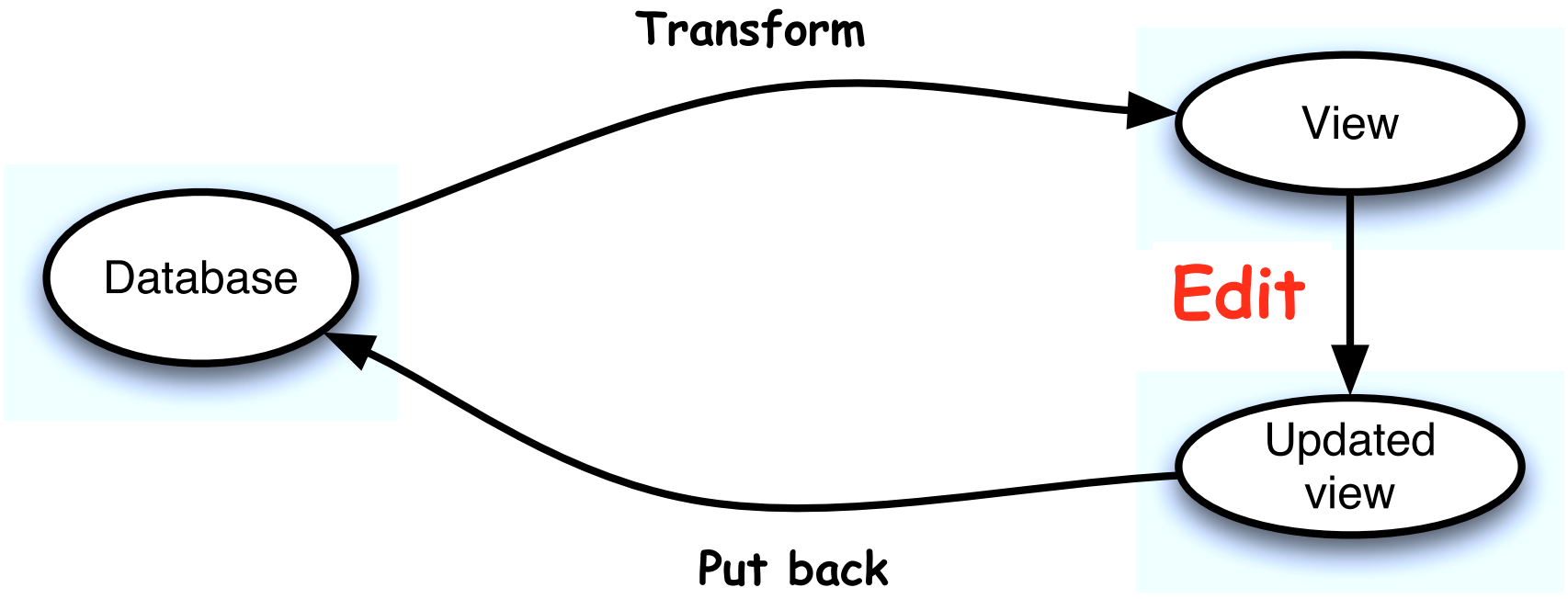
Combinators for Bi-Directional Tree Transformations:

