Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem

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

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

J. NATHAN FOSTER
University of Pennsylvania
MICHAEL B. GREENWALD
Bell Labs, Lucent Technologies
JONATHAN T. MOORE
University of Pennsylvania
BENJAMIN C. PIERCE
University of Pennsylvania
ALAN SCHMITT
INRIA Rhône-Alpes

We propose a novel approach to the *view update problem* for 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 begin by identifying a natural mathematical space of well-behaved bi-directional transformations over arbitrary structures, studying definedness and continuity in this setting. We then instantiate this semantic framework in the form of a collection of *lens combinators* that can be assembled to describe bi-directional 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, copying, merging, etc.). We illustrate the expressiveness of these combinators by developing a number of bi-directional list-processing transformations as derived forms. An extended example shows how our combinators can be used to define a lens that translates between a native HTML representation of browser bookmarks and a generic abstract bookmark format.

Categories and Subject Descriptors: D.3.2 [**Programming Languages**]: Language Classifications—*Specialized application languages*

General Terms: Languages

Additional Key Words and Phrases: Bi-directional programming, Harmony, XML, lenses, view update problem

1. INTRODUCTION

Computing is full of situations where some structure must be converted to a different form—a *view*—in such a way that changes made to the view can be reflected 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 [Foster et al. 2006]. 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. The ultimate aim of the project is to provide a platform on which a Harmony programmer can quickly assemble a high-quality synchronizer for a new type of tree-structured data stored in a standard low-level format such as XML. 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 structures that may be stored in disparate concrete formats, we define a single common abstract format and a collection of *lenses* that transform each concrete format into this abstract one. For example, we can synchronize a Mozilla bookmark file with an Internet Explorer bookmark file by transforming each into an *abstract bookmark structure* and propagating changed information between these. Afterwards, we need to take the updated abstract structures and reflect the corresponding updates back into the original 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 putting an updated abstract view back into the original concrete view to yield an updated concrete view. We call these the *get* and *putback* components, respectively. The intuition is that the mapping from concrete to abstract is commonly some sort of projection, so the *get* direction involves getting the abstract part out of a larger concrete structure, while the *putback* direction amounts to putting a new abstract part into an old concrete structure. We show a concrete example of this process in Section 2.

The difficulty of the view update problem springs from a fundamental tension between *expressiveness* and *robustness*. The richer we make the set of possible transformations in the *get* direction, the more difficult it becomes to define corresponding functions in the *putback* direction in such as way that each lens is both *well behaved*—its *get* and *putback* behaviors fit together in a sensible way—and *total*—its *get* and *putback* functions are defined on all the inputs to which they may be applied.

To reconcile this tension, a successful 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 in-

stances. 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. (We do hope that our work will suggest new attacks on these problems, however; a first step in this direction is described in [Bohannon et al. 2006].)

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

The first step in our formal development (Section 3) is identifying a natural mathematical space of well-behaved lenses over arbitrary data structures. There is a good deal of territory to be explored at this semantic level. First, we must phrase our basic definitions to allow the underlying functions in lenses to be partial, since there will in general be structures to which a given lens cannot sensibly be applied. The sets of structures to which we *do* intend to apply a given lens are specified by associating it with a type of the form $C \rightleftharpoons A$, where C is a set of concrete "source structures" and A is a set of abstract "target structures." Second, we define a notion of well-behavedness that captures our intuitions about how the *get* and *putback* parts of a lens should behave in concert. For example, if we use the *get* part of a lens to extract an abstract view a from a concrete view c and then use the *putback* part to push the very same a back into c, we should get c back. Third, we deploy standard tools from domain theory to define monotonicity and continuity for lens combinators parameterized on other lenses, establishing a foundation for defining lenses by recursion. (Recursion is needed because the trees that our lenses manipulate may in general have arbitrarily deep nested structure—e.g., when they represent directory hierarchies, bookmark folders, etc.) Finally, to allow lenses to be used to create new concrete structures rather than just updating existing ones (needed, for example, when new records are added to a database in the abstract view), we adjoin a special "missing" element to the structures manipulated by lenses and establish suitable conventions for how it is treated.

With these semantic foundations in hand, we proceed to syntax. In Section 4, we present a group of generic lens combinators (identities, composition, and constants), which can work with any kind of data. In Section 5, we focus attention on tree-structured data and present several more combinators that perform various manipulations on trees (hoisting, splitting, mapping, etc.); we also show how to assemble these primitives, along with the generic combinators from before, to yield some useful derived forms. Section 6 introduces another set of generic combinators implementing various sorts of bi-directional conditionals. Section 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, including lenses for projecting the head and tail of a list, mapping over a list, grouping the elements of a list, concatenating two lists, and—our most complex example—implementing a bi-directional filter lens whose *putback* function performs a rather intricate "weaving" operation to recombine an updated abstract list with the concrete list elements that were filtered away by the *get*. This example also demonstrates the use of the reasoning techniques developed in Section 3 for establishing totality of recursive lenses. Section 8 further illustrates the use of our combinators in real-world lens programming by walking through a substantial example derived from the Harmony bookmark synchronizer.

Section 9 presents some first steps into a somewhat different region of the lens design space: lenses for dealing with relational data encoded as trees. We define three more primitives—a "flattening" combinator that transforms a list of (keyed) records into a bush, a "pivoting" combinator that can be used to promote a key field to a higher position in the tree, and a "transposing" combinator related to the outer join operation on databases. The first two combinators play an important role in Harmony instances for relational data such as address books encoded as XML trees.

Section 10 surveys related work and Section 11 sketches directions for future research.

To keep things moving, we defer all proofs to an electronic appendix, which is available on both the Harmony and TOPLAS web pages.

2. A SMALL EXAMPLE

Suppose our concrete tree c is a simple 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 curly 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 actually stands for "$\{\texttt{333-4444} \mapsto \{\}\}$." When trees are linearized in running text, we separate children with commas for easier reading.

Now, suppose that we want to edit the data from this concrete tree in a yet simpler 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 generated 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 and the edits are going to be performed by a human who is only interested in phone information and doesn't want to see URLs. Whatever the reason, we are going to make our changes to the abstract tree a, yielding a new abstract tree a' of

the same form but with modified content.[1] For example, let us change Pat's phone number, drop Chris, and add a new friend, Jo.

