The Weird World of Bi-Directional Programming

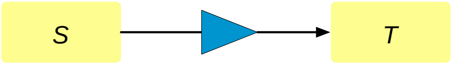
Benjamin C. Pierce

Microsoft Research, Cambridge
(on leave from University of Pennsylvania)

March, 2006

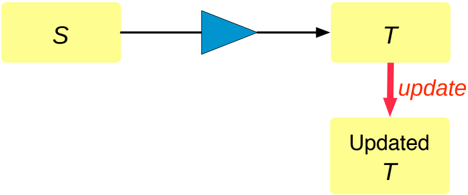
The View Update Problem

- We apply a function to transform source to target



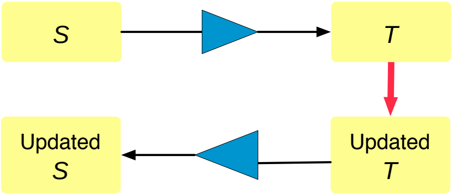
The View Update Problem

- We apply a function to transform source to target
- Someone updates target



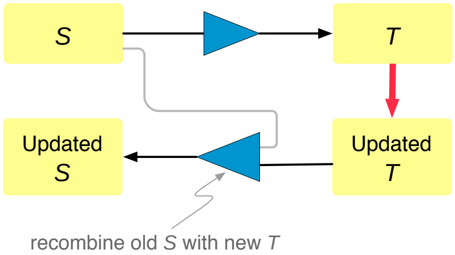
The View Update Problem

- We apply a function to transform source to target
- Someone updates target
- We must now "translate" this update to obtain an appropriately updated source



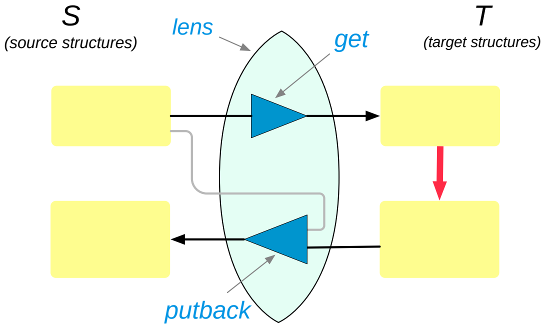
The View Update Problem

- We apply a function to transform source to target
- Someone updates target
- We must now "translate" this update to obtain an appropriately updated source



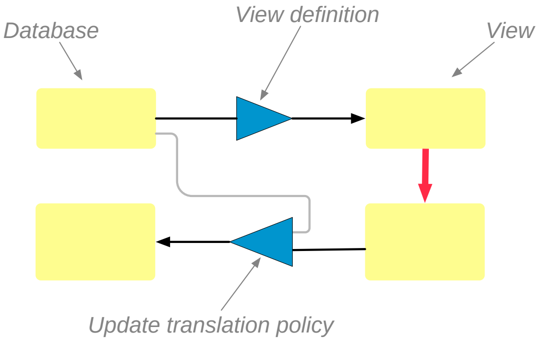
The View Update Problem: Terminology

Let's call the function from source to target *get* and the other *putback*. The two functions together form a lens.



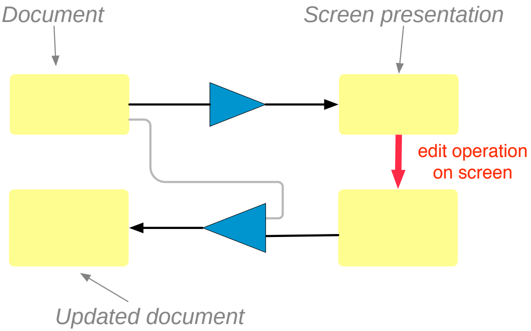
The View Update Problem In Practice

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



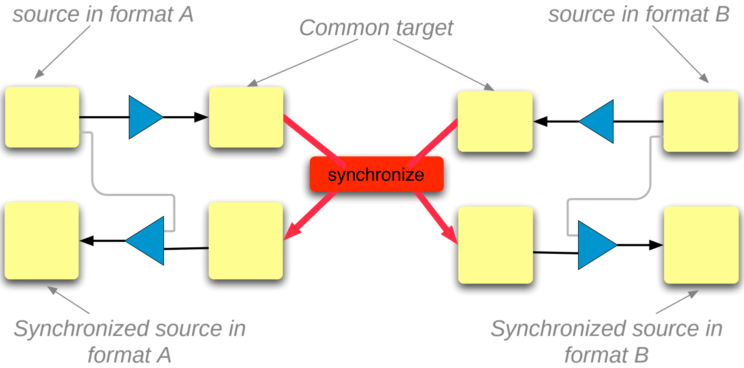
The View Update Problem In Practice

This problem arises in many contexts besides “straight” databases — for example in editors for structured documents...



The View Update Problem In Practice

...and data synchronizers such as the Harmony system being built at the University of Pennsylvania.



Why is This Hard?

A Simple Solution?

We can "solve" the problem just by sticking together two arbitrary functions of appropriate types, each written separately in any programming language you like.

A Simple Non-Solution

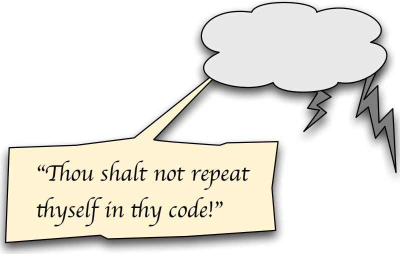
We can "solve" the problem just by sticking together two arbitrary functions of appropriate types, each written separately in any programming language you like.

But this is tricky to get right... and even trickier to maintain!

A Simple Non-Solution

We can "solve" the problem just by sticking together two arbitrary functions of appropriate types, each written separately in any programming language you like.

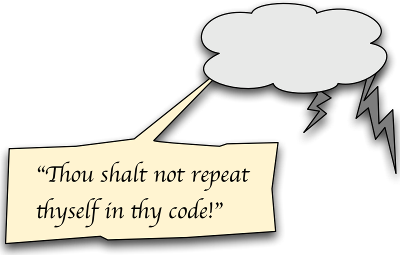
But this is tricky to get right... and even trickier to maintain!



A Simple Non-Solution

We can "solve" the problem just by sticking together two arbitrary functions of appropriate types, each written separately in any programming language you like.

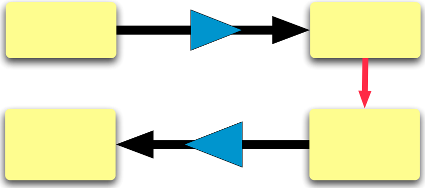
But this is tricky to get right... and even trickier to maintain!



Need to find a way of deriving both functions from a single description.

A Class of Simple Solutions

Things become easier if we restrict attention to bijective transformations.



A Class of Simple Solutions

Things become easier if we restrict attention to bijective transformations. Lots of success stories...

- `xsugar` [Møller and Braband]
- bijective macro tree transducers [Hosoya]
- *correspondences* and *structuring schemas* for XML [Abiteboul, Cluet, and Milo]
- pickling combinators [Kennedy]
- embedded interpreters [Benton]
- updatable views of o-o databases [Ohori and Tajima]
- inter-compilation between Jekyll and C [Ennals]
- etc., etc., etc.

From Bijectiveness to Bi-Directionality

But bijectiveness is a strong restriction.

Often the whole point of defining a "view" is to hide some information that is present in the source!

- presenting just a small part of a huge database
- ignoring ordering of records when synchronizing XML databases
- etc.

$\Longrightarrow$ Also important to address the more general "bi-directional" case, where the *putback* function weaves updates back into the original source.

Constructivism

Another issue: It is not enough to know that a *putback* exists.

- E.g., even in the bijective case, just giving the *get* function and proving, somehow, that it is bijective doesn't do the whole job.

We need a description that allows us to compute both *get* and *putback* functions.

(This is why the bijective case is already interesting!)

Constructivism

Possible approaches:

- Monolithic:
  - programmer writes *get* function in some standard notation (e.g., SQL)
  - read off its semantics in some form
  - from this, calculate an appropriate *putback*
- Compositional:
  - Build complex bi-directional transformations from simpler bi-directional components.

Monolithic Approaches

Problem:

- In general, the *get* direction may not provide enough information to determine the *putback* function (More on this later...)

Possible solutions:

- Restrict the set of *get* functions to ones with "obvious" *putback* functions [e.g., Oracle "updatable views"]
- Put an ordering on the possible results of the *putback* and look for an algorithm that finds the best [e.g., Buneman, Khanna, and Tan]
- Restrict the set of update operations (e.g., to single-record insertions or deletions) [Hu, Mu, and Takeichi]

Compositional Approaches

- Build complex bi-directional transformations by combining simpler bi-directional transformations
- I.e., design languages in which every program can be read...

  from left to right as a *get* function
  from right to left as a *putback* function
- Compositional reasoning about lens properties
- Type systems play a crucial role

Compositional Approaches

- Build complex bi-directional transformations by combining simpler bi-directional transformations
- I.e., design languages in which every program can be read...

  from left to right as a *get* function
  from right to left as a *putback* function

- Compositional reasoning about lens properties
- Type systems play a crucial role
  - well-typedness $\Longrightarrow$ "reasonableness"
  - well-typedness of components (plus simple local reasoning) $\Longrightarrow$ well-typedness of compound expression

Harmony

In the Harmony group at Penn, we have applied this compositional approach in two concrete domains:

- a language for bi-directional tree transformations
- a language for updatable relational views

Today

A guided tour of this weird world.

