Combinators for Bi-Directional Tree Transformations

A Linguistic Approach to the View Update Problem

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore,
Benjamin C. Pierce, and Alan Schmitt
University of Pennsylvania

ABSTRACT

We propose a novel approach to the well-known *view update problem* for the case of tree-structured data: a domain-specific programming language in which all expressions denote bi-directional transformations on trees. In one direction, these transformations—dubbed *lenses*—map a "concrete" tree into a simplified "abstract view"; in the other, they map a modified abstract view, together with the original concrete tree, to a correspondingly modified concrete tree. Our design emphasizes both robustness and ease of use, guaranteeing strong well-behavedness and totality properties for well-typed lenses.

We identify a natural space of well-behaved bi-directional transformations over arbitrary structures, study definedness and continuity in this setting, and state a precise connection with the classical theory of "update translation under a constant complement" from databases. We then instantiate this semantic framework in the form of a collection of *lens combinators* that can be assembled to describe transformations on trees. These combinators include familiar constructs from functional programming (composition, mapping, projection, conditionals, recursion) together with some novel primitives for manipulating trees (splitting, pruning, merging, etc.). We illustrate the expressiveness of these combinators by developing a number of bi-directional list-processing transformations as derived forms.

Categories and Subject Descriptors.

D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*

General Terms. Languages

Keywords. Bi-directional programming, Harmony, XML, lenses, view update problem

1. INTRODUCTION

Computing is full of situations where one wants to transform some structure into a different form—a *view*—in such a way that changes made to the view can be reflected back as updates to the original structure. This *view update problem* is a classical topic in the database literature, but has so far been little studied by programming language researchers.

This paper addresses a specific instance of the view update problem that arises in a larger project called Harmony [26]. Harmony is a generic framework for synchronizing tree-structured data—a tool for propagating updates between different copies of tree-shaped data structures, possibly stored in different formats. For example, Harmony can be used to synchronize the bookmark files of several different web browsers, allowing bookmarks and bookmark folders to be added, deleted, edited, and reorganized in any browser and propagated to the others. Other Harmony instances currently in daily use or under development include synchronizers for calendars (Palm DateBook, ical, and iCalendar formats), address books, slide presentations, structured documents, and generic XML and HTML.

Views play a key role in Harmony: to synchronize disparate data formats, we define one common abstract view and a collection of *lenses* that transform each concrete format into this abstract view. For example, we can synchronize a Mozilla bookmark file with an Explorer bookmark file by transforming each into an *abstract bookmark structure* and synchronizing the results. Having done so, we need to take the updated abstract structures and perform the corresponding updates to the concrete structures. Thus, each lens must include not one but *two* functions—one for extracting an abstract view from a concrete one and another for pushing an updated abstract view back into the original concrete view to yield an updated concrete view. We call these the *get* and *putback* components, respectively. The intuition is that the mapping from concrete to abstract is commonly some sort of projection, so the *get* direction involves getting the abstract part out of a larger concrete structure, while the *putback* direction amounts to putting a new abstract part into an old concrete structure. We present a concrete example in §2.

The difficulty of the view update problem springs from a fundamental tension between *expressiveness* and *robustness*. The richer we make the set of possible transformations in the *get* direction, the more difficult it becomes to define corresponding functions in the *putback* direction so that each lens is both *well behaved*—its *get* and *putback* behaviors fit together in a sensible way—and *total*—its *get* and *putback* functions are defined on all the inputs to which they may be applied. To reconcile this tension, any approach to the view update problem must be carefully designed with a particular application domain in mind. The approach described here is tuned to the kinds of projection-and-rearrangement transformations on trees and lists that we have found useful for implementing Harmony instances. It does not directly address some well-known difficulties with view update in the classical setting of relational databases—such as the difficulty of "inverting" queries involving joins—though we hope that our work may suggest new attacks on these problems.

A second difficulty concerns *ease of use*. In general, there are many ways to equip a given *get* function with a *putback* function to form a well-behaved and total lens; we need some means of specifying which *putback* is intended that is natural for the application domain and that does not involve onerous proof obligations or checking of side conditions. We adopt a linguistic approach to this issue, proposing a set of lens *combinators*—a small domain-specific language—in which every expression simultaneously specifies both a *get* function and the corresponding *putback*. Moreover, each combinator is accompanied by a *type declaration*, designed so that the well-behavedness and—for non-recursive lenses—totality of composite lens expressions can be verified by straightforward, compositional checks.

The first step in our formal development, in §3, is identifying a natural space of well-behaved lenses over arbitrary data structures. There is a good deal of territory to be explored at this semantic level. First, we must phrase our basic definitions to allow the underlying functions in lenses to be partial, since there will in general be structures to which a given lens cannot sensibly be applied. The sets of structures to which we *do* intend to apply a given lens are specified by associating it with a type of the form $C \rightleftharpoons A$, where C is a set of concrete "source structures" and A is a set of abstract "target structures." Second, we define a notion of well-behavedness that captures our intuitions about how the *get* and *putback* parts of a lens should behave in concert. Third, we use standard tools to define monotonicity and continuity for lens combinators, establishing a foundation for defining lenses by recursion. Finally, to allow lenses to be used to create new concrete structures rather than just updating existing ones (which can happen, e.g., when new records are added to a database in the abstract view), we show how to adjoin a special "missing" element to the structures manipulated by lenses and establish suitable conventions for how it is treated.

We next proceed to syntax. We first (§4) present a group of generic lens combinators (identities, composition, and constants), which can work with any kind of data. Next (§5), we present several more combinators that perform various manipulations on trees (hoisting, splitting, mapping, etc.) and show how to assemble these primitives to yield some useful derived forms. §6 introduces another set of generic combinators implementing various sorts of bi-directional conditionals. §7 gives a more ambitious illustration of the expressiveness of these combinators by implementing a number of bi-directional list-processing transformations; our main example is a bi-directional `list_filter` lens whose *putback* direction performs a rather intricate "weaving" operation to recombine an updated abstract list with the concrete list elements that were filtered away by the *get*. A more pragmatic illustration of the use of our combinators in real-world lens programming may be found in the accompanying technical report [12], where we walk through a substantial example derived from the Harmony bookmark synchronizer.

§8 surveys a variety of related work and states a precise correspondence (amplified in [25]) between our well-behaved lenses and "update translation under a constant complement" from databases. §9 sketches directions for future research. Omitted proofs can be found in [12].

2. A SMALL EXAMPLE

Suppose our concrete tree c is a small address book:

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

(We draw trees sideways to save space. Each set of hollow braces corresponds to a tree node, and each "$\mathtt{X} \mapsto ...$" denotes a child labeled with the string `X`. The children of a node are unordered. To avoid clutter, when an edge leads to an empty tree, we usually omit the braces, the $\mapsto$ symbol, and the final childless node—e.g., "`333-4444`" above stands for "$\{\!|\mathtt{333\text{-}4444} \mapsto \{\!|\!\}|\!\}$." When trees are linearized in running text, we separate children with commas.)

Now, suppose that we want to edit the data from this concrete tree in a simplified format where each name is associated directly with a phone number.

$$a \quad = \quad \left\{\!\!\left|\begin{array}{l}\mathtt{Pat} \mapsto \mathtt{333\text{-}4444} \\ \mathtt{Chris} \mapsto \mathtt{888\text{-}9999}\end{array}\right|\!\!\right\}$$

Why would we want this? Perhaps because the edits are going to be performed by synchronizing this abstract tree with another replica of the same address book in which no URL information is recorded. Or perhaps there is no synchronizer involved, but the edits are going to be performed by a human who is only interested in phone information and whose screen should not be cluttered with URLs. Whatever the reason, we are going to make our changes to the abstract tree a, yielding a new abstract tree a' of the same form but with modified content.[1] For example, let us change `Pat`'s phone number, drop `Chris`, and add a new friend, `Jo`.

$$a' \quad = \quad \left\{\!\!\left|\begin{array}{l}\mathtt{Pat} \mapsto \mathtt{333\text{-}4321} \\ \mathtt{Jo} \mapsto \mathtt{555\text{-}6666}\end{array}\right|\!\!\right\}$$

Lastly, we want to compute a new concrete tree c' reflecting the new abstract tree a'. That is, we want the parts of c' that were kept when calculating a (e.g., `Pat`'s phone number) to be overwritten with the corresponding information from a', while the parts of c that were filtered out (e.g., `Pat`'s `URL`) have their values carried over from c.

$$c' \quad = \quad \left\{\!\!\left|\begin{array}{l} \mathtt{Pat} \mapsto \left\{\!\!\left|\begin{array}{l}\mathtt{Phone} \mapsto \mathtt{333\text{-}4321} \\ \mathtt{URL} \mapsto \mathtt{http://pat.com}\end{array}\right|\!\!\right\} \\ \mathtt{Jo} \mapsto \left\{\!\!\left|\begin{array}{l}\mathtt{Phone} \mapsto \mathtt{555\text{-}6666} \\ \mathtt{URL} \mapsto \mathtt{http://google.com}\end{array}\right|\!\!\right\} \end{array}\right|\!\!\right\}$$

We also need to "fill in" appropriate values for the parts of c' (in particular, `Jo`'s `URL`) that were created in a' and for which c therefore contains no information. Here, we simply set the `URL` to a constant default.

[1]Note that we are interested here in the final tree a', not the particular sequence of edit operations that was used to transform a into a'. This is important in the context of Harmony, where we only have access to the current states of the replicas, rather than a trace of modifications; see [26].

Together, the transformations from c to a and from a' and c to c' form a lens. Our goal is to find a set of combinators that can be assembled to describe a wide variety of lenses in a concise, natural, and mathematically coherent manner. (Just to whet the reader's appetite, the lens expression that implements the transformation sketched above is written `map (focus Phone` $\{\!\!|$`URL` $\mapsto$ `http://google.com`$|\!\!\}$`)`.)

3. SEMANTIC FOUNDATIONS

Although many of our combinators work on trees, their semantic underpinnings can be presented in an abstract setting parameterized by the data structures ("views") manipulated by lenses. In this section—and in §4, where we discuss generic combinators—we simply assume some fixed set $\mathcal{U}$ of views; from §5 on, we will choose $\mathcal{U}$ to be the set of trees.

Basic Structures When f is a partial function, we write $f(a)\downarrow$ if f is defined on argument a and $f(a) = \bot$ otherwise. We write $f(a) \sqsubseteq b$ for $f(a) = \bot \vee f(a) = b$. We write $\mathsf{dom}(f)$ for the set of arguments on which f is defined. When $S \subseteq \mathcal{U}$, we write $f(S)$ for $\{r \mid s \in S \ \wedge\ f(s)\downarrow \wedge\ f(s) = r\}$. We take function application to be strict, i.e., $f(g(x))\downarrow$ implies $g(x)\downarrow$.

3.1 Definition [Lenses]: A *lens* l comprises a partial function $l\nearrow$ from $\mathcal{U}$ to $\mathcal{U}$, called the *get function* of l, and a partial function $l\searrow$ from $\mathcal{U} \times \mathcal{U}$ to $\mathcal{U}$, called the *putback function*.

3.2 Definition [Well-behaved lenses]: Let l be a lens and let C and A be subsets of $\mathcal{U}$. We say that l is a *well behaved* lens from C to A, written $l \in C \rightleftharpoons A$, iff it maps arguments in C to results in A and vice versa

$$l\nearrow(C) \subseteq A \qquad (\textsc{Get})$$
$$l\searrow(A \times C) \subseteq C \qquad (\textsc{Put})$$

and its *get* and *putback* functions obey the following laws:

$$l\searrow(l\nearrow c,\ c) \sqsubseteq c \qquad \text{for all } c \in C \qquad (\textsc{GetPut})$$
$$l\nearrow(l\searrow(a,\ c)) \sqsubseteq a \qquad \text{for all } (a, c) \in A \times C \qquad (\textsc{PutGet})$$

We call C the *source* and A the *target* in $C \rightleftharpoons A$.

Intuitively, the GetPut law states that, if we *get* some abstract view a from a concrete view c and immediately *putback* a (with no modifications) into c, we must get back exactly c (if both operations are defined). PutGet, on the other hand, demands that the *putback* function must capture all of the information contained in the abstract view: if putting a view a into a concrete view c yields a view c', then the abstract view obtained from c' is exactly a.