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

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

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

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

Together, the transformations from c to a and from a' plus 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. To whet the reader's appetite, the lens expression that implements the transformations above is `map (focus Phone {|URL ↦ http://google.com|})`.

3. SEMANTIC FOUNDATIONS

Although many of our combinators work on trees, their semantic underpinnings can be presented in an abstract setting parameterized by the data structures (which we call "views") manipulated by lenses.[2] In this section—and in Section 4, where we discuss generic combinators—we simply assume some fixed set $\mathcal{V}$ of views; from Section 5 on, we will choose $\mathcal{V}$ to be the set of trees.

Basic Structures

When f is a partial function, we write $f(a) \downarrow$ if f is defined on argument a and $f(a) = \bot$ otherwise. We write $f(a) \sqsubseteq b$ for $f(a) = \bot \vee f(a) = b$. We write $\mathsf{dom}(f)$ for $\{s \mid f(s) \downarrow\}$, the set of arguments on which f is defined. When $S \subseteq \mathcal{V}$, we write

[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, which is designed to support synchronization of off-the-shelf applications, where in general we only have access to the current states of the replicas, rather than a trace of modifications; the tradeoffs between state-based and trace-based synchronizers are discussed in detail elsewhere [Pierce and Vouillon 2004; Foster et al. 2006].

[2] We use the word "view" here in a slightly different sense than some of the database papers that we cite, where a view is a *query* that maps concrete to abstract states—i.e., it is a function that, for each concrete database state, picks out a view in our sense. Also, note that we use "view" to refer uniformly to both concrete and abstract structures—when we come to programming with lenses, the distinction will be merely a matter of perspective anyway, since the output of one lens is often the input to another.

$f(S)$ for $\{r \mid s \in S \ \wedge \ f(s)\downarrow \ \wedge f(s) = r\}$ and $\mathsf{ran}(f)$ for $f(\mathcal{V})$. We take function application to be strict: $f(g(x))\downarrow$ implies $g(x)\downarrow$.

3.1 Definition [Lenses]: A *lens* l comprises a partial function $l\nearrow$ from $\mathcal{V}$ to $\mathcal{V}$, called the *get function* of l, and a partial function $l\searrow$ from $\mathcal{V} \times \mathcal{V}$ to $\mathcal{V}$, called the *putback 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 *putback* part "pushes down" a new abstract view into an existing concrete view. We often say "put a into c (using l)" instead of "apply the *putback* function (of l) to (a, c)."

3.2 Definition [Well-behaved lenses]: Let l be a lens and let C and A be subsets of $\mathcal{V}$. We say that l is a *well behaved* lens from C to A, written $l \in C \rightleftharpoons A$, if 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 *putback* 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 $C \rightleftharpoons A$. Note that a given l may be a well-behaved lens from C to A for many different Cs and As; in particular, every l is trivially a well-behaved lens from $\emptyset$ to $\emptyset$, while the everywhere-undefined lens belongs to $C \rightleftharpoons A$ for every C and A.

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

An example of a lens satisfying PUTGET but not GETPUT is the following. Suppose $C = \texttt{string} \times \texttt{int}$ and $A = \texttt{string}$, and define l by:

$$l\nearrow(s, n) = s \qquad\qquad l\searrow(s',\, (s, n)) = (s', 0)$$

Then $l\searrow(l\nearrow(s, 1),\, (s, 1)) = (s, 0) \not\sqsubseteq (s, 1)$. Intuitively, the law fails because the *putback* function has "side effects": it modifies information in 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 :

$$l\nearrow s = (s, 0) \qquad\qquad l\searrow((s', n),\, s) = s'$$

PUTGET fails here because some information contained in the abstract view does not get propagated to the new concrete view. For example, $l\nearrow(l\searrow((s', 1),\, s)) = l\nearrow s' = (s', 0) \not\sqsubseteq (s', 1)$.

The GETPUT and PUTGET laws reflect fundamental expectations about the behavior of lenses; removing either law significantly weakens the semantic foundation.

We may also consider an optional third law, called PUTPUT:

$$l \searrow (a',\, l \searrow (a,\, c)) \sqsubseteq l \searrow (a',\, c) \quad \text{for all } a, a' \in A \text{ and } c \in C.$$

This law states that the effect of a sequence of two *putback*s is (modulo definedness) just the effect of the second: the first gets completely overwritten. Alternatively, a series of changes to an abstract view may be applied either incrementally or all at once, resulting in the same final concrete view. We say that a well-behaved lens that also satisfies PUTPUT is *very well behaved*. Both well-behaved and very well behaved lenses correspond to familiar classes of "update translators" from the classical database literature; see Section 10.

The foundational development in this section is valid for both well-behaved and very well behaved lenses. However, when we come to defining our lens combinators for tree transformations, we will not require PUTPUT because some of our lens combinators—in particular, `map`, `flatten`, `merge`, and conditionals—fail to satisfy it for reasons that seem pragmatically unavoidable (see Sections 5 and 9).

For now, a simple example of a lens that is well behaved but not very well behaved is as follows. Consider the following lens, where $C = \texttt{string} \times \texttt{int}$ and $A = \texttt{string}$. The second component of each concrete view intuitively represents a version number.

$$l \nearrow (s, n) = s \qquad\qquad l \searrow (s,\, (s', n)) = \begin{cases} (s, n) & \text{if } s = s' \\ (s, n{+}1) & \text{if } s \neq s' \end{cases}$$

The *get* function of l projects away the version number and yields just the "data part." The *putback* function overwrites the data part, checks whether the new data part is the same as the old one, and, if not, increments the version number. This lens satisfies both GETPUT and PUTGET but not PUTPUT, as we have $l \searrow (s,\, l \searrow (s',\, (c, n))) = (s, n+2) \not\sqsubseteq (s, n+1) = l \searrow (s,\, (c, n))$.

Another critical property of lenses is *totality* with respect to a given source and target.

