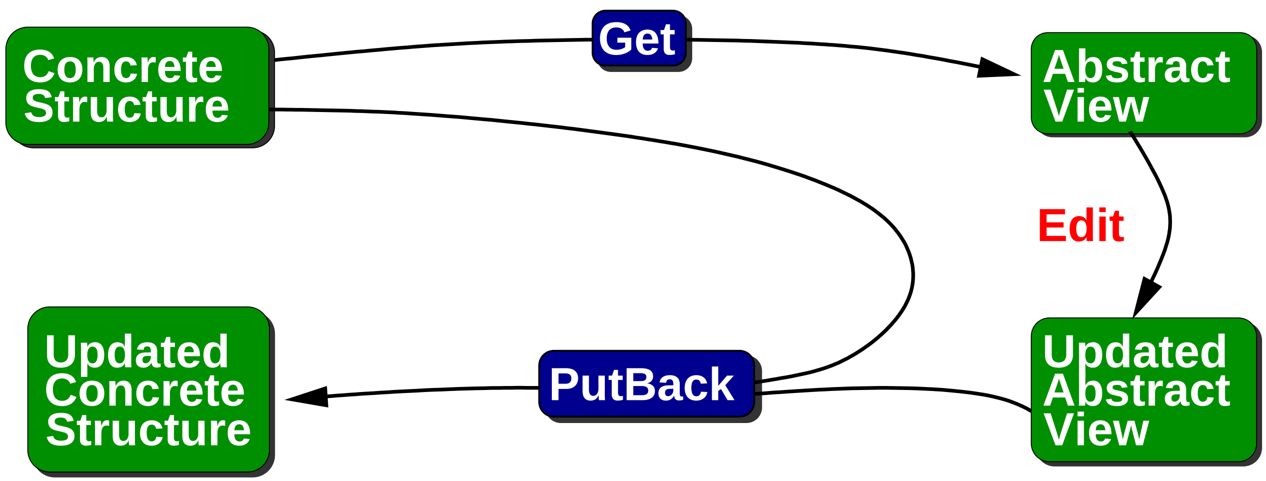
# Combinators for Bi-Directional Tree Transformations: A Linguistic Approach to the View Update Problem


| | |
|---|---|
| J. Nathan Foster | (Penn) |
| Michael B. Greenwald | (Lucent) |
| Jon Moore | (Penn) |
| Benjamin C. Pierce | (Penn) |
| Alan Schmitt | (INRIA) |

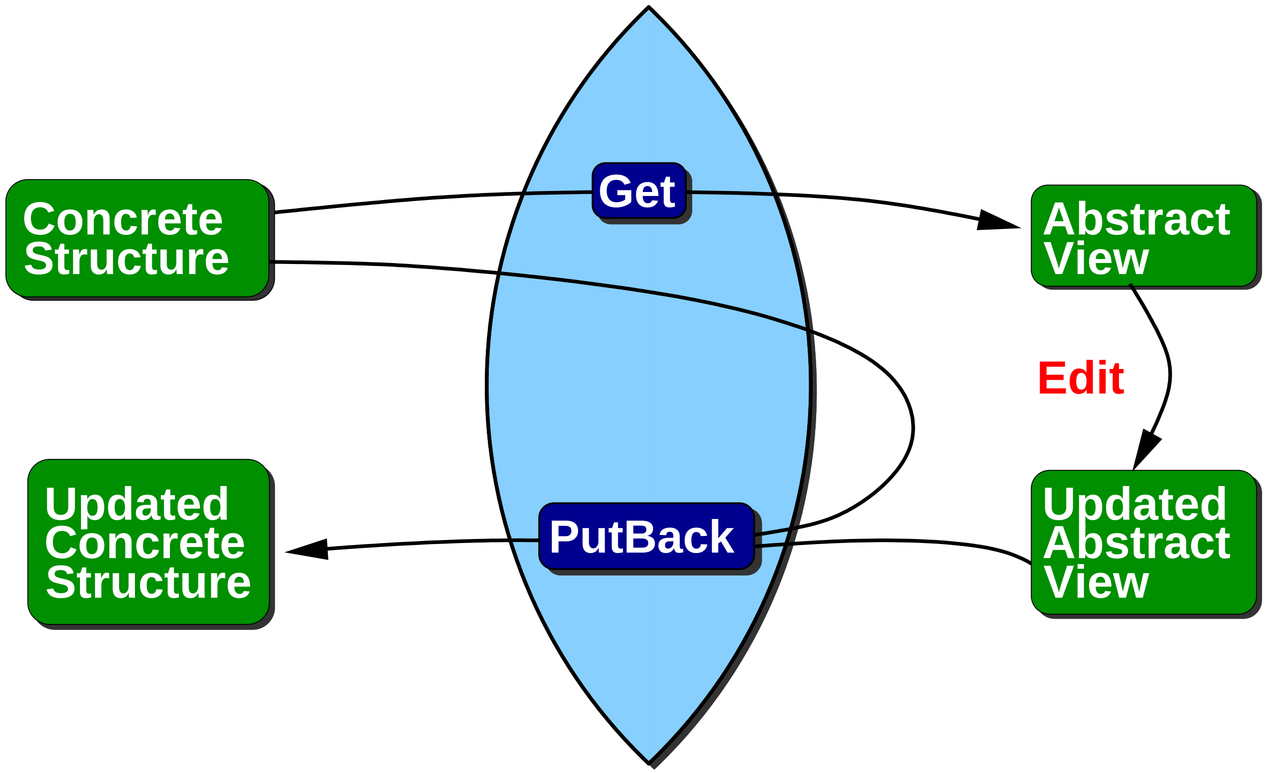
# View Update

An old problem from the database community:



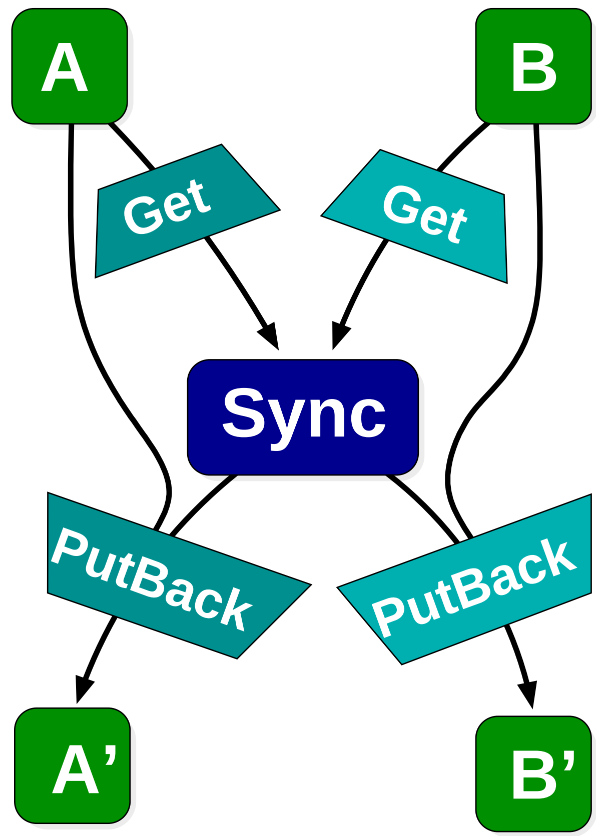
# View Update

Our approach: a domain specific language for writing ***get*** and ***putback*** at once. A **lens** is a bi-directional map between concrete structures and abstract views.



# Lenses and Synchronization

Harmony project goal: a generic synchronization framework for *heterogeneous* data:



# Example

- Our data model is unordered, edge-labelled trees of finite width where every node has at most one child for every name $n$.
- Equivalently a trees is a finite map from names to trees.
- (We draw trees sideways to save space.)

Suppose that we have an address book represented as a tree:

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

# Example

... and we only want to synchronize phone numbers and add or drop complete entries. Using the ***get*** component of a lens, we transform

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

into

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

# Example

Now we synchronize the abstract view, yielding a tree:

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

and ***putback*** the updated abstract view into the original tree:

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

# Contributions

1. A natural semantic space of well-behaved lenses.
2. A domain specific language where
   - reasoning about well-behavedness is *compositional*
   - every well-typed program denotes a well-behaved lens.
3. A concrete application: a synchronizer built using lenses.

# Semantic Foundations

# Lenses

Let $C$ be a set of concrete structures and $A$ a set of abstract views.

A (total) *lens* $l$ between $C$ and $A$ is a pair of functions

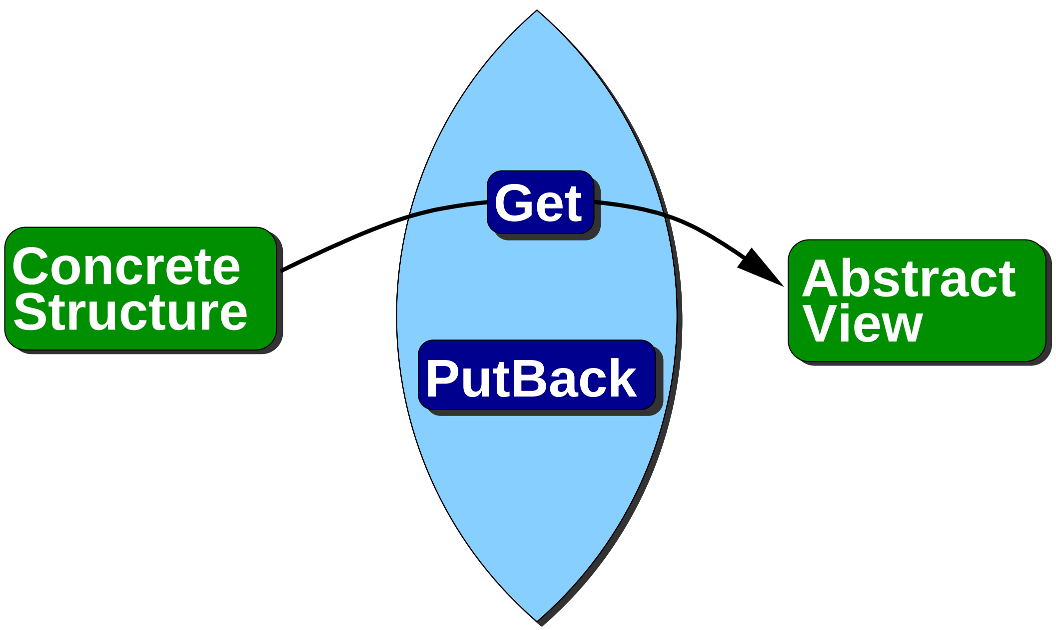
- $l\nearrow$ from $C$ to $A$ [***Get***]
- $l\searrow$ from $A \times C$ to $C$ [***PutBack***]

But we don't want any pair of functions with these types...

# Well-Behaved Lenses

... we need guarantees on round-trip behavior

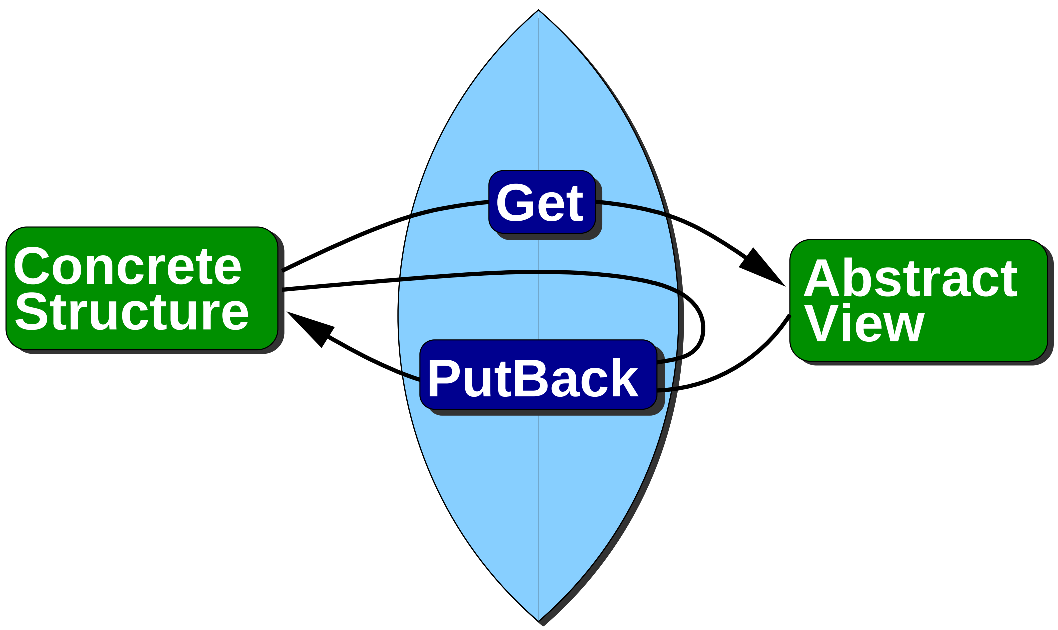
$$l \searrow (l \nearrow c,\ c) = c$$ [***GetPut***]