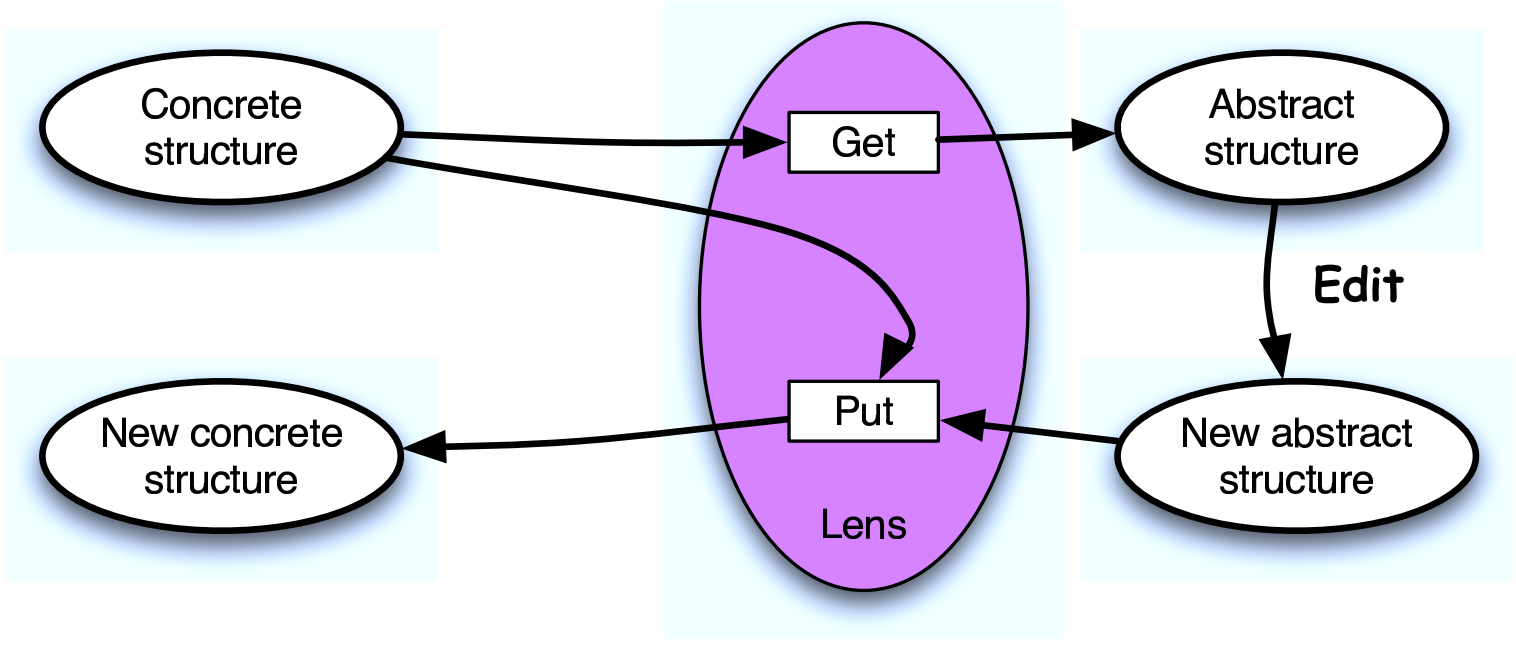
A Linguistic Approach to the View Update Problem

J. Nathan Foster (Penn),
Michael B. Greenwald (Lucent),
Jonathan Moore (Penn),
Benjamin C. Pierce (Penn)
Alan Schmitt (INRIA)

The View Update Problem



View Update, Functionally



Terminology: A lens is a pair of a get function from concrete to abstract structures and a put function from abstract plus concrete back to concrete.

A Small Example

Example

Suppose we have an address book, represented as a tree...

$$\left\{\begin{array}{l}\texttt{Pat} \mapsto \left\{\begin{array}{l}\texttt{Phone} \mapsto \left\{\texttt{333-4444} \mapsto \{\!\}\right\} \\ \texttt{URL} \mapsto \left\{\texttt{http://pat.com} \mapsto \{\!\}\right\}\end{array}\right\} \\ \texttt{Chris} \mapsto \left\{\begin{array}{l}\texttt{Phone} \mapsto \left\{\texttt{888-9999} \mapsto \{\!\}\right\} \\ \texttt{URL} \mapsto \left\{\texttt{http://chris.org} \mapsto \{\!\}\right\}\end{array}\right\}\end{array}\right\}$$

(We work, throughout, with unordered, edge-labeled trees with all edges from a node labeled distinctly—i.e., a tree is a finite function from labels to subtrees. We draw trees sideways.)

Example

... and we want to edit just the names and phone numbers.

Example

We would use a lens that maps from full address books to phone-numbers-only address books. Its get component maps

$$\left\{\left|\begin{array}{l}\texttt{Pat} \mapsto \left\{\left|\begin{array}{l}\texttt{Phone} \mapsto \left\{\texttt{333-4444} \mapsto \{\}\right\} \\ \texttt{URL} \mapsto \left\{\texttt{http://pat.com} \mapsto \{\}\right\}\end{array}\right|\right\} \\ \texttt{Chris} \mapsto \left\{\left|\begin{array}{l}\texttt{Phone} \mapsto \left\{\texttt{888-9999} \mapsto \{\}\right\} \\ \texttt{URL} \mapsto \left\{\texttt{http://chris.org} \mapsto \{\}\right\}\end{array}\right|\right\}\end{array}\right|\right\}$$

into:

$$\left\{\left|\begin{array}{l}\texttt{Pat} \mapsto \left\{\texttt{333-4444} \mapsto \{\}\right\} \\ \texttt{Chris} \mapsto \left\{\texttt{888-9999} \mapsto \{\}\right\}\end{array}\right|\right\}$$