An example of a lens satisfying PutGet but not GetPut is the following. Suppose $C = \texttt{string}\times\texttt{int}$ and $A = \texttt{string}$, and define l by $l\nearrow(s, n) = s$ and $l\searrow(s', (s, n)) = (s', 0)$. Then $l\searrow(l\nearrow(s, 1), (s, 1)) = (s, 0) \neq (s, 1)$. Intuitively, the law fails because the *putback* function has "side effects": it modifies information from the concrete view that is not reflected in the abstract view.

An example of a lens satisfying GetPut but not PutGet is the following. Let $C = \texttt{string}$ and $A = \texttt{string} \times \texttt{int}$, and define l by $l\nearrow s = (s, 0)$ and $l\searrow((s', n),\ s) = s'$. PutGet fails here because some information contained in the abstract view does not get propagated to the concrete view. For example, $l\nearrow(l\searrow((s', 1),\ s)) = l\nearrow s' = (s', 0) \neq (s', 1)$.

The GetPut and PutGet laws reflect fundamental expectations about the behavior of lenses; removing either law significantly weakens the semantic foundation. We may also consider an optional third law, called PutPut, stating that $l\searrow(a',\ l\searrow(a,\ c)) \sqsubseteq l\searrow(a',\ c)$ for all $a, a' \in A$ and $c \in C$. This law states that the effect of a sequence of two *putback*s is (modulo undefinedness) just the effect of the second: the first gets completely overwritten. Alternatively, a series of changes to an abstract view may be applied either incrementally or all at once, resulting in the same final concrete view. We say that a well-behaved lens that also satisfies PutPut is *very well behaved*. Both well-behaved and very well behaved lenses correspond to well-known classes of "update translators" from the classical database literature (see §8).

However, when we come to defining our lens combinators for tree transformations in §5, we will not require PutPut because one of our most important lens combinators, `map`, fails to satisfy it(see §5).

A final important property of lenses is *totality*.

3.3 Definition [Totality]: A lens $l \in C \rightleftharpoons A$ is said to be *total*, written $l \in C \Longleftrightarrow A$, if $C \subseteq \mathsf{dom}(l\nearrow)$ and $A \times C \subseteq \mathsf{dom}(l\searrow)$.

The reasons for considering both partial and total lenses instead of building totality into the definition of well-behavedness are much the same as in conventional functional languages. In practice, we always want lenses to be total: to make Harmony synchronizers work predictably, lenses must be defined on the whole of the domains where they are used. All of our primitive lenses are designed to be total, and all of our lens combinators map total lenses to total lenses—with the sole, but important, exception of lenses defined by recursion (to which we will turn in a moment); as usual, recursive lenses must be constructed in the semantics as limits of chains of increasingly defined partial lenses. At the level of types, the type annotations we give for our lens combinators can be used to prove that *any* well-typed lens expression is well-behaved, but only recursion-free expressions can be shown total by completely compositional reasoning; for recursive lenses, more global arguments are required. The long version of the paper gives several of these arguments in detail, showing that all our derived forms involving recursion are actually total. In what follows here, however, we focus on well-behavedness for the sake of brevity.

Recursion Since we will be interested in lenses over trees, and since trees in many interesting application domains may have unbounded depth (e.g., a bookmark item can be either a link or a folder containing a list of items), we will often want to define lenses by recursion.

The development follows familiar lines. We introduce an information ordering on lenses and show that the set of lenses equipped with this ordering is a complete partial order (cpo). We then apply standard tools from domain theory to interpret a variety of common syntactic forms from programming languages—in particular, functional abstraction and application ("higher-order lenses") and lenses defined by single or mutual recursion.

We say that a lens l' is *more informative* than a lens l, written $l \prec l'$, if both the *get* and *putback* functions of l' have domains that are at least as large as those of l and if their results agree on their common domains.

A *cpo* is a partially ordered set in which every increasing chain of elements has a least upper bound in the set. A *cpo* *with bottom* is a cpo with an element $\perp$ that is smaller than every other element. In our setting, $\perp$ is the lens whose *get* and *putback* functions are everywhere undefined.

3.4 Lemma: Let $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ be an increasing chain of lenses. The lens l defined by

$$\begin{array}{ll} l \searrow (a,\, c) = l_i \searrow (a,\, c) & \text{if } l_i \searrow (a,\, c) \downarrow \text{ for some } i \\ l \nearrow c = l_i \nearrow c & \text{if } l_i \nearrow c \downarrow \text{ for some } i \end{array}$$

and undefined elsewhere is a least upper bound for the chain.

3.5 Lemma: Let $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ be an increasing chain of lenses, and let $C_0 \subseteq C_1 \subseteq \ldots$ and $A_0 \subseteq A_1 \subseteq \ldots$ be increasing chains of subsets of $\mathcal{U}$. Then$(\forall i \in \omega.\ l_i \in C_i \rightleftharpoons A_i) \implies (\bigsqcup_n l_n) \in (\bigcup_i C_i) \rightleftharpoons (\bigcup_i A_i)$.

3.6 Theorem: Let $\mathcal{L}$ be the set of well-behaved lenses from C to A. Then $(\mathcal{L}, \prec)$ is a cpo with bottom.

We can now apply standard domain theory to interpret recursive definitions: the least fixed point of a continuous function on well-behaved lenses is a well-behaved lens.

Dealing with Creation In practice, there will be cases where we need to apply a *putback* function, but where no old concrete view is available, as we saw with `Jo`'s `URL` in §2. We deal with these cases by enriching the universe $\mathcal{U}$ of views with a special placeholder Ω, pronounced "missing," which we assume is not already in $\mathcal{U}$. When $S \subseteq \mathcal{U}$, we write S_Ω for $S \cup \{\Omega\}$.

Intuitively, $l \searrow (a,\, \Omega)$ means "create a *new* concrete view from the information in the abstract view a." By convention, Ω is only used in an interesting way when it is the second argument to the *putback* function: in all of the lenses defined below, 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 $C \overset{\Omega}{\rightleftharpoons} A$ for the set of well-behaved lenses from C_Ω to A_Ω obeying these conventions. For brevity in the lens definitions below, we always assume that $c \neq \Omega$ when defining $l \nearrow c$ and that $a \neq \Omega$ when defining $l \searrow (a,\, c)$, since the results in these cases are uniquely determined by these conventions. (There are other, formally equivalent, ways of handling missing concrete views. The advantages of this one are discussed in §5.)

4. GENERIC LENSES

With these semantic foundations in hand, we are ready to move on to syntax. We begin in this section with several *generic* lens combinators, whose definitions are independent of the particular choice of universe $\mathcal{U}$. Each definition is accompanied by a type declaration asserting its well-behavedness under certain conditions (e.g., "the identity lens belongs to $C \overset{\Omega}{\rightleftharpoons} C$ for any C").

Most of the lens definitions in this and following sections are parameterized on one or more arguments. These may be of various types: views, other lenses, predicates on views, edge labels, predicates on labels, etc. The long version includes proofs that every lens we define is well behaved (i.e., that the type declaration accompanying its definition is a theorem) and total, and that every lens that takes other lenses as parameters is continuous in these parameters and maps total lenses to total lenses.

The identity lens copies the concrete view in the *get* direction and the abstract view in the *putback* direction.

$$\boxed{\begin{array}{rcl} \texttt{id} \nearrow c & = & c \\ \texttt{id} \searrow (a,\, c) & = & a \\ \hline \multicolumn{3}{c}{\forall C \subseteq \mathcal{U}. \quad \texttt{id} \in C \overset{\Omega}{\rightleftharpoons} C} \end{array}}$$

The lens composition combinator $l;k$ places two lenses l and k in sequence.

$$\boxed{\begin{array}{rcl} (l;k) \nearrow c & = & k \nearrow (l \nearrow c) \\ (l;k) \searrow (a,\, c) & = & l \searrow (k \searrow (a,\, l \nearrow c),\, c) \\ \hline \multicolumn{3}{l}{\forall A,B,C \subseteq \mathcal{U}.\ \forall l \in C \overset{\Omega}{\rightleftharpoons} B.\ \forall k \in B \overset{\Omega}{\rightleftharpoons} A.} \\ \multicolumn{3}{l}{\quad l;k \in C \overset{\Omega}{\rightleftharpoons} A} \end{array}}$$

The *get* direction applies the *get* function of l to yield a first abstract view, on which the *get* function of k is applied. In the other direction, the two *putback* functions are applied in turn: first, the *putback* function of k is used to put a into the concrete view that the *get* of k was applied to, i.e., $l \nearrow c$; the result is then put into c using the *putback* function of l. (If the concrete view c is Ω, then, $l \nearrow c$ will also be Ω by our conventions on Ω, so the effect of $(l;k) \searrow (a,\, \Omega)$ is to use k to put a into Ω and then l to put the result into Ω.)

Another simple combinator is $\texttt{const}\ v\ d$, which transforms any view into the constant view v in the *get* direction. In the *putback* direction, `const` simply restores the old concrete view if one is available; if the concrete view is Ω, it returns a default view d.

$$\boxed{\begin{array}{rcll} (\texttt{const}\ v\ d) \nearrow c & = & v & \\ (\texttt{const}\ v\ d) \searrow (a,\, c) & = & c & \text{if } c \neq \Omega \\ & & d & \text{if } c = \Omega \\ \hline \multicolumn{4}{l}{\forall C \subseteq \mathcal{U}.\ \forall v \in \mathcal{U}.\ \forall d \in C. \quad \texttt{const}\ v\ d \in C \overset{\Omega}{\rightleftharpoons} \{v\}} \end{array}}$$

Note that the type declaration demands that the *putback* direction only be applied to the abstract argument v.

We will define a few more generic lenses in §6; for now, though, let us turn to some lens combinators that work on tree-structured data, so that we can ground our definitions in specific examples.

5. LENSES FOR TREES

To keep our lens definitions as straightforward as possible, we work with an extremely simple form of trees: unordered, edge-labeled trees with no repeated labels. This does not give us—primitively—all the structure we need for some applications; in particular, we will need to deal with ordered data such as lists and XML documents via an encoding instead of primitively. Experience has shown that the reduction in the complexity of the lens *definitions* obtained in this way far outweighs the increase in complexity of lens *programs* due to manipulating ordered data in encoded form.

Notation From this point on, we choose the universe $\mathcal{U}$ to be the set $\mathcal{T}$ of finite, unordered, edge-labeled trees with labels drawn from some infinite set $\mathcal{N}$ of *names*—e.g., character strings—and with the children of a given node all labeled with distinct names.

A tree is essentially a finite partial function from names to other trees. It will be more convenient, though, to adopt a slightly different perspective: we will consider a tree $t \in \mathcal{T}$ to be a *total* function from $\mathcal{N}$ to $\mathcal{T}_\Omega$ that yields Ω on all but a finite number of names. We write $\mathsf{dom}(t)$ for the domain of t—i.e., the set of the names for which it returns something other than Ω—and $t(n)$ for the subtree associated to name n in t, or Ω if $n \notin \mathsf{dom}(t)$.

Tree values are written using hollow curly braces. The empty tree is written $\{\!|\,|\!\}$. (Note that $\{\!|\,|\!\}$, a node with no children, is different from Ω.) We often describe trees by comprehension, writing $\{\!| n \mapsto F(n) \mid n \in N |\!\}$, where F is some function from $\mathcal{N}$ to $\mathcal{T}_\Omega$ and $N \subseteq \mathcal{N}$ is some set of names. When t and t' have disjoint domains, we write $t \cdot t'$ or $\{\!| t\ t' |\!\}$ (the latter especially in multi-line displays) for the tree mapping n to $t(n)$ for $n \in \mathsf{dom}(t)$, to $t'(n)$ for $n \in \mathsf{dom}(t')$, and to Ω otherwise.

When $p \subseteq \mathcal{N}$ is a set of names, we write $\overline{p}$ for $\mathcal{N}\backslash p$, the complement of p. We write $t|_p$ for the restriction of t to children with names from p—i.e., the tree $\{\!| n \mapsto t(n) \mid n \in p \cap \mathsf{dom}(t) |\!\}$—and $t\backslash_p$ for $\{\!| n \mapsto t(n) \mid n \in \mathsf{dom}(t)\backslash p |\!\}$. When p is just a singleton set $\{n\}$, we drop the set braces and write just $t|_n$ and $t\backslash_n$ instead of $t|_{\{n\}}$ and $t\backslash_{\{n\}}$.

To shorten some of the lens definitions, we adopt the conventions that $\mathsf{dom}(\Omega) = \emptyset$, and that $\Omega|_p = \Omega$ for any p.