# Well-Behaved Lenses

... we need guarantees on round-trip behavior:

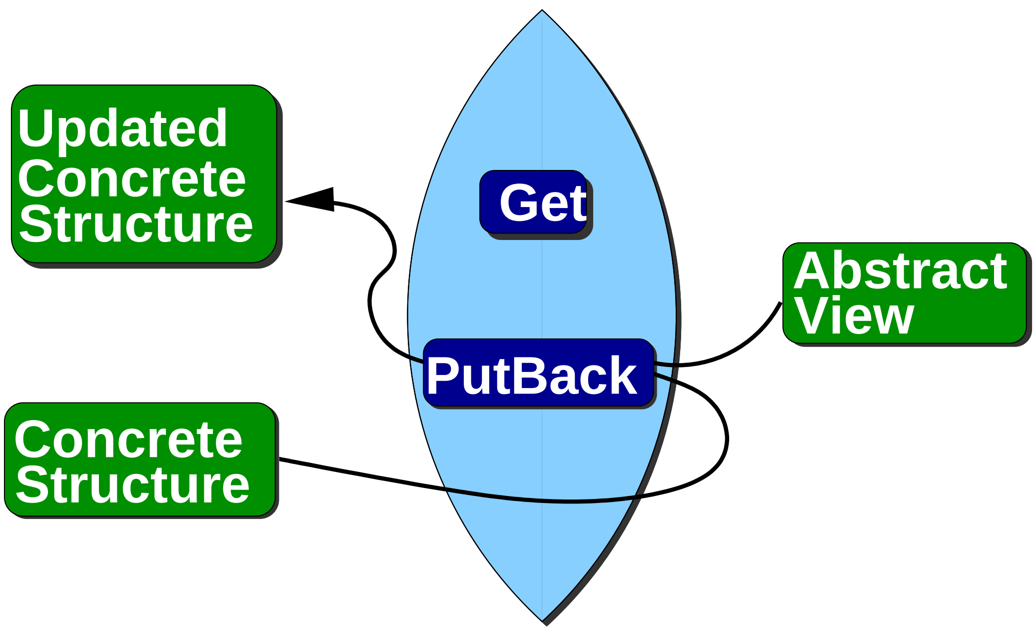
$$l \searrow (l \nearrow c,\ c) = c \qquad \text{[GetPut]}$$



# Well-Behaved Lenses

... in both directions:

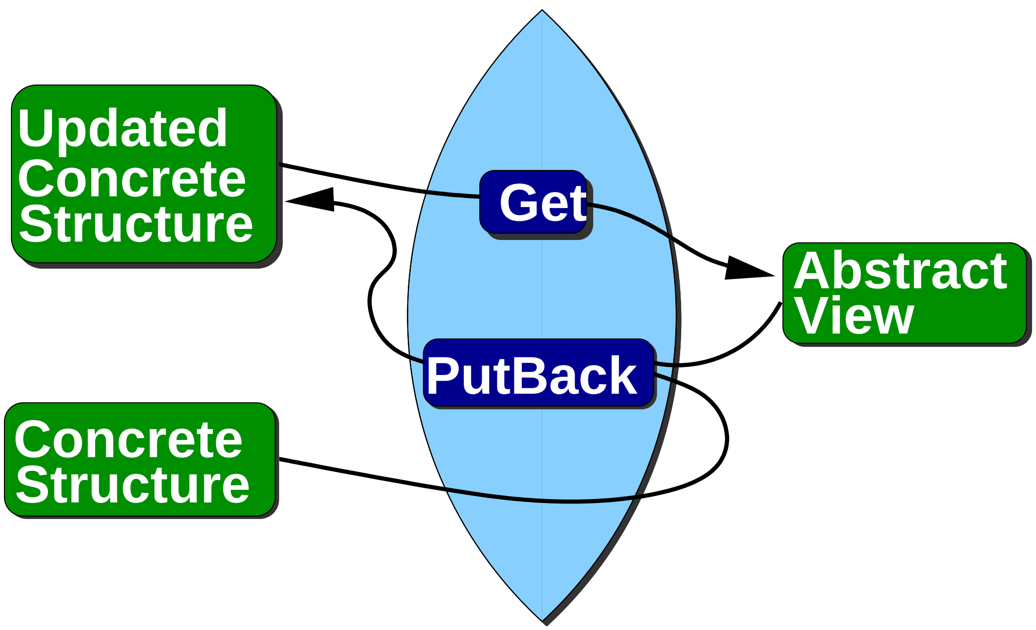
$$l \nearrow l \searrow (a, c) = a$$ **[*PutGet*]**



# Well-Behaved Lenses

... in both directions:

$$l \nearrow l \searrow (a,\ c) = a$$ [***PutGet***]




Write $l \in C \Longleftrightarrow A$ for a well-behaved lens between $C$ and $A$.

# Recursive Lenses

We want to define lenses by recursion.

We can refine lenses to a partial setting and take fixed points using standard techniques.

See paper for details; in this talk, we'll only look at total lenses.

# A Lens Language

# Identity

$$\texttt{id} \in C \Longleftrightarrow C$$

---

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

The ***get*** function yields $c$;

the ***putback*** function ignores $c$ and yields $a$.