Example

Its put component maps the edited abstract tree

$$\left\{\begin{array}{l}\texttt{Pat} \mapsto \left\{\texttt{333-4321} \mapsto \{\}\right\} \\ \texttt{Jo} \mapsto \left\{\texttt{555-6666} \mapsto \{\}\right\}\end{array}\right\}$$

together with the original concrete tree to a new concrete tree:

$$\left\{\begin{array}{l}\texttt{Pat} \mapsto \left\{\begin{array}{l}\texttt{Phone} \mapsto \left\{\texttt{333-4321} \mapsto \{\}\right\} \\ \texttt{URL} \mapsto \left\{\texttt{http://pat.com} \mapsto \{\}\right\}\end{array}\right\} \\ \texttt{Jo} \mapsto \left\{\begin{array}{l}\texttt{Phone} \mapsto \left\{\texttt{555-6666} \mapsto \{\}\right\} \\ \texttt{URL} \mapsto \left\{\texttt{http://google.com} \mapsto \{\}\right\}\end{array}\right\}\end{array}\right\}$$

Some Questions

1. What are lenses, exactly? What properties must a pair of functions have to be called a "well-behaved" lens?

2. How can we construct well-behaved lenses?
   - Write by hand and prove well-behavedness :-|
   - Design a new programming language :-)

3. Can we use lenses to do useful things?

Contributions

1. A natural semantic space of well-behaved lenses.

2. A domain-specific programming language in which every well-typed expression denotes a well-behaved lens.

```
map (filter {Phone} {URL = http://google.com})
```

3. A concrete application of this programming language in a generic data synchronization tool.

What are Lenses?

Semantics of Lenses

Let's begin by forgetting about trees and just thinking about what kinds of properties we would want lenses to have in general.

Let C be some set of "concrete structures" and A a set of "abstract structures."

Lenses, Formally

A (total) well-behaved lens l from C to A is a pair of functions

- $l\nearrow$ from C to A (get)
- $l\searrow$ from $A \times C$ to C (put)

satisfying two laws:

1. $l \searrow (l \nearrow c,\ c) = c$ (GETPUT)
2. $l \nearrow (l \searrow (a,\ c)) = a$ (PUTGET)

We write $C \Longleftrightarrow A$ for the set of well-behaved lenses from C to A.

Properties of Lenses

Some useful properties follow directly from the definition.

Theorem: The put component of a well-behaved lens l is always injective in its abstract argument:

$$a_1 \neq a_2 \quad \Longrightarrow \quad l\searrow(a_1, c) \neq l\searrow(a_2, c).$$

Theorem: If $p \in A \times C \to C$ is injective in its abstract argument, then there is a unique $g \in C \to A$ such that (g, p) is a total well-behaved lens.

I.e., we can design lenses by thinking just about injective put functions.

Creation

There are cases where we want to apply the put function of a lens, but where no old concrete structure is available — as we saw with `Jo`'s `URL` in the example.

We deal with this by enriching C and A with a special placeholder Ω ("missing").

Intuitively, $l \searrow (a, \Omega)$ means "create a new concrete structure using the information in the abstract structure a."

Creation

By convention, Ω is only used in an interesting way when it is the second argument to the put function. In all the lenses that we define later, we maintain the invariants that

1. $l \nearrow \Omega = \Omega$,
2. $l \searrow (\Omega, c) = \Omega$ for any c,
3. $l \nearrow c \neq \Omega$ for any $c \neq \Omega$, and
4. $l \searrow (a, c) \neq \Omega$ for any $a \neq \Omega$ and any c (including Ω).

We write $C \stackrel{\Omega}{\Longleftrightarrow} A$ for the set of total well-behaved lenses from $C \cup \{\Omega\}$ to $A \cup \{\Omega\}$ obeying these conventions.

Partiality

The real story is a little more complicated because we want to define lenses over trees by recursion — as limits of sequences of partially defined lenses.

In the paper, we define suitable notions of partial well-behaved lenses, ordering, and limits, and show that the set of partial well-behaved lenses forms a CPO — i.e., a setting in which we can solve recursive equations.

I'll elide these details here and talk only about total lenses.

Variation: One more lens law

There are several other laws we could consider imposing. Here is one particularly natural one:

$$l \searrow (a_1,\ (l \searrow (a_2,\ c))) = l \searrow (a_2,\ c) \qquad \text{(PUTPUT)}$$

However, one of our most important combinators (map) does not satisfy this law.

Fortunately, GETPUT and PUTGET have proved strong enough in practice to place useful constraints on the design of lens primitives.