For writing down types,[2] we extend these tree notations to sets of trees. If $T \subseteq \mathcal{T}$ and $n \in \mathcal{N}$, then $\{\!| n \mapsto T |\!\}$ denotes the set of singleton trees $\{\{\!| n \mapsto t |\!\} \mid t \in T\}$. If $T \subseteq \mathcal{T}$ and $N \subseteq \mathcal{N}$, then $\{\!| N \mapsto T |\!\}$ denotes the set of trees $\{t \mid \mathsf{dom}(t) = N \text{ and } \forall n \in N.\ t(n) \in T\}$ and $\{\!| N \overset{?}{\mapsto} T |\!\}$ denotes the set of trees $\{t \mid \mathsf{dom}(t) \subseteq N \text{ and } \forall n \in N.\ t(n) \in T_\Omega\}$. We write $T_1 \cdot T_2$ for $\{t_1 \cdot t_2 \mid t_1 \in T_1, t_2 \in T_2\}$ and $T(n)$ for $\{t(n) \mid t \in T\} \setminus \{\Omega\}$. If $T \subseteq \mathcal{T}$, then $\mathsf{dom}(T) = \{\mathsf{dom}(t) \mid t \in T\}$.

A *value* is a tree of the special form $\{\!| k \mapsto \{\!|\,|\!\} |\!\}$, often written just k. For instance, the phone number $\{\!| \texttt{333-4444} \mapsto \{\!|\,|\!\} |\!\}$ in the example of §2 is a value.

Hoisting and Plunging Let's warm up with some combinators that perform simple structural transformations on trees of very simple shapes. We will see shortly how to combine these with a powerful "forking" operator to perform related operations on more general sorts of trees.

The lens `hoist` n is used to shorten a tree by removing an edge at the top. In the *get* direction, it expects a tree that has exactly one child, named n. It returns this child, removing the edge n. In the *putback* direction, the value of the old concrete tree is ignored and a new one is created, with a single edge n pointing to the given abstract tree.

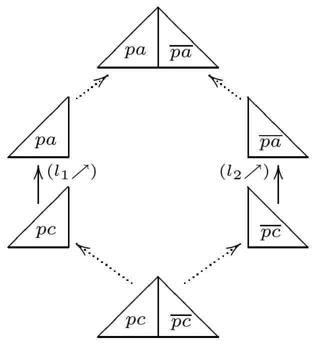
$$\boxed{\begin{array}{l}
\begin{array}{rcll}
(\texttt{hoist}\ n)\nearrow c & = & t & \text{if } c = \{\!| n \mapsto t |\!\} \\
(\texttt{hoist}\ n)\searrow (a,\ c) & = & \{\!| n \mapsto a |\!\} &
\end{array} \\
\hline
\forall C\subseteq\mathcal{T}.\ \forall n\in\mathcal{N}.\quad \texttt{hoist}\ n \in \{\!| n \mapsto C |\!\} \stackrel{\Omega}{\rightleftharpoons} C
\end{array}}$$

Conversely, the `plunge` lens is used to deepen a tree by adding an edge at the top. In the *get* direction, a new tree is created, with a single edge n pointing to the given concrete tree. In the *putback* direction, the value of the old concrete tree is ignored and the abstract tree is required to have exactly one subtree, labeled n, which becomes the result of the `plunge`.

[2]Note that, although we are defining a syntax for lens expressions, the types used to classify these expressions are semantic—they are just sets of lenses or views. We are not (yet!—see §9) proposing an algebra of types or an algorithm for mechanically checking membership of lens expressions in type expressions.

$$\boxed{\begin{array}{l}
\begin{array}{rcll}
(\texttt{plunge}\ n)\nearrow c & = & \{\!| n \mapsto c |\!\} & \\
(\texttt{plunge}\ n)\searrow (a,\ c) & = & t & \text{if } a = \{\!| n \mapsto t |\!\}
\end{array} \\
\hline
\forall C\subseteq\mathcal{T}.\ \forall n\in\mathcal{N}.\quad \texttt{plunge}\ n \in C \stackrel{\Omega}{\rightleftharpoons} \{\!| n \mapsto C |\!\}
\end{array}}$$

Forking The lens combinator `xfork` applies different lenses to different parts of a tree: it splits the tree into two parts according to the names of its immediate children, applies a different lens to each, and concatenates the results. Formally, `xfork` takes as arguments two sets of names and two lenses. The *get* direction of `xfork` $pc\ pa\ l_1\ l_2$ can be visualized as in the inset figure (the concrete tree is at the bottom). The triangles labeled pc denote trees whose immediate child edges have labels in pc; dotted arrows represent splitting or concatenating trees. The result of applying $l_1\nearrow$ to $c|_{pc}$ (the tree formed by dropping the immediate children of c whose names are not in pc) must be a tree whose top-level labels are in the set pa; similarly, the result of applying $l_2\nearrow$ to $c\backslash_{pc}$ must be in $\overline{pa}$. That is, the lenses l_1 and l_2 are allowed to change the sets of names in the trees they are given, but each must map from its own part of pc to its own part of pa. Conversely, in the *putback* direction, l_1 must map from pa to pc and l_2 from $\overline{pa}$ to $\overline{pc}$. Here is the full definition:



$$\boxed{\begin{array}{l}
\begin{array}{rcl}
(\texttt{xfork}\ pc\ pa\ l_1\ l_2)\nearrow c & = & (l_1\nearrow c|_{pc}) \cdot (l_2\nearrow c\backslash_{pc}) \\
(\texttt{xfork}\ pc\ pa\ l_1\ l_2)\searrow (a,\ c) & = & \\
\multicolumn{3}{r}{(l_1\searrow (a|_{pa},\ c|_{pc})) \cdot (l_2\searrow (a\backslash_{pa},\ c\backslash_{pc}))}
\end{array} \\
\hline
\forall pc, pa\subseteq\mathcal{N}.\ \forall C_1\subseteq\mathcal{T}|_{pc}.\ \forall A_1\subseteq\mathcal{T}|_{pa}. \\
\forall C_2\subseteq\mathcal{T}\backslash_{pc}.\ \forall A_2\subseteq\mathcal{T}\backslash_{pa}. \\
\forall l_1 \in C_1 \stackrel{\Omega}{\rightleftharpoons} A_1.\ \forall l_2 \in C_2 \stackrel{\Omega}{\rightleftharpoons} A_2. \\
\quad \texttt{xfork}\ pc\ pa\ l_1\ l_2 \in (C_1 \cdot C_2) \stackrel{\Omega}{\rightleftharpoons} (A_1 \cdot A_2)
\end{array}}$$

We rely here on our convention that $\Omega|_p = \Omega$ to avoid explicitly splitting out the Ω case in the *putback* direction.

We have now defined enough basic lenses to implement several useful derived forms for manipulating trees.

In many uses of `xfork`, the sets of names specifying where to split the concrete tree and where to split the abstract tree are identical. We can define a simpler `fork` as:

$$\boxed{\begin{array}{l}
\texttt{fork}\ p\ l_1\ l_2 \quad = \quad \texttt{xfork}\ p\ p\ l_1\ l_2 \\
\hline
\forall p\subseteq\mathcal{N}.\ \forall C_1, A_1\subseteq\mathcal{T}|_p.\ \forall C_2, A_2\subseteq\mathcal{T}\backslash_p. \\
\forall l_1 \in C_1 \stackrel{\Omega}{\rightleftharpoons} A_1.\ \forall l_2 \in C_2 \stackrel{\Omega}{\rightleftharpoons} A_2. \\
\quad \texttt{fork}\ p\ l_1\ l_2 \in (C_1 \cdot C_2) \stackrel{\Omega}{\rightleftharpoons} (A_1 \cdot A_2)
\end{array}}$$

We can use `fork` to define a lens that discards all of the children of a tree whose names do not belong to some set p:

$$\boxed{\begin{array}{l}
\texttt{filter}\ p\ d \quad = \quad \texttt{fork}\ p\ \texttt{id}\ (\texttt{const}\ \{\!|\,|\!\}\ d) \\
\hline
\forall C\subseteq\mathcal{T}.\ \forall p\subseteq\mathcal{N}.\ \forall d \in C\backslash_p. \\
\quad \texttt{filter}\ p\ d \in (C|_p \cdot C\backslash_p) \stackrel{\Omega}{\rightleftharpoons} C|_p
\end{array}}$$

In the *get* direction, this lens takes a concrete tree, keeps the children with names in p (using `id`), and throws away the rest (using $\texttt{const}\ \{\!\}\ d$). The tree d is used when putting an abstract tree into a missing concrete tree, providing a default for information that does not appear in the abstract tree but is required in the concrete tree. The type of `filter` follows directly from the types of the three primitive lenses: $\texttt{const}\ \{\!\}\ d$, with type $C\backslash_p \overset{\Omega}{\Longleftrightarrow} \{\{\!\}\}$, the lens `id`, with type $C|_p \overset{\Omega}{\Longleftrightarrow} C|_p$, and `fork` (with the observation that $C|_p = C|_p \cdot \{\!\}$).

Another way to thin a tree is to explicitly specify a child that should be removed if it exists:

$$\boxed{\begin{array}{l} \texttt{prune}\ n\ d \;=\; \texttt{fork}\ \{n\}\ \big(\texttt{const}\ \{\!\}\ \{\!|n \mapsto d|\!\}\big)\ \texttt{id} \\ \hline \forall C \subseteq \mathcal{T}.\ \forall n \in \mathcal{N}.\ \forall d \in C(n). \\ \quad \texttt{prune}\ n\ d \in (C|_n \cdot C\backslash_n) \overset{\Omega}{\rightleftharpoons} C\backslash_n \end{array}}$$

This lens is similar to `filter`, except that (1) the name given is the child to be removed, and (2) the default tree is the one to go under n if the concrete tree is Ω.

Conversely, we can grow a tree in the *get* direction by explicitly adding a child. The type annotation disallows changes in the newly added tree, so it can be dropped in the *putback*.

$$\boxed{\begin{array}{l} \texttt{add}\ n\ t \;=\; \texttt{xfork}\ \{\}\ \{n\}\ (\texttt{const}\ t\ \{\!\};\ \texttt{plunge}\ n)\ \texttt{id} \\ \hline \forall n \in \mathcal{N}.\ \forall C \subseteq \mathcal{T}\backslash_n.\ \forall t \in \mathcal{T}. \\ \quad \texttt{add}\ n\ t \in C \overset{\Omega}{\rightleftharpoons} \{\!|n \mapsto \{t\}|\!\} \cdot C \end{array}}$$

Another lens focuses attention on a single child n:

$$\boxed{\begin{array}{l} \texttt{focus}\ n\ d \;=\; (\texttt{filter}\ \{n\}\ d);\ (\texttt{hoist}\ n) \\ \hline \forall n \in \mathcal{N}.\ \forall C \subseteq \mathcal{T}\backslash_n. \forall d \in C.\ \forall D \subseteq \mathcal{T}. \\ \quad \texttt{focus}\ n\ d \in (C \cdot \{\!|n \mapsto D|\!\}) \overset{\Omega}{\rightleftharpoons} D \end{array}}$$

In the *get* direction, `focus` filters away all other children, then removes the edge n and yields n's subtree. As usual, the default tree is only used in the case of creation, where it is the default for children that have been filtered away. Again the type of `focus` follows from the types of the lenses from which it is defined, observing that $\texttt{filter}\ \{n\}\ d \in (C \cdot \{\!|n \mapsto D|\!\}) \overset{\Omega}{\Longleftrightarrow} \{\!|n \mapsto D|\!\}$ and that $\texttt{hoist n} \in \{\!|n \mapsto D|\!\} \overset{\Omega}{\Longleftrightarrow} D$.

A last derived lens renames a single child.

$$\boxed{\begin{array}{l} \texttt{rename}\ m\ n \;= \\ \qquad\qquad \texttt{xfork}\ \{m\}\ \{n\}\ (\texttt{hoist}\ m;\ \texttt{plunge}\ n)\ \texttt{id} \\ \hline \forall m, n \in \mathcal{N}.\ \forall C \subseteq \mathcal{T}.\ \forall D \subseteq \mathcal{T}\backslash_{\{m,n\}}. \\ \quad \texttt{rename}\ m\ n \in (\{\!|m \mapsto C|\!\} \cdot D) \overset{\Omega}{\rightleftharpoons} (\{\!|n \mapsto C|\!\} \cdot D) \end{array}}$$

In the *get* direction, `rename` splits the concrete tree in two. The first tree has a single child m (which is guaranteed to exist by the type annotation) and is hoisted up, removing the edge named m, and then plunged under n. The rest of the original tree is passed through the `id` lens. Similarly, the *putback* direction splits the abstract view into a tree with a single child `n`, and the rest of the tree. The tree under `n` is put back using the lens ($\texttt{hoist}\ m$; $\texttt{plunge}\ n$), which first removes the edge named n and then plunges the resulting tree under m. Note that the type annotation on `rename` demands that the concrete view have a child named m and that the abstract view have a child named n.