# Hoist & Plunge

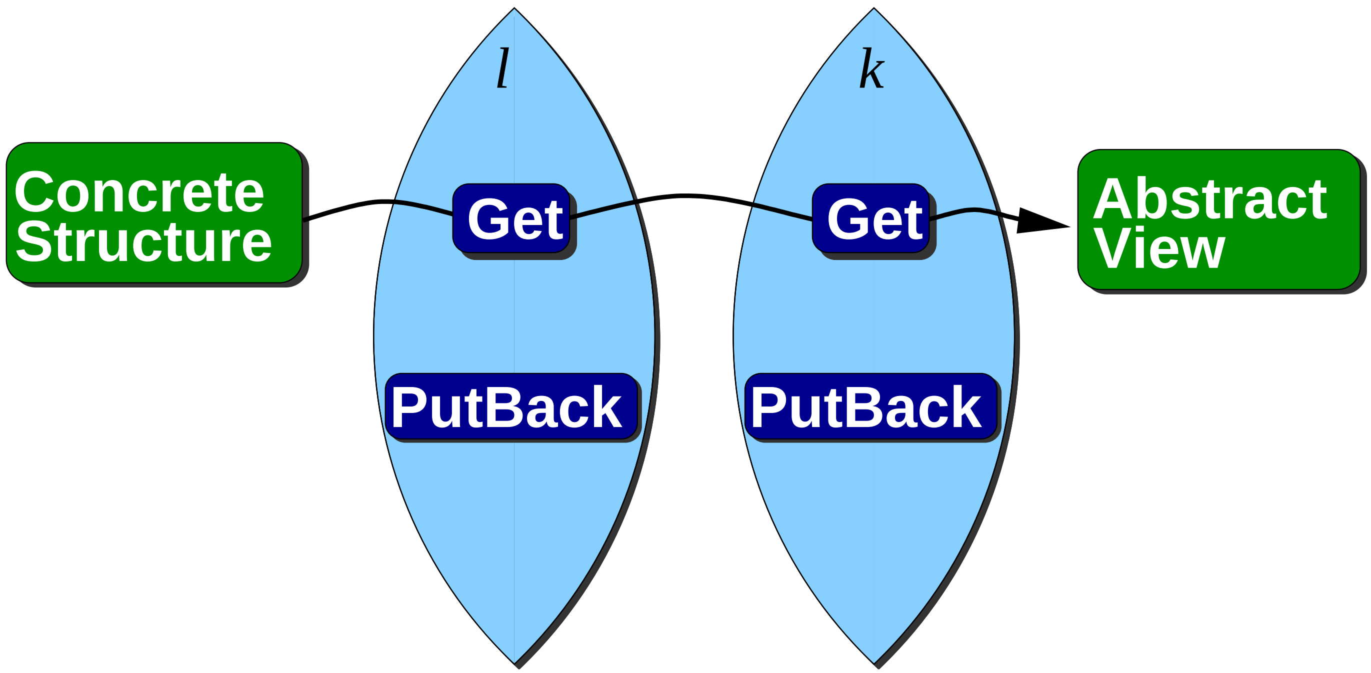
$$\frac{\texttt{hoist}\ n \in \{n \mapsto C\} \Longleftrightarrow C}{\begin{array}{rcll} \texttt{hoist}\ n \nearrow c & = & t & \text{if } c = \{n \mapsto t\} \\ \texttt{hoist}\ n \searrow (a, c) & = & \{n \mapsto a\} & \end{array}}$$

$$\frac{\texttt{plunge}\ n \in C \Longleftrightarrow \{n \mapsto C\}}{\begin{array}{rcll} \texttt{plunge}\ n \nearrow c & = & \{n \mapsto c\} & \\ \texttt{plunge}\ n \searrow (a, c) & = & t & \text{if } a = \{n \mapsto t\} \end{array}}$$

# Composition

If $l \in C \Longleftrightarrow B$ and $k \in B \Longleftrightarrow A$ then $(l;k) \in C \Longleftrightarrow A$.

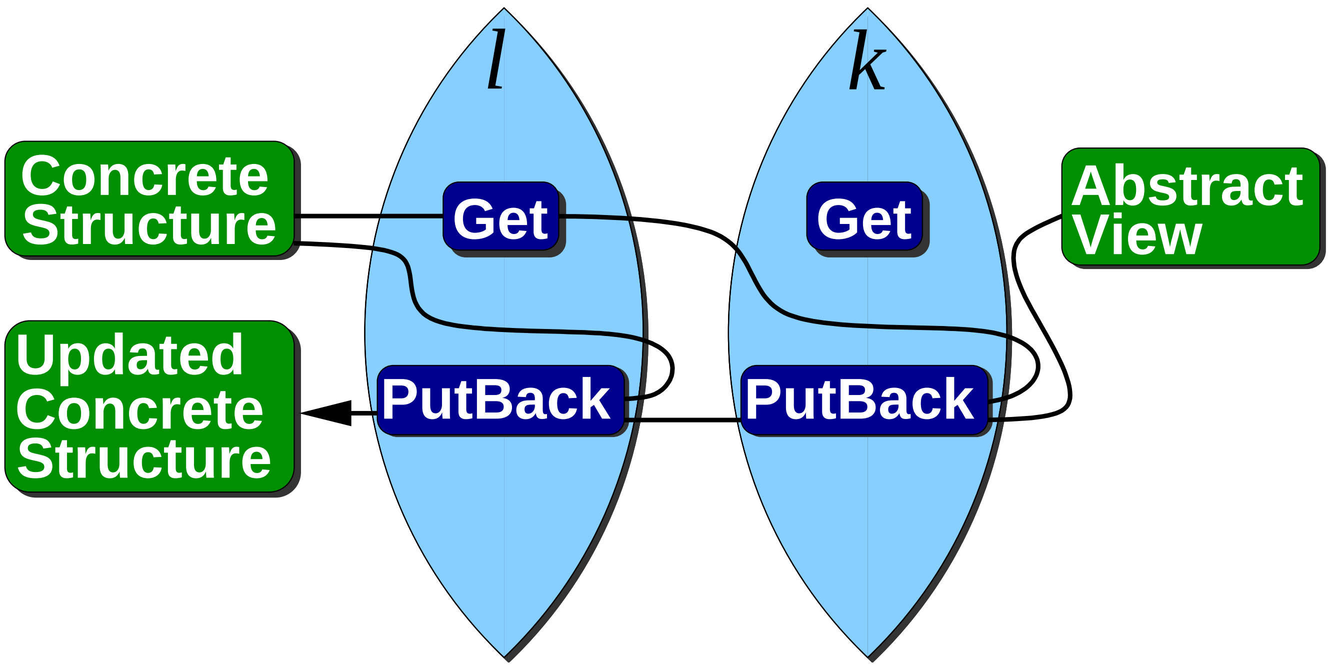
$$(l;k) \nearrow c = k \nearrow (l \nearrow c)$$ [***Get***]