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

The reasons for considering both partial and total lenses instead of building totality into the definition of well-behavedness are much the same as the reasons for considering partial functions in conventional functional languages. In practice, we want lenses to be total:[3] to guarantee that Harmony synchronizers will work predictably, lenses must be defined on the whole of the domains where they are used; the *get* direction should be defined for any structure in the concrete set, and the *putback* direction should be capable of putting back any possible updated version from the abstract set.[4] All of our primitive lenses are designed to be total, and all of our lens

[3]Indeed, well-behavedness is rather 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, slightly less trivially, to be defined only on inputs of the form $(l\nearrow c, c)$.

[4]Since we intend to use lenses to build synchronizers, the updated structures here will be results of synchronization. A fundamental property of the core synchronization algorithm in Harmony is that, if all of the updates between synchronizations occur in just one of the replicas, then the effect

combinators map total lenses to total lenses—with the sole, but important, exception of lenses defined by recursion; as usual, recursive lenses must be constructed in the semantics as limits of chains of increasingly defined partial lenses. The soundness of the type annotations we give for our syntactic lens combinators guarantees that *every* well-typed lens expression is well-behaved, but only recursion-free expressions can be shown total by completely compositional reasoning with types; for recursive lenses, more global arguments are required, as we shall see.

Basic Properties

We now explore some simple but useful consequences of the lens laws. All the proofs can be found in the electronic appendix.

3.4 Definition: Let f be a partial function from $A \times C$ to C and $P \subseteq A \times C$. We say that f is *semi-injective on* P if it is injective (in the standard sense) in the first component of arguments drawn from P—i.e., if, for all views a, a', c, and c' with $(a, c) \in P$ and $(a', c') \in P$, if $f(a, c) \downarrow$ and $f(a', c') \downarrow$, then $a \neq a'$ implies $f(a, c) \neq f(a', c')$.

3.5 Lemma: If $l \in C \rightleftharpoons A$, then $l\searrow$ is semi-injective on $\{(a, c) \mid (a, c) \in A \times C \ \wedge \ l\nearrow(l\searrow(a, c))\downarrow\}$.

The main application of this lemma is the following corollary, which provides an easy way to show that a lens is *not* well behaved. We used it many times while designing our combinators, to quickly generate and test candidates.

3.6 Corollary: If $l \in C \Longleftrightarrow A$, then $l\searrow$ is semi-injective on $A \times C$.

An important special case arises when the *putback* function of a lens is completely insensitive to its concrete argument.

3.7 Definition: A lens l is said to be *oblivious* if $l\searrow(a, c) = l\searrow(a, c')$ for all $a, c, c' \in \mathcal{V}$.

Oblivious lenses have some special properties that make them simpler to reason about than lenses in general. For example:

3.8 Lemma: If l is oblivious and $l \in C_1 \rightleftharpoons A_1$ and $l \in C_2 \rightleftharpoons A_2$, then $l \in (C_1 \cup C_2) \rightleftharpoons (A_1 \cup A_2)$.

3.9 Lemma: If $l \in C \Longleftrightarrow A$ is oblivious, then $l\nearrow$ is a bijection from C to A.

Conversely, every bijection between C and A induces a total oblivious lens from C to A—that is, the set of bijections between subsets of $\mathcal{V}$ forms a subcategory of the category of total lenses. Many of the combinators defined below actually live in this simpler subcategory, as does much of the related work surveyed in Section 10.

---

of synchronization will be to propagate all these changes to the other replica. This implies that the *putback* function in the lens associated with the other replica must be prepared to accept any value from the abstract domain. In other settings, different notions of totality may be appropriate. For example, Hu, Mu, and Takeichi [Hu et al. 2004] have argued that, in the context of interactive editors, a reasonable definition of totality is that $l\searrow(a, c)$ should be defined whenever a differs by at most one edit operation from $l\nearrow c$.

Recursion

Since we will be interested in lenses over trees, and since trees in many application domains may have unbounded depth (e.g., a bookmark can be either a link or a folder containing a list of bookmarks), we will often want to define lenses by recursion. Our next task 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 *putback* functions of l' have domains that are at least as large as those of l and their results agree on their common domains:

3.10 Definition: $l \prec l'$ iff $\mathsf{dom}(l\nearrow) \subseteq \mathsf{dom}(l'\nearrow)$, $\mathsf{dom}(l\searrow) \subseteq \mathsf{dom}(l'\searrow)$, $l\nearrow c = l'\nearrow c$ for all $c \in \mathsf{dom}(l\nearrow)$, and $l\searrow(a,\, c) = l'\searrow(a,\, c)$ for all $(a, c) \in \mathsf{dom}(l\searrow)$.

3.11 Lemma: $\prec$ is a partial order on lenses.

A *cpo* is a partially ordered set in which every increasing chain of elements has a least upper bound in the set. If $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ is an increasing chain, we write $\bigsqcup_{n\in\omega} l_n$ (often shortened to $\bigsqcup_n l_n$) for its least upper bound. A *cpo with bottom* is a cpo with an element $\bot$ that is smaller than every other element. In our setting, the bottom element $\bot_l$ is the lens whose *get* and *putback* functions are everywhere undefined. It is obviously the smallest lens according to $\prec$ and is well-behaved at any lens type (it trivially satisfies all equations).

3.12 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

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

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

3.13 Corollary: Let $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ be an increasing chain of lenses. For every $a, c \in \mathcal{V}$, we have:

(1) $(\bigsqcup_n l_n)\nearrow c = v \quad \text{iff} \quad \exists i.\ l_i\nearrow c = v.$
(2) $(\bigsqcup_n l_n)\searrow(a,\, c) = v \quad \text{iff} \quad \exists i.\ l_i\searrow(a,\, c) = v.$

3.14 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 $\mathcal{V}$. Then:

(1) Well-behavedness commutes with limits:
$(\forall i \in \omega.\ l_i \in C_i \rightleftharpoons A_i) \quad \text{implies} \quad \bigsqcup_n l_n \in (\bigcup_i C_i) \rightleftharpoons (\bigcup_i A_i).$
(2) Totality commutes with limits:
$(\forall i \in \omega.\ l_i \in C_i \iff A_i) \quad \text{implies} \quad \bigsqcup_n l_n \in (\bigcup_i C_i) \iff (\bigcup_i A_i).$

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

When defining lenses, we will make heavy use of the following standard theorem from domain theory (e.g., [Winskel 1993]). Recall that a function f between two cpos is *continuous* if it is monotonic and if, for all increasing chains $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$, we have $f(\bigsqcup_n l_n) = \bigsqcup_n f(l_n)$. A fixed point of f is a function $\mathit{fix}(f)$ satisfying $\mathit{fix}(f) = f(\mathit{fix}(f))$.