Mapping So far, all of our lens combinators do things near the root of the trees they are given. Of course, we also want to be able to perform transformations in the interior of trees. The `map` combinator is our fundamental means of doing this. When combined with recursion, it also allows us to iterate over structures of arbitrary depth.

The `map` combinator is parameterized on a single lens l. In the *get* direction, `map` applies $l\nearrow$ to each subtree of the root and combines the results together into a new tree. (Later, we will define a more general combinator, called `wmap`, that can apply a different lens to each subtree.) The *putback* direction is more interesting. In the simple case where a and c have equal domains, its behavior is straightforward: it uses $l\searrow$ to combine concrete and abstract subtrees with identical names and assembles the results into a new concrete tree, c'. In general, however, the abstract tree a in the *putback* direction need not have the same domain as c (i.e., the edits that produced the new abstract view may have involved adding and deleting children); the behavior of `map` in this case is a little more involved. Observe, first, that the domain of c' is determined by the domain of the abstract argument. Since we aim at building total lenses, we may suppose that $(\texttt{map}\ l)\nearrow((\texttt{map}\ l)\searrow(a, c))$ is defined, in which case it must be equal to a by rule PutGet. Thus $\mathsf{dom}((\texttt{map}\ l)\nearrow((\texttt{map}\ l)\searrow(a, c))) = \mathsf{dom}(a)$, hence $\mathsf{dom}((\texttt{map}\ l)\searrow(a, c)) = \mathsf{dom}(a)$ as the *get* of `map` does not change the domain of the tree. This means we can simply drop children that occur in $\mathsf{dom}(c)$ but not in $\mathsf{dom}(a)$. Children bearing names that occur both in $\mathsf{dom}(a)$ and $\mathsf{dom}(c)$ are dealt with as described above. This leaves the children that only appear in $\mathsf{dom}(a)$, which need to be passed through l so that they can be included in c'; to do this, we need some concrete argument to pass to $l\searrow$. There is no corresponding child in c, so instead these abstract trees are put into the missing tree Ω—indeed, this case is precisely why we introduced Ω. Formally, the behavior of `map` is defined as follows. (It relies on the convention that $c(n) = \Omega$ if $n \notin \mathsf{dom}(c)$; the type declaration also involves some new notation, explained below.)

$$\boxed{\begin{array}{l} \quad (\texttt{map}\ \mathrm{l})\nearrow c \;=\; \{\!|n \mapsto l\nearrow c(n) \mid n \in \mathsf{dom}(c)|\!\} \\ (\texttt{map}\ \mathrm{l})\searrow(a, c) \;= \\ \qquad\qquad \{\!|n \mapsto l\searrow(a(n), c(n)) \mid n \in \mathsf{dom}(a)|\!\} \\ \hline \forall C, A \subseteq \mathcal{T}\ \textit{with}\ C = C^{\circlearrowleft},\ A = A^{\circlearrowleft},\ \mathsf{dom}(C) = \mathsf{dom}(A). \\ \forall l \in (\bigcap_{n \in \mathcal{N}} . \ C(n) \overset{\Omega}{\rightleftharpoons} A(n)). \quad\ \ \texttt{map}\ l \in C \overset{\Omega}{\rightleftharpoons} A \end{array}}$$

Because of the way that it takes tree apart, transforms the pieces, and reassembles them, the typing of `map` is a little subtle. For example, in the *get* direction, `map` does not modify the names of the immediate children of the concrete tree and in the *putback* direction, the names of the abstract tree are left unchanged; we might therefore expect a simple typing rule stating that, if $l \in (\bigcap_{n \in \mathcal{N}} C(n) \overset{\Omega}{\rightleftharpoons} A(n))$—i.e., if l is a well-behaved lens from the concrete subtree type $C(n)$ to the abstract subtree type $A(n)$ for each child n—then $\texttt{map}\ l \in C \overset{\Omega}{\rightleftharpoons} A$. Unfortunately, for arbitrary C and A, the `map` lens is not guaranteed to be well-behaved at this type. In particular, if $\mathsf{dom}(C)$, the set of domains of trees in C, is not equal to $\mathsf{dom}(A)$, then the *putback* function can produce a tree that is not in C, as the following example shows. Consider the sets of trees $C = \{\{\!|\texttt{x} \mapsto \texttt{m}|\!\},\ \{\!|\texttt{y} \mapsto \texttt{n}|\!\}\}$ and $A = C \cup \{\{\!|\texttt{x} \mapsto \texttt{m},\ \texttt{y} \mapsto \texttt{n}|\!\}\}$, and observe that with trees $a = \{\!|\texttt{x} \mapsto \texttt{m},\ \texttt{y} \mapsto \texttt{n}|\!\}$ and $c = \{\!|\texttt{x} \mapsto \texttt{m}|\!\}$, we have

$\texttt{map id} \searrow (a,\, c) = a$, a tree that is not in C. This shows that the type of `map` must include the requirement that $\mathsf{dom}(C) = \mathsf{dom}(A)$. (Recall that for any type T the set $\mathsf{dom}(T)$ is a set of sets of names.)

A related problem arises when the sets of trees A and C have dependencies between the names of children and the trees that may appear under those names. Again, one might naively expect that, if l has type $C(n) \overset{\Omega}{\rightleftharpoons} A(m)$ for each name m, then $\texttt{map}\ l$ would have type $C \overset{\Omega}{\rightleftharpoons} A$. Consider, however, the set $A = \{\{\!|\texttt{x} \mapsto \texttt{m},\ \texttt{y} \mapsto \texttt{p}|\!\},\ \{\!|\texttt{x} \mapsto \texttt{n},\ \texttt{y} \mapsto \texttt{q}|\!\}\}$, in which the value `m` only appears under `x` when `p` appears under `y`, and the set $C = \{\{\!|\texttt{x} \mapsto \texttt{m},\ \texttt{y} \mapsto \texttt{p}|\!\},\ \{\!|\texttt{x} \mapsto \texttt{m},\ \texttt{y} \mapsto \texttt{q}|\!\},\ \{\!|\texttt{x} \mapsto \texttt{n},\ \texttt{y} \mapsto \texttt{p}|\!\},\ \{\!|\texttt{x} \mapsto \texttt{n},\ \texttt{y} \mapsto \texttt{q}|\!\}\}$, where both `m` and `n` appear with both `p` and `q`. When we consider just the projections of C and A at specific names, we obtain the same sets of subtrees: $C(\texttt{x}) = A(\texttt{x}) = \{\{\!|\texttt{m}|\!\}, \{\!|\texttt{n}|\!\}\}$ and $C(\texttt{y}) = A(\texttt{y}) = \{\{\!|\texttt{p}|\!\}, \{\!|\texttt{q}|\!\}\}$. The lens `id` has type $C(\texttt{x}) \overset{\Omega}{\rightleftharpoons} A(\texttt{x})$ and $C(\texttt{y}) \overset{\Omega}{\rightleftharpoons} A(\texttt{y})$ (and $C(z) = \emptyset \overset{\Omega}{\rightleftharpoons} \emptyset = A(z)$ for all other names z). But it is clearly not the case that $\texttt{map id} \in C \overset{\Omega}{\rightleftharpoons} A$. To avoid this error (but still give a type for `map` that is precise enough to derive interesting types for lenses defined in terms of `map`), we require that the source and target sets in the type of `map` be closed under the "shuffling" of their children. Formally, if T is a set of trees, then the set of *shufflings* of T, denoted $T^{\circlearrowleft}$, is $T^{\circlearrowleft} = \bigcup_{D \in \mathsf{dom}(T)} \{\!|n \mapsto T(n) \mid n \in D|\!\}$ where $\{\!|n \mapsto T(n) \mid n \in D|\!\}$ is the set of trees with domain D whose children under n are taken from the set $T(n)$. We say that T is *shuffle closed* iff $T = T^{\circlearrowleft}$. For instance, in the example above, $A^{\circlearrowleft} = C^{\circlearrowleft} = C$—i.e., C is shuffle closed, but A is not.

In the situations where `map` is used, shuffle closure is typically easy to check. For example, any set of trees whose elements each have singleton domains is shuffle closed. Also, for every set of trees T, the encoding introduced in §7 of lists with elements in T is shuffle closed, which justifies using `map` (with recursion) to implement operations on lists. Furthermore, types of the form $\{\!|n \mapsto T \mid n \in \mathcal{N}|\!\}$ with infinite domain but with the same structure under each edge, which are heavily used in database examples (where the top-level names are keys and the structures under them are records) are shuffle closed.

Another point to note about `map` is that it does not obey the PutPut law. Consider a lens l and $(a, c) \in \mathsf{dom}(l\searrow)$ such that $l \searrow (a,\, c) \neq l \searrow (a,\, \Omega)$. We have

$$\begin{array}{rl} & (\texttt{map}\ l) \searrow (\{\!|\texttt{n} \mapsto a|\!\},\ ((\texttt{map}\ l) \searrow (\{\!||\!\},\ \{\!|\texttt{n} \mapsto c|\!\}))) \\ = & (\texttt{map}\ l) \searrow (\{\!|\texttt{n} \mapsto a|\!\},\ \{\!||\!\}) \\ = & \{\!|\texttt{n} \mapsto l \searrow (a,\, \Omega)|\!\} \end{array}$$

whereas

$$\{\!|\texttt{n} \mapsto l \searrow (a,\, c)|\!\} \quad = \quad (\texttt{map}\ l) \searrow (\{\!|\texttt{n} \mapsto a|\!\},\ \{\!|\texttt{n} \mapsto c|\!\}).$$

Intuitively, there is a difference between, on the one hand, modifying a child n and, on the other, removing it and then adding it back: in the first case, any information in the concrete view that is "projected away" in the abstract view will be carried along to the new concrete view; in the second, such information will be replaced with default values. This difference seems pragmatically reasonable, so we prefer to keep `map` and lose PutPut.[3]

A final point of interest is the relation between `map` and the missing tree Ω. The *putback* function of every other lens combinator only results in a *putback* into the missing tree if the combinator itself is called on Ω. In the case of `map` l, calling its *putback* function on some a and c where c is not the missing tree may result in the application of the *putback* of l to Ω if a has some children that are not in c. In an earlier variant of `map`, we dealt with missing children by providing a default concrete child tree, which would be used when no actual concrete tree was available. However, we discovered that, in practice, it is often difficult to find a single default concrete tree that fits all possible abstract trees, particularly because of `xfork` (where different lenses are applied to different parts of the tree) and recursion (where the depth of a tree is unknown). We tried parameterizing this default concrete tree by the abstract tree and the lens, but noticed that most primitive lenses ignore the concrete tree when defining the *putback* function, as enough information is available in the abstract tree. The natural choice for a concrete tree parameterized by a and l was thus $l \searrow (a,\, \Omega)$, for some special tree Ω. The only lens for which the *putback* function needs to be defined on Ω is `const`, as it is the only lens that discards information. This led us to the present design, where only the `const` lens (and other lenses defined from it, such as `focus`) expects a default tree d. This approach is much more local than the others we tried, since one only needs to provide a default tree at the exact point where information is discarded.

We now define a more general form of `map` that is parameterized on a total function from names to lenses.

$$\boxed{\begin{array}{l} (\texttt{wmap}\ m) \nearrow c \quad = \\ \qquad\qquad \{\!|n \mapsto m(n) \nearrow c(n) \mid n \in \mathsf{dom}(c)|\!\} \\ (\texttt{wmap}\ m) \searrow (a,\, c) \quad = \\ \qquad\qquad \{\!|n \mapsto m(n) \searrow (a(n),\, c(n)) \mid n \in \mathsf{dom}(a)|\!\} \\ \hline \forall C, A \subseteq \mathcal{T}\ \textit{with}\ C = C^{\circlearrowleft},\ A = A^{\circlearrowleft},\ \mathsf{dom}(C) = \mathsf{dom}(A). \\ \forall m \in (\Pi n {\in} \mathcal{N}.\ C(n) \overset{\Omega}{\rightleftharpoons} A(n)). \qquad \texttt{wmap}\ m \in C \overset{\Omega}{\rightleftharpoons} A \end{array}}$$

In the type annotation, we use the dependent type notation $m \in \Pi n.\ C(n) \overset{\Omega}{\rightleftharpoons} A(n)$ to mean that m is a total function mapping each name n to a well-behaved lens from $C(n)$ to $A(n)$. Although m is a total function, we will often describe it by giving its behavior on a finite set of names and adopting the convention that it maps every other name to `id`. For example, the lens $\texttt{wmap}\ \{x \mapsto \texttt{plunge a}\}$ maps `plunge a` over trees under x and `id` over the subtrees of every other child.

