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

August 28, 2004

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.

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 [32]. 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 a single common abstract view and a collection of *lenses* that transform each 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 *put* 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 *put* 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 *put* direction in such a way that each lens is both *well behaved*, in the sense that its *get* and *put* behaviors fit together in a sensible way, and *total*, in the sense that its *get* and *put* functions are guaranteed to be 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 *put* function to form a well-behaved and total lens; we need some means of specifying which *put* 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 *put*. 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. (Proving totality of recursive lenses, like ordinary recursive programs, requires global reasoning that goes beyond types.)

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 abstract 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 is 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 *put* parts of a lens should behave in concert. (E.g., if we use the *get* part of a lens to extract a from c and then use the *put* part to push the very same a back into c, we should get c.) Third, we use standard tools to define monotonicity and continuity for lens combinators, establishing a foundation for defining lenses by recursion (which we need because the trees that our lenses manipulate may in general have arbitrarily deep nested structure). 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 focus attention on tree-structured data and present several more combinators that perform various manipulations on trees (hoisting, splitting, mapping, etc.) and show how to assemble these primitives, along with the generic combinators from before, 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 as derived forms; our main example is a bi-directional `list_filter` lens whose *put* direction must perform a rather intricate "weaving" operation to recombine a potentially 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 [15], 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 [31]) 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 [15].

2 A Small Example

Suppose our concrete tree c is a small address book:

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

(We draw trees sideways to save space. Each set of hollow braces corresponds to a tree node, and each "$\texttt{X} \mapsto$ ..." 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 "$\{\!\{\texttt{333-4444} \mapsto \{\!\{\}\!\}\}\!\}$." 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.

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

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`.

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

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.

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

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, but in general we might want to compute it from other information.

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` $\{\!\{\texttt{URL} \mapsto \texttt{http://google.com}\}\!\}$).)

[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 [32].

3 Semantic Foundations

Although many of our combinators are designed to perform various transformations 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.

3.1 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 $g(x)\downarrow$. We extend function application to sets of arguments in a pointwise fashion, writing $f(C)$ for $\{f(c) \mid c \in C \cap \mathsf{dom}(f)\}$.

3.1.1 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 *put function*.

The intuition behind the notations $l\nearrow$ and $l\searrow$ is that the *get* part of a lens "lifts" an abstract view out of a concrete one, while the *put* part "pushes down" a new abstract view into an existing concrete view.

3.1.2 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

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

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

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

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 *put* a (with no modifications) back into c, we must get back exactly c (if both operations are defined). PutGet, on the other hand, demands that the *put* 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 *put* 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. The long version of this paper describes a third law, PutPut. Well-behaved lenses that also obey PutPut are *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).

A final important property that lenses may satisfy (on a given domain) is *totality*.

3.1.3 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 $A \times C \subseteq \mathsf{dom}(l\searrow)$.

Note that well-behavedness is trivial in the absence of totality: for *any* function $l\nearrow$ from C to A, we can obtain a well-behaved lens by taking $l\searrow$ to be undefined on all inputs (or—very slightly less trivially—to be defined only on inputs of the form $(l\nearrow c, c)$).

This is consistent with the pragmatic intuition that we always want our lenses to be defined on the whole of the domains where we intend to use them. However, totality of lenses—like totality of ordinary recursive functions or termination of while loops—is more difficult to reason about than simple well-behavedness, requiring invention of global termination measures, in contrast to the purely local reasoning used to show well-behavedness. This is why we formulate it as a separate condition rather than building it into the definition of well-behavedness.

3.2 Recursion

Since our lens framework will be instantiated for the universe of 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 bookmark items), we will need to define lenses by recursion. Our next task in this foundational section is to set up the necessary structure for interpreting such definitions.

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 *put* 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 $\bot$ that is smaller than every other element. In our setting, $\bot$ is the lens whose *get* and *put* functions are everywhere undefined.

3.2.1 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

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

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

3.2.2 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$(\forall i \in \omega.\ l_i \in C_i \rightleftharpoons A_i) \implies (\bigsqcup_n l_n) \in (\bigcup_i C_i) \rightleftharpoons (\bigcup_i A_i)$.

3.2.3 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.

3.3 Dealing with Creation

In practice, there will be cases where we need to apply a *put* 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 *put* 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, and $C \overset{\Omega}{\Longleftrightarrow} A$ for the set of total lenses 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.)

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.

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

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

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

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

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 *put* direction, the two *put* functions are applied in turn: first, the *put* 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 of this *put* is then put into c using the *put* 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 $(l; k) \searrow (a, \Omega)$ will be to use k to put a into Ω and then l to put the result into Ω.) Note that we record two different type declarations for composition: one for the case where the parameter lenses l and k are only known to be well behaved, and another for the case where they are also known to be total.

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

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

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

We will define a few more generic lenses in §6; now, though, let us turn to 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* that we obtain in this way far outweighs the increase in complexity of lens *programs* due to manipulating ordered data in encoded form.

5.1 Notation

From this point forward, we will 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. The variables a, c, d, and t range over $\mathcal{T}$; by convention, we use a for trees that are thought of as abstract and c or d for concrete trees.

A tree is essentially a finite partial function from names to other trees. It will be more convenient, though, to choose a slightly different definition: 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 $\{\!\}$, the tree 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 "pointwise" 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 $\left\{\!\middle|N \stackrel{?}{\mapsto} T\right|\!\}$ 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 $\{\!|\mathsf{333\text{-}4444} \mapsto \{\!\}|\!\}$ in the example of §2 is a value.