Variation: Weaker Lens Laws

Weaker lens laws have also been advocated. For example, [Hu, Mu, and Takeichi] propose these:

$$l \nearrow l \searrow (a, c) = a \text{ where } a = l \nearrow c \qquad \text{(GETPUTGET)}$$

$$l \searrow (l \nearrow c', c') = c' \text{ where } c' = l \searrow (a, c) \qquad \text{(PUTGETPUT)}$$

However, the reason these weak laws are needed in their setting is to allow a primitive (a form of copying) that is also not total. Since we want all lenses to be total, we don't want this primitive, so we can keep the stronger laws.

Combinators for Lenses

A Language of Lenses

Question: How do we make it easy for people to write well-behaved lenses?

Answer: By defining a domain-specific language in which all well-typed expressions denote well-behaved lenses over trees.

The language is roughly similar to a combinator-style ("point-free") functional language like Backus's FP, but with many novel details.

Design Constraints

1. All primitive lenses must be well-behaved and total
2. All lens combinators must map total well-behaved lenses to total well-behaved lenses (perhaps subject to some side conditions)
3. These side conditions must be compositional — i.e., the well-formedness of a composite lens must always follow from the types of its component lenses.

(Caveat: as might be expected... for recursive lenses, well-behavedness follows compositionally, but totality requires some global reasoning.)

Identity

In the get direction, the identity lens copies its concrete argument.
In the put direction, it copies its abstract argument (and ignores its concrete argument).

$$\begin{array}{rcl} \mathtt{id}\nearrow c & = & c \\ \mathtt{id}\searrow(a,\, c) & = & a \end{array}$$

$$\forall C.\quad \mathtt{id} \in C \overset{\Omega}{\Longleftrightarrow} C$$

Constant

In the get direction, the `const` lens discards its concrete argument and returns a fixed abstract structure v.
In the put direction, it restores its constant argument (unless this is Ω, in which case it returns a default structure d).

$$
\begin{array}{rcll}
(\texttt{const}\ v\ d)\nearrow c & = & v & \\
(\texttt{const}\ v\ d)\searrow (a, c) & = & c & \textsf{if } c \neq \Omega \\
 & & d & \textsf{if } c = \Omega
\end{array}
$$

$$
\forall C.\ \forall v.\ \forall d{\in}C.\quad \texttt{const}\ v\ d \in C \overset{\Omega}{\Longleftrightarrow} \{v\}
$$

Hoist

In the get direction, the `hoist` lens clips out a top-level edge labeled n. In the put direction, it pushes the abstract tree under an edge labeled n.

$$\begin{array}{rcll} (\texttt{hoist}\ n) \nearrow c & = & t & \text{if } c = \{\!|n \mapsto t|\!\} \\ (\texttt{hoist}\ n) \searrow (a, c) & = & \{\!|n \mapsto a|\!\} & \end{array}$$

$$\forall C.\ \forall n \in \mathcal{N}.\quad \texttt{hoist}\ n \in \{\!|n \mapsto C|\!\} \overset{\Omega}{\Longleftrightarrow} C$$

Plunge

The `plunge` lens does the opposite of `hoist`.

$$
\begin{array}{rcll}
(\texttt{plunge}\ n) \nearrow c & = & \{\!|n \mapsto c|\!\} & \\
(\texttt{plunge}\ n) \searrow (a, c) & = & t & \text{if } a = \{\!|n \mapsto t|\!\} \\
\hline
\multicolumn{4}{c}{\forall C.\ \forall n \in \mathcal{N}.\quad \texttt{plunge}\ n \in C \overset{\Omega}{\Longleftrightarrow} \{\!|n \mapsto C|\!\}}
\end{array}
$$

Composition

The composite lens $l;k$ applies l and k sequentially.

$$
\begin{array}{rcl}
(l;k)\nearrow c & = & k\nearrow(l\nearrow c) \\
(l;k)\searrow(a,\,c) & = & l\searrow(k\searrow(a,\,l\nearrow c),\,c)
\end{array}
$$

$$
\begin{array}{l}
\forall A,B,C.\ \forall l\in C \overset{\Omega}{\Longleftrightarrow} B.\ \forall k\in B \overset{\Omega}{\Longleftrightarrow} A. \\
\qquad l;k\in C \overset{\Omega}{\Longleftrightarrow} A
\end{array}
$$

Map

The lens `map` l "applies l to all immediate children."