Goals:

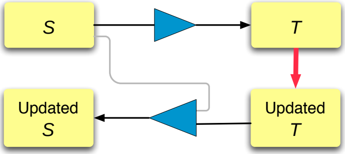
- To give a sense of the design space of bi-directional languages
- To illustrate a particular point in this space (Harmony's language of bi-directional tree transformations) and sketch some of its interesting structures

Harmony Demo

The Design Space

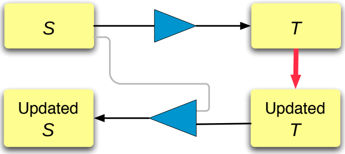
What is an “Update”?

Before we can talk about what it means to translate an update, we must first say precisely what we mean by an update.



What is an “Update”?

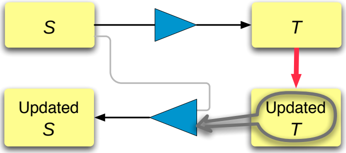
Before we can talk about what it means to translate an update, we must first say precisely what we mean by an update.



Is it...

What is an "Update"?

Before we can talk about what it means to translate an update, we must first say precisely what we mean by an update.

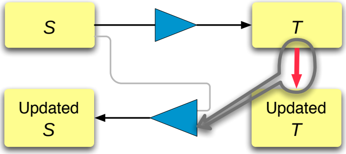


Is it...

- the new state of T

What is an "Update"?

Before we can talk about what it means to translate an update, we must first say precisely what we mean by an update.

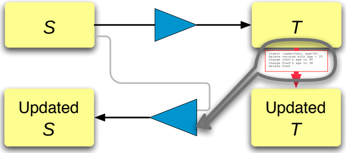


Is it...

- the new state of T
- a (mathematical) function from T to T?

What is an "Update"?

Before we can talk about what it means to translate an update, we must first say precisely what we mean by an update.



Is it...

- the new state of T
- a (mathematical) function from T to T?
- a (syntactic) program denoting such a function?

What is an “Update”?

All of these are sensible answers.

Tradeoffs:

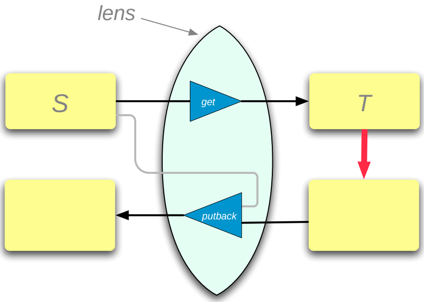
- state-based approach (update = new state):
  - \+ mathematically simpler
  - \+ describes “loosely coupled” systems: update translator need not know what operation was applied — just its result
- operation-based approaches (update = function / program):
  - \+ more expressive / flexible
  - \+ directly captures intuition of “manipulating (small) deltas to (huge) databases”

For this talk, we’ll adopt the simpler state-based approach.

Lenses (Formally)

A lens between a set of source structures S and a set of target structures T is a pair of functions

get	from S to T
putback	from $T \times S$ to S



A Sample Lens

```
Xml.flatten;
hoist "contacts"; List.hd []; hoist "contact";
List.map (mapp {"n"} (List.hd []; hoist "pcdata";
                      List.hd []);
          pivot "n");
List.flatten;
map (List.hd [];
     map (List.map (hoist "pcdata"; List.hd []));
     acond {} [] (const [] {})  (hoist "studio"))
```

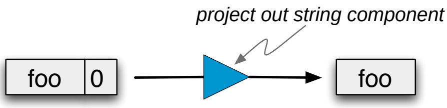
What is a “Reasonable” Lens?

To design a nice programming language, we need some design principles to

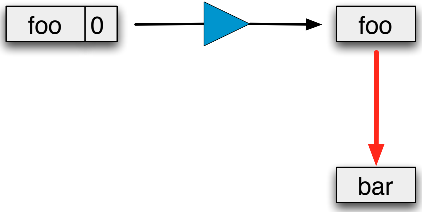
- allow us to recognize and reject bad (unreasonable) primitives and bad (non-reasonableness-preserving) combining forms
- give users a means to understand and predict the behavior of programs in our language

An Unreasonable Example

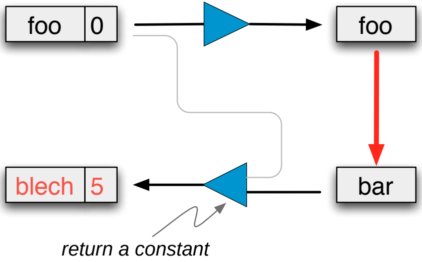
An Unreasonable Example



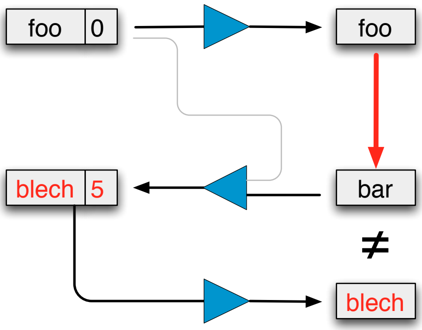
An Unreasonable Example



An Unreasonable Example



An Unreasonable Example



Acceptability

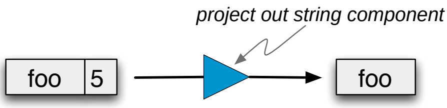
Principle:

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

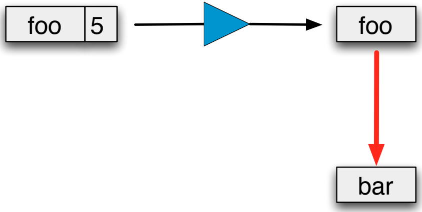
Formally:

$$get(putback(t, s)) \quad = \quad t$$

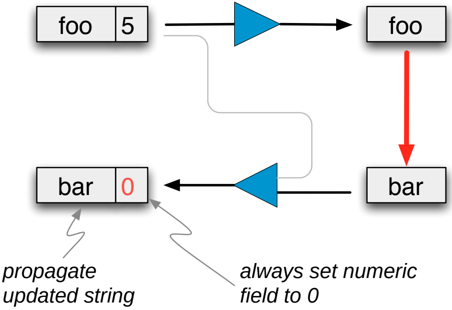
Another Unreasonable Example



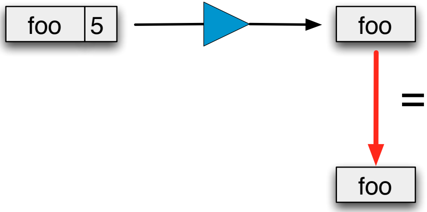
Another Unreasonable Example



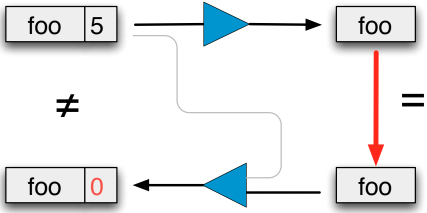
Another Unreasonable Example



Another Unreasonable Example



Another Unreasonable Example



Stability

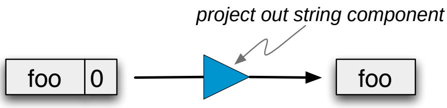
Principle:

> *If the target does not change, neither should the source.*

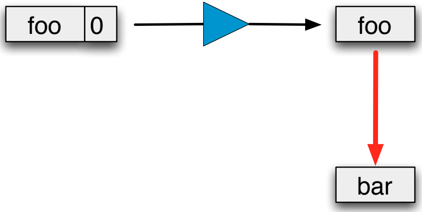
Formally:

$$putback(get(s), s) \quad = \quad s$$

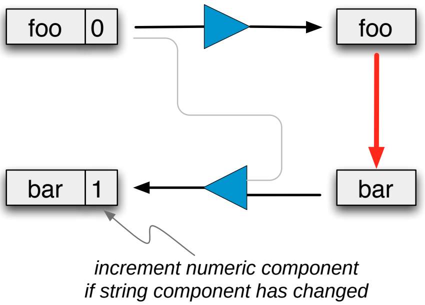
A Debatable Example



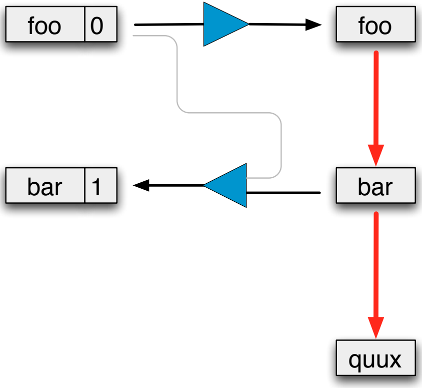
A Debatable Example



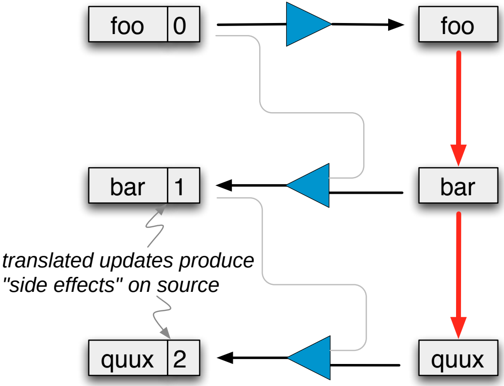
A Debatable Example



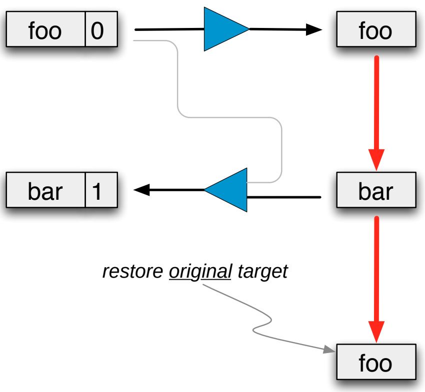
A Debatable Example



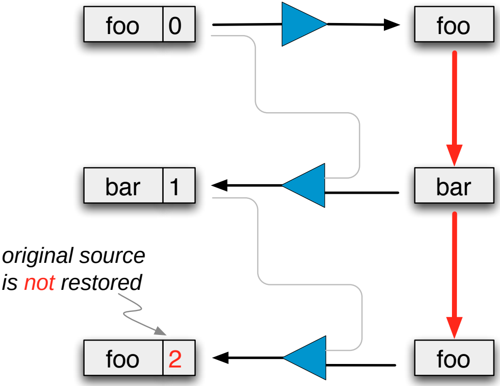
A Debatable Example



A Debatable Example



A Debatable Example



Forgetfulness

Principle:

> *Each update should completely overwrite the effect of the previous one. Thus, the effect of two putbacks in a row should be the same as just the second.*

Formally:

$$putback(t_2,\ putback(t_1,\ s)) \quad = \quad putback(t_2,\ s)$$

Nice properties:

- Implies that S is isomorphic to $T \times U$ for some U
- Bancilhon and Spyratos's notion of preserving a "constant complement" is a slight refinement of this.

Forgetfulness

Principle:

> *Each update should completely overwrite the effect of the previous one. Thus, the effect of two putbacks in a row should be the same as just the second.*

Formally:

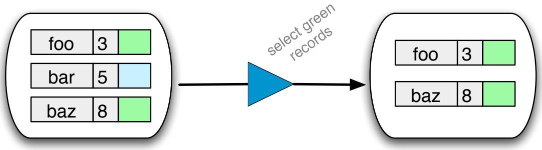
$$\mathit{putback}(t_2,\ \mathit{putback}(t_1,\ s)) \quad = \quad \mathit{putback}(t_2,\ s)$$

Nice properties:

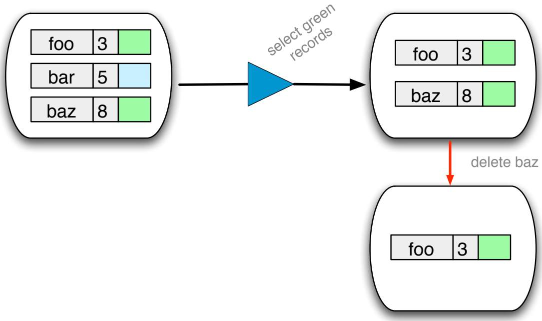
- Implies that S is isomorphic to $T \times U$ for some U
- Bancilhon and Spyratos's notion of preserving a "constant complement" is a slight refinement of this.

Seems sensible. But do we want to require it of *all* lenses?

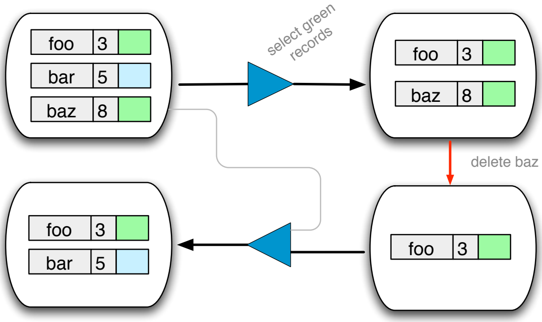
More Examples To Think About



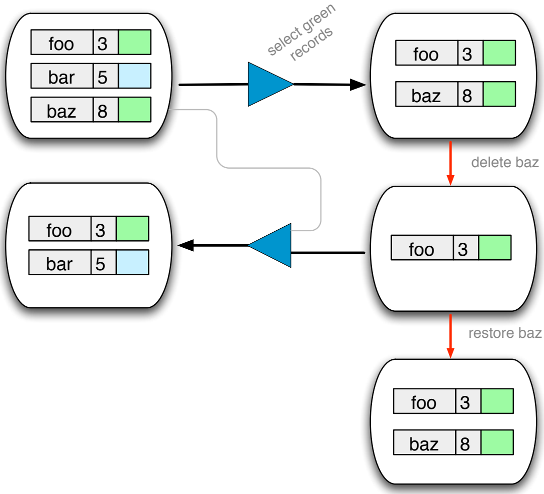
More Examples To Think About



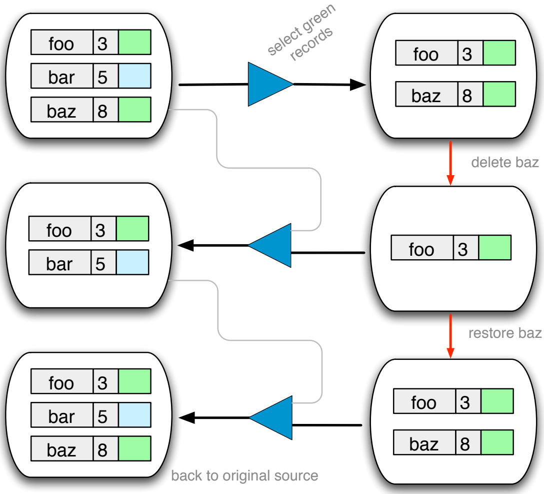
More Examples To Think About



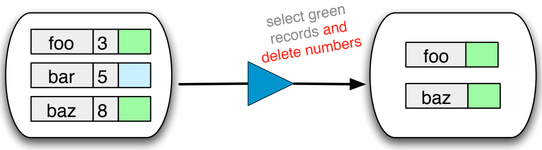
More Examples To Think About



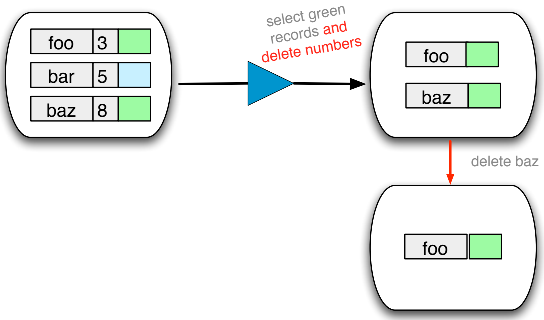
More Examples To Think About



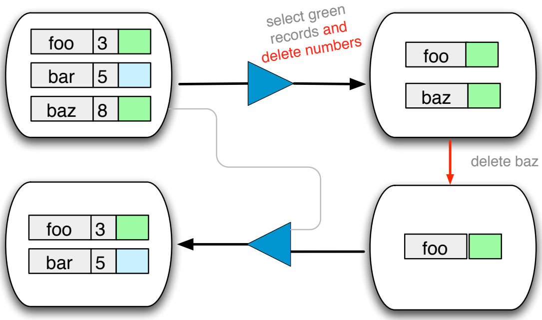
More Examples To Think About



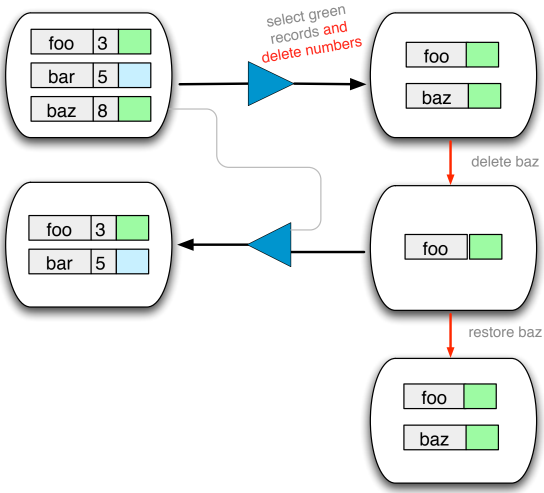
More Examples To Think About



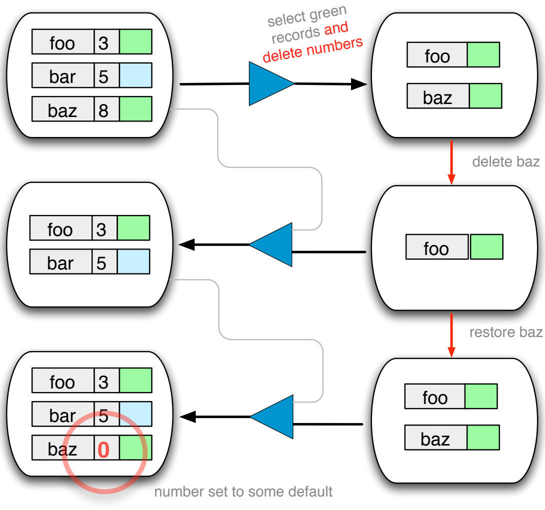
More Examples To Think About



More Examples To Think About



More Examples To Think About



What To Do?