[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.

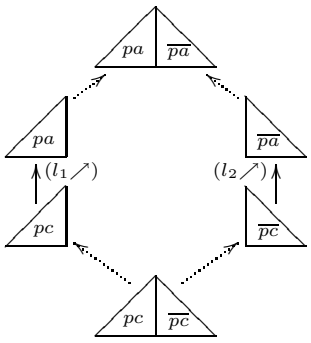

Figure 1: The *get* direction of `xfork`

5.2 Hoisting and Plunging

Let's warm up with some combinators that perform simple structural transformations on trees of very simple shapes. We will see in §5.3 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 *put* direction, the value of the old concrete tree is ignored and a new concrete tree is created, with a single edge n pointing to the given abstract tree.

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

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

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 *put* 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`.

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

$$\forall C{\subseteq}\mathcal{T}.\ \forall n{\in}\mathcal{N}. \quad \texttt{plunge}\ n \in C \overset{\Omega}{\rightleftharpoons} \{\!|n \mapsto C|\!\}$$

5.3 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 Figure 1 (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, and, 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 *put* direction, l_1 must map from pa to pc and l_2 from $\overline{pa}$ to $\overline{pc}$. Here is the full definition:

$$\begin{array}{rcl}(\texttt{xfork}\ pc\ pa\ l_1\ l_2)\nearrow c & = & (l_1\nearrow c|_{pc})\cdot(l_2\nearrow c\backslash_{pc}) \\ (\texttt{xfork}\ pc\ pa\ l_1\ l_2)\searrow(a,\ c) & = & (l_1\searrow(a|_{pa},\ c|_{pc}))\cdot(l_2\searrow(a\backslash_{pa},\ c\backslash_{pc}))\end{array}$$

$$\begin{array}{l}\forall pc, pa{\subseteq}\mathcal{N}.\ \forall C_1{\subseteq}\mathcal{T}|_{pc}.\ \forall A_1{\subseteq}\mathcal{T}|_{pa}. \\ \forall C_2{\subseteq}\mathcal{T}\backslash_{pc}.\ \forall A_2{\subseteq}\mathcal{T}\backslash_{pa}. \\ \forall l_1\in C_1 \overset{\Omega}{\rightleftharpoons} A_1.\ \forall l_2\in C_2 \overset{\Omega}{\rightleftharpoons} A_2. \\ \quad \texttt{xfork}\ pc\ pa\ l_1\ l_2 \in (C_1\cdot C_2) \overset{\Omega}{\rightleftharpoons} (A_1\cdot A_2)\end{array}$$

We rely here on our convention that $\Omega|_p = \Omega$ to avoid explicitly splitting out the Ω case in the *put* 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 define the simpler `fork` as:

$$\begin{array}{rcl}\texttt{fork}\ p\ l_1\ l_2 & = & \texttt{xfork}\ p\ p\ l_1\ l_2\end{array}$$

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

We may now define a lens that discards all of the children of a tree whose names do not belong to some set p:

$$\begin{array}{rcl}\texttt{filter}\ p\ d & = & \texttt{fork}\ p\ \texttt{id}\ (\texttt{const}\ \{\!|\}\ d)\end{array}$$

$$\begin{array}{l}\forall C{\subseteq}\mathcal{T}.\ \forall p{\subseteq}\mathcal{N}.\ \forall d\in C\backslash_p. \\ \quad \texttt{filter}\ p\ d \in (C|_p\cdot C\backslash_p) \overset{\Omega}{\rightleftharpoons} C|_p\end{array}$$

In the *get* direction, this lens takes a concrete tree, keeps the part of the tree whose children have names in p (using `id`), and throws away the rest of the tree (using `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: `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 $C|_p = C|_p\cdot\{\!|\}$.)

The next derived lens focuses attention on a single child n:

$$\begin{array}{rcl}\texttt{focus}\ n\ d & = & (\texttt{filter}\ \{n\}\ d);\ (\texttt{hoist}\ n)\end{array}$$

$$\forall n{\in}\mathcal{N}.\ \forall C{\subseteq}\mathcal{T}\backslash_n.\forall d{\in}C.\ \forall D{\subseteq}\mathcal{T}.\quad \texttt{focus}\ n\ d \in (C\cdot\{\!|n\mapsto D|\!\}) \overset{\Omega}{\rightleftharpoons} D$$

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 `filter` $\{n\}\ d \in (C\cdot\{\!|n\mapsto D|\!\}) \overset{\Omega}{\Longleftrightarrow} \{\!|n\mapsto D|\!\}$ and that `hoist n` $\in \{\!|n\mapsto D|\!\} \overset{\Omega}{\Longleftrightarrow} D$.

Our next derived lens renames a single child.

$$\begin{array}{rcl}\texttt{rename}\ m\ n & = & \texttt{xfork}\ \{m\}\ \{n\}\ (\texttt{hoist}\ m;\ \texttt{plunge}\ n)\ \texttt{id}\end{array}$$

$$\begin{array}{l}\forall m, n{\in}\mathcal{N}.\ \forall C{\subseteq}\mathcal{T}.\ \forall D{\subseteq}\mathcal{T}\backslash_{\{m,n\}}. \\ \quad \texttt{rename}\ m\ n \in (\{\!|m\mapsto C|\!\}\cdot D) \overset{\Omega}{\rightleftharpoons} (\{\!|n\mapsto C|\!\}\cdot D)\end{array}$$

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 *put* 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 (`hoist` m; `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 astract view has a child named n.