Merging It sometimes happens that a concrete representation requires equality between two distinct subtrees within a view. A `merge` lens is one way to preserve this invariant when the abstract view is updated. In the *get* direction, `merge` takes a tree with two (equal) branches and deletes one of them. In the *putback* direction, `merge` copies the updated value of the remaining branch to *both* branches in the concrete view.

[3] Alternatively, we could use a refinement of the type system to track when PutPut does hold, annotating some of the lens combinators with the fact that they are *oblivious* (i.e., ignore their concrete argument), and then give `map` two types: the one we gave here plus another saying "when `map` is applied to an oblivious lens, the result is very well behaved."

$$(\texttt{merge}\ m\ n)\nearrow c \;=\; c\backslash_n$$
$$(\texttt{merge}\ m\ n)\searrow(a,\,c) \;=\; \begin{cases} a\cdot\{\!|n\mapsto a(m)|\!\} & \text{if } c(m)=c(n)\\ a\cdot\{\!|n\mapsto c(n)|\!\} & \text{if } c(m)\neq c(n)\end{cases}$$
$$\forall m,n{\in}\mathcal{N}.\ \forall C{\subseteq}\mathcal{T}\backslash_{\{m,n\}}.\ \forall D{\subseteq}\mathcal{T}.\quad \texttt{merge}\ m\ n \in (C\cdot\{\!|m\mapsto D_\Omega,\ n\mapsto D_\Omega|\!\}) \overset{\Omega}{\rightleftharpoons} (C\cdot\{\!|m\mapsto D_\Omega|\!\})$$

There is some freedom in the type of `merge`. On one hand, we can give it a precise type that expresses the intended equality constraint in the concrete view; the lens is well-behaved and total at that type. Alternatively, we can give it a more permissive type (as we do) by ignoring the equality constraint—even if the two original branches are unequal, `merge` is still defined and well-behavedness is preserved.

Other Tree Lenses The long version of the paper defines several additional combinators for manipulating tree structures: a `copy` lens, which duplicates information in the *get* direction and re-integrates the copied children in the *putback* direction after performing an equality check (we also discuss problems with the typing of `copy` and compare it with some variants that have been proposed by others), and three lenses for dealing with relational data encoded as trees—a "flattening" combinator that transforms a list of (keyed) records into a bush, a "pivoting" combinator that can be used to promote a key field to a higher position in the tree, and a "transposing" combinator related to the outer join operation on databases.

6. CONDITIONALS

Conditional lens combinators, which can be used to selectively apply one lens or another to a view, are necessary for writing many interesting derived lenses. Whereas `xfork` and its variants split their input trees into two parts, send each part through a separate lens, and recombine the results, a conditional lens performs some test and sends the *whole* trees through one or the other of its sub-lenses.

The requirement that makes conditionals tricky is totality: we want to be able to take a concrete view, put it through a conditional lens to obtain some abstract view, and then take *any* other abstract view of suitable type and push it back down. But this will only work if either (1) we somehow ensure that the abstract view is guaranteed to be sent to the same sub-lens on the way down as we took on the way up, or else (2) the two sub-lenses are constrained to behave coherently. Since we want reasoning about well-behavedness and totality to be compositional in the absence of recursion, the second is unacceptable.

Interestingly, once we adopt the first approach, we can give a *complete* characterization of all possible conditional lenses: we argue (in the long version of the paper) that every binary conditional operator that yields well-behaved and total lenses is an instance of the general `cond` combinator presented below. Since this general `cond` is a little complex, however, we start by discussing two particularly useful special cases.

Our first conditional, `ccond`, is parameterized on a predicate C_1 on views and two lenses, l_1 and l_2. In the *get* direction, it tests the concrete view c and applies the *get* of l_1 if c satisfies the predicate and l_2 otherwise. In the *putback* direction, `ccond` again examines the concrete view, and applies the *putback* of l_1 if it satisfies the predicate and l_2 otherwise.

$$(\texttt{ccond}\ C_1\ l_1\ l_2)\nearrow c \;=\; \begin{cases} l_1\nearrow c & \text{if } c\in C_1\\ l_2\nearrow c & \text{if } c\notin C_1\end{cases}$$
$$(\texttt{ccond}\ C_1\ l_1\ l_2)\searrow(a,\,c) \;=\; \begin{cases} l_1\searrow(a,\,c) & \text{if } c\in C_1\\ l_2\searrow(a,\,c) & \text{if } c\notin C_1\end{cases}$$
$$\forall C,C_1,A{\subseteq}\mathcal{U}.\ \forall l_1\in C{\cap}C_1 \overset{\Omega}{\rightleftharpoons} A.\ \forall l_2\in C\backslash C_1 \overset{\Omega}{\rightleftharpoons} A.\quad \texttt{ccond}\ C_1\ l_1\ l_2 \in C \overset{\Omega}{\rightleftharpoons} A$$

A quite different way of defining a conditional lens is to make it ignore its *concrete* argument in the *putback* direction, basing its decision whether to use $l_1\searrow$ or $l_2\searrow$ entirely on its abstract argument.

$$(\texttt{acond}\ C_1\ A_1\ l_1\ l_2)\nearrow c \;=\; \begin{cases} l_1\nearrow c & \text{if } c\in C_1\\ l_2\nearrow c & \text{if } c\notin C_1\end{cases}$$
$$(\texttt{acond}\ C_1\ A_1\ l_1\ l_2)\searrow(a,\,c) \;=\; \begin{cases} l_1\searrow(a,\,c) & \text{if } a\in A_1\wedge c\in C_1\\ l_1\searrow(a,\,\Omega) & \text{if } a\in A_1\wedge c\notin C_1\\ l_2\searrow(a,\,c) & \text{if } a\notin A_1\wedge c\notin C_1\\ l_2\searrow(a,\,\Omega) & \text{if } a\notin A_1\wedge c\in C_1\end{cases}$$
$$\forall C,A,C_1,A_1{\subseteq}\mathcal{U}.\ \forall l_1\in C{\cap}C_1 \overset{\Omega}{\rightleftharpoons} A{\cap}A_1.\ \forall l_2\in (C\backslash C_1) \overset{\Omega}{\rightleftharpoons} (A\backslash A_1).\quad \texttt{acond}\ C_1\ A_1\ l_1\ l_2 \in C \overset{\Omega}{\rightleftharpoons} A$$

The general conditional, `cond`, is essentially obtained by combining the behaviors of `ccond` and `acond`. The concrete conditional requires that the targets of the two lenses be identical, while the abstract conditional requires that they be disjoint. More generally, we can let them overlap arbitrarily, behaving like `ccond` in the region where they do overlap (i.e., for arguments (a,c) to *putback* where a is in the intersection of the targets) and like `acond` in the regions where the abstract argument to *putback* belongs to just one of the targets. To this we can add one additional observation: that the use of Ω in the definition of `acond` is actually arbitrary. All that is required is that, when we use the *putback* of l_1, the concrete argument should come from $(C_1)_\Omega$, so that l_1 is guaranteed to do something good with it. These considerations lead us to the following definition.

$$(\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2)\nearrow c \;=\; \begin{cases} l_1\nearrow c & \text{if } c\in C_1\\ l_2\nearrow c & \text{if } c\notin C_1\end{cases}$$
$$(\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2)\searrow(a,\,c) \;=\; \begin{cases} l_1\searrow(a,\,c) & \text{if } a\in A_1{\cap}A_2\wedge c\in C_1\\ l_2\searrow(a,\,c) & \text{if } a\in A_1{\cap}A_2\wedge c\notin C_1\\ l_1\searrow(a,\,c) & \text{if } a\in A_1\backslash A_2\wedge c\in (C_1)_\Omega\\ l_1\searrow(a,f_{21}(c)) & \text{if } a\in A_1\backslash A_2\wedge c\notin (C_1)_\Omega\\ l_2\searrow(a,\,c) & \text{if } a\in A_2\backslash A_1\wedge c\notin C_1\\ l_2\searrow(a,f_{12}(c)) & \text{if } a\in A_2\backslash A_1\wedge c\in C_1\end{cases}$$
$$\forall C,C_1,A_1,A_2\subseteq\mathcal{U}.\ \forall l_1\in (C{\cap}C_1) \overset{\Omega}{\rightleftharpoons} A_1.\ \forall l_2\in (C\backslash C_1) \overset{\Omega}{\rightleftharpoons} A_2.\ \forall f_{21}\in (C\backslash C_1)\to(C{\cap}C_1)_\Omega.\ \forall f_{12}\in (C{\cap}C_1)\to(C\backslash C_1)_\Omega.\quad \texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2 \in C \overset{\Omega}{\rightleftharpoons} (A_1{\cup}A_2)$$

When a is in the targets of both l_1 and l_2, $\texttt{cond}\searrow$ chooses between them based solely on c (as does `ccond`, whose targets always overlap). If a lies in the range of only l_1 or l_2, then `cond`'s choice of lens for *putback* is predetermined (as with `acond`, whose targets are disjoint). Once $l\searrow$ is chosen to be either $l_1\searrow$ or $l_2\searrow$, if the old value of c is not in $\texttt{ran}(l\searrow)_\Omega$, then we apply a "fixup function," f_{21} or f_{12}, to c to choose a new value from $\texttt{ran}(l\searrow)_\Omega$. Ω is one possible result of the fixup functions, but it is sometimes useful to compute a more interesting one, as we will see in §7.

7. DERIVED LENSES FOR LISTS

XML and many other concrete data formats make heavy use of ordered lists. We describe in this section how we can represent lists as trees, using a standard cons cell encoding, and introduce some derived lenses to manipulate them. We begin with very simple lenses for projecting the head and tail of a list encoded as a cons cell. We then define recursive lenses implementing some more complex operations on lists: mapping and filtering. In the long version of the paper, we also show how to derive a list-reversing lens that takes a list encoded as a tree and yields the same list in reverse order (in both directions, ignoring its concrete argument in the *putback* direction). Other list-processing derived forms that we have implemented include a "grouping" lens that, in the *get* direction, takes a list whose elements alternate between elements of D and elements of E and returns a list of pairs of Ds and Es—e.g., it maps `[d1 e1 d2 e2 d3 e3]` to `[[d1 e1] [d2 e2] [d3 e3]]`.

A tree t is said to be a *list* iff either it is empty or it has exactly two children, one named `*h` and another named `*t`, and $t(\texttt{*t})$ is also a list. In the following, we use the lighter notation $\texttt{[}t_1 \ldots t_n\texttt{]}$ for the tree $\{\!|\texttt{*h} \mapsto \texttt{t}_1\ \texttt{*t} \mapsto \{\!|\ldots \mapsto \{\!|\texttt{*h} \mapsto \texttt{t}_n\ \texttt{*t} \mapsto \{\!|\!\}|\!\}|\!\}|\!\}$. In types, we write `[]` for the set $\{\{\!|\!\}\}$ containing only the empty list, $C :: D$ for the set $\{\!|\texttt{*h} \mapsto C,\ \texttt{*t} \mapsto D|\!\}$ of "cons cell trees" whose head belongs to C and whose tail belongs to D, and $\texttt{[}C\texttt{]}$ for the set of lists with elements in C—i.e., the smallest set of trees satisfying $\texttt{[}C\texttt{]} = \texttt{[]} \cup (C :: \texttt{[}C\texttt{]})$. The interleaving of a list of type $\texttt{[}B\texttt{]}$ and a list of type $\texttt{[}C\texttt{]}$, taking elements from the first list and elements from the second in an arbitrary fashion but maintaining the relative order of each, is written $\texttt{[}B\texttt{]}\,\&\,\texttt{[}C\texttt{]}$.

Our first list lenses extract the head or tail of a cons cell.