getting $\left\{\begin{vmatrix} n_1 \mapsto t_1 \\ \dots \\ n_k \mapsto t_k \end{vmatrix}\right\}$ yields $\left\{\begin{vmatrix} n_1 \mapsto l \nearrow t_1 \\ \dots \\ n_k \mapsto l \nearrow t_k \end{vmatrix}\right\}$

putting $\left\{\begin{vmatrix} n_1 \mapsto t_1 \\ \dots \\ n_k \mapsto t_k \end{vmatrix}\right\}$ into $\left\{\begin{vmatrix} n_1 \mapsto t'_1 \\ \dots \\ n_k \mapsto t'_k \end{vmatrix}\right\}$ yields $\left\{\begin{vmatrix} n_1 \mapsto l \searrow (t_1, t'_1) \\ \dots \\ n_k \mapsto l \searrow (t_k, t'_k) \end{vmatrix}\right\}$

More on Map

In the general case, we must consider what happens when the domains of the abstract and concrete trees are different.

- Children missing from the abstract tree may be deleted from the final concrete tree.
- Children added to the abstract tree must be created in the resulting concrete tree. We can achieve this by putting them into Ω.

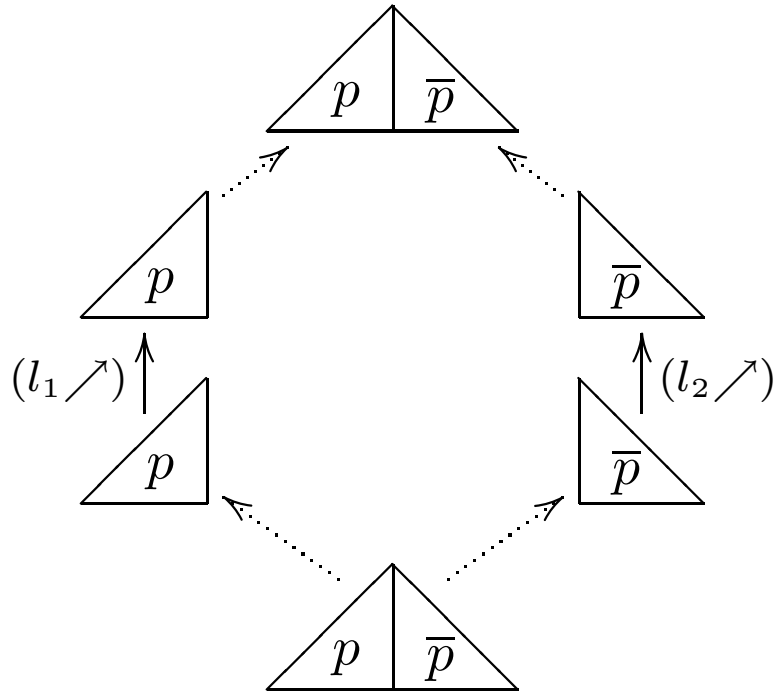
More on Map

Full definition of put direction:

$$(map\ l) \searrow (a,\ c) = \left\{ \begin{array}{l} n \mapsto l \searrow (a(n),\ c(n)) \\ \qquad\qquad n \in dom(a) \cap dom(c) \\ n \mapsto l \searrow (a(n),\ \Omega) \\ \qquad\qquad n \in dom(a) \setminus dom(c) \end{array} \right\}$$

A Fun Primitive Lens: `fork`

The get direction of `fork` $p\ l_1\ l_2$:



Some Derived Lenses

- `filter` p = `fork` p `id` (`const` $\{\}$)
- `focus` n = `filter` $\{n\}$; `hoist` n

List Processing with Lenses...

Representing Lists

The list

$$[\mathtt{v}_1 \ldots \mathtt{v}_n]$$

can be represented by the tree

$$\left\{\begin{array}{l} \texttt{*h} \mapsto \mathtt{v}_1 \\ \texttt{*t} \mapsto \left\{\begin{array}{l} \texttt{*h} \mapsto \mathtt{v}_2 \\ \texttt{*t} \mapsto \left\{\ldots \mapsto \left\{\begin{array}{l} \texttt{*h} \mapsto \mathtt{v}_n \\ \texttt{*t} \mapsto \{\!\}\end{array}\right\}\right\}\end{array}\right\}\end{array}\right\}$$

where `*h` and `*t` are special distinguished labels.

Derived Lenses for Lists

- `hd` = `focus *h`
- `tl` = `focus *t`
- `mapp` p l = `fork` p (`map` l) `id`
- `map_list` l = `mapp {*h}` l; `mapp {*t} (map_list` l`)`

More Primitives

Several more primitives are defined in the paper:

- conditionals (two completely different ones!)
- renaming
- pivoting
- copying and merging
- flattening and (a limited form of) joining
- etc.