# Composition

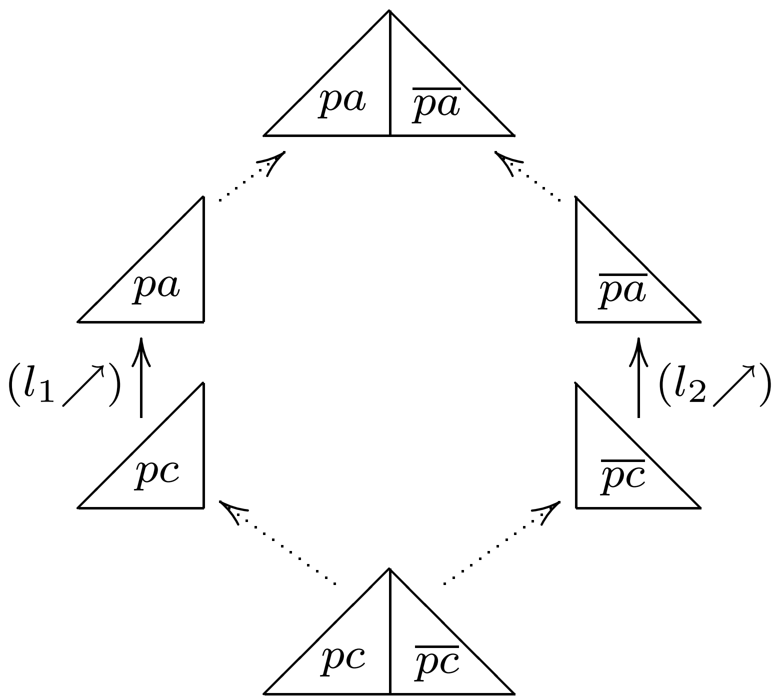
If $l \in C \Longleftrightarrow B$ and $k \in B \Longleftrightarrow A$ then $(l;k) \in C \Longleftrightarrow A$.

$$(l;k) \searrow (a,\, c) = l \searrow (k \searrow (a,\, l \nearrow c),\, c) \qquad \textbf{[\textit{PutBack}]}$$



# XFork

`xfork` $pc$ $pa$ $l_1$ $l_2$ splits the tree and applies a different lens to each part:



# Map

Map applies a lens one level deeper in the tree.

The ***get*** function is easy:

$$(\texttt{map}\ l)\nearrow\left(\left\{\begin{array}{c} n_1 \mapsto t_1 \\ \vdots \\ n_k \mapsto t_k \end{array}\right\}\right) = \left\{\begin{array}{c} n_1 \mapsto l\nearrow t_1 \\ \vdots \\ n_k \mapsto l\nearrow t_k \end{array}\right\}$$

When $a$ and $c$ have the same children the ***putback*** function is also easy:

$$(\texttt{map}\ l)\searrow\left(\left\{\begin{array}{c} n_1 \mapsto t_1 \\ \vdots \\ n_k \mapsto t_k \end{array}\right\}, \left\{\begin{array}{c} n_1 \mapsto t_1' \\ \vdots \\ n_k \mapsto t_k' \end{array}\right\}\right) = \left\{\begin{array}{c} n_1 \mapsto l\searrow(t_1,\ t_1') \\ \vdots \\ n_k \mapsto l\searrow(t_k,\ t_k') \end{array}\right\}$$

In general, $a$ and $c$ might have different children...

# Map

A natural choice for the ***putback*** of $(\mathtt{map}\ l)$ is to keep the children in $a$, and discard children that only appear in $c$. (In fact ***PutGet*** requires it.)

- Children appearing only in $c$ are dropped;
- Children in both $a$ and $c$ are ***putback*** as in simple case;
- Children appearing only in $a$ are ***putback*** with what?
  - Use special tree, $\Omega$ ("missing") to mark where a default is needed.

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

# Constant

Lenses whose ***get*** functions are projections need to handle handle $\Omega$ (by providing defaults).

$$\texttt{const}\; t\; d \in C \Longleftrightarrow \{t\}$$

$$\begin{array}{rcll} \texttt{const}\; t\; d \nearrow c & = & t & \\ \texttt{const}\; t\; d \searrow (a,\, c) & = & c & \text{if } c \neq \Omega \text{ and } a = t \\ & & d & \text{if } c = \Omega \text{ and } a = t \end{array}$$

The ***get*** function discards the entire concrete tree.

The ***putback*** function restores the original concrete tree, or a default if $c$ is $\Omega$:

# Conditionals

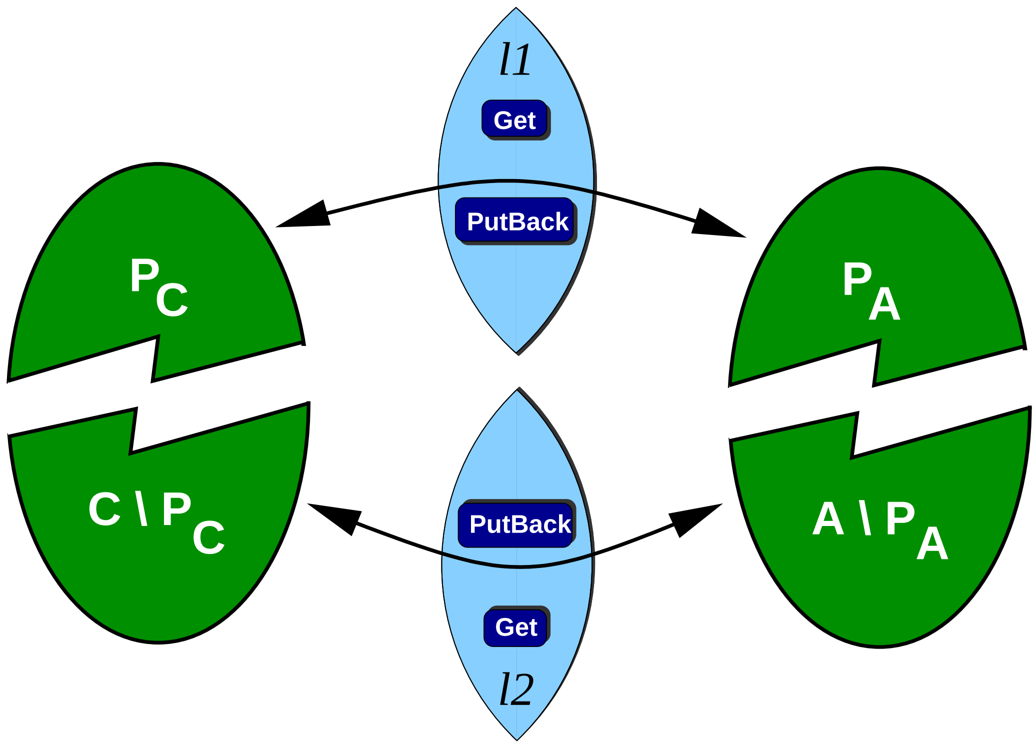
Conditionals are a fun challenge in a bi-directional setting.

Have to select a lens in ***both*** directions.

# ACond

If $l_1 \in (C \cap P_C) \Longleftrightarrow (A \cap P_A)$ and $l_2 \in (C \setminus P_C) \Longleftrightarrow (A \setminus P_A)$

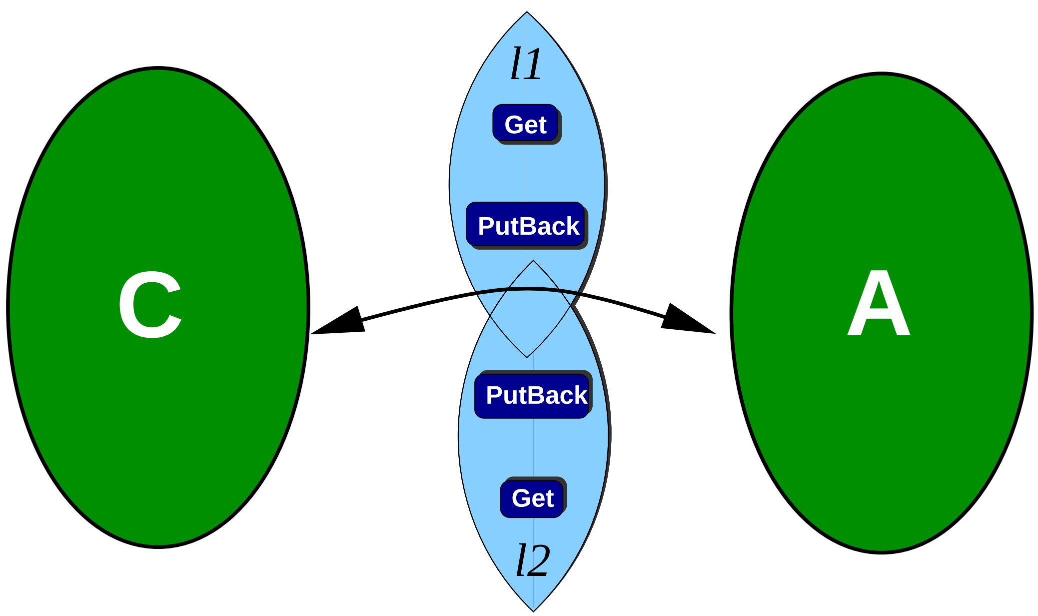
then `acond` $P_C\ P_A\ l_1\ l_2 \in C \Longleftrightarrow A$.