Should we...

- Demand forgetfulness and lose the ability to handle deletion in some cases?
- Not demand forgetfulness and lose the guarantee of undoability?

What To Do?

Should we...

- Demand forgetfulness and lose the ability to handle deletion in some cases?
- Not demand forgetfulness and lose the guarantee of undoability?

Better: keep both as possibilities

- Do not demand forgetfulness of all lenses
- But provide a way to easily check that it holds in particular cases

Another Special Case: Bijective Lenses

A lens whose *putback* function ignores its second (source) argument is called bijective.

Too strong for many applications.

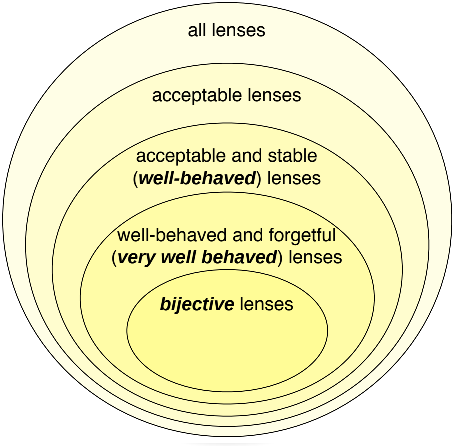
But nice when it holds!

- behavior very predictable and easy to understand
- simplifies notations (allowing defaults to be omitted, etc.)

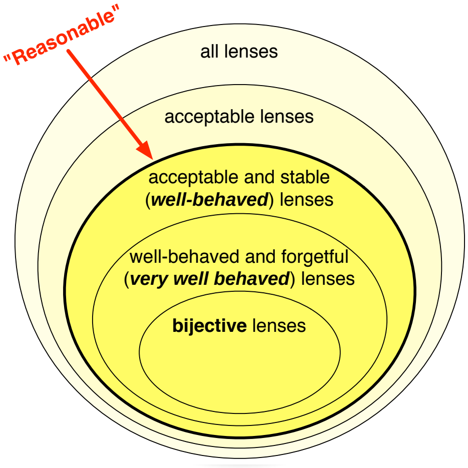
Again, we should keep both possibilities open:

- do not demand bijectiveness of all lenses
- but provide a way to tell when it holds

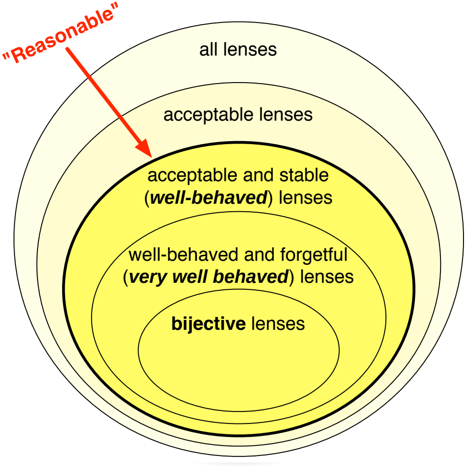
A Lens Bestiary



A Lens Bestiary



A Lens Bestiary



Also, if we had time: *partial*, *monotone*, ...

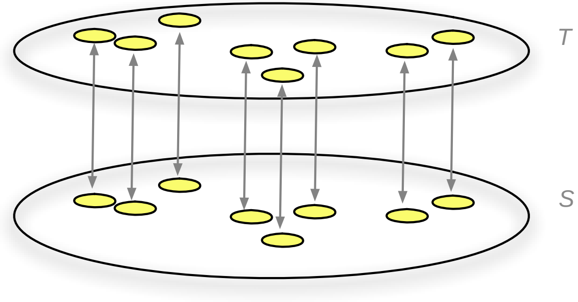
Notation: Lens Types

$S \overset{wb}{\Longleftrightarrow} T$	well-behaved lenses from S to T
$S \overset{vwb}{\Longleftrightarrow} T$	very well behaved lenses from S to T
$S \overset{bij}{\Longleftrightarrow} T$	bijective lenses from S to T
$S \overset{\alpha}{\Longleftrightarrow} T$	lenses with property $\alpha \in \{wb, vwb, bij\}$

How Many Putbacks?

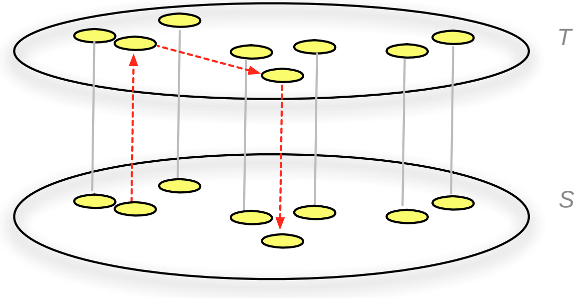
To deepen intuitions about these different subclasses of reasonable lenses, let's try a little visualization exercise...

How Many Putbacks? (Bijective Case)



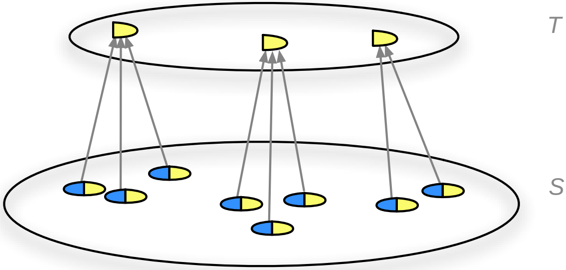
A bijective lens defines a one-to-one correspondence between S and T.

How Many Putbacks? (Bijective Case)



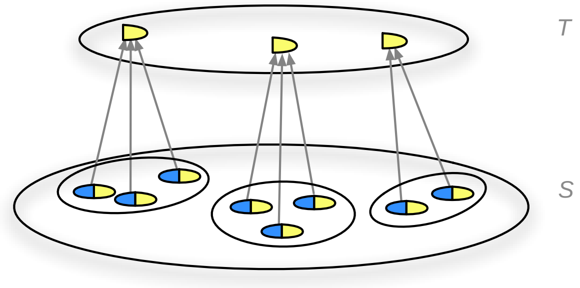
The behavior of the *putback* function is thus completely fixed by the behavior of *get*.

How Many Putbacks? (Very Well Behaved Case)



If we are defining a very well behaved lens, then many structures from S can map onto the same structure from T.

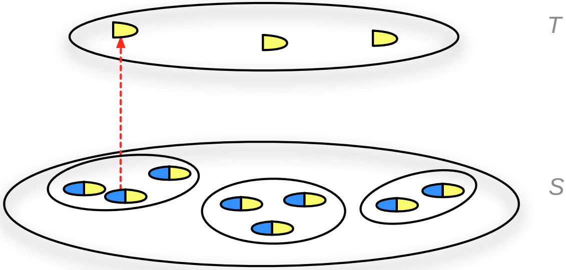
How Many Putbacks? (Very Well Behaved Case)



Source structures partitioned by the equivalence relation

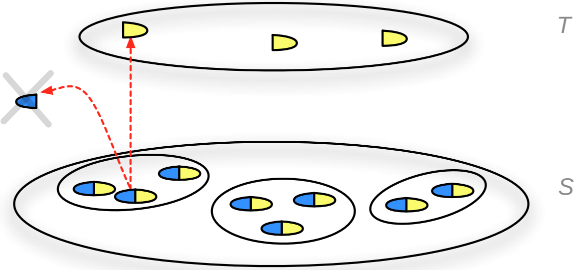
$$s_1 \sim s_2 \quad \Longleftrightarrow \quad get(s_1) = get(s_2)$$

How Many Putbacks? (Very Well Behaved Case)



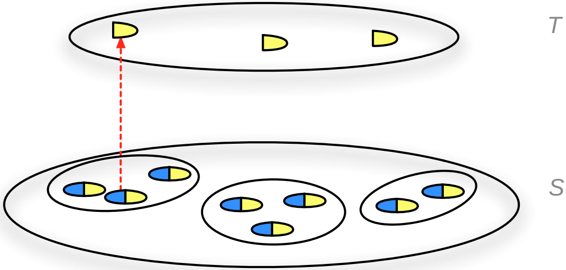
The *get* function projects out part of the information in the source structure...