More List Processing

Using these combinators, we can write:

- list reverse
- list filter
- group list into sublists of given length
- etc.

(N.b.: Not trivial!)

A Real Example

Representing XML

The XML element

```
<tag attr1="val1" ... attrm="valm">
   subelt1 ... subeltn
</tag>
```

is represented by the tree

$$\left\{\left|\begin{array}{l}\texttt{tag} \mapsto \left\{\left|\begin{array}{l}\texttt{attr1} \mapsto \left\{\left|\texttt{val1} \mapsto \{\!|\}\right|\right\} \\ \quad\vdots \\ \texttt{attrm} \mapsto \left\{\left|\texttt{valm} \mapsto \{\!|\}\right|\right\} \\ \\ \texttt{*content} \mapsto \left[\begin{array}{l}\langle\texttt{subelt1}\rangle \\ \quad\vdots \\ \langle\texttt{subeltn}\rangle\end{array}\right.\end{array}\right.\end{array}\right.\right\}\right|\right\}$$

Mozilla Bookmarks (as HTML)

```
<html>
  <head> <title>Bookmarks</title> </head>
  <body>
    <h3>Bookmarks Folder</h3>
    <dl>
      <dt> <a href="www.google.com"
              add_date="1032458036">Google</a> </dt>
      <dd>
        <h3>Conferences Folder</h3>
        <dl>
          <dt> <a href="www.cs.luc.edu/icfp"
                  add_date="1032528670">ICFP</a> </dt>
        </dl>
      </dd>
    </dl>
  </body>
</html>
```

Mozilla Bookmarks (as a tree)

```
{*contents ->
 [{html -> {*contents ->
   [{head -> {*contents -> [{title ->
                  {*contents ->
                   [{PCDATA -> Bookmarks}]}}]}}
    {body -> {*contents ->
     [{h3 -> {*contents ->
             [{PCDATA -> Bookmarks Folder}]}}
      {dl -> {*contents ->
       [{dt -> {*contents ->
         [{a -> {*contents -> [{PCDATA -> Google}]
                add_date -> 1032458036
                href -> www.google.com}}]}}
        {dd -> {*contents ->
         [{h3 -> {*contents -> [{PCDATA ->
                          Conferences Folder}]}}
          {dl -> {*contents ->
           [{dt -> {*contents ->
              [{a ->
                {*contents -> [{PCDATA -> POPL}]
                 add_date -> 1032528670
                 href -> cristal.inria.fr/POPL2004
                }}]}}]}}]}}]}}]}}]}}]}}]
```

Bookmarks (abstract tree)

```
{name -> Bookmarks Folder
 contents ->
  [{link -> {name -> Google
             url -> www.google.com}}
   {folder ->
     {name -> Conferences Folder
      contents ->
       [{link ->
         {name -> POPL
          url -> cristal.inria.fr/POPL2004}}]}}]}
```

Bookmark lens

```
link = rename {dt = link};
       map (hoist *contents;
            hd {};
            hoist a;
            rename {href = url, *contents = name};
            prune add_date {$today};
            mapp {name} (hd {}; hoist PCDATA))

folder = rename {dd = folder};
         map (hoist *contents; folder_contents)

folder_contents =
   hoist_list [{h3} {dl}];
   rename {h3 = name, dl = contents};
   mapp {name} (hoist *contents; hd {}; hoist PCDATA);
   mapp {contents} (hoist *contents; map_list item)
```

Bookmark lens (continued)

```
item =
  dispatch [({dd},{folder},folder)
            ({dt},{link},link)]

bookmarks =
  hoist *contents; hd {}; hoist html; hoist *contents;
  tl {head -> {*contents -> [{title -> {*contents ->
                  [{PCDATA -> Bookmarks}]}}]}};
  hd {}; hoist body; hoist *contents;
  folder_contents
```


What Can We Do With Lenses?