3.16 Theorem [Fixed-Point Theorem]: Let f be a continuous function from D to D, where D is a cpo with bottom. Define

$$\mathit{fix}(f) = \bigsqcup_n f^n(\bot)$$

Then $\mathit{fix}(f)$ is a fixed point, in fact the least fixed point, of f.

Theorem 3.15 tells us that we can apply Theorem 3.16 to continuous functions from lenses to lenses—i.e., it justifies defining lenses by recursion. The following corollary packages up this argument in a convenient form; we will appeal to it many times in later sections to show that recursive derived forms are well behaved and total.

3.17 Corollary: Suppose f is a continuous function from lenses to lenses.

(1) If $l \in C \rightleftharpoons A$ implies $f(l) \in C \rightleftharpoons A$ for all l, then $\mathit{fix}(f) \in C \rightleftharpoons A$.

(2) Suppose $\emptyset = C_0 \subseteq C_1 \subseteq \ldots$ and $\emptyset = A_0 \subseteq A_1 \subseteq \ldots$ are increasing chains of subsets of $\mathcal{V}$. If $l \in C_i \Longleftrightarrow A_i$ implies $f(l) \in C_{i+1} \Longleftrightarrow A_{i+1}$ for all i and l, then $\mathit{fix}(f) \in (\bigcup_i C_i) \Longleftrightarrow (\bigcup_i A_i)$.

We can now apply standard domain theory to interpret a variety of constructs for defining continuous lens combinators. We say that an expression e is continuous in the variable x if the function $\lambda x.e$ is continuous. An expression is said to be continuous in its variables, or simply continuous, if it is continuous in every variable separately. Examples of continuous expressions are variables, constants, tuples (of continuous expressions), projections (from continuous expressions), applications of continuous functions to continuous arguments, lambda abstractions (whose bodies are continuous), let bindings (of continuous expressions in continuous bodies), case constructions (of continuous expressions), and the fixed point operator itself. Tupling and projection let us define mutually recursive functions: if we want to define f as $F(f, g)$ and g as $G(f, g)$, where both F and G are continuous, we define $(f, g) = \mathit{fix}(\lambda(x, y).(F(x, y), G(x, y)))$.

When proving the totality of recursive lenses, we sometimes need to use a more powerful induction scheme in which a lens is proved, simultaneously, to be total on a whole collection of different types (any of which can be used in the induction step). This is supported by a generalization of the proof technique in 3.17(2).

We specify a *total type* by a pair (C, A) of subsets of $\mathcal{V}$, and say that a lens l has this type, written $l \in (C, A)$ iff $l \in C \Longleftrightarrow A$. We use the variable τ to range over total types and $\mathbb{T}$ for sets of total types. We write $(C, A) \subseteq (C', A')$ iff $C \subseteq C'$ and $A \subseteq A'$ and write $(C, A) \cup (C', A')$ for $(C \cup C', A \cup A')$.

3.18 Definition: The increasing chain $\tau_0 \subseteq \tau_1 \subseteq \ldots$ is an *increasing instance* of the sequence $\mathbb{T}_0, \mathbb{T}_1, \ldots$ iff $\tau_i \in \mathbb{T}_i$ for all i.

Note that $\mathbb{T}_0, \mathbb{T}_1, \ldots$ here is an arbitrary sequence of sets of total types—the sequence need not be increasing. This is the trick that makes this proof technique work: we start with a sequence of sets of total types $\mathbb{T}_0, \mathbb{T}_1, \ldots$ that, a priori, have nothing to do with each other; we then show that some continuous function f on lenses gets us from each $\mathbb{T}_i$ to $\mathbb{T}_{i+1}$, in the sense that f takes any lens l that belongs to *all* of the total types in $\mathbb{T}_i$ to a lens $f(l)$ that belongs to all of the total types in $\mathbb{T}_{i+1}$. Finally, we identify an increasing *chain* of particular total types $\tau_0 \subseteq \tau_1 \subseteq \ldots$ whose limit is the total type that we desire to show for the fixed point of f and such that each τ_i belongs to $\mathbb{T}_i$, and hence is a type for $f^i(\bot_l)$.

Here is the generalization of Corollary 3.17(2) to increasing instances of sequences of sets of total types. It will be used in Section 7.

3.19 Lemma: Suppose f is a continuous function from lenses to lenses and $\mathbb{T}_0, \mathbb{T}_1, \ldots$ is a sequence of sets of total types with $\mathbb{T}_0 = \{(\emptyset, \emptyset)\}$. If for all l and i we have $(\forall \tau \in \mathbb{T}_i.\ \ l \in \tau)$ implies $(\forall \tau \in \mathbb{T}_{i+1}.\ \ f(l) \in \tau)$, then for every increasing instance $\tau_0 \subseteq \tau_1 \subseteq \ldots$ of $\mathbb{T}_0, \mathbb{T}_1, \ldots$ we have $\mathit{fix}(f) \in \bigcup_i \tau_i$.

Dealing with Creation

In practice, there will be cases where we need to apply a *putback* function, but where no old concrete view is available, as we saw with `Jo`'s `URL` in Section 2. We deal with these cases by enriching the universe $\mathcal{V}$ of views with a special placeholder Ω, pronounced "missing," which we assume is not already in $\mathcal{V}$. (There are other, formally equivalent, ways of handling missing concrete views. The advantages of this one are discussed in Section 5.) When $S \subseteq \mathcal{V}$, we write S_Ω for $S \cup \{\Omega\}$.