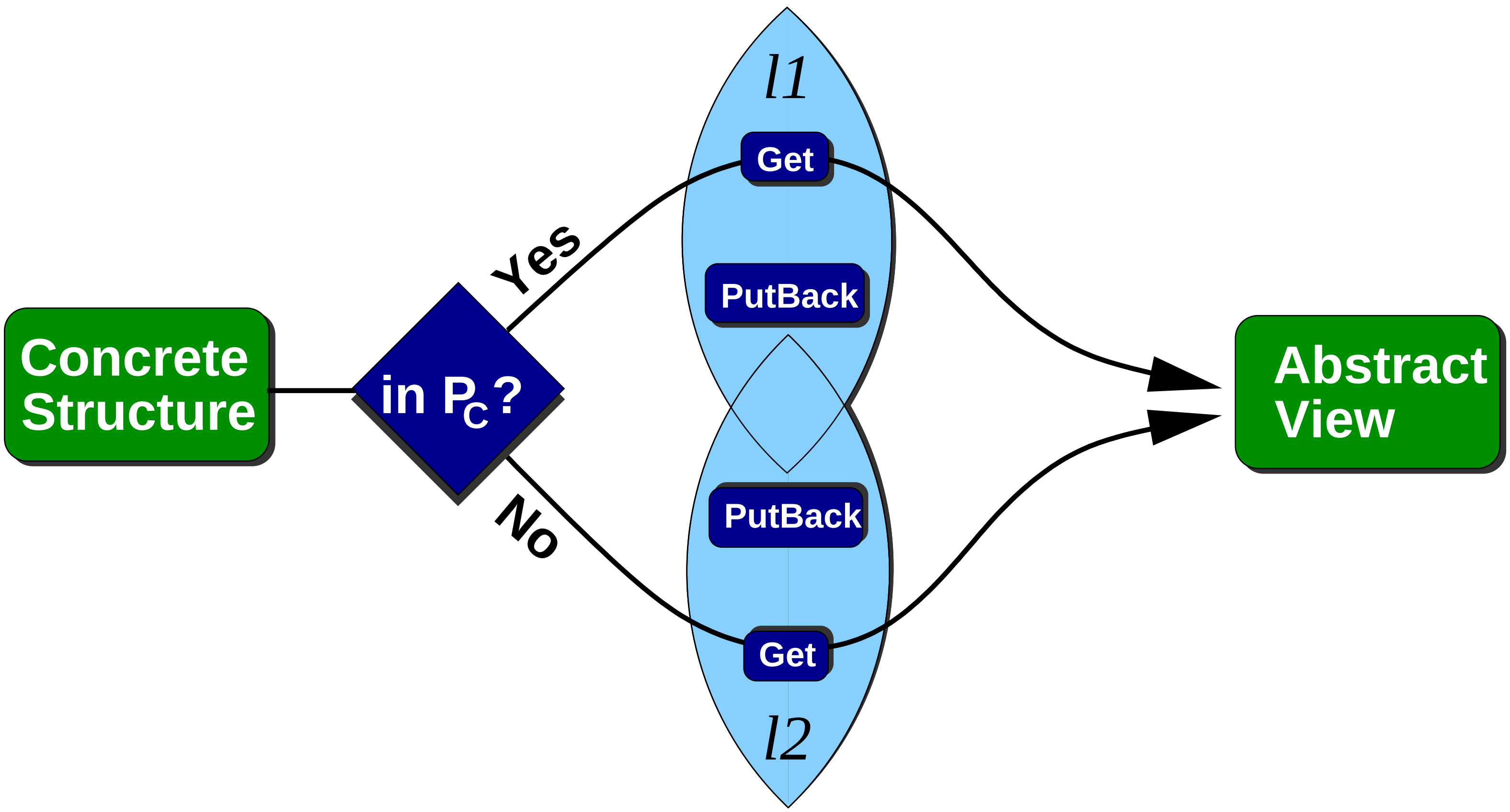
# ACond

If $l_1 \in (C \cap P_C) \Longleftrightarrow (A \cap P_A)$ and $l_2 \in (C \setminus P_C) \Longleftrightarrow (A \setminus P_A)$

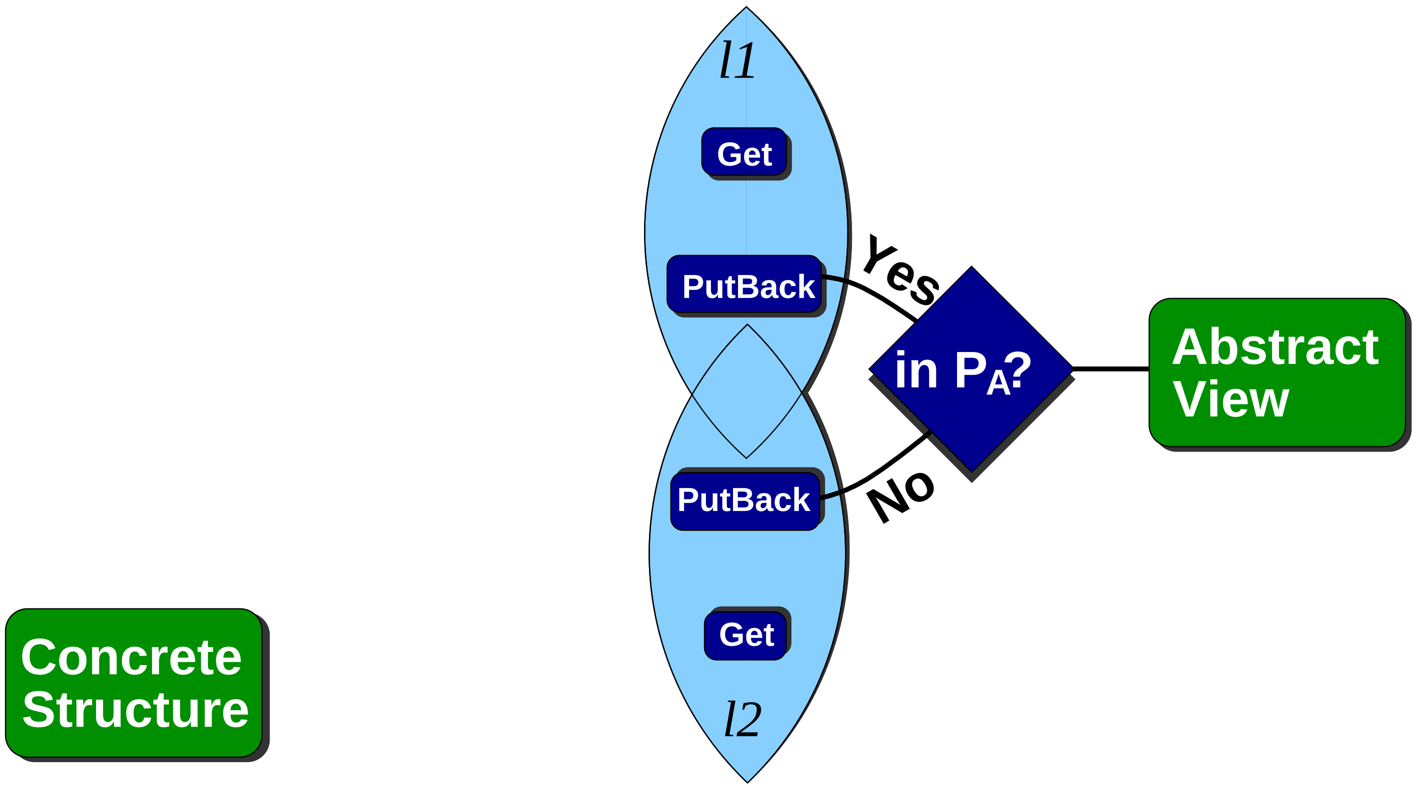
then $\texttt{acond}\ P_C\ P_A\ l_1\ l_2 \in C \Longleftrightarrow A$.