5.4 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 (and sometimes conditionals), 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 in the section, we will define a more general combinator, called `wmap`, that applies a different lens to each subtree.) The *put* direction of `map` 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. In general, however, the abstract tree in the *put* direction need not have the same domain as the concrete tree (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. First, note that the domain of the result is determined by the domain of the abstract argument. If $(\texttt{map}\ l){\searrow}(a,\, c)$ is defined, then, by rule PUTGET, we should have $(\texttt{map}\ l){\nearrow}((\texttt{map}\ l){\searrow}(a,\, c)) \sqsubseteq a$; thus we necessarily have $\mathsf{dom}((\texttt{map}\ l){\searrow}(a,\, c)) = \mathsf{dom}(a)$ (if the *put* is defined). This means we can simply drop children that occur in $\mathsf{dom}(c)$ but not $\mathsf{dom}(a)$. 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 $\mathsf{dom}(a)$. These need to be passed through l so that they can be included in the result; 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.)

$$
\begin{array}{rcl}
(\texttt{map}\ \mathrm{l}){\nearrow}\, c & = & \{\!|n \mapsto l{\nearrow}\, c(n) \mid n \in \mathsf{dom}(c)|\!\} \\
(\texttt{map}\ \mathrm{l}){\searrow}(a,\, c) & = & \{\!|n \mapsto l{\searrow}(a(n),\, c(n)) \mid n \in \mathsf{dom}(a)|\!\}
\end{array}
$$

$$
\begin{array}{l}
\forall C, A{\subseteq}\mathcal{T}\ \textit{with}\ C = C^{\circlearrowleft},\ A = A^{\circlearrowleft},\ \textit{and}\ \mathsf{dom}(C) = \mathsf{dom}(A). \\
\forall l \in (\bigcap_{n\in\mathcal{N}}.\ C(n) \overset{\Omega}{\rightleftharpoons} A(n)). \\
\quad \texttt{map}\ l \in C \overset{\Omega}{\rightleftharpoons} A
\end{array}
$$

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 *put* 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 *put* function can produce a tree that is not in C, as the following example shows. Consider the sets of trees

$$
\begin{array}{l}
C = \{\, \{\!|\mathtt{x} \mapsto \mathtt{m}|\!\},\ \{\!|\mathtt{y} \mapsto \mathtt{n}|\!\} \,\} \\
A = C \cup \{\, \{\!|\mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{n}|\!\} \,\}
\end{array}
$$

and observe that with trees

$$
\begin{array}{l}
a = \{\!|\mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{n}|\!\} \\
c = \{\!|\mathtt{x} \mapsto \mathtt{m}|\!\}
\end{array}
$$

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

$$
A = \{\, \{\!|\mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{p}|\!\},\ \{\!|\mathtt{x} \mapsto \mathtt{n},\ \mathtt{y} \mapsto \mathtt{q}|\!\} \,\},
$$

in which the value `m` only appears under `x` when `p` appears under `y`, and the set

$$C = \{\{\!|\mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{p}|\!\}, \{\!|\mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{q}|\!\}, \{\!|\mathtt{x} \mapsto \mathtt{n},\ \mathtt{y} \mapsto \mathtt{p}|\!\}, \{\!|\mathtt{x} \mapsto \mathtt{n},\ \mathtt{y} \mapsto \mathtt{q}|\!\}\},$$

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: $C(\mathtt{x}) = A(\mathtt{x}) = \{\{\!|\mathtt{m}|\!\}, \{\!|\mathtt{n}|\!\}\}$ and $C(\mathtt{y}) = A(\mathtt{y}) = \{\{\!|\mathtt{p}|\!\}, \{\!|\mathtt{q}|\!\}\}$, and the lens `id` has type $C(\mathtt{x}) \stackrel{\Omega}{\rightleftharpoons} A(\mathtt{x})$ and $C(\mathtt{y}) \stackrel{\Omega}{\rightleftharpoons} A(\mathtt{y})$ (and $C(z) = \emptyset \stackrel{\Omega}{\rightleftharpoons} \emptyset = A(z)$ for all other names z). But it is clearly not the case that $\mathtt{map\ id} \in C \stackrel{\Omega}{\rightleftharpoons} A$. 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, $A^{\circlearrowleft} = C^{\circlearrowleft} = C$.

In the situations where `map` is used, shuffle closure is typically very 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.

A final point worth emphasizing is the relation between the `map` lens combinator and the missing tree Ω. The *put* function of every other lens combinator only results in a *put* into the missing tree if the combinator itself is called on Ω. In the case of `map` l, calling its *put* function on some a and c where c is not the missing tree may result in the application of the *put* of l to Ω if a has some children that are not in c. In an earlier version 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 *put* 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 *put* 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 the general form of `map`, parameterized on a total function from names to lenses.

$$\begin{array}{rcl} (\mathtt{wmap}\ m) \nearrow c & = & \{\!|n \mapsto m(n) \nearrow c(n) \mid n \in \mathsf{dom}(c)|\!\} \\ (\mathtt{wmap}\ m) \searrow (a,\ c) & = & \{\!|n \mapsto m(n) \searrow (a(n),\ c(n)) \mid n \in \mathsf{dom}(a)|\!\} \end{array}$$

$$\begin{array}{l} \forall C, A \subseteq \mathcal{T}\ \textit{with}\ C = C^{\circlearrowleft},\ A = A^{\circlearrowleft},\ \textit{and}\ \mathsf{dom}(C) = \mathsf{dom}(A). \\ \forall m \in (\Pi n \in \mathcal{N}.\ C(n) \stackrel{\Omega}{\rightleftharpoons} A(n)). \\ \quad \mathtt{wmap}\ m \in C \stackrel{\Omega}{\rightleftharpoons} A \end{array}$$

In the type annotation, we use the dependent type notation $m \in \Pi n.\ C(n) \stackrel{\Omega}{\rightleftharpoons} A(n)$ 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 $\mathtt{wmap}\ \{x \mapsto \mathtt{plunge\ a}\}$ maps `plunge a` to trees under x and `id` to the subtrees of every other child.