$\texttt{hd}\ d \;=\; \texttt{focus *h}\ \{\!|\texttt{*t} \mapsto d|\!\}$
$\forall C, D \subseteq \mathcal{T}.\ \forall d \in D.\quad \texttt{hd}\ d \in (C :: D) \overset{\Omega}{\rightleftharpoons} C$

$\texttt{tl}\ d \;=\; \texttt{focus *t}\ \{\!|\texttt{*h} \mapsto d|\!\}$
$\forall C, D \subseteq \mathcal{T}.\ \forall d \in C.\quad \texttt{tl}\ d \in (C :: D) \overset{\Omega}{\rightleftharpoons} D$

The lens `hd` expects a default tree, which it uses in the *putback* direction as the tail of the created tree when the concrete tree is missing; in the *get* direction, it returns the tree under `*h`. The lens `tl` works analogously. Note that the types of these lenses apply to both homogeneous lists (the type of `hd` implies $\forall C \subseteq \mathcal{T}.\ \forall d \in \texttt{[}C\texttt{]}.\ \texttt{hd}\ d \in \texttt{[}C\texttt{]} \overset{\Omega}{\Longleftrightarrow} C$) as well as cons cells whose head and tail have unrelated types. The types of `hd` and `tl` follow from the type of `focus`.

The `list_map` lens applies l to each element of a list:

$\texttt{list\_map}\ l \;=\; \texttt{wmap}\ \{\texttt{*h} \mapsto l,\ \texttt{*t} \mapsto \texttt{list\_map}\ l\}$
$\forall C, A \subseteq \mathcal{T}.\ \forall l \in C \overset{\Omega}{\rightleftharpoons} A.\quad \texttt{list\_map}\ l \in \texttt{[}C\texttt{]} \overset{\Omega}{\rightleftharpoons} \texttt{[}A\texttt{]}$

The *get* direction applies l to the subtree under `*h` and recurses on the subtree under `*t`. The *putback* direction uses $l\searrow$ on corresponding pairs of elements from the abstract and concrete lists. The result has the same length as the abstract list; if the concrete list is longer, the extra tail is thrown away. If it is shorter, each extra element of the abstract list is *putback* into Ω.

Since `list_map` is our first recursive lens, it is worth making a few observations about the way recursive calls are made in each direction. The *get* function of the `wmap` lens simply applies l to the head and $\texttt{list\_map}\ l$ to the tail until it reaches a tree with no children. Similarly, in the *putback* direction, `wmap` applies l to the head of the abstract tree and either the head of the concrete tree (if it is present) or Ω, and it applies $\texttt{list\_map}\ l$ to the tail of the abstract tree and the tail of the concrete tree (if it is present) or Ω. In both directions, the recursive calls continue until the entire tree—concrete (for the *get*) or abstract (for the *putback*)—has been traversed.

Our most interesting derived lens, `list_filter`, is parameterized on two sets of views, D and E, which we assume to be disjoint and non-empty. In the *get* direction, it takes a list whose elements belong to either D or E and projects away those that belong to E, leaving an abstract list containing only Ds; in the *putback* direction, it restores the projected-away Es from the concrete list. Its definition utilizes our most complex lens combinators—`wmap` and two forms of conditionals—and mutual recursion, yielding a lens that is well-behaved and total on lists of arbitrary length.

In the *get* direction, the desired behavior of $\texttt{list\_filter}\ D\ E$ (for brevity, let us call it l) is clear. In the *putback* direction, things are more interesting because there are many ways that we could restore projected elements from the concrete list. The lens laws impose some constraints on the behavior of $l\searrow$. The GetPut law forces the *putback* function to restore each of the filtered elements when the abstract list is put into the original concrete list. For example (letting `d` and `e` be elements of D and E) we must have $l\searrow(\texttt{[d]},\ \texttt{[e d]}) = \texttt{[e d]}$. The PutGet law forces the *putback* function to include every element of the abstract list in the resulting concrete list, and these elements must be the only Ds in the result (there is however no restriction on the Es). The behavior that seems most natural to us is to overwrite elements of D in c with elements of D from a, element-wise, until either c or a runs out of elements of D. If c runs out first, then $l\searrow$ appends the rest of the elements of a at the end of c. If a runs out first, then $l\searrow$ restores the remaining Es from the end of c and discards any remaining Ds in c (as it must to satisfy PutGet).

These choices lead us to the following specification for a single step of the *putback* part of a recursively defined lens implementing l. If the abstract list a is empty, then we restore all the Es from c. If c is empty and a is not empty, then we return a. If a and c are both cons cells whose heads are in D, then we return a cons cell whose head is the head of a and whose tail is the result obtained by recursing on the tails of both a and c. Otherwise (i.e., c has type $E :: (\texttt{[}D\texttt{]}\,\&\,\texttt{[}E\texttt{]})$) we restore the head of c and recurse on a and the tail of c. Translating this into lens combinators leads to the definition below of `list_filter` and a helper lens, `inner_filter`, by mutual recursion.[4] The definitions involve a little new notation and a few additional technicalities, explained below.

$$\boxed{\begin{array}{l}
\texttt{list\_filter}\ D\ E = \\
\qquad \texttt{cond}\ \texttt{[}E\texttt{]}\ \texttt{[]}\ (D :: \texttt{[}D\texttt{]})\ \mathit{fltr}_E\ (\lambda c.\ c@\texttt{[}\mathit{any}_D\texttt{]}) \\
\qquad\quad (\texttt{const}\ \texttt{[]}\ \texttt{[]}) \\
\qquad\quad (\texttt{inner\_filter}\ D\ E) \\
\\
\texttt{inner\_filter}\ D\ E = \\
\qquad \texttt{ccond}\ (E :: ((D :: \texttt{[}D\texttt{]}) \& \texttt{[}E\texttt{]})) \\
\qquad\qquad (\texttt{tl}\ \mathit{any}_E;\ \texttt{inner\_filter}\ D\ E) \\
\qquad\qquad (\texttt{wmap}\ \{\texttt{*h} \mapsto \texttt{id}, \texttt{*t} \mapsto \texttt{list\_filter}\ D\ E\}) \\
\hline
\forall D, E \subseteq \mathcal{T}.\ \mathit{with}\ D \cap E = \emptyset\ \mathit{and}\ D \neq \emptyset\ \mathit{and}\ E \neq \emptyset. \\
\quad \texttt{list\_filter}\ D\ E \in \texttt{[}D\texttt{]} \& \texttt{[}E\texttt{]} \overset{\Omega}{\rightleftharpoons} \texttt{[}D\texttt{]}\ \mathit{and} \\
\quad \texttt{inner\_filter}\ D\ E \in (D :: \texttt{[}D\texttt{]}) \& \texttt{[}E\texttt{]} \overset{\Omega}{\rightleftharpoons} (D :: \texttt{[}D\texttt{]})
\end{array}}$$

The "choice operator" any_D denotes an arbitrary element of the (non-empty) set D.[5] The function fltr_E is the usual list-filtering *function*, which for present purposes we simply assume has been defined as a primitive. (In our implementation, we actually use $\texttt{list\_filter}\nearrow$.) Finally, the function $\lambda c.\ c@\texttt{[}\mathit{any}_D\texttt{]}$ appends some arbitrary element of D to the right-hand end of a list c. These "fixup functions" are applied in the *putback* direction by the `cond` lens.

The behavior of the *get* function of `list_filter` can be described as follows. If $c \in \texttt{[}E\texttt{]}$, then the outermost `cond` selects the `const [] []` lens, which produces `[]`. Otherwise the `cond` selects `inner_filter`, which uses a `ccond` instance to test if the head of the list is in E. If this test succeeds, it strips away the head using `tl` and recurses; if not, it retains the head and filters the tail using `wmap`.

In the *putback* direction, if $a = \texttt{[]}$ then the outermost `cond` lens selects the `const[] []` lens, with c as the concrete argument if $c \in \texttt{[}E\texttt{]}$ and $(\mathit{fltr}_E\ c)$ otherwise. This has the effect of restoring all of the Es from c. Otherwise, if $a \neq \texttt{[]}$ then the `cond` instance selects the *putback* of the `inner_filter` lens, using c as the concrete argument if c contains at least one D, and $(\lambda c.\ c@\texttt{[}\mathit{any}_D\texttt{]})\ c$, which appends a dummy value of type D to the tail of c, if not. The dummy value, any_D, is required because `inner_filter` expects a concrete argument that contains at least one D. Intuitively, the dummy value marks the point where the head of a should be placed.

To illustrate how all this works, let us step through some examples in detail. In each example, the concrete type is $\texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]}$ and the abstract type is $\texttt{[}D\texttt{]}$. We will write $\texttt{d}_i$ and $\texttt{e}_j$ to stand for elements of D and E respectively. To shorten the presentation, we will write l for $\texttt{list\_filter}\ D\ E$ and i for $\texttt{inner\_filter}\ D\ E$. In the first example, the abstract tree a is $\texttt{[d}_1\texttt{]}$, and the concrete tree c is $\texttt{[e}_1\ \texttt{d}_2\ \texttt{e}_2\texttt{]}$. At each step, we underline the term that is simplified in the next step.

$$\begin{array}{ll}
\multicolumn{2}{l}{\underline{l \searrow (a,\ c)} = \underline{i \searrow (a,\ c)}} \\
& \text{by the definition of } \texttt{cond}, \\
& \text{as } a \in (D :: \texttt{[}D\texttt{]}) \text{ and } c \in (\texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]}) \setminus \texttt{[}E\texttt{]} \\
= & \underline{(\texttt{tl}\ \mathit{any}_E;\ i) \searrow (a,\ c)} \\
& \text{by the definition of } \texttt{ccond}, \\
& \text{as } c \in E :: ((D :: \texttt{[}D\texttt{]})\&\texttt{[}E\texttt{]}) \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(i \searrow \Big(a,\ \underline{(\texttt{tl}\ \mathit{any}_E) \nearrow c}\Big),\ c\Big) \\
& \text{by the definition of composition} \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\underline{i \searrow (a,\ \texttt{[d}_2\ \texttt{e}_2\texttt{]})},\ c\Big) \\
& \text{reducing } (\texttt{tl}\ \mathit{any}_E) \nearrow c \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\underline{\texttt{wmap}\ \{\texttt{*h} \mapsto \texttt{id}, \texttt{*t} \mapsto l\} \searrow (a,\ \texttt{[d}_2\ \texttt{e}_2\texttt{]})},\ c\Big) \\
& \text{by the definition of } \texttt{ccond}, \\
& \text{as } \texttt{[d}_2\ \texttt{e}_2\texttt{]} \notin E :: ((D :: \texttt{[}D\texttt{]})\&\texttt{[}E\texttt{]}) \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\texttt{d}_1 :: (\underline{l \searrow (\texttt{[]},\ \texttt{[e}_2\texttt{]})}),\ c\Big) \\
& \text{by the definition of } \texttt{wmap} \text{ with } \texttt{id} \searrow (\texttt{d}_1,\ \texttt{d}_2) = \texttt{d}_1 \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\texttt{d}_1 :: (\underline{(\texttt{const [] []}) \searrow (\texttt{[]},\ \texttt{[e}_2\texttt{]})}),\ c\Big) \\
& \text{by the definition of } \texttt{cond}, \text{ as } \texttt{[]} \in \texttt{[]} \text{ and } \texttt{[e}_2\texttt{]} \in \texttt{[}E\texttt{]} \\
= & \underline{(\texttt{tl}\ \mathit{any}_E) \searrow (\texttt{d}_1 :: \texttt{[e}_2\texttt{]},\ c)} \\
& \text{by the definition of } \texttt{const} \\
= & \texttt{[e}_1\ \texttt{d}_1\ \texttt{e}_2\texttt{]} \quad \text{by the definition of } \texttt{tl}.
\end{array}$$

The final two examples illustrate how the "fixup functions" supplied to the `cond` lens are used. The first function, fltr_E, is used when the abstract list is empty and the concrete list is not in $\texttt{[}E\texttt{]}$. Let $a = \texttt{[]}$ and $c = \texttt{[d}_1\ \texttt{e}_1\texttt{]}$.

$$\begin{array}{ll}
\multicolumn{2}{l}{\underline{l \searrow (a,\ c)} = (\texttt{const [] []}) \searrow \Big(\texttt{[]},\ \underline{\mathit{fltr}_E \texttt{[d}_1\ \texttt{e}_1\texttt{]}}\Big)} \\
& \text{by the definition of } \texttt{cond}, \text{ as } a = \texttt{[]} \text{ but } c \notin \texttt{[}E\texttt{]} \\
= & \underline{(\texttt{const [] []}) \searrow (\texttt{[]},\ \texttt{[e}_1\texttt{]})} \\
& \text{by the definition of } \mathit{fltr}_E \\
= & \texttt{[e}_1\texttt{]} \quad \text{by definition of } \texttt{const}.
\end{array}$$