HARMONY

The Harmony Project

Goal:

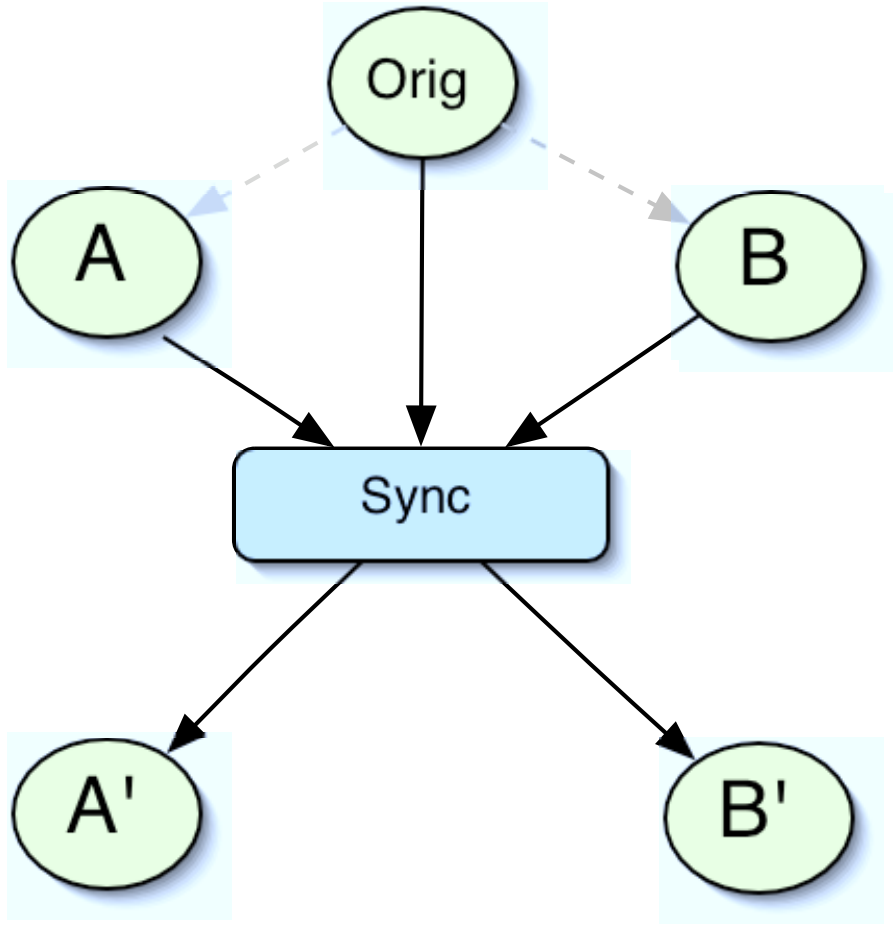
Build a generic synchronization framework for tree-structured data (stored, e.g., as XML documents).

Instances:

calendars (ical, iCalendar, palm datebook)	running
bookmarks (Mozilla, Safari, IE5, ...)	~running
address books (palm, vcard, xcard, ...)	underway
presentations (Keynote, PPT, ...)	planned
structured documents (docbook, Word, ...)	planned
preference files, etc., etc.	planned

User base: 2 :-)