5.5 Copying and 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, the `merge` lens takes a tree with two (equal) branches and deletes one of them. In the *put* direction, `merge` copies the updated value of the remaining branch to *both* branches in the concrete view.

There is some freedom in the type of `merge`. We can either give it a precise type that captures the 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 (which we do) by ignoring the equality constraint — if the two original branches are unequal, `merge` is still defined and well-behavedness is preserved. This is possible because the old concrete view is an argument to the *put* function, and can be tested to see whether the two branches were equal or not in c. If not, then the value in a does not overwrite the value in the deleted branch, allowing `merge` to obey PutGet.

$$
\begin{array}{rcl}
(\texttt{merge}\ m\ n) \nearrow c & = & c\backslash_n \\
(\texttt{merge}\ m\ n) \searrow (a,\, c) & = & \begin{cases} a \cdot \{\!|n \mapsto a(m)|\!\} & \text{if } c(m) = c(n) \\ a \cdot \{\!|n \mapsto c(n)|\!\} & \text{if } c(m) \neq c(n) \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\forall m, n \in \mathcal{N}.\ \forall C \subseteq \mathcal{T}\backslash_{\{m,n\}}.\ \forall D \subseteq \mathcal{T}. \\
\quad \texttt{merge}\ m\ n \in \\
\qquad (C \cdot \{\!|m \mapsto D_\Omega,\ n \mapsto D_\Omega|\!\}) \stackrel{\Omega}{\rightleftharpoons} (C \cdot \{\!|m \mapsto D_\Omega|\!\})
\end{array}
$$

It is also possible (as we show in the long version) to define a `copy` lens that duplicates a subtree in the *get* direction and performs an equality comparison in the *put* direction. Unfortunately, because of the asymmetry between *get* and *put*, this lens cannot be given a permissive type like the one we gave to `merge`: the fact that the duplicated subtrees must be kept equal shows up as a constraint on the target type of `copy`. This, in turn, means that it is nearly impossible to use `copy` as part of interesting composite lenses, except as the very last step.

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 our 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, (2) 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 B 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 *put* direction, `ccond` again examines the concrete view and applies the *put* of l_1 if it satisfies the predicate and l_2 otherwise.

$$
\begin{array}{rcl}
(\texttt{ccond}\ C_1\ l_1\ l_2)\nearrow c & = & \begin{cases} l_1\nearrow c & \text{if } c \in C_1 \\ l_2\nearrow c & \text{if } c \notin C_1 \end{cases} \\
(\texttt{ccond}\ C_1\ l_1\ l_2)\searrow(a,\, c) & = & \begin{cases} l_1\searrow(a,\, c) & \text{if } c \in C_1 \\ l_2\searrow(a,\, c) & \text{if } c \notin C_1 \end{cases}
\end{array}
$$

$$
\forall C, C_1, A \subseteq \mathcal{U}.\ \forall l_1 \in C \cap C_1 \overset{\Omega}{\rightleftharpoons} A.\ \forall l_2 \in C \backslash C_1 \overset{\Omega}{\rightleftharpoons} A. \qquad \texttt{ccond}\ C_1\ l_1\ l_2 \in C \overset{\Omega}{\rightleftharpoons} A
$$

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

$$
\begin{array}{rcl}
(\texttt{acond}\ C_1\ A_1\ l_1\ l_2)\nearrow c & = & \begin{cases} l_1\nearrow c & \text{if } c \in C_1 \\ l_2\nearrow c & \text{if } c \notin C_1 \end{cases} \\
(\texttt{acond}\ C_1\ A_1\ l_1\ l_2)\searrow(a,\, c) & = & \begin{cases} l_1\searrow(a,\, c) & \text{if } a \in A_1 \text{ and } c \in C_1 \\ l_1\searrow(a,\, \Omega) & \text{if } a \in A_1 \text{ and } c \notin C_1 \\ l_2\searrow(a,\, c) & \text{if } a \notin A_1 \text{ and } c \notin C_1 \\ l_2\searrow(a,\, \Omega) & \text{if } a \notin A_1 \text{ and } c \in C_1 \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\forall C, A, C_1, A_1 \subseteq \mathcal{U}. \\
\forall l_1 \in C \cap C_1 \overset{\Omega}{\rightleftharpoons} A \cap A_1.\ \forall l_2 \in (C \backslash C_1) \overset{\Omega}{\rightleftharpoons} (A \backslash A_1). \\
\quad \texttt{acond}\ C_1\ A_1\ l_1\ l_2 \in C \overset{\Omega}{\rightleftharpoons} A
\end{array}
$$

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 *put* where a is in the intersection of the targets) and like `acond` in the regions where the abstract argument to *put* 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 *put* 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.