How Many Putbacks? (Very Well Behaved Case)



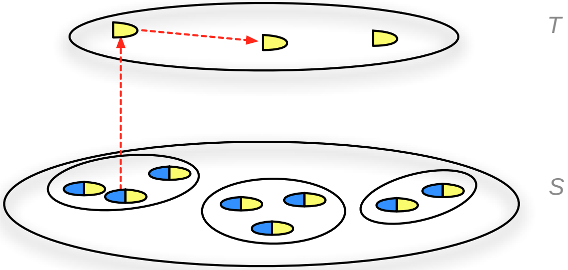
The *get* function projects out part of the information in the source structure... and throws away the rest.

How Many Putbacks? (Very Well Behaved Case)



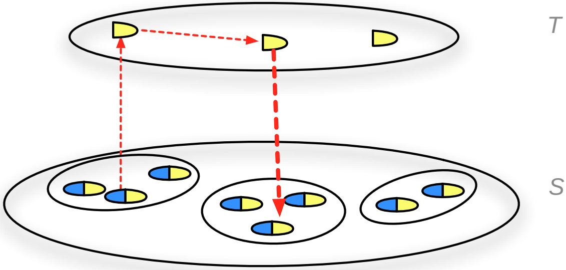
If the target structure is modified...

How Many Putbacks? (Very Well Behaved Case)



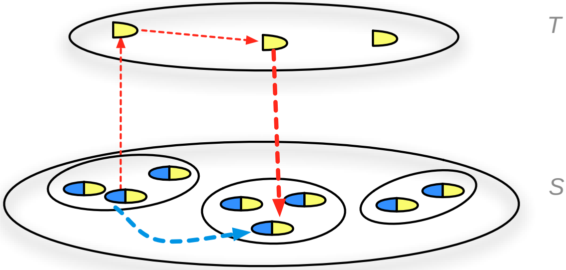
If the target structure is modified. . .

How Many Putbacks? (Very Well Behaved Case)



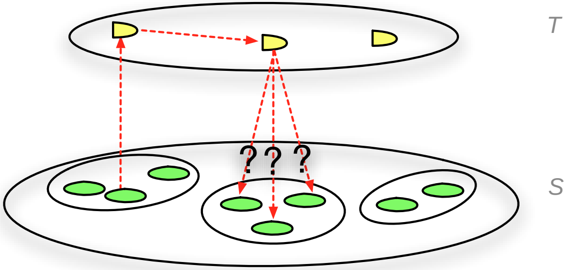
If the target structure is modified. . . the “target part” of the new source structure is fixed by *acceptability*...

How Many Putbacks? (Very Well Behaved Case)



If the target structure is modified. . . the "target part" of the new source structure is fixed by *acceptability*... and the "projected away part" is fixed by *forgetfulness* to be exactly the one from the original source.

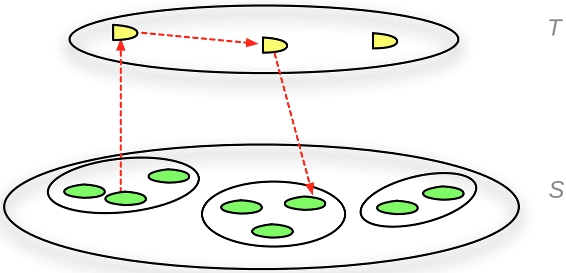
How Many Putbacks? (Well-Behaved Case)



However, if we are defining a well-behaved lens, the behavior of *putback* is constrained only by *acceptability*.

Many *putback*s to choose from!

How Many Putbacks? (Well-Behaved Case)



Need extra information to select one.

Lenses for Trees

Lenses for Trees

The rest of the talk focuses on Harmony's language for bi-directional tree transformations.

Applications include:

- mappings from various XML/HTML bookmark forms to a common "abstract bookmark" schema
- mappings between calendar formats (icalendar, palm calendar)
- mappings between address book formats (xcard, csv)
- (under construction) translators for XMI files, drawings, bibtex files, MS Access databases, ...

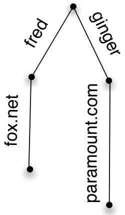
Overview

- Generic lenses
  - identity, composition, conditionals, recursion
- Structure manipulation lenses
  - Lenses that modify the shape of the tree near the root
  - `hoist`, `plunge`, `pivot`, ...
- Tree navigation lenses
  - Apply different lenses to different parts of the tree, or one lens deeper in the tree
  - `map`, `fork`, ...
- "Database-like" lenses
  - `flatten`, `join`, ...
- Structure replication lenses
  - `merge`, `copy`, ...

Trees

Core data model: unordered, edge-labeled trees with no duplicate edge labels at a given node.

(I.e., a tree is just a partial function from labels to subtrees.)



More complex concrete data formats (lists, XML, etc.) are straightforwardly encoded as unordered trees.

Tree Types

Types are sets of trees.

$\Longrightarrow$ Type algebra based on regular tree types is a natural fit.

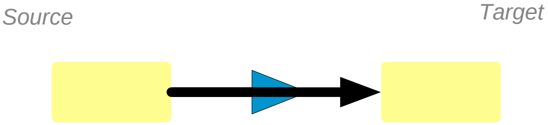
Notation:

$$
\begin{array}{lll}
T ::= & \{n \mapsto T\} & \text{child named } n \text{ with subtree in } T \\
 & \{! \mapsto T\} & \text{child with any name and subtree in } T \\
 & \{* \mapsto T\} & \text{any number of children with subtrees in } T \\
 & T_1 \bullet T_2 & \text{concatenation of } T_1 \text{ and } T_2 \\
 & \ldots & \text{(plus some others we don't need today)}
\end{array}
$$

The Identity Lens

The identity is a bijective lens from any set U to itself.

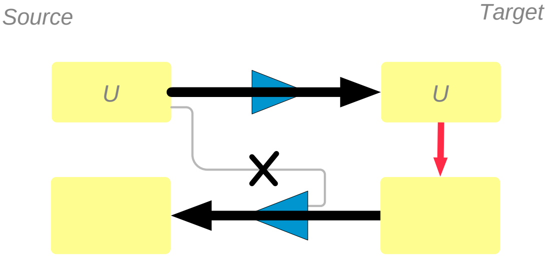
$$\boxed{\texttt{id} \in U \stackrel{bij}{\Longleftrightarrow} U}$$



The Identity Lens

The identity is a bijective lens from any set U to itself.

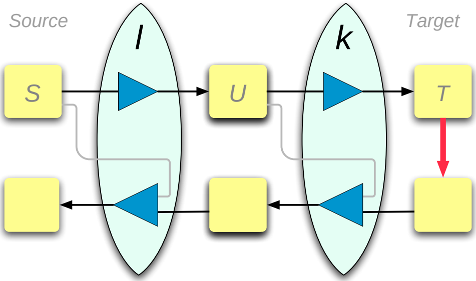
$$\boxed{\texttt{id} \in U \overset{bij}{\Longleftrightarrow} U}$$



Lens Composition

If $l \in S \overset{\alpha}{\Longleftrightarrow} U$ and $k \in U \overset{\alpha}{\Longleftrightarrow} T$ then

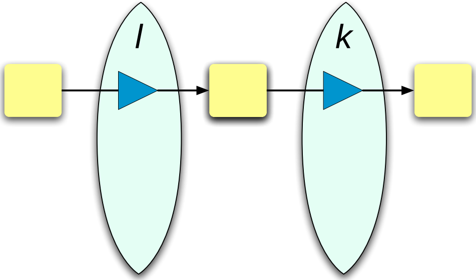
$$\boxed{(l;k) \quad \in \quad S \overset{\alpha}{\Longleftrightarrow} T}$$



Lens Composition

If $l \in S \overset{\alpha}{\Longleftrightarrow} U$ and $k \in U \overset{\alpha}{\Longleftrightarrow} T$ then

$$\boxed{(l;k) \quad \in \quad S \overset{\alpha}{\Longleftrightarrow} T}$$

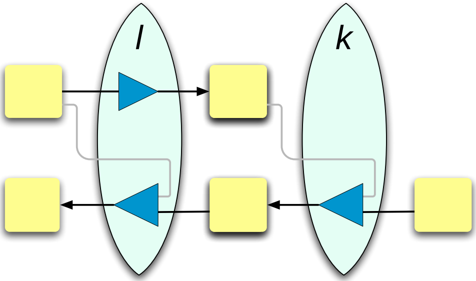


$$get_{l;k}(s) = get_k(get_l(s))$$

Lens Composition

If $l \in S \overset{\alpha}{\Longleftrightarrow} U$ and $k \in U \overset{\alpha}{\Longleftrightarrow} T$ then

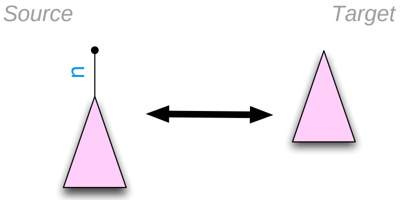
$$\boxed{(l;k) \quad \in \quad S \overset{\alpha}{\Longleftrightarrow} T}$$



$$putback_{l;k}(t, s) = putback_l(putback_k(t, get_l(s)), s)$$

Hoist

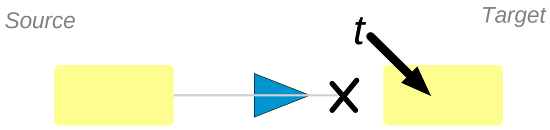
$$\boxed{\texttt{hoist} \quad \in \quad \{n \mapsto U\} \stackrel{bij}{\Longleftrightarrow} U}$$



The *get* function hoists the child under n.

