power and
beauty.
```

# Challenges

- How to combine global alignment with structural transformations?
  - conditional?
  - composition?
- What is a good algorithm for performing global alignment?
  - ordinary `diff`?
  - fancier algorithm allowing "block moves"?

    (many now available, thanks to work in genome matching!)

# Another Challenge: Ignoring Inessential Differences

# Ignoring Inessential Differences

- The lens laws demand round-tripping "on the nose"

$$l.put\ (l.get\ s)\ s = s \qquad (\textsc{GetPut})$$

$$l.get\ (l.put\ t\ s) = t \qquad (\textsc{PutGet})$$

$$l.get\ (l.create\ t) = t \qquad (\textsc{CreateGet})$$

# Ignoring Inessential Differences

- The lens laws demand round-tripping "on the nose"
- But often this is more than we want
  - e.g., in XML documents, want to ignore most whitespace differences, ordering of attributes, etc.

$$l.put\ (l.get\ s)\ s = s \qquad \text{(GetPut)}$$

$$l.get\ (l.put\ t\ s) = t \qquad \text{(PutGet)}$$

$$l.get\ (l.create\ t) = t \qquad \text{(CreateGet)}$$

# Ignoring Inessential Differences

- The lens laws demand round-tripping "on the nose"
- But often this is more than we want
  - e.g., in XML documents, want to ignore most whitespace differences, ordering of attributes, etc.
- Idea: Loosen lens laws to hold only "up to an equivalence"

$$l.put\ (l.get\ s)\ s \sim s \qquad \text{(GetPut)}$$

$$l.get\ (l.put\ t\ s) \sim t \qquad \text{(PutGet)}$$

$$l.get\ (l.create\ t) \sim t \qquad \text{(CreateGet)}$$

# Ignoring Inessential Differences

- The lens laws demand round-tripping "on the nose"
- But often this is more than we want
  - e.g., in XML documents, want to ignore most whitespace differences, ordering of attributes, etc.
- Idea: Loosen lens laws to hold only "up to an equivalence"

$$t \sim l.get\ s \Longrightarrow l.put\ t\ s \sim s \qquad (\textsc{GetPut})$$

$$l.get\ (l.put\ t\ s) \sim t \qquad (\textsc{PutGet})$$

$$l.get\ (l.create\ t) \sim t \qquad (\textsc{CreateGet})$$

# Ignoring Inessential Differences

- The lens laws demand round-tripping "on the nose"
- But often this is more than we want
  - e.g., in XML documents, want to ignore most whitespace differences, ordering of attributes, etc.
- Idea: Loosen lens laws to hold only "up to an equivalence"
- Problem: Not clear how composition should work

$$t \sim l.get\ s \implies l.put\ t\ s \sim s \qquad \text{(GetPut)}$$

$$l.get\ (l.put\ t\ s) \sim t \qquad \text{(PutGet)}$$

$$l.get\ (l.create\ t) \sim t \qquad \text{(CreateGet)}$$

# Another Challenge: Different Data Models

# Some Other Lens Languages

[POPL 05, PLANX 07]

Data model: Trees (XML)

Computation model: Local tree transformations plus mapping, conditionals, composition, recursion.

Types: Regular tree languages.

[Bohannon et al PODS 06]

Data model: Relations

Computation model: Relational algebra, augmented with extra parameters to determine *put* behavior.

Types: Schemas with functional dependencies.

# Open questions

## Streaming data

- Process source and target incrementally
  - e.g., SwissProt sources are about 1Gb!
  - [cf. recent work by Alexandre Pilkiewicz]

## Graphs

- e.g., UML models
- Issue: What is a nice *compositional* language for describing graph transformations?

(Any graph transformation experts in the audience?)

# Wrapping Up...

# Related Work

- Semantic Framework — *many* related ideas in database literature
  - [Dayal, Bernstein '82] "exact translation"
  - [Bancilhon, Spryatos '81] "translators under constant complement"
  - [Gottlob, Paolini, Zicari '88] "dynamic views"
- Bijective languages — *many*
- Bidirectional languages
  - [Meertens] — language for constaint maintainers; similar behavioral laws
  - [Hu, Mu, Takeichi, *et al.*] — several languages for structured document editors

See our TOPLAS paper for details...

# Want to Play?

Our prototype Boomerang implementation is now available for download...

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

# Thank You!

Recent collaborators on this work: Aaron Bohannon, Nate Foster, Michael Greenberg, Alexandre Pilkiewicz, Alan Schmitt

Other Harmony contributors: 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/`

# Extra Slides

# Refined Semantics

# Quasi-Obliviousness

We want a property to distinguish the behavior of the first composers lens from the version with chunks and keys.

Intuition: the *put* function is agnostic to the order of chunks having different keys.

Let $\sim \subseteq S \times S$ be the equivalence relation that identifies sources up to key-respecting reorderings of chunks.

The dictionary composers lens obeys

$$\frac{s \sim s'}{l.put\ t\ s = l.put\ t\ s'} \qquad \text{(EQUIVPUT)}$$

but the basic lens does not.

# Quasi-Obliviousness

More generally we can let $\sim$ be an arbitrary equivalences on $S$.

The EquivPut law characterizes some important special cases of lenses:

- Every lens is quasi-oblivious wrt the identity relation.
- Bijective lenses are quasi-oblivious wrt the total relation.
- For experts: Recall the PutPut law:

$$put(t_2, put(t_1, s)) = put(t_2, s)$$

  which captures the notion of "constant complement" from databases. A lens obeys this law iff each equivalence classes of the coarsest $\sim$ maps via *get* to $T$.

# Another Challenge:
# Higher-Level Syntax

# Syntax

Writing finite-state transducers as annotated regular expressions is simple, natural, and familiar.

But not always convenient...

- some transformations (e.g., lexing) are simpler to express by writing FSAs directly
- others are naturally written using some form of binding