$$
\begin{array}{rcl}
(\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2)\nearrow c & = & \begin{cases} l_1\nearrow c & \text{if } c \in C_1 \\ l_2\nearrow c & \text{if } c \notin C_1 \end{cases} \\
(\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2)\searrow(a,\, c) & = & \begin{cases} l_1\searrow(a,\, c) & \text{if } a \in A_1 \cap A_2 \text{ and } c \in C_1 \\ l_2\searrow(a,\, c) & \text{if } a \in A_1 \cap A_2 \text{ and } c \notin C_1 \\ l_1\searrow(a,\, c) & \text{if } a \in A_1 \backslash A_2 \text{ and } c \in (C_1)_\Omega \\ l_1\searrow(a,\, f_{21}(c)) & \text{if } a \in A_1 \backslash A_2 \text{ and } c \notin (C_1)_\Omega \\ l_2\searrow(a,\, c) & \text{if } a \in A_2 \backslash A_1 \text{ and } c \notin C_1 \\ l_2\searrow(a,\, f_{12}(c)) & \text{if } a \in A_2 \backslash A_1 \text{ and } c \in C_1 \end{cases}
\end{array}
$$

$$
\begin{array}{l}
\forall C, C_1, A_1, A_2 \subseteq \mathcal{U}. \\
\forall l_1 \in (C \cap C_1) \overset{\Omega}{\rightleftharpoons} A_1. \\
\forall l_2 \in (C \backslash C_1) \overset{\Omega}{\rightleftharpoons} A_2. \\
\forall f_{21} \in (C \backslash C_1) \to (C \cap C_1)_\Omega. \\
\forall f_{12} \in (C \cap C_1) \to (C \backslash C_1)_\Omega. \\
\quad \texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2 \in C \overset{\Omega}{\rightleftharpoons} (A_1 \cup A_2)
\end{array}
$$

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

Somewhat surprisingly, all this generality can actually be quite useful in practice! We will see an example depending on the full power of `cond` in the next section.

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 some very simple lenses for projecting the head and tail of a list encoded as a cons cell. We then define some recursive lenses implementing more complex operations on lists: mapping, reversal, and filtering. The simplest of these lenses, `list_map`, uses `wmap` and recursion to apply a lens to every element of a list. The next lens reverses the order of elements in a list. We conclude with a quite intricate derived form, `list_filter`, that uses the general conditional, `cond`, to filter lists according to some predicate.

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 *put* direction). Other list-processing derived forms that we have implemented (but do not show here) 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 (no children) or it has exactly two children, one named `*h` and another named `*t`, with t(`*t`) also a list. In the following, we use the lighter notation $[t_1 \ldots t_n]$ 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 `[`C`]` for the set of lists with elements in C—i.e., the smallest set of trees satisfying `[`C`]` $=$ `[]` $\cup$ (C :: `[`C`]`). We sometimes refine this notation to describe lists of specific lengths, writing `[`$D^{i..j}$`]` for lists of Ds whose length is at least i and at most j. The interleaving of a list of type `[`$B^{i..j}$`]` and a list of type `[`$C^{n..m}$`]`, taking elements from the first list and elements from the second in an arbitrary fashion but maintaining the relative order of each, is written `[`$B^{i..j}$`]`&`[`$C^{m..n}$`]`.

Our first two list lenses extract the head or tail of a list (or, more generally, any cons cell).

$$\texttt{hd}\ d \quad = \quad \texttt{focus *h}\ \{\!|\texttt{*t} \mapsto d|\!\}$$

$$\forall C, D \subseteq \mathcal{T}.\ \forall d \in D. \quad \texttt{hd}\ d \in (C :: D) \stackrel{\Omega}{\rightleftharpoons} C$$

$$\texttt{tl}\ d \quad = \quad \texttt{focus *t}\ \{\!|\texttt{*h} \mapsto d|\!\}$$

$$\forall C, D \subseteq \mathcal{T}.\ \forall d \in C. \quad \texttt{tl}\ d \in (C :: D) \stackrel{\Omega}{\rightleftharpoons} D$$

The lens `hd` expects a default tree, which it uses in the *put* direction as the tail of the created tree when the concrete tree is missing. In the *get* direction, `hd` 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 $\forall C \subseteq \mathcal{T}.\ \forall d \in$ `[`C`]`$.\ \texttt{hd}\ d \in$ `[`C`]` $\stackrel{\Omega}{\Longleftrightarrow} C$) as well as cons cells whose head and tail have arbitrary types. The types of `hd` and `tl` follow straightforwardly from the type of `focus`.

The `list_map` lens iterates over a list, applying a lens l to every element of the list:

$$\texttt{list\_map}\ l \quad = \quad \texttt{wmap}\ \{\texttt{*h} \mapsto l,\ \texttt{*t} \mapsto \texttt{list\_map}\ l\}$$

$$\forall C, A \subseteq \mathcal{T}.\ \forall l \in C \stackrel{\Omega}{\rightleftharpoons} A. \quad \texttt{list\_map}\ l \in \texttt{[}C\texttt{]} \stackrel{\Omega}{\rightleftharpoons} \texttt{[}A\texttt{]}$$

The *get* direction of this lens applies l to the subtree under `*h` and recurses on the subtree under `*t`. The *put* direction uses $l\!\searrow$ to put back 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 *put* into Ω.

It is worth noting how the recursive calls in `list_map` terminate. In the *get* direction, the `wmap` lens simply applies l to the head and `list_map` l to the tail until it reaches a tree with no children. Similarly, in the *put* direction, the lens is applied at each level of the abstract tree, using the corresponding part of the

concrete tree, if it is present, and Ω otherwise. In either case, the recursive calls continue until the entire tree has been traversed.