Intuitively, $l \searrow (a, \Omega)$ means "create a *new* concrete view from the information in the abstract view a." By convention, Ω is only used in an interesting way when it is the second argument to the *putback* function: in all of the lenses defined below, we maintain the invariants that (1) $l \nearrow \Omega = \Omega$, (2) $l \searrow (\Omega, c) = \Omega$ for any c, (3) $l \nearrow c \neq \Omega$ for any $c \neq \Omega$, and (4) $l \searrow (a, c) \neq \Omega$ for any $a \neq \Omega$ and any c (including Ω). We write $C \stackrel{\Omega}{\rightleftharpoons} A$ for the set of well-behaved lenses from C_Ω to A_Ω obeying these conventions and $C \stackrel{\Omega}{\Longleftrightarrow} A$ 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. A useful consequence of these conventions is that a lens $l \in C \stackrel{\Omega}{\rightleftharpoons} A$ also has type $C \rightleftharpoons A$.

3.20 Lemma: For any lens l and sets of views C and A: $l \in C \stackrel{\Omega}{\rightleftharpoons} A$ implies $l \in C \rightleftharpoons A$ and (2) $l \in C \stackrel{\Omega}{\Longleftrightarrow} A$ implies $l \in C \Longleftrightarrow A$.

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 (we will usually say just *lenses* from now on), whose definitions are independent of the particular choice of universe $\mathcal{V}$. Each definition is accompanied by a type declaration asserting its well-behavedness under certain conditions—e.g., "the identity lens belongs to $C \stackrel{\Omega}{\rightleftharpoons} C$ for any C".

Many of the lens definitions are parameterized on one or more arguments. These may be of various types: views (e.g., `const`), other lenses (e.g., composition), predicates on views (e.g., the conditional lenses in Section 6), or—in some of the lenses for trees in Section 5—edge labels, predicates on labels, etc.

Electronic Appendix A contains representative proofs that the lenses we define are well behaved (i.e., that the type declaration accompanying its definition is a theorem) and total, and that lenses that take other lenses as parameters are continuous in these parameters and map total lenses to total lenses. Indeed, nearly all of the lenses we define are *very* well behaved (if their lens arguments are), the only exceptions being `map`, `flatten`, `merge`, and conditionals; we do not prove very well behavedness, however, since we are mainly interested just in the well-behaved case.

Identity

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

$$\begin{array}{l} \texttt{id}\nearrow c \;=\; c \\ \texttt{id}\searrow(a,\,c) \;=\; a \\ \hline \forall C{\subseteq}\mathcal{V}. \quad \texttt{id} \in C \overset{\Omega}{\Longleftrightarrow} C \end{array}$$

Having defined `id`, we must prove that it is well behaved and total—i.e., that its type declaration is a theorem. We state the properties explicitly as lemmas and give proofs (in electronic Appendix A) for `id` and a few representative lenses. For the rest, we elide both the statements of the properties, which can be read off from each lens's definition, and the proofs, which are largely calculational.

4.1 Lemma [Well-behavedness]: $\forall C{\subseteq}\mathcal{V}.\ \texttt{id} \in C \overset{\Omega}{\rightleftharpoons} C$

4.2 Lemma [Totality]: $\forall C{\subseteq}\mathcal{V}.\ \texttt{id} \in C \overset{\Omega}{\Longleftrightarrow} C$

For each lens definition, the statements of the totality lemma and well-behavedness lemmas are almost identical, just replacing $\overset{\Omega}{\rightleftharpoons}$ by $\overset{\Omega}{\Longleftrightarrow}$. In the case of `id`, we could just as well combine the two into a single lemma, because every lens with a total type is also well-behaved at that type. However, for lens definitions that are parameterized on other lenses (like composition, just below), the totality of the compound lens depends on the totality (not just well-behavedness) of its argument lenses, while we can establish the well-behavedness of the composite even if the arguments are only well-behaved and not necessarily total. Since we expect this situation will be common in practice—programmers will always want to check that their lenses are well-behaved, since the reasoning involved is simple and local, but may not want to go to the trouble of setting up the more intricate global reasoning needed to prove that their recursive lens definitions are total—we state the two lemmas (i.e., typings) separately.

Composition

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

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

$$\forall A,B,C{\subseteq}\mathcal{V}.\ \forall l \in C \stackrel{\Omega}{\rightleftharpoons} B.\ \forall k \in B \stackrel{\Omega}{\rightleftharpoons} A. \qquad l;k \in C \stackrel{\Omega}{\rightleftharpoons} A$$

$$\forall A,B,C{\subseteq}\mathcal{V}.\ \forall l \in C \stackrel{\Omega}{\Longleftrightarrow} B.\ \forall k \in B \stackrel{\Omega}{\Longleftrightarrow} A. \qquad l;k \in C \stackrel{\Omega}{\Longleftrightarrow} A$$

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

Once again, proofs that the composition operator has the types mentioned above are given in electronic Appendix A.

4.3 Lemma [Well-behavedness]:
$\forall A,B,C{\subseteq}\mathcal{V}.\ \forall l \in C \stackrel{\Omega}{\rightleftharpoons} B.\ \forall k \in B \stackrel{\Omega}{\rightleftharpoons} A. \qquad l;k \in C \stackrel{\Omega}{\rightleftharpoons} A$

4.4 Lemma [Totality]:
$\forall A,B,C{\subseteq}\mathcal{V}.\ \forall l \in C \stackrel{\Omega}{\Longleftrightarrow} B.\ \forall k \in B \stackrel{\Omega}{\Longleftrightarrow} A. \qquad l;k \in C \stackrel{\Omega}{\Longleftrightarrow} A$

Besides well-behavedness and totality, we must also show that lens composition is continuous in its arguments. This will justify using composition in recursive lens definitions: in order for a recursive lens defined as $\mathit{fix}(\lambda l.\ l_1;l_2)$ (where l_1 and l_2 may both mention l) to be well formed, we need to apply Theorem 3.16, which requires that $\lambda l.\ l_1;l_2$ be continuous in l. The following lemma shows that this will be the case whenever l_1 and l_2 are continuous in l.