The other fixup function, $(\lambda c.\ c@\texttt{[}\mathit{any}_D\texttt{]})$, inserts a dummy D element when `list_filter` is called with a non-empty abstract list and a concrete list whose elements are all in E. Let $a = \texttt{[d}_1\texttt{]}$ and $c = \texttt{[e}_1\texttt{]}$ and assume that $\mathit{any}_D = \texttt{d}_0$.

$$\begin{array}{ll}
\multicolumn{2}{l}{\underline{l \searrow (a,\ c)} = i \searrow \Big(a,\ \underline{(\lambda c.\ c@\texttt{[}\mathit{any}_D\texttt{]})\ c}\Big)} \\
& \text{by the definition of } \texttt{cond}, \text{ as } a \in (D :: \texttt{[}D\texttt{]}) \text{ and } c \in \texttt{[}E\texttt{]} \\
= & \underline{i \searrow (a,\ \texttt{[e}_1\ \texttt{d}_0\texttt{]})} \\
& \text{by the definition of } (\lambda c.\ c@\texttt{[}\mathit{any}_D\texttt{]}) \\
= & \underline{(\texttt{tl}\ \mathit{any}_E;\ i) \searrow (a,\ \texttt{[e}_1\ \texttt{d}_0\texttt{]})} \\
& \text{by the definition of } \texttt{ccond}, \\
& \text{as } \texttt{[e}_1\ \texttt{d}_0\texttt{]} \in E :: ((D :: \texttt{[}D\texttt{]})D\&\texttt{[}E\texttt{]}) \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(i \searrow \Big(a,\ \underline{(\texttt{tl}\ \mathit{any}_E) \nearrow \texttt{[e}_1\ \texttt{d}_0\texttt{]}}\Big),\ \texttt{[e}_1\ \texttt{d}_0\texttt{]}\Big) \\
& \text{by the definition of composition} \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\underline{i \searrow (a,\ \texttt{[d}_0\texttt{]})},\ \texttt{[d}_0\ \texttt{e}_1\texttt{]}\Big) \\
& \text{reducing } (\texttt{tl}\ \mathit{any}_E) \nearrow \texttt{[d}_0\ \texttt{e}_1\texttt{]} \\
= & (\texttt{tl}\ \mathit{any}_E) \\
& \searrow \Big(\underline{\texttt{wmap}\ \{\texttt{*h} \mapsto \texttt{id}, \texttt{*t} \mapsto l\} \searrow (a,\ \texttt{[d}_0\texttt{]})},\ \texttt{[e}_1\ \texttt{d}_0\texttt{]}\Big) \\
& \text{by the definition of } \texttt{ccond}, \\
& \text{as } \texttt{[d}_0\texttt{]} \notin E :: ((D :: \texttt{[}D\texttt{]})\&\texttt{[}E\texttt{]})
\end{array}$$

[4]The singly recursive variant where `inner_filter` is inlined has the same dynamic behavior as the version presented here. We split out `inner_filter` so that we can give it a more precise type, facilitating reasoning about well-behavedness and totality: in the *get* direction it maps lists containing at least one D to $(D :: \texttt{[}D\texttt{]})$; the corresponding types for `list_filter` include empty lists.

[5]We are dealing with countable sets of finite trees here, so this construct poses no metaphysical conundrums; alternatively, but less readably, we can pass `list_filter` an extra argument $d \in D$.

$$= \quad (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\texttt{d}_1 :: (\underline{l \searrow (\texttt{[]},\ \texttt{[]})}),\ \texttt{[e}_1\ \texttt{d}_0\texttt{]}\Big)$$
by the definition of `wmap` with $\texttt{id} \searrow (\texttt{d}_1,\ \texttt{d}_0) = \texttt{d}_1$

$$= \quad (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\texttt{d}_1 :: (\underline{(\texttt{const [] []}) \searrow (\texttt{[]},\ \texttt{[]})}),\ \texttt{[e}_1\ \texttt{d}_0\texttt{]}\Big)$$
by the definition of `cond`, as $\texttt{[]} \in \texttt{[]}$ and $\texttt{[]} \in \texttt{[}E\texttt{]}$

$$= \quad \underline{(\texttt{tl}\ \mathit{any}_E) \searrow (\texttt{d}_1 :: \texttt{[]},\ \texttt{[e}_1\ \texttt{d}_0\texttt{]})}$$
by the definition of `const`

$$= \quad \texttt{[e}_1\ \texttt{d}_1\texttt{]} \quad \text{by the definition of } \texttt{tl}.$$

8. RELATED WORK

Our lens combinators evolved in the setting of the Harmony data synchronizer. The overall architecture of Harmony and the role of lenses in building synchronizers for various forms of data are described in [26], along with a detailed discussion of related work on synchronization.

Our foundational structures—lenses and their laws—are not new: closely related structures have been studied for decades in the database community. However, our treatment of these structures is arguably simpler (transforming states rather than "update functions") and somewhat more refined (treating well-behavedness as a form of type assertion). Our formulation is also novel in considering the issue of continuity, thus supporting a rich variety of surface language structures including definition by recursion. The idea of defining programming languages for constructing bi-directional transformations of various sorts has also been explored previously in diverse communities. We appear to be the first to take totality as a primary goal (while connecting the language with a formal semantic foundation, choosing primitives that can be combined into composite lenses whose totality is guaranteed by construction), and the first to emphasize types (i.e., compositional reasoning) as an organizing design principle.

Foundations of View Update The foundations of view update translation were studied intensively by database researchers in the late '70s and '80s. This thread of work is closely related to our semantics of lenses in §3.

Dayal and Bernstein [11] gave a seminal formal account of the theory of "correct update translation." Their notion of "exactly performing an update" corresponds to our PutGet law. Their "absence of side effects" corresponds to our GetPut and PutPut laws. Their requirement of preservation of semantic consistency corresponds to the partiality of our *putback* functions.

Bancilhon and Spyratos [6] developed an elegant semantic characterization of update translation, introducing the notion of *complement* of a view, which must include at least all information missing from the view. When a complement is fixed, there exists at most one update of the database that reflects a given update on the view while leaving the complement unmodified—i.e., that "translates updates under a constant complement." In general, a view may have many complements, each corresponding to a possible strategy for translating view updates to database updates. The problem of translating view updates then becomes a problem of finding, for a given view, a suitable complement.

Gottlob, Paolini, and Zicari [13] offered a more refined theory based on a syntactic translation of view updates. They identified a hierarchy of restricted cases of their framework, the most permissive form being their "dynamic views" and the most restrictive, called "cyclic views with constant complement," being formally equivalent to Bancilhon and Spyratos's update translators.

In a companion report [25], we state a precise correspondence between our lenses and the structures studied by Bancilhon and Spyratos and by Gottlob, Paolini, and Zicari. Briefly, our set of very well behaved lenses is isomorphic to the set of *translators under constant complement* in the sense of Bacilhon and Spyratos, while our set of well-behaved lenses is isomorphic to the set of *dynamic views* in the sense of Gottlob, Paolini, and Zicari. To be precise, both of these results must be qualified by an additional condition regarding partiality. The frameworks of Bacilhon and Spyratos and of Gottlob, Paolini, and Zicari are both formulated in terms of translating *update functions* on A into update functions on C, i.e., their *putback* functions have type $(A \longrightarrow A) \longrightarrow (C \longrightarrow C)$, while our lenses translate abstract *states* into update functions on C, i.e., our *putback* functions have type (isomorphic to) $A \longrightarrow (C \longrightarrow C)$. Moreover, in both of these frameworks, "update translators" (the analog of our *putback* functions) are defined only over some particular chosen set U of abstract update functions, not over all functions from A to A. These update translators return *total* functions from C to C. Our *putback* functions, on the other hand, are defined over all abstract states and return *partial* functions from C to C. Finally, the *get* functions of lenses are allowed to be partial, whereas the corresponding functions (called *views*) in the other two frameworks are assumed to be total. In order to make the correspondences tight, our sets of well-behaved and very well behaved lenses need to be restricted to subsets that are also total in a suitable sense.

Recent work by Lechtenbörger [17] establishes that translations of view updates under constant complements are possible precisely if view update effects may be undone using further view updates.

In the literature on programming languages, laws similar to our lens laws (but somewhat simpler, dealing only with total *get* and *putback* functions) appear in Oles' category of "state shapes" [24] and in Hofmann and Pierce's work on "positive subtyping" [14].

Languages for Bi-Directional Transformations At the level of syntax, different forms of bi-directional programming have been explored across a surprisingly diverse range of communities, including programming languages, databases, program transformation, constraint-based user interfaces, and quantum computing. One useful way of classifying these languages is by the "shape" of the semantic space in which their transformations live. We identify three major classes: *Bi-directional languages*, including ours, form lenses by pairing a *get* function of type $C \to A$ with a *putback* function of type $A \times C \to C$. In general, the *get* function can project away some information from the concrete view, which must then be restored by the *putback* function. In *bijective languages*, the *putback* function has the simpler type $A \to C$—it is given no concrete argument to refer to. To avoid loss of information, the *get* and *putback* functions must form a (perhaps partial) bijection between C and A. *Reversible languages* go a step further, demanding only that the work performed by any function to produce a given output can be undone by applying the function "in reverse" working backwards from this output to produce the original input. Here, there is no separate *putback* function at all: instead, the *get* function itself is constructed so that each step can be run in reverse. We survey relevant work in the first two classes in detail next; information on reversible languages (whose concerns are less closely related) can be found in the long version of the paper.

In the first class, the work that is fundamentally most similar to ours is Meertens's formal treatment of *constraint maintainers* for constraint-based user interfaces [21]. Meertens's semantic setting is actually even more general: he takes *get* and *putback* to be *relations*, not just functions, and his constraint maintainers are symmetric: *get* relates pairs from $C \times A$ to elements of A and *putback* relates pairs in $A \times C$ to elements of C. Taking the special case where the *get* relation is actually a function (which is important for Meertens because this is the case where composition [in the sense of our ; combinator] is guaranteed to preserve well-behavedness), yields essentially our very well behaved lenses. Meertens proposes a variety of combinators for building constraint maintainers, most of which have analogs among our lenses, but does not directly deal with definition by recursion; also, some of his combinators do not support compositional reasoning about well-behavedness. He considers constraint maintainers for structured data such as lists, as we do for trees, but here adopts a rather different point of view from ours, focusing on constraint maintainers that work with structures not directly but in terms of the "edit scripts" that might have produced them. In the terminology of synchronization, he switches from a state-based to an operation-based treatment at this point.

Recent work of Mu, Hu, and Takeichi on "injective languages" for view-update-based structure editors [22] adopts a similar perspective. Although their transformations obey our GetPut law, their notion of well-behaved transformations is informed by different goals than ours, leading to a weaker form of the PutGet law. A primary concern is using the view-to-view transformations to simultaneously restore invariants *within* the source view as well as update the concrete view. Another paper by Hu, Mu, and Takeichi [15] applies a bi-directional programming language quite closely related to ours to the design of "programmable editors" for structured documents. As in [22], they support preservation of local invariants in the *putback* direction. Here, instead of modifying the abstract view, they assume that a *putback* or a *get* occurs after *every* modification to either view. They use this "only one update" assumption to choose the correct inverse for the lens that copied data in the *get* direction — because only one branch can have been modified at any given time. Consequently, they can *putback* the data from the modified branch and overwrite the unmodified branch. Here, as in [22], the notion of well-behavedness must be weakened. We discuss these variants further in the full paper, in the section discussing our `copy` and `merge` lenses.

The TRIP2 system (e.g., [19]) uses bidirectional transformations specified as collections of Prolog rules as a means of implementing direct-manipulation interfaces for application data structures. The *get* and *putback* components of these mappings are written separately by the user.

Languages for Bijective Transformations An active thread of work in the program transformation community concerns *program inversion* and *inverse computation*—see, for example, [3, 4] and many other papers cited there. Program inversion derives the inverse program from the forward program. Inverse computation computes a possible input of a program from a particular output. One approach to inverse computation is to design languages that produce easily invertible expressions—for example, languages that can only express injective functions, where every program is trivially invertible. These languages bear some intriguing similarities to ours, but differ in a number of ways, primarily in their focus on the bijective case.

In the database community, Abiteboul, Cluet, and Milo [1] defined a declarative language of *correspondences* between parts of trees in a data forest. In turn, these correspondence rules can be used to translate one tree format into another through non-deterministic Prolog-like computation. This process assumes an isomorphism between the two data formats. The same authors [2] later defined a system for bi-directional transformations based around the concept of *structuring schemas* (parse grammars annotated with semantic information). Thus their *get* functions involved parsing, whereas their *putback*s consisted of unparsing. Again, to avoid ambiguous abstract updates, they restricted themselves to *lossless* grammars that define an isomorphism between concrete and abstract views.