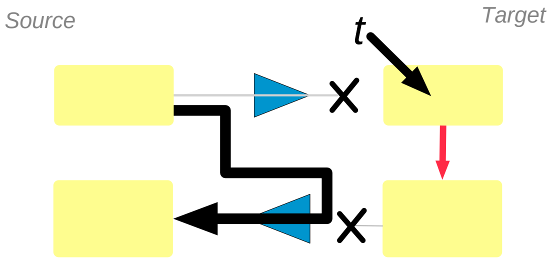
The *putback* function restores the edge n.

The Constant Lens (first version)



The *get* function discards the entire source structure and always yields the tree t.

The Constant Lens (first version)

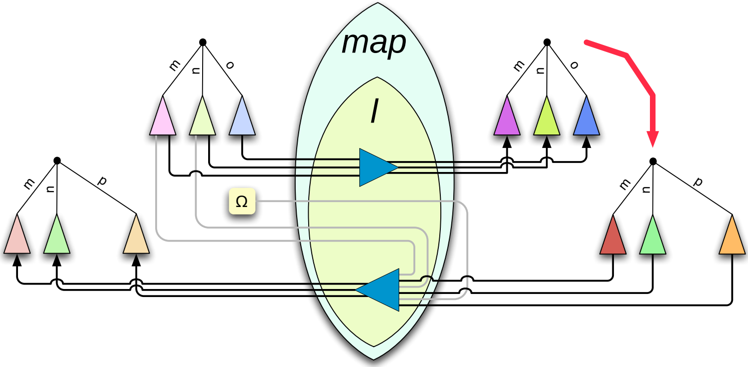


The *get* function discards the entire source structure and always yields the tree *t*.

The *putback* function restores the original source structure.

Map

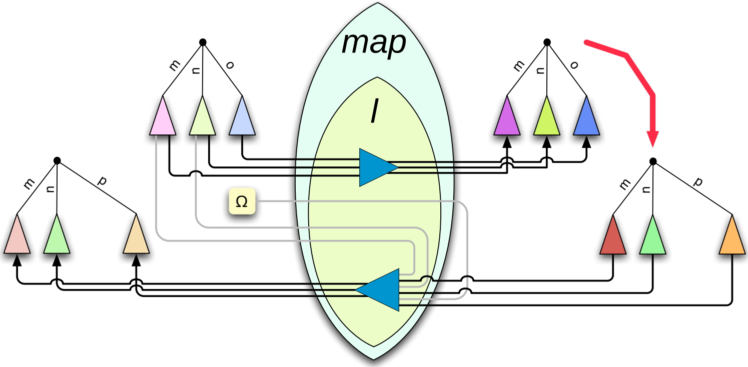
`map` *l* applies a lens *l* to all the children of the root node



Map

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \overset{wb}{\Longleftrightarrow} \{* \mapsto T\}}$$

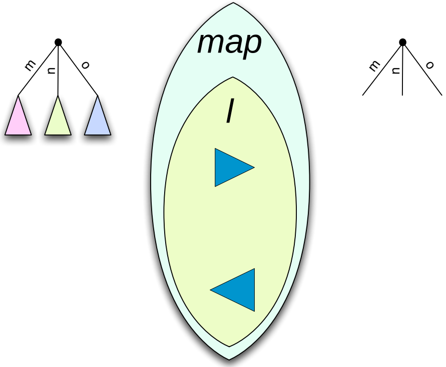
if $l \in S \overset{wb}{\Longleftrightarrow} T$



Map

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \overset{wb}{\Longleftrightarrow} \{* \mapsto T\}}$$

The *get* direction replicates the source structure...

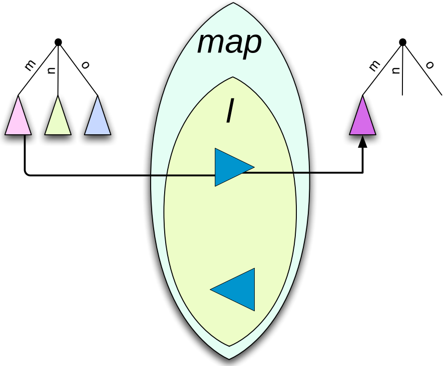


Map

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \overset{wb}{\Longleftrightarrow} \{* \mapsto T\}}$$

The *get* direction replicates the source structure...

and applies `l` to every child

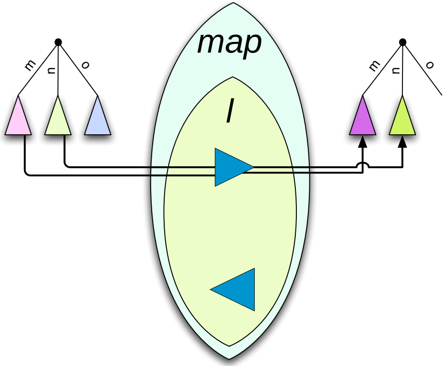


Map

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \overset{wb}{\Longleftrightarrow} \{* \mapsto T\}}$$

The *get* direction replicates the source structure...

and applies `l` to every child

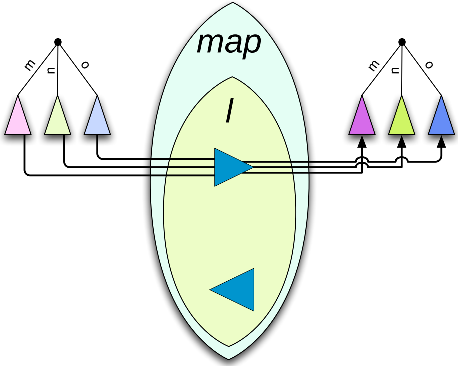


Map

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \overset{wb}{\Longleftrightarrow} \{* \mapsto T\}}$$

The *get* direction replicates the source structure. . .

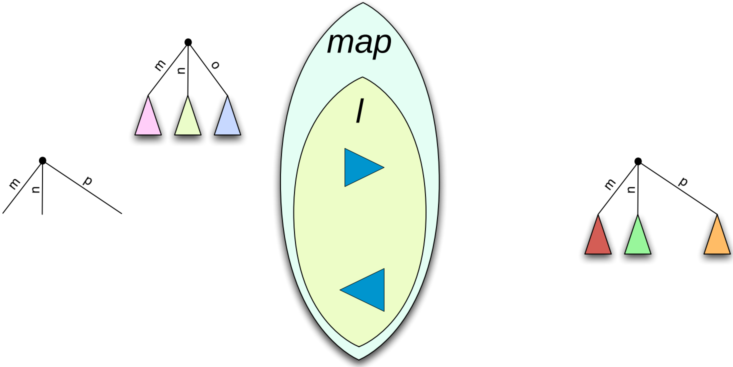
and applies `l` to every child



Map

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \overset{wb}{\Longleftrightarrow} \{* \mapsto T\}}$$

The *putback* direction replicates the target structure...

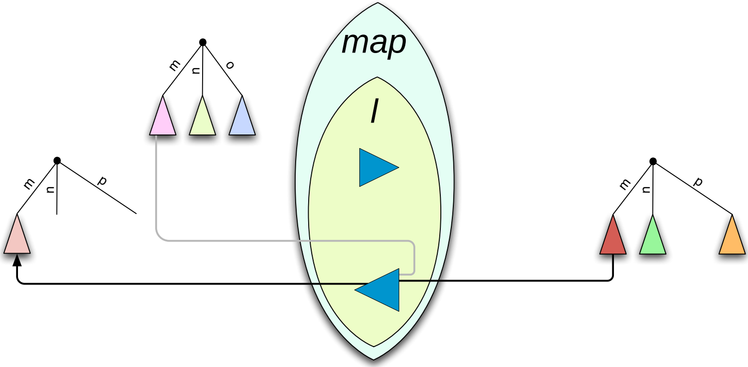


Map

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \overset{wb}{\Longleftrightarrow} \{* \mapsto T\}}$$

The *putback* direction replicates the target structure...

and applies `l` to every child using the source trees

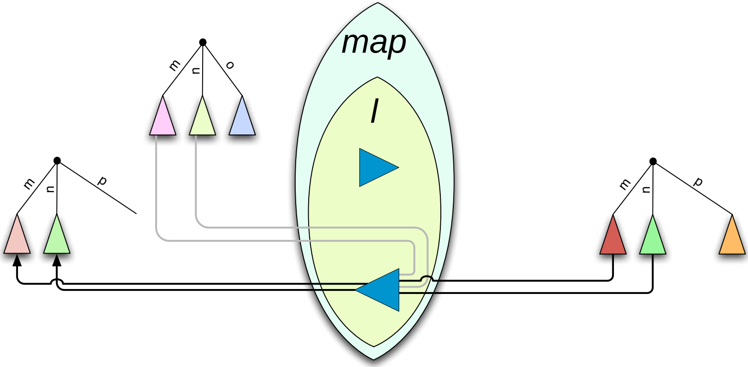


Map

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \overset{wb}{\Longleftrightarrow} \{* \mapsto T\}}$$