4.5 Lemma [Continuity]: Let F and G be continuous functions from lenses to lenses. Then the function $\lambda l.\ (F(l);G(l))$ is continuous.

We have proved an analogous lemma for each of our lens combinators that takes other lenses as parameters, so that the continuity of every lens expression will follow from the continuity of its immediate constituents, but we will not bother to state these continuity lemmas explicitly in what follows.

Constant

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

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

$$\forall C{\subseteq}\mathcal{V}.\ \forall v{\in}\mathcal{V}.\ \forall d{\in}C. \qquad \texttt{const}\ v\ d \in C \stackrel{\Omega}{\Longleftrightarrow} \{v\}$$

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

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

5. LENSES FOR TREES

To keep the definitions of our lens primitives as straightforward as possible, we work with an extremely simple form of trees: unordered, edge-labeled trees with no repeated labels among the children of a given node. This model is a natural fit for applications where the data is unordered, such as the keyed address books described in Section 2. Unfortunately, unordered trees do not have all the structure we need for other applications; in particular, we will need to deal with ordered data such as lists and XML documents via an encoding (shown in Section 8). A more direct treatment of ordered trees is a worthwhile topic for future work, but, in the context of the Harmony system, where we are interested in both ordered and unordered data, the choice of a simpler foundation seems to have been a good one: the increase in complexity of lens *programs* that must manipulate ordered data in encoded form is more than made up by the reduction in the complexity of the definitions of lens *primitives* due to the simpler data model.

Notation

From this point on, we choose the universe $\mathcal{V}$ 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. Trees of this form (often extended with labels on internal nodes as well as on children) are sometimes called *deterministic* trees or *feature trees* (e.g., [Niehren and Podelski 1993]). 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 adopt a slightly different perspective: we will consider a tree $t \in \mathcal{T}$ to be a *total* function from $\mathcal{N}$ to $\mathcal{T}_\Omega$ that yields Ω on all but a finite number of names. We write $\mathsf{dom}(t)$ for the domain of t—i.e., the set of the names for which it returns something other than Ω—and $t(n)$ for the subtree associated to name n in t, or Ω if $n \notin \mathsf{dom}(t)$.

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

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

For writing down types,[5] we extend these tree notations to sets of trees. If $T \subseteq \mathcal{T}$ and $n \in \mathcal{N}$, then $\{\!|n \mapsto T|\!\}$ denotes the set of singleton trees $\{\{\!|n \mapsto t|\!\} \mid t \in T\}$. If $T \subseteq \mathcal{T}$ and $N \subseteq \mathcal{N}$, then $\{\!|N \mapsto T|\!\}$ denotes the set of trees $\{t \mid \mathsf{dom}(t) = N \text{ and } \forall n \in N.\ t(n) \in T\}$ and $\left\{\!\middle|N \stackrel{?}{\mapsto} T\right|\!\}$ 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 $\mathsf{doms}(T) = \{\mathsf{dom}(t) \mid t \in T\}$. Note that $\mathsf{doms}(T)$ is a set of sets of names, while $\mathsf{dom}(t)$ is a set of names.

A *value* is a tree of the special form $\{\!|k \mapsto \{\!\}|\!\}$, often written just k. For instance, the phone number $\{\!|\texttt{333-4444} \mapsto \{\!\}|\!\}$ in the example of Section 2 is a value. We write *Val* for the type whose denotation is the set of all values.

Hoisting and Plunging

Let's warm up with some combinators that perform simple structural transformations on trees. The lens `hoist` n is used to shorten a tree by removing an edge at the top. In the *get* direction, it expects a tree that has exactly one child, named n. It returns this child, removing the edge n. In the *putback* direction, the value of the old concrete tree is ignored and a new one is created, with a single edge n pointing to the given abstract tree. (Later we will meet a derived form, `hoist_nonunique`, that works on bushier trees.)

$$\begin{array}{rcl} (\texttt{hoist}\ n) \nearrow c & = & c(n) \\ (\texttt{hoist}\ n) \searrow (a,\, c) & = & \{\!|n \mapsto a|\!\} \\ \hline \multicolumn{3}{l}{\forall C \subseteq \mathcal{T}.\ \forall n \in \mathcal{N}.\quad \texttt{hoist}\ n \in \{\!|n \mapsto C|\!\} \stackrel{\Omega}{\Longleftrightarrow} C} \end{array}$$

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

$$\begin{array}{rcl} (\texttt{plunge}\ n) \nearrow c & = & \{\!|n \mapsto c|\!\} \\ (\texttt{plunge}\ n) \searrow (a,\, c) & = & a(n) \\ \hline \multicolumn{3}{l}{\forall C \subseteq \mathcal{T}.\ \forall n \in \mathcal{N}.\quad \texttt{plunge}\ n \in C \stackrel{\Omega}{\Longleftrightarrow} \{\!|n \mapsto C|\!\}} \end{array}$$

Forking

The lens combinator `xfork` applies different lenses to different parts of a tree. More precisely, 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 children have labels in *pc*;

[5] 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 Section 11) proposing an algebra of types or an algorithm for mechanically checking membership of lens expressions in type expressions.

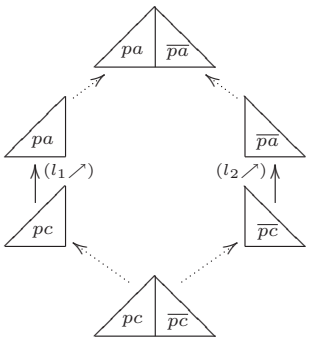


Fig. 1. The *get* direction of `xfork`

dotted arrows represent splitting or concatenating trees. The result of applying $l_1\nearrow$ to $c|_{pc}$ (the tree formed by dropping the immediate children of c whose names are not in pc) must be a tree whose top-level labels are in the set pa; similarly, the result of applying $l_2\nearrow$ to $c\backslash_{pc}$ must be in $\overline{pa}$. That is, the lens l_1 may change the names of immediate children of the tree it is given, but it must map the part of the tree with immediate children belonging to pc to a tree with children belonging to pa. Likewise, l_2 must map the part of the tree with immediate children belonging to $\overline{pc}$ to a tree with children in $\overline{pa}$. Conversely, in the *putback* direction, l_1 must map from pa to pc and l_2 from $\overline{pa}$ to $\overline{pc}$. Here is the full definition:

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

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

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

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

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

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

$$\texttt{fork}\ p\ l_1\ l_2\ =\ \texttt{xfork}\ p\ p\ l_1\ l_2$$

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

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

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

In the *get* direction, this lens takes a concrete tree, keeps the children with names in p (using `id`), and throws away the rest (using `const` $\{\!|\}$ d). The tree d is used when putting an abstract tree back 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 used to define it: `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 $C|_p = C|_p \cdot \{\!|\}$).