# ACond

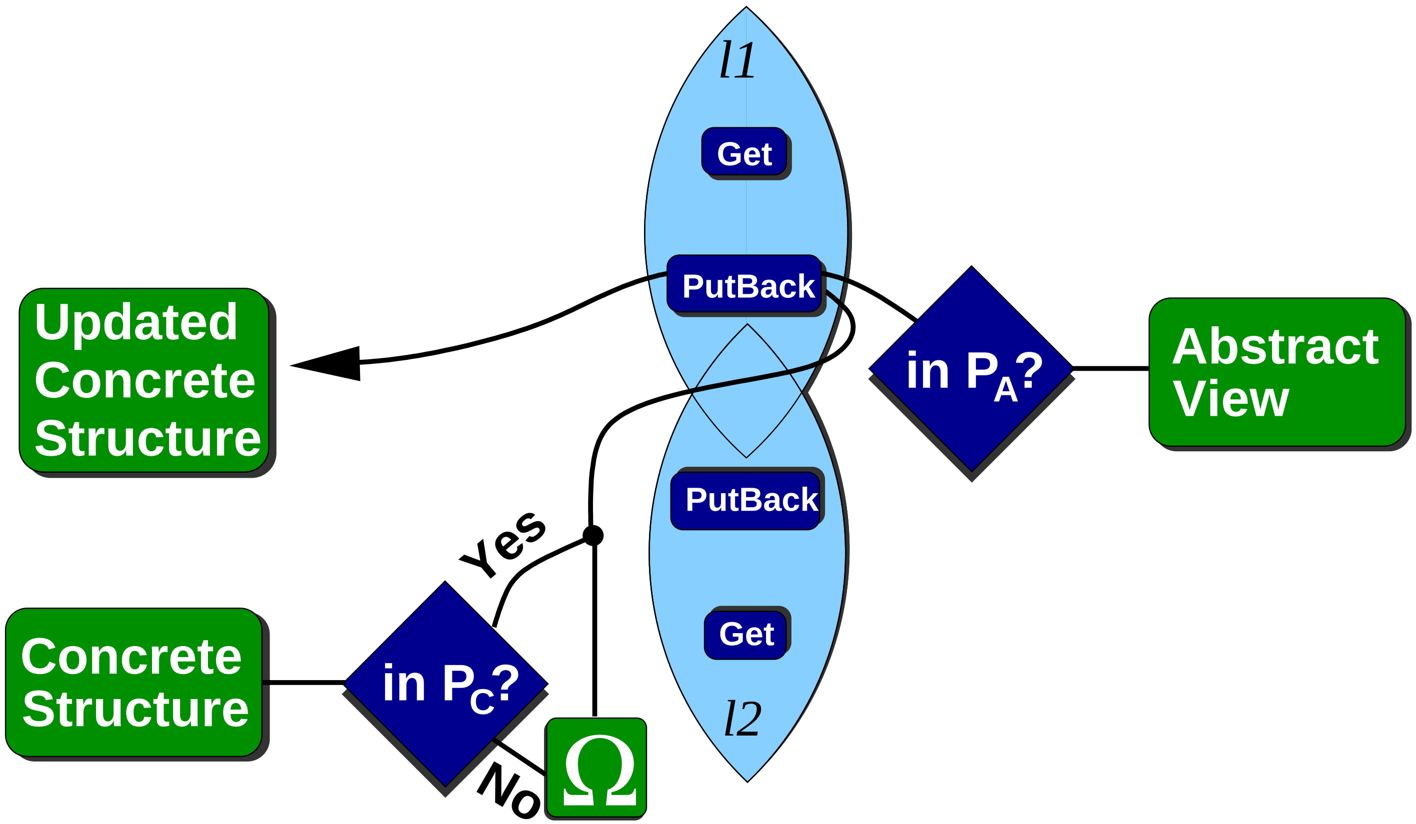

Concrete Structure
in $P_C$?
Yes
No
l1
Get
PutBack
PutBack
Get
l2
Abstract View

# ACond


l1
Get
PutBack
Yes
in $P_A$?
Abstract View
PutBack
No
Get
l2
Concrete Structure

# ACond

$$l_1 \in (C \cap P_C) \iff (A \cap P_A)$$



# Lenses for Lists

Can encode lists using standard "cons cells".

The list $[\mathtt{v}_1 \ldots \mathtt{v}_n]$ is represented by the tree

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

Lenses implementing functions on lists are derived forms.

Demo

# Lenses for Lists

```
let hd = xfork {*h} {*h} id (const {} {*t=[]});
         hoist *h

let tl = xfork {*t} {*t} id (const {} {*h={}});
         hoist *t

let rec list_map l =
  xfork {*h} {*h} (map l) (map (list_map l))
```

# Lenses for Lists

```
let rename x y = xfork {x} {y} (hoist x; plunge y) id
let swaphd =
  rename *h tmp;
  xfork {*t} {*h *t} (hoist *t) id;
  xfork {tmp *t} {*t} (rename tmp *h; plunge *t) id
let rec rotate =
  acond isSingletonOrEmptyList isSingletonOrEmptyList
    id
    (swaphd; xfork {*t} {*t} (map rotate) id)
let rec list_reverse =
  xfork {*t} {*t} (map list_reverse) id; rotate
```

# Other Lenses

We have investigated several other lenses:

- pivoting, copying, and merging
- conditionals (two additional ones!)
- filtering and flattening (for lists)

and have built several applications using these lenses:

- a bookmark synchronizer
- a calendar synchronizer
- an addressbook synchronizer

# Future Work

1. Semantic Framework
   - Explore stronger lens laws (e.g., in a metric space).
2. A Lens Language
   - Mechanical type checking for lenses.
   - Characterization of the expressive power of lenses and our language.
   - Beyond trees (e.g., relational lenses).
3. Applications of Lenses
   - End-to-end typed synchronizer.
   - More applications.

# Related Work

- Semantic Framework - many related ideas in database literature (see paper).
  - [Bancilhon, Spryatos '81] "translators under constant complement".
  - [Gottlob, Paolini, Zicari '88] "dynamic views".
- Bi-Directional Languages
  - [Meertens] - language for constaint maintainers; similar behavioral laws.
  - [Hu, Mu, Takeichi '04] - language at core of a structured document editor.
- Bijective and Reversible Languages

HARMONY

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