Ohori and Tajima [23] developed a statically-typed polymorphic record calculus for defining views on object-oriented databases. They specifically restricted which fields of a view are updatable, allowing only those with a ground (simple) type to be updated, whereas our lenses can accommodate structural updates as well.

A related idea from the functional programming community, called *views* [29], extends algebraic pattern matching to abstract data types using programmer-supplied *in* and *out* operators.

Update Translation for Tree Views There have been many proposals for query languages for trees (e.g., XQuery and its forerunners, UnQL, StruQL, and Lorel), but these either do not consider the view update problem at all or else handle update only in situations where the abstract and concrete views are isomorphic.

For example, Braganholo, Davidson, and Heuser [8] and others studied the problem of updating relational databases "presented as XML." Their solution requires a 1:1 mapping between XML view elements and objects in the database, to make updates unambiguous. Tatarinov, Ives, Halevy, and Weld [28] described a mechanism for translating updates on XML structures that are stored in an underlying relational database. In this setting there is again an isomorphism between the concrete relational database and the abstract XML view, so updates are unambiguous—rather, the problem is choosing the most efficient way of translating a given XML update into a sequence of relational operations.

The view update problem has also been studied in the context of object-oriented databases. School, Laasch, and Tresch [27] restrict the notion of views to queries that preserve object identity. The view update problem is greatly simplified in this setting, as the objects contained in the view are the objects of the database, and an update on the view is directly an update on objects of the database.

Update Translation for Relational Views Research on view update translation in the database literature has tended to focus on taking an existing language for defining *get* functions (e.g., relational algebra) and then considering how to infer corresponding *putback* functions, either automatically or with some user assistance. By contrast, we have designed a new language in which the definitions of *get* and *putback* go hand-in-hand. Our approach also goes beyond classical work in the relational setting by directly transforming and updating tree-structured data, rather than flat relations. (Of course, trees can be encoded as relations, but it is not clear how our tree-manipulation primitives could be expressed using the recursion-free relational languages considered in previous work in this area.) We briefly review the most relevant research from the relational setting.

Masunaga [18] described an automated algorithm for translating updates on views defined by relational algebra. The core idea was to annotate where the "semantic ambiguities" arise, indicating they must be resolved either with knowledge of underlying database semantic constraints or by interactions with the user.

Keller [16] catalogued all possible strategies for handling updates to a select-project-join view and showed that these are exactly the set of translations that satisfy a small set of intuitive criteria. Building on this foundation, Barsalou, Siambela, Keller, and Wiederhold [7] described a scheme for interactively constructing update translators for object-based views of relational databases.

Medeiros and Tompa [20] presented a design tool for exploring the effects of choosing a view update policy. This tool shows the update translation for update requests supplied by the user; by considering all possible valid concrete states, the tool predicts whether the desired update would in fact be reflected back into the view after applying the translated update to the concrete database.

Atzeni and Torlone [5] described a tool for translating views and observed that if one can translate any concrete view to and from a *meta-model* (shared abstract view), one then gets bi-directional transformations between any pair of concrete views. They limited themselves to mappings where the concrete and abstract views are isomorphic.

Complexity bounds have also been studied for various versions of the view update inference problem. In one of the earliest, Cosmadakis and Papadimitriou [10] considered the view update problem for a single relation, where the view is a projection of the underlying relation, and showed that there are polynomial time algorithms for determining whether insertions, deletions, and tuple replacements to a projection view are translatable into concrete updates. More recently, Buneman, Khanna, and Tan [9] established a variety of intractability results for the problem of inferring "minimal" view updates in the relational setting for query languages that include both join and either project or union.

9. CONCLUSIONS AND FUTURE WORK

We have worked to design a collection of combinators that fit together in a sensible way and that are easy to program with. Starting with lens laws that define "reasonable behavior," adding type annotations, and proving that each of our lenses is total, has imposed strong constraints on our design of new lenses—constraints that, paradoxically, make the design process easier. In the early stages of the Harmony project, working in an under-constrained design space, we found it extremely difficult to converge on a useful set of primitive lenses. Later, when we understood how to impose the framework of type declarations and the demand for compositional reasoning, we experienced a *huge* increase in manageability. The types helped not just in finding programming errors in derived lenses, but in exposing design mistakes in the primitives at an early stage.

Naturally, the progress we have made on lens combinators raises a host of further challenges. The most urgent of these is automated typechecking. At present, it is the lens programmers' responsibility to check the well-behavedness of the lenses that they write. But the types of the primitive combinators have been designed so that these checks are both local and essentially mechanical. The obvious next step is to reformulate the type declarations as a type *algebra* and find a mechanical procedure for checking (or, more ambitiously, inferring) types.

A number of other interesting questions are related to static analysis of lenses. For instance, can we characterize the complexity of programs built from these combinators? Is there an algebraic theory of lens combinators that would underpin optimization of lens expressions in the same way that the relational algebra and its algebraic theory are used to optimize relational database queries? (For example, the combinators we have described here have the property that `map` l_1; `map` l_2 = `map` $(l_1; l_2)$ for all l_1 and l_2, but the latter should run substantially faster.)

This algebraic theory will play a crucial role in a more serious implementation effort. Our current prototype performs a straightforward translation from a concrete syntax similar to the one used in this paper to a combinator library written in OCaml. This is fast enough for experimenting with lens programming (Malo Denielou has built an interactive programming environment that recompiles and reapplies lenses on every keystroke) and for small demos (our calendar lenses can process a few thousands of appointments in under a minute), but we would like to apply the Harmony system to applications such as synchronization of biological databases that will require much higher throughput.

Another area for further investigation is the design of additional combinators. While we have found the ones we have described here to be expressive enough to code a large number of examples—both intricate structural manipulations such as the list transformations in §7 and more prosaic application transformations such as the ones needed by our bookmark synchronizer—there are some areas where we would like more general forms of the lenses we have (e.g., a more flexible form of `xfork`, where the splitting and recombining of trees is not based on top-level names, but involves deeper structure), lenses expressing more global transformations on trees (including analogs of database operations such as `join`), or lenses addressing completely different sorts of transformations (e.g., none of our combinators do any significant processing on edge labels, which might include string processing, arithmetic, etc.). Higher-level combinators embodying more global transformations on trees—perhaps modeled on a familiar notation such as XSLT—are another interesting possibility.

More generally, what are the limits of bi-directional programming? How expressive are the combinators we have defined here? Do they cover any known or succinctly characterizable classes of computations (in the sense that the set of *get* parts of the total lenses built from these combinators coincide with this class)? We have put considerable energy into these questions, but at the moment we can only report that they are challenging! One reason for this is that questions about expressiveness tend to have trivial answers when phrased semantically. For example, it is not hard to show that *any* surjective *get* function can be equipped with a *putback* function—indeed, typically many—to form a total lens. Indeed, if the concrete domain C is recursively enumerable, then this *putback* function is even computable. The real problems are thus syntactic—how to conveniently pick out a *putback* function that does what is wanted for a given situation.

Finally, we intend to experiment with instantiating our semantic framework with other structures besides trees—in particular, with relations, to establish closer links with existing research on the view update problem in databases.

Acknowledgements

The Harmony project was begun in collaboration with Zhe Yang. Owen Gunden and, more recently, Malo Denielou have also collaborated with us on many aspects of the Harmony design and implementation. Trevor Jim provided the initial push to start the project. Conversations with Martin Hofmann, Zack Ives, Nitin Khandelwal, Sanjeev Khanna, William Lovas, Kate Moore, Cyrus Najmabadi, Penny Anderson, and Steve Zdancewic helped us sharpen our ideas. Serge Abiteboul, Zack Ives, Dan Suciu, and Phil Wadler pointed us to related work. We would also like to thank Karthik Bhargavan, Vanessa Braganholo, Peter Buneman, Michael Hicks, Norman Ramsey, Wang-Chiew Tan, Stephen Tse, and several anonymous referees for helpful comments. The Harmony project is supported by the National Science Foundation under grant ITR-0113226, *Principles and Practice of Synchronization*.

10. REFERENCES

[1] S. Abiteboul, S. Cluet, and T. Milo. Correspondence and translation for heterogeneous data. In *Proceedings of 6th Int. Conf. on Database Theory (ICDT)*, 1997.

[2] S. Abiteboul, S. Cluet, and T. Milo. A logical view of structure files. *VLDB Journal*, 7(2):96–114, 1998.

[3] S. M. Abramov and R. Glück. The universal resolving algorithm: Inverse computation in a functional language. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837, pages 187–212. Springer-Verlag, 2000.

[4] S. M. Abramov and R. Glück. Principles of inverse computation and the universal resolving algorithm. In T. Mogensen, D. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation*, volume 2566 of *Lecture Notes in Computer Science*, pages 269–295. Springer-Verlag, 2002.

[5] P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. In *Proceedings of EDBT'96, LNCS 1057*, 1996.

[6] F. Bancilhon and N. Spyratos. Update semantics of relational views. *TODS*, 6(4):557–575, 1981.

[7] T. Barsalou, N. Siambela, A. M. Keller, and G. Wiederhold. Updating relational databases through object-based views. In *PODS'91*, pages 248–257, 1991.

[8] V. Braganholo, S. Davidson, and C. Heuser. On the updatability of XML views over relational databases. In *WebDB 2003*, 2003.

[9] P. Buneman, S. Khanna, and W.-C. Tan. On propagation of deletions and annotations through views. In *PODS'02*, pages 150–158, 2002.

[10] S. S. Cosmadakis and C. H. Papadimitriou. Updates of relational views. *Journal of the ACM*, 31(4):742–760, 1984.

[11] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *TODS*, 7(3):381–416, September 1982.

[12] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bi-directional tree transformations: A linguistic approach to the view update problem. Technical Report MS-CIS-04-15, University of Pennsylvania, Aug. 2004. An earlier version appeared in the *Workshop on Programming Language Technologies for XML (PLAN-X)*, 2004, under the title "A Language for Bi-Directional Tree Transformations".

[13] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems (TODS)*, 13(4):486–524, 1988.

[14] M. Hofmann and B. Pierce. Positive subtyping. In *ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL), San Francisco, California*, pages 186–197, Jan. 1995. Full version in *Information and Computation*, volume 126, number 1, April 1996. Also available as University of Edinburgh technical report ECS-LFCS-94-303, September 1994.

[15] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bi-directional transformations. In *Partial Evaluation and Program Manipulation (PEPM)*, 2004.

[16] A. M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *PODS'85*, 1985.

[17] J. Lechtenbörger. The impact of the constant complement approach towards view updating. In *Proceedings of the 22nd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 49–55. ACM, June 9–12 2003. San Diego, CA.

[18] Y. Masunaga. A relational database view update translation mechanism. In *VLDB'84*, 1984.

[19] S. Matsuoka, S. Takahashi, T. Kamada, and A. Yonezawa. A general framework for bi-directional translation between abstract and pictorial data. *ACM Transactions on Information Systems*, 10(4):408–437, October 1992.

[20] C. M. B. Medeiros and F. W. Tompa. Understanding the implications of view update policies. In *VLDB'85*, 1985.

[21] L. Meertens. Designing constraint maintainers for user interaction, 1998. Manuscript.

[22] S.-C. Mu, Z. Hu, and M. Takeichi. An algebraic approach to bi-directional updating. In *ASIAN Symposium on Programming Languages and Systems (APLAS)*, Nov. 2004.

[23] A. Ohori and K. Tajima. A polymorphic calculus for views and object sharing. In *PODS'94*, 1994.

[24] F. J. Oles. Type algebras, functor categories, and block structure. In M. Nivat and J. C. Reynolds, editors, *Algebraic Methods in Semantics*. Cambrige University Press, 1985.

[25] B. C. Pierce and A. Schmitt. Lenses and view update translation. Manuscript, 2003.

[26] B. C. Pierce, A. Schmitt, and M. B. Greenwald. Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania, 2003.

[27] M. H. Scholl, C. Laasch, and M. Tresch. Updatable Views in Object-Oriented Databases. In C. Delobel, M. Kifer, and Y. Yasunga, editors, *Proc. 2nd Intl. Conf. on Deductive and Object-Oriented Databases (DOOD)*, number 566. Springer, 1991.

[28] I. Tatarinov, Z. G. Ives, A. Y. Halevy, and D. S. Weld. Updating XML. In *SIGMOD Conference*, 2001.

[29] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *ACM Symposium on Principles of Programming Languages (POPL), Munich, Germany.* 1987.