Let us see how `filter` behaves in an example. Let the concrete tree $c = \{\!|\texttt{name} \mapsto \texttt{Pat},\ \texttt{phone} \mapsto \texttt{333-4444}|\!\}$, and lens l = `filter {name} {|}`. We calculate $l \nearrow c$, underlining the next term to be simplifed at each step.

$$\begin{array}{ll} & l \nearrow c = \underline{(\texttt{fork}\ \{\texttt{name}\}\ \texttt{id}\ (\texttt{const}\{\!|\}\ d)) \nearrow \{\!|\texttt{name} \mapsto \texttt{Pat},\ \texttt{phone} \mapsto \texttt{333-444}|\!\}} \\ & \qquad \text{by the definition of } l \\ = & \underline{\texttt{id} \nearrow \{\!|\texttt{name} \mapsto \texttt{Pat}|\!\}} \cdot \underline{(\texttt{const}\ \{\!|\}\ d) \nearrow \{\!|\texttt{phone} \mapsto \texttt{333-4444}|\!\}} \\ & \qquad \text{by the definition of } \texttt{fork} \text{ and splitting } c \text{ using } \{\texttt{name}\} \\ = & \{\!|\texttt{name} \mapsto \texttt{Pat}|\!\} \cdot \{\!|\}\ \ = \ \ \{\!|\texttt{name} \mapsto \texttt{Pat}|\!\} = a \\ & \qquad \text{by the definitions of } \texttt{id} \text{ and } \texttt{const} \end{array}$$

Now suppose that we update this tree, a, to $\{\!|\texttt{name} \mapsto \texttt{Patty}|\!\}$. Let us calculate the result of putting back a into c. To save space, we write k for (`const` $\{\!|\}$ $\{\!|\}$).

$$\begin{array}{ll} & l \searrow (a,\, c) \\ = & \underline{(\texttt{fork}\ \{\texttt{name}\}\ \texttt{id}\ k) \searrow (\{\!|\texttt{name} \mapsto \texttt{Pat}|\!\},\ \{\!|\texttt{name} \mapsto \texttt{Pat},\ \texttt{phone} \mapsto \texttt{333-4444}|\!\})} \\ & \qquad \text{by the definition of } l \\ = & \underline{\texttt{id} \searrow (\{\!|\texttt{name} \mapsto \texttt{Patty}|\!\},\ \{\!|\texttt{name} \mapsto \texttt{Pat}|\!\})} \cdot \underline{k \searrow (\{\!|\}, \{\!|\texttt{phone} \mapsto \texttt{333-4444}|\!\})} \\ & \qquad \text{by the definition of } \texttt{fork} \text{ and splitting } a \text{ and } c \text{ using } \{\texttt{name}\} \\ = & \{\!|\texttt{name} \mapsto \texttt{Patty},\ \texttt{phone} \mapsto \texttt{333-4444}|\!\} \\ & \qquad \text{by the definition of } \texttt{id} \text{ and } \texttt{const} \end{array}$$

Note that the *putback* function restores the filtered part of the concrete tree and propagates the change made to the abstract tree. In the case of creation—i.e., if we put back an abstract tree using Ω— then the default argument to `const` is concatenated to the abstract tree to form the result, since there is no filtered part of the concrete tree to restore.

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

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

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

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

$$\frac{\texttt{add}\ n\ t\ =\ \texttt{xfork}\ \{\}\ \{n\}\ (\texttt{const}\ t\ \{\!\{\}\!\};\ \texttt{plunge}\ n)\ \texttt{id}}{\begin{array}{c}\forall n{\in}\mathcal{N}.\ \forall C{\subseteq}\mathcal{T}\backslash_n.\ \forall t \in \mathcal{T}.\\ \texttt{add}\ n\ t \in C \overset{\Omega}{\Longleftrightarrow} \left\{\!\!\left\{n \mapsto \{t\}\right\}\!\!\right\} \cdot C\end{array}}$$

Let us explore the behavior of `add` through an example. Let $c = \left\{\!\!\left\{\texttt{a} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}$ and $l = \texttt{add b}\left\{\!\!\left\{\texttt{x} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}$. To save space, write k for $\texttt{const}\ \left\{\!\!\left\{\texttt{x} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\ \{\!\{\}\!\}$ and p for $\texttt{plunge b}$. We calculate $l \nearrow c$ directly, underlining the term to be simplifed at each step.

$$\begin{aligned}
l \nearrow c &= \underline{(\texttt{xfork}\ \{\}\ \{\texttt{b}\}\ (k;\ p)\ \texttt{id}) \nearrow c}\\
&\quad \text{by the definition of } l\\
&= \underline{(k;\ p) \nearrow \{\!\{\}\!\}} \cdot \underline{\texttt{id} \nearrow \left\{\!\!\left\{\texttt{a} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}}\\
&\quad \text{by the definition of } \texttt{xfork} \text{ and splitting } c \text{ using } \{\}\\
&= p \nearrow (\underline{k \nearrow \{\!\{\}\!\}}) \cdot \left\{\!\!\left\{\texttt{a} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\\
&\quad \text{by the definitions of the composition and } \texttt{id}\\
&= \left(\underline{p \nearrow \left\{\!\!\left\{\texttt{x} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}}\right) \cdot \left\{\!\!\left\{\texttt{a} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\\
&\quad \text{by the definition of } k\\
&= \left\{\!\!\left\{\texttt{a} \mapsto \{\!\{\}\!\},\ \texttt{b} \mapsto \left\{\!\!\left\{\texttt{x} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\right\}\!\!\right\}\\
&\quad \text{by the definition of } p
\end{aligned}$$