Basic Harmony Architecture

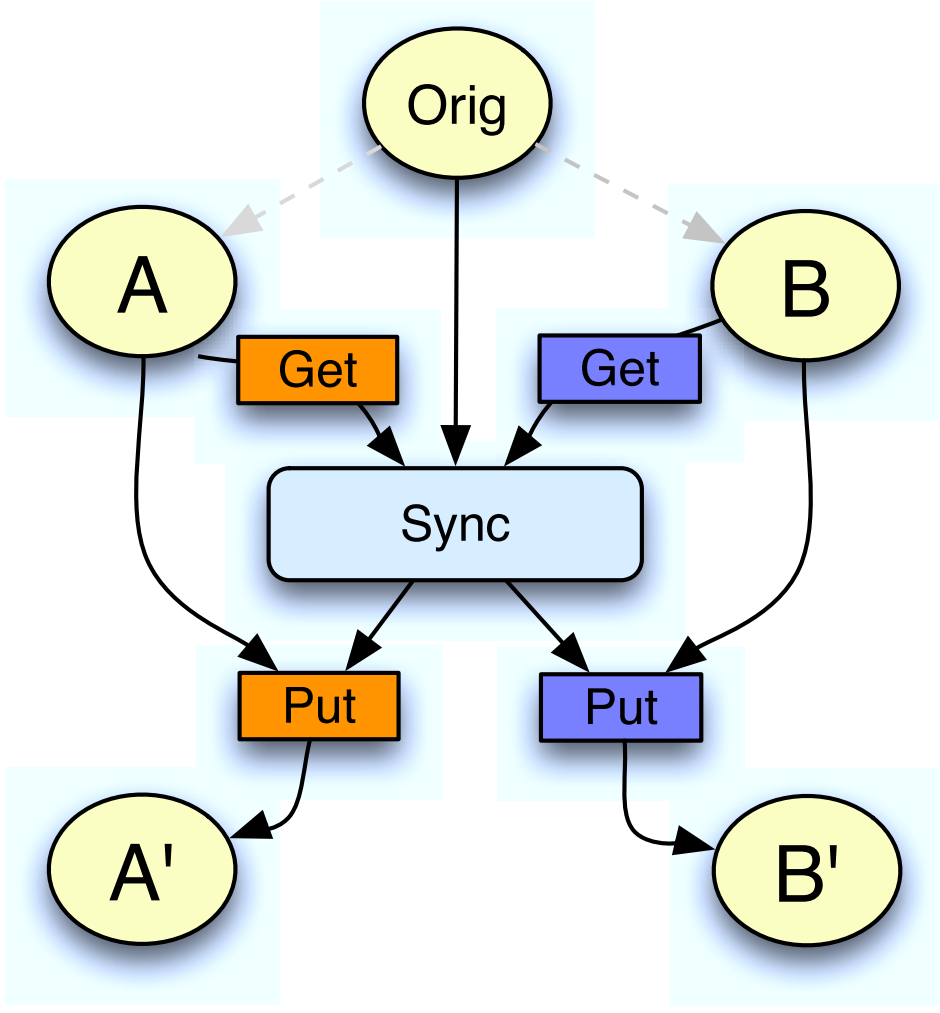


Heterogeneity

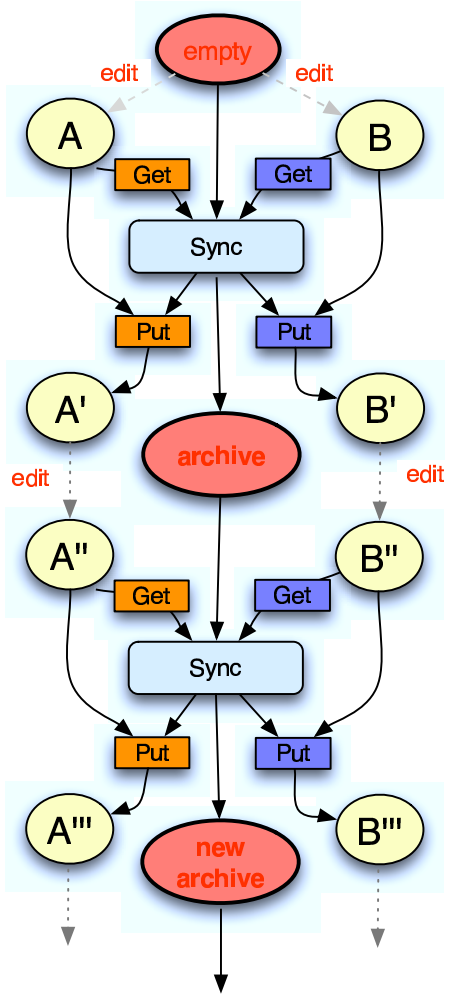
We want to synchronize many different concrete formats for, e.g., bookmarks, without writing n^2 different synchronizers.

This is achieved in Harmony by writing lenses mapping each particular concrete format to a single common abstract format.

Full Harmony Architecture



The Real Story



Finishing Up...

Related Work

Lots! [See paper for details...]

Main categories:

- Bidirectional languages
- Bijective languages
- Reversible languages

Bidirectional Languages

In bidirectional languages, the get direction can throw away information and the put direction restores it.

- our work
- later work by Hu, Mu, and Takeichi (uses weaker lens laws and does not require totality)
- earlier work by Meertens (similar, but never published or implemented; no type system)
- classical database literature, e.g., Bancilhon&Spyratos (closely related to our semantic framework)

Bijective Languages

In bijective languages, the get and put functions form an isomorphism

Many instances in various areas...

Reversible Languages

Reversible languages have a quite different flavor: there is no "editing" of the abstract view—we just want to be able to run programs backwards from output to input.

Such languages arise mainly in connection with quantum computing. Not too relevant to our setting.

Further Challenges...

Typechecking

Types play a critical role both in the design of primitive lenses and in programming with lenses.

At the moment, though, our "type declarations" are just statements in set theory. "Typechecking" must be performed by hand by programmers.

We are working on designing a syntactic type algebra with a mechanical typechecking procedure.

The Relation to Relations

The classical view update problem arose in the setting of relational databases.

Does our approach shed any new light?

The Issue of Expressiveness

What tree transformations can we write using our present combinators?

What could we write if we added new primitives?

Are there any transformations that could never be expressed in any extension of our language because they cannot, in principle, be "reversed"?

http://www.cis.upenn.edu/~bcpierce/harmony