The *putback* direction replicates the target structure...

and applies `l` to every child using the source trees

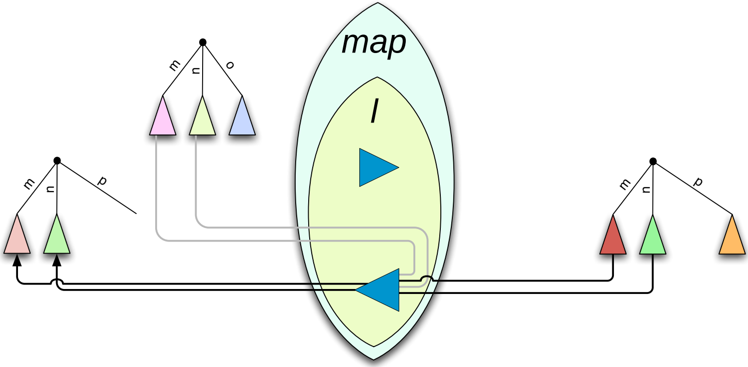


Map

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \stackrel{wb}{\Longleftrightarrow} \{* \mapsto T\}}$$

The *putback* direction replicates the target structure. . .

deleting children that are absent

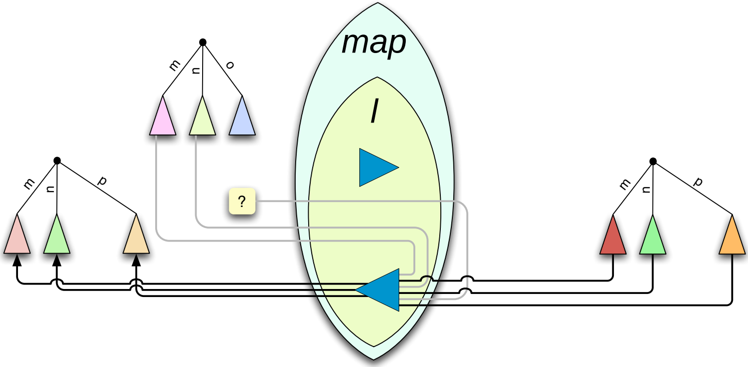


Map

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \overset{wb}{\Longleftrightarrow} \{* \mapsto T\}}$$

The *putback* direction replicates the target structure...

creating new children, using which source tree?

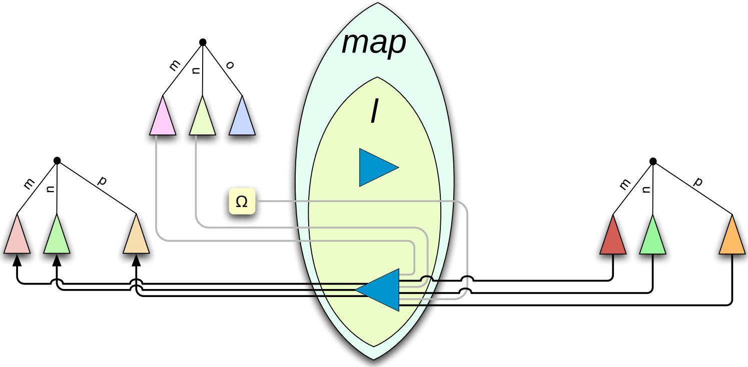


Map

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \stackrel{wb}{\Longleftrightarrow} \{* \mapsto T\}}$$

The *putback* direction replicates the target structure. . .

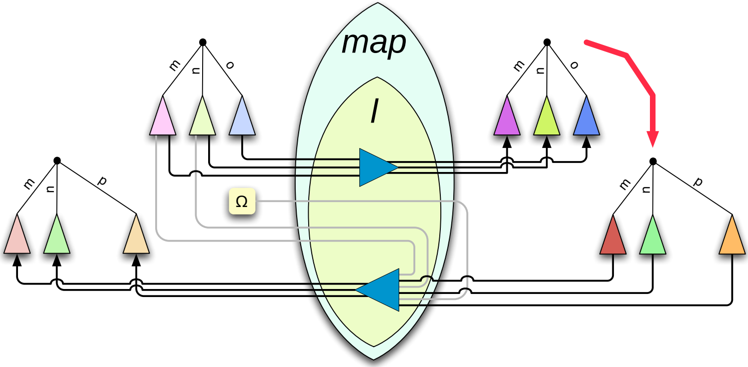
creating new children, using the special “missing tree” Ω



Map (alternate typing)

$$\boxed{\texttt{map}\ l \quad \in \quad \{* \mapsto S\} \overset{vwb}{\Longleftrightarrow} \{* \mapsto T\}}$$

if $l \in S \overset{bij}{\Longleftrightarrow} T$



Creation

Doing *putback*(t, Ω) corresponds to creating an element of S given just an element $t \in T$.

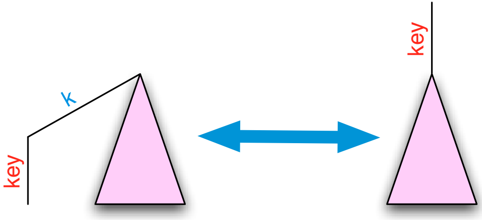
Formally, we enrich the source and target of all lenses with the element Ω.

Lenses whose *get* functions discard information (like `const`) now need to be extended to handle Ω (by providing defaults).

Pivot

$$\boxed{\texttt{pivot}\ k \quad \in \quad \{k \mapsto \{! \mapsto \{\}\} \bullet U \overset{bij}{\Longleftrightarrow} \{! \mapsto U\}}$$

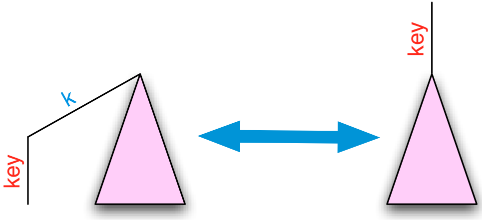
$\texttt{pivot}\ k$ performs fetches the key under k and puts it at the root of the tree.



Pivot

$$\boxed{\texttt{pivot}\ k \quad \in \quad \{k \mapsto \{! \mapsto \{\}\} \bullet U \overset{bij}{\Longleftrightarrow} \{! \mapsto U\}}$$

$\texttt{pivot}\ k$ performs fetches the key under k and puts it at the root of the tree.

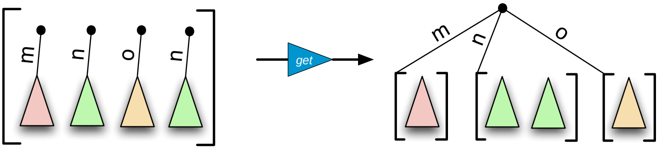


Typical use: choose a key before a `flatten`...

Flatten

$$\boxed{\texttt{flatten} \quad \in \quad \texttt{List}(\{! \mapsto U\}) \overset{wb}{\Longleftrightarrow} \{* \mapsto \texttt{List}(U)\}}$$

The `flatten` lens takes a list of keyed trees and flattens it into a tree of lists, where the top-level children are the keys from the original list:



putback: exercise!

Conditional

Any serious language needs some kind of conditional.

How could this work in a bi-directional setting?

Conditional

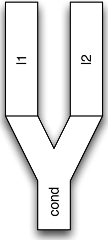
Any serious language needs some kind of conditional.

How could this work in a bi-directional setting?

Critical issue: How do we ensure reasonableness, when all we know about the two branches of the conditional is that each is reasonable by itself (i.e., if both *get* and *putback* go through the same branch)?