Because `list_map` is defined recursively, proving it is well behaved requires (just) a little more work than has been needed for the derived lenses we have seen above: we need to show that it has a particular type *assuming* that the recursive use of `list_map` has the same type. This is nothing very surprising: exactly the same reasoning process is used in typing recursive functional programs.

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 *put* direction, it restores the projected-away Es from the concrete list. Unlike `list_reverse`, the *put* function for `list_filter` depends on both the abstract and concrete views. 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 `list_filter` D E (for brevity, let us call it l) is clear. In the *put* direction, things are more interesting. To begin with, the lens laws impose some key constraints on the behavior of $l\diagdown$. The GetPut law forces the *put* 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 $l\diagdown(\texttt{[}d\texttt{]}, \texttt{[}e\ d\texttt{]}) = \texttt{[}e\ d\texttt{]}$. The PutGet law forces the *put* function to include every element of the abstract list in the resulting concrete list and to only take Es (not Ds) from the concrete list. In the general case, where the abstract list a is different from the filtered concrete list $l\nearrow c$, there is some freedom in how $l\diagdown$ behaves. First, it may selectively restore only some of the elements of E from the concrete list (or indeed, even less intuitively, it might add some new elements of E that it somehow makes up). Second, it may interleave the restored Es with the Ds from the abstract list in any order, as long as the order of the Ds is preserved from a. From these possibilities, 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 $l\diagdown$ appends the rest of the elements of a at the end. If a runs out first, then $l\diagdown$ keeps any remaining Es that may be left at the end of c (discarding any remaining Ds in c, as it must to satisfy PutGet). For example, $l\diagdown(\texttt{[]}, \texttt{[}d\ e\texttt{]})$ yields `[`e`]`, not `[]`, and $l\diagdown(\texttt{[}d\texttt{]}, \texttt{[}e\texttt{]})$ is `[`$e\ d$`]`, not `[`$d\ e$`]`.

These choices lead us to the following specification for a single step, in the *put* direction, of a recursively defined lens implementing l. If the abstract list a and concrete list c are both cons cells whose heads are in D, then it yields the head of a and recurses on both tails. If c begins with an E (i.e., c has type $E :: \texttt{[}D\texttt{]}\,\&\,\texttt{[}E\texttt{]}$), then it restores the head of c and recurses on a and the tail of c. If a is empty and c begins with a D (c has type $D :: \texttt{[}D\texttt{]}\,\&\,\texttt{[}E\texttt{]}$), then it restores all the remaining Es from c and returns. Translating this into the lens combinators defined above leads (modulo a little new notation and a few additional technicalities, explained below) to the definition below of `list_filter` and a helper lens, `inner_filter`, by mutual recursion. The singly recursive variant with `inner_filter` inlined has the same 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.

$$
\begin{array}{l}
\texttt{list\_filter}\ D\ E = \\
\quad \texttt{cond}\ [\texttt{E}]\ []\ [D^{1..\omega}]\ \mathit{fltr}_E\ (\lambda c.\ c@[\mathit{any}_D]) \\
\qquad (\texttt{const}\ []\ []) \\
\qquad (\texttt{inner\_filter}\ D\ E) \\
\\
\texttt{inner\_filter}\ D\ E = \\
\quad \texttt{ccond}\ (E :: ([D^{1..\omega}]\&[\texttt{E}])) \\
\qquad\quad (\texttt{tl}\ \mathit{any}_E;\ \texttt{inner\_filter}\ D\ E) \\
\qquad\quad (\texttt{wmap}\ \{\texttt{*t} \mapsto \texttt{list\_filter}\ D\ E\}) \\
\hline
\forall D, E \subseteq \mathcal{T}.\ \mathit{with}\ D \cap E = \emptyset\ \mathit{and}\ D \neq \emptyset\ \mathit{and}\ E \neq \emptyset. \\
\quad \texttt{list\_filter}\ D\ E \in [D]\&[\texttt{E}] \stackrel{\Omega}{=\!=} [D]\ \mathit{and} \\
\quad \texttt{inner\_filter}\ D\ E \in [D^{1..\omega}]\&[\texttt{E}] \stackrel{\Omega}{=\!=} [D^{1..\omega}]
\end{array}
$$

The "choice operator" any_D denotes an arbitrary element of a non-empty set D.[3] The function fltr_E is used by the `cond` to strip out any Ds from the tail of c remaining when the a argument becomes empty; this 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 `list_filter`↗ here; but for expository purposes we prefer to avoid this extra bit of recursiveness.) Finally, the function $\lambda c.\ c@[\mathit{any}_D]$ appends some arbitrary element of D to the right-hand end of a list c. It is used by `cond` for the case where a non-empty a is being *put* into a list c that does not contain any Ds; by adding a dummy d at the end of c, it produces a concrete list that can validly be passed to `inner_filter`, which expects at least one D in its concrete argument marking the point where the head of a should be placed.