Conditional: *get* direction

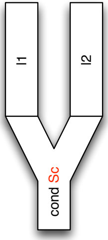
Choose a lens depending on some property of the structure



Conditional: *get* direction

Choose a lens depending on some property of the structure

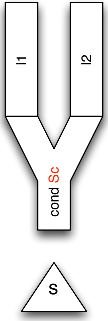
In the get direction: test the source structure (i.e., check if $s \in S_c \subseteq S$)



Conditional: *get* direction

Choose a lens depending on some property of the structure

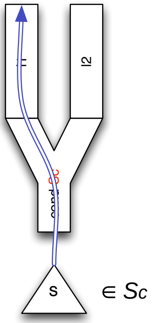
In the get direction: test the source structure (i.e., check if $s \in S_c \subseteq S$)



Conditional: *get* direction

Choose a lens depending on some property of the structure

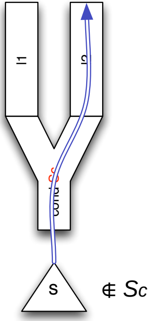
In the get direction: test the source structure (i.e., check if $s \in S_c \subseteq S$)



$$l_1 \in S_c \overset{\alpha}{\Longleftrightarrow} T_1$$

Conditional: *get* direction

Choose a lens depending on some property of the structure
In the get direction: test the source structure (i.e., check if $s \in S_c \subseteq S$)

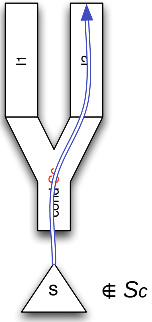


$$l_1 \in S_c \overset{\alpha}{\Longleftrightarrow} T_1 \qquad l_2 \in \overline{S_c} \overset{\alpha}{\Longleftrightarrow} T_2$$

Conditional: *get* direction

Choose a lens depending on some property of the structure

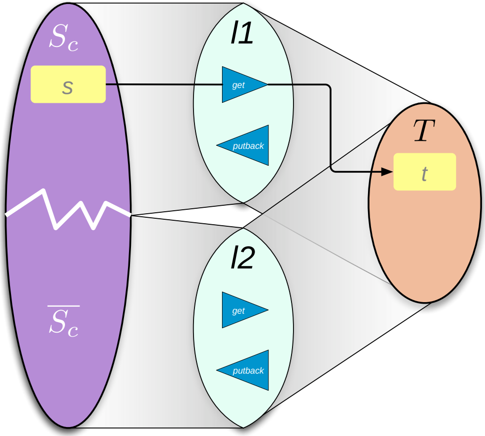
In the get direction: test the source structure (i.e., check if $s \in S_c \subseteq S$)



$$l_1 \in S_c \overset{\alpha}{\Longleftrightarrow} T_1 \qquad l_2 \in \overline{S_c} \overset{\alpha}{\Longleftrightarrow} T_2$$

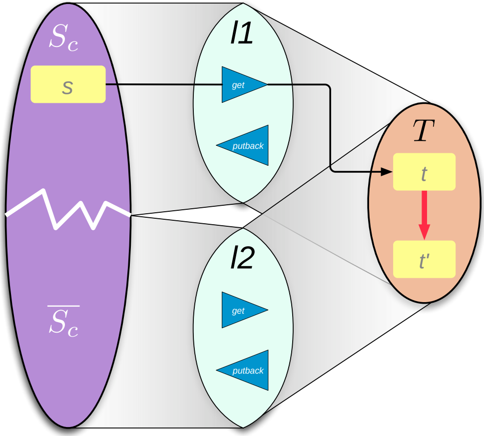
What about the way back?

Conditional: `ccond`



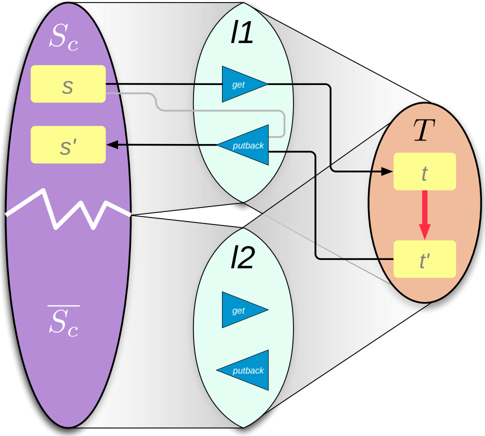
Choose according to the source argument

Conditional: `ccond`



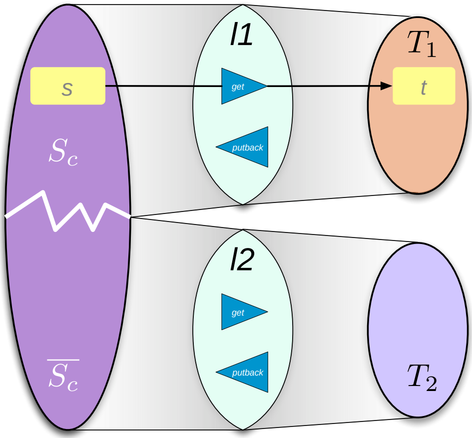
Choose according to the source argument

Conditional: `ccond`



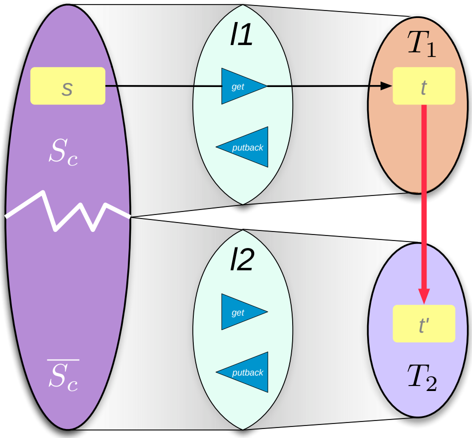
Choose according to the source argument

Conditional: acond



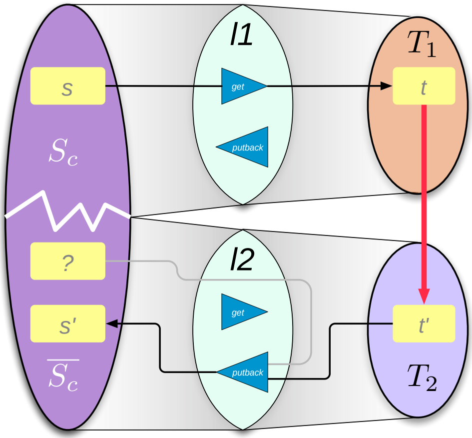
Another alternative: Choose according to the target argument

Conditional: `acond`



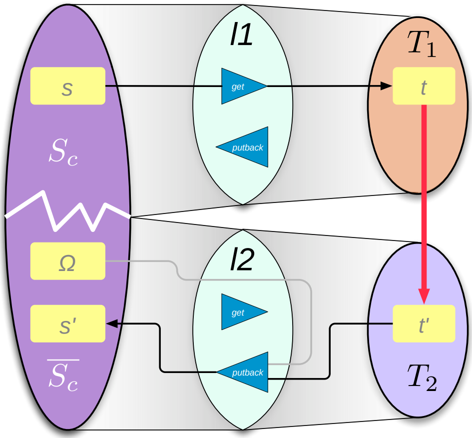
But what about switching domains?

Conditional: acond



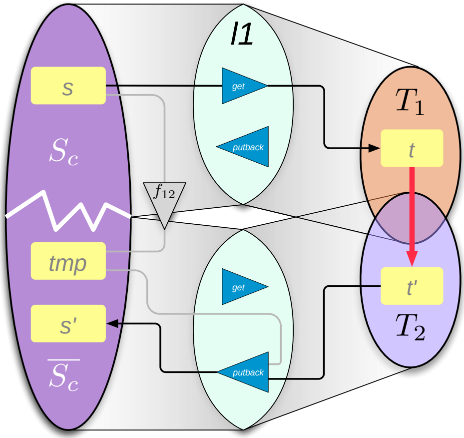
A source structure from $\overline{S_c}$ is needed...

Conditional: `acond`



Consider it as creation

Conditional: The General Case



Combine both and use a fix up function instead of Ω.

Going Further

Derived forms for a wide variety of more complex transformations can be implemented in terms of these primitives.

- more complex structure manipulations
- list processing (map, reverse, filter, group, ...)
- XML processing
- etc., etc.

Lenses For Relations

Quick Sketch

Data model: source and target structures are relational databases (named collections of tables)

Primitives: Operators from relational algebra, each augmented with enough parameters to determine *putback* behavior.

Type system: Built using standard tools from databases

- predicates on rows of tables
- functional dependencies between columns (restricted to "tree form")

See our upcoming PODS 2006 paper for more.

Finishing Up. . .

Related Work

- Semantic Framework — *many* related ideas in database literature (see paper)
  - [Dayal, Bernstein '82] "exact translation"
  - [Bancilhon, Spryatos '81] "translators under constant complement"
  - [Gottlob, Paolini, Zicari '88] "dynamic views"
- Bijective and Reversible Languages (lots)
- Bi-Directional Languages
  - [Meertens] — language for constaint maintainers; similar behavioral laws
  - [Hu, Mu, Takeichi '04] — language at core of a structured document editor

Harmony Status

- Prototype implementation and several demo applications working well
- Distributed operation via integration with Unison file synchronizer
- Starting to be used seriously outside of Penn
  - We're looking for more users... Join the fun!
- Extensive set of demos (including a lens programming playground) available on the web

Ongoing and Future Work

- More / larger applications
- Formal characterizations of expressiveness
  - Is this set of primitives *complete* in some interesting sense?
- Programming puzzles
  - can flatten be written as a derived form?
  - can a linear-time reverse be written as a derived form?
- Algorithmic aspects of static typechecking with tree types
- Relational lenses / database integration
- Lenses over other structures (graphs, streams, ...)
- Lens programming by example (i.e., much higher-level languages sharing the same semantic basis)

Thank You!

Mail collaborators on this work: Aaron Bohannon, Nate Foster, Michael Greenwald, Alan Schmitt, Jeff Vaughan

Other Harmony contributors: Malo Denielou, Michael Greenwald, Owen Gunden, Martin Hofmann, Sanjeev Khanna, Keshav Kunal, Stéphane Lescuyer, Jon Moore, Zhe Yang

Resources: Papers, slides, (open source) code, and online demos:

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