To illustrate how all this works, let us step through two examples in detail. In both, the concrete type is $[D]\&[\texttt{E}]$ and the abstract type is $[D]$ where $D = \{\texttt{d}\}$ and $E = \{\texttt{e}\}$. For the first example, let the abstract tree $a = [\texttt{d}]$, and the concrete tree $c = [\texttt{e d e}]$. At each step, we underline the next term to be reduced.

$$
\begin{array}{rl}
 & \underline{(\texttt{list\_filter}\ D\ E) \searrow (a,\ c)} \\
= & \underline{(\texttt{inner\_filter}\ D\ E) \searrow (a,\ c)} \\
 & \quad \text{by the definition of } \texttt{cond}, \text{ as } a = [\texttt{d}] \in D :: [D] \text{ and } c \in ([D]\&[\texttt{E}]) \setminus [\texttt{E}] \\
= & \underline{(\texttt{tl}\ \mathit{any}_E;\ \texttt{inner\_filter}\ D\ E) \searrow (a,\ c)} \\
 & \quad \text{by the definition of } \texttt{ccond}, \text{ as } c = [\texttt{e d e}] \in E :: ([D^{1..\omega}]\&[\texttt{E}]) \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big((\texttt{inner\_filter}\ D\ E) \searrow \big(a,\ \underline{(\texttt{tl}\ \mathit{any}_E) \nearrow c}\big),\ c\Big) \\
 & \quad \text{by the definition of composition} \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\underline{(\texttt{inner\_filter}\ D\ E) \searrow (a,\ [\texttt{d e}])},\ c\Big) \\
 & \quad \text{reducing } (\texttt{tl}\ \mathit{any}_E) \nearrow c \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\underline{(\texttt{wmap}\ \{\texttt{*t} \mapsto \texttt{list\_filter}\ D\ E\}) \searrow (a,\ [\texttt{d e}])},\ c\Big) \\
 & \quad \text{by the definition of } \texttt{ccond}, \text{ as } a = [\texttt{d}] \notin E :: ([D^{1..\omega}]\&[\texttt{E}]) \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\texttt{d} :: (\underline{(\texttt{list\_filter}\ D\ E) \searrow ([],\ [\texttt{e}])}),\ c\Big) \\
 & \quad \text{by the definition of } \texttt{wmap} \text{ plus } \texttt{id} \searrow (\texttt{d},\ \texttt{d}) = \texttt{d} \\
= & (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\texttt{d} :: (\underline{(\texttt{const}\ []\ []) \searrow ([],\ [\texttt{e}])}),\ c\Big) \\
 & \quad \text{by the definition of } \texttt{cond}, \text{ as } [] \in [] \text{ and } [\texttt{e}] \in [\texttt{E}] \\
= & \underline{(\texttt{tl}\ \mathit{any}_E) \searrow (\texttt{d} :: [\texttt{e}],\ c)} \\
 & \quad \text{by the definition of } \texttt{const} \\
= & [\texttt{e d e}] \\
 & \quad \text{by the definition of } \texttt{tl}
\end{array}
$$

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

The second example illustrates how the "fixup functions" supplied to the `cond` lens are used. Let a = `[]` and c = `[d e]`.

$$
\begin{aligned}
& \underline{(\texttt{list\_filter}\ D\ E) \searrow (a,\ c)} \\
= \ & (\texttt{const []\ []}) \searrow \Big(\texttt{[]},\ \underline{(\lambda c.\mathit{fltr}_E\ c)\texttt{[d e]}}\Big) \\
& \quad \text{by the definition of } \texttt{cond}\text{, as } a = \texttt{[]} \text{ but } c \notin [E] \\
= \ & \underline{(\texttt{const []\ []}) \searrow (\texttt{[]},\ \texttt{[e]})} \\
& \quad \text{by the definition of } \mathit{fltr}_E \\
= \ & \texttt{[e]} \\
& \quad \text{by definition of } \texttt{const}
\end{aligned}
$$

8 Related Work

The lens combinators described in this paper 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 [32], 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 "programming language treatment" of these structures has led us to a formulation that 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 [13] 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 *put* functions.

Bancilhon and Spyratos [7] 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 [16] 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 [31], 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 *put* functions have type $(A \longrightarrow A) \longrightarrow (C \longrightarrow C)$, while our lenses translate abstract *states* into update functions on C, i.e., our *put* functions have type (isomorphic to) $A \longrightarrow (C \longrightarrow C)$. Moreover, in both of these frameworks, "update translators" (the analog of our *put* 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 *put* 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.

In the literature on programming languages, laws similar to our lens laws (but somewhat simpler, dealing only with total *get* and *put* functions) appear in Oles' category of "state shapes" [30] and in Hofmann and Pierce's work on "positive subtyping" [17].

Recent work by Lechtenbörger [21] establishes that translations of view updates under constant complements are possible precisely if view update effects may be undone using further view updates.

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 put 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 *put* function.
- In *bijective languages*, the *put* 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 *put* 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 *put* function at all: instead, the *get* function itself is constructed so that each step can be run in reverse.

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 [26]. Meertens's semantic setting is actually even more general: he takes *get* and *put* to be *relations*, not just functions, and his constraint maintainers are symmetric: *get* relates pairs from $C \times A$ to elements of A and *put* relates pairs in $A \times C$ to elements of C; the idea is that a constraint maintainer forms a connection between two graphical objects on the screen so that, whenever one of the objects is changed by the user, the change can be propagated by the maintainer to the other object such that some desired relationship between the objects is always maintained. 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 [27] 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. For example, an abstract view may maintain two lists where the name field of each element in one list must match the name field in the corresponding element in the other list. If an element is added to the first list, then not only must the change be propagated to the concrete view, it must also add a new element to the second list in the abstract view. It is easy to see that PutGet cannot hold if the abstract view, itself, is—in this sense—modified by the *put*. Similarly, they assume that edits to the abstract view mark all modified fields as "updated." These marks are removed when the *put* lens computes the modifications to the concrete view—another change to the abstract view that must violate PutGet. Consequently, to support invariant preservation within the abstract view, and to support edit lists, their transformations only obey a much weaker variant of PutGet (described above in Section 5.5).

Another paper by Hu, Mu, and Takeichi [18] applies a bi-directional programming language quite closely related to ours to the design of "programmable editors" for structured documents. As in [27], they support preservation of local invariants in the *put* direction. Here, instead of annotating the abstract view with modification marks, they assume that a *put* 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 *put* the data from the modified branch and overwrite the unmodified branch.

The TRIP2 system (e.g. [23]) uses bidirectional transformations specified as collections of Prolog rules as a means of implementing direct-manipulation interfaces for application data structures. The *get* and *put* 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 [14] derives the inverse program from the forward program. Inverse computation [24] computes a possible input of a program from the program and a particular output. One approach to inverse computation is to design languages that produce easily invertible expressions. For example, designing languages that can only express injective functions (in which case 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 *put*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 [29] 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* [35], extends algebraic pattern matching to abstract data types using programmer-supplied *in* and *out* operators.

Languages for Reversible Transformations Our work is the first (of which we are aware) in which totality and compositional reasoning about totality are taken as primary design goals. Nevertheless, in all of the languages discussed above there is an expectation that programmers will want their transformations to be "total enough"—i.e., that the sets of inputs for which the *get* and *put* functions are defined should be large enough for some given purpose. In particular, we expect that *put* functions should be accept a suitably large

set of abstract inputs for each given concrete input, since the whole point of these languages is to allow editing through a view. A quite different class of languages have been designed to support *reversible* computation, in which the *put* functions are only ever applied to the results of the corresponding *get* functions. While the goals of these languages are quite different from ours—they have nothing to do with view update—there are intriguing similarities in the basic approach.

Landauer [20] observed that non-injective functions were logically irreversible, and that this irreversibility requires the generation and dissipation of some heat per machine cycle. Bennet [9] demonstrated that this irreversibility was not inevitable by constructing a *reversible Turing machine*, showing that thermodynamically reversible computers were plausible. Baker [6] argued that irreversible primitives were only part of the problem; irreversibility at the "highest levels" of computer usage cause the most difficulty due to information loss. Consequently, he advocated the design of programs that "conserve information." Because deciding reversibility of large programs is unsolvable, he proposed designing languages that guaranteed that all well-formed programs are reversible, i.e. designing languages whose primitives were reversible, and whose combinators preserved reversibility. A considerable body of work has developed around these ideas (e.g. [28]).

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 [10] 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 [34] 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 [33] 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 *put* functions, either automatically or with some user assistance. By contrast, we have designed a new language in which the definitions of *get* and *put* 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 [22] 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 [19] 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 [8] described a scheme for interactively constructing update translators for object-based views of relational databases.

Medeiros and Tompa [25] 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 [12] 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 [11] 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 taken care to find 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 constraints on our design of new lenses. These strong constraints, paradoxically, make the design process easier. In the early stages of the Harmony, 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 lenses themselves 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 re-applies 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 both intricate structural manipulations such as the list transformations in §7 as well as 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.).

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!

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; Zhe contributed numerous insights whose generic material can be found (generally in much-recombined form) in this paper. Owen Gunden and, more recently, Malo Denielou have also collaborated with us on many aspects of the Harmony design and implementation; in particular, Malo's compiler and programming environment for the combinators described in this paper have contributed enormously. Trevor Jim provided the initial push to start the project by observing that the next step beyond the Unison file synchronizer (of which Trevor was a co-designer) would be synchronizing XML. 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, Malo Denielou, Owen Gunden, Michael Hicks, Zack Ives, Trevor Jim, Kate Moore, Wang-Chiew Tan, Stephen Tse, and Zhe Yang, for very helpful comments on earlier drafts of this paper. The Harmony project is supported by the National Science Foundation under grant ITR-0113226, *Principles and Practice of Synchronization.*

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. Proceedings*, 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] H. G. Baker. NREVERSAL of fortune – the thermodynamics of garbage collection. In *Proc. Int'l Workshop on Memory Management*, September 1992. St. Malo, France. Springer LNCS 637, 1992.

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

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

[9] C. H. Bennet. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

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

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

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

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

[14] E. W. Dijkstra. Program inversion. In F. L. Bauer and M. Broy, editors, *Program Construction, International Summer School, July 26 - August 6, 1978, Marktoberdorf, germany*, volume 69 of *Lecture Notes in Computer Science*. Springer, 1979.

[15] 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.

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

[17] M. Hofmann and B. Pierce. Positive subtyping. In *POPL'95*, 1995.

[18] 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. To appear.

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

[20] R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5(3):183–191, 1961. (Republished in IBM Jour. of Res. and Devel., 44(1/2):261-269, Jan/Mar. 2000).

[21] 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.

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

[23] 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.

[24] J. McCarthy. The inversion of functions defined by turing machines. In C. E. Shannon and J. McCarthy, editors, *Automata Studies, Annals of Mathematical Studies*, number 34, pages 177–181. Princeton University Press, 1956.

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

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

[27] 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. To appear.

[28] S.-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *Seventh International Conference on Mathematics of Program Construction (MPC)*, 2004.

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

[30] 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.

[31] B. C. Pierce and A. Schmitt. Lenses and view update translation. Manuscript; available at `http://www.cis.upenn.edu/~bcpierce/harmony`, 2003.

[32] 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. Submitted for publication.

[33] 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.

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

[35] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *POPL'87*. 1987.