Now suppose we modify this tree by renaming the child `a` to `c`, obtaining $a = \left\{\!\!\left\{\texttt{c} \mapsto \{\!\{\}\!\},\ \texttt{b} \mapsto \left\{\!\!\left\{\texttt{x} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\right\}\!\!\right\}$. The result of the *putback* function, $l \searrow (a,\ c)$, is calculated as follows:

$$\begin{aligned}
l \searrow (a,\ c) &= \underline{(\texttt{xfork}\ \{\}\ \{\texttt{b}\}\ (k;\ p)\ \texttt{id}) \searrow (a,\ c)}\\
&\quad \text{by the definition of } l\\
&= \left((k;\ p) \searrow \left(\left\{\!\!\left\{\texttt{b} \mapsto \left\{\!\!\left\{\texttt{x} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\right\}\!\!\right\},\ \{\!\{\}\!\}\right)\right) \cdot \left(\underline{\texttt{id} \searrow (\left\{\!\!\left\{\texttt{c} \mapsto \{\!\{\}\!\}\right\}\!\!\right\},\ \left\{\!\!\left\{\texttt{a} \mapsto \{\!\{\}\!\}\right\}\!\!\right\})}\right)\\
&\quad \text{by the definition of } \texttt{xfork}, \text{ splitting } a \text{ using } \{\texttt{b}\} \text{ and } c \text{ using } \{\}\\
&= \left(\underline{(k;\ p) \searrow \left(\left\{\!\!\left\{\texttt{b} \mapsto \left\{\!\!\left\{\texttt{x} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\right\}\!\!\right\},\ \{\!\{\}\!\}\right)}\right) \cdot \left\{\!\!\left\{\texttt{c} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\\
&\quad \text{by the definition of } \texttt{id}\\
&= \left(k \searrow \left(\underline{p \searrow \left(\left\{\!\!\left\{\texttt{b} \mapsto \left\{\!\!\left\{\texttt{x} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\right\}\!\!\right\},\ k \nearrow \{\!\{\}\!\}\right)},\ \{\!\{\}\!\}\right)\right) \cdot \left\{\!\!\left\{\texttt{c} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\\
&\quad \text{by the definition of composition}\\
&= \left(\underline{k \searrow (\left\{\!\!\left\{\texttt{x} \mapsto \{\!\{\}\!\}\right\}\!\!\right\},\ \{\!\{\}\!\})}\right) \cdot \left\{\!\!\left\{\texttt{c} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\\
&\quad \text{by the definition of } p\\
&= \{\!\{\}\!\} \cdot \left\{\!\!\left\{\texttt{c} \mapsto \{\!\{\}\!\}\right\}\!\!\right\} \quad = \quad \left\{\!\!\left\{\texttt{c} \mapsto \{\!\{\}\!\}\right\}\!\!\right\}\\
&\quad \text{by the definition of } k
\end{aligned}$$

Another derived lens focuses attention on a single child n:

$$\texttt{focus}\ n\ d\ =\ (\texttt{filter}\ \{n\}\ d);\ (\texttt{hoist}\ n)$$

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

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. The type of `focus` follows from the types of the lenses from which it is defined, observing that $\texttt{filter}\ \{n\}\ d \in (C \cdot \{\!|n \mapsto D|\!\}) \Longleftrightarrow^{\Omega} \{\!|n \mapsto D|\!\}$ and that $\texttt{hoist n} \in \{\!|n \mapsto D|\!\} \Longleftrightarrow^{\Omega} D$.

The `hoist` primitive defined earlier requires that the name being hoisted be the unique child of the concrete tree. It is often useful to relax this requirement, hoisting one child out of many. This generalized version of `hoist` is annotated with the set p of possible names of the grandchildren that will become children after the hoist, which must be disjoint from the names of the existing children.

$$\texttt{hoist\_nonunique}\ n\ p\ =\ \texttt{xfork}\ \{n\}\ p\ (\texttt{hoist}\ n)\ \texttt{id}$$

$$\forall n{\in}\mathcal{N}.\ \forall p{\subseteq}\mathcal{N}.\ \forall D{\subseteq}\mathcal{T}\backslash_{\{n\}\cup p}.\ \forall C{\subseteq}\mathcal{T}|_p.\ \texttt{hoist\_nonunique}\ n\ p \in (\{\!|n \mapsto C|\!\} \cdot D) \Longleftrightarrow^{\Omega} (C \cdot D)$$

A last derived lens renames a single child.

$$\texttt{rename}\ m\ n\ =\ \texttt{xfork}\ \{m\}\ \{n\}\ (\texttt{hoist}\ m;\ \texttt{plunge}\ n)\ \texttt{id}$$

$$\forall m, n{\in}\mathcal{N}.\ \forall C{\subseteq}\mathcal{T}.\ \forall D{\subseteq}\mathcal{T}\backslash_{\{m,n\}}.\ \texttt{rename}\ m\ n \in (\{\!|m \mapsto C|\!\} \cdot D) \Longleftrightarrow^{\Omega} (\{\!|n \mapsto C|\!\} \cdot D)$$

In the *get* direction, `rename` splits the concrete tree in two. The first tree has a single child m (which is guaranteed to exist by the type annotation) and is hoisted up, removing the edge named m, and then plunged under n. The rest of the original tree is passed through the `id` lens. Similarly, the *putback* direction splits the abstract view into a tree with a single child `n`, and the rest of the tree. The tree under `n` is put back using the lens ($\texttt{hoist}\ m$; $\texttt{plunge}\ n$), which first removes the edge named n and then plunges the resulting tree under m. Note that the type annotation on `rename` demands that the concrete view have a child named m and that the abstract view have a child named n. In Section 6 we will see how to wrap this lens in a conditional to obtain a lens with a more flexible type.

Mapping

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

The `map` combinator is parameterized on a single lens l. In the *get* direction, `map` applies $l\nearrow$ to each subtree of the root and combines the results together into a

new tree. (Later in this section, we will define a more general combinator, called `wmap`, that can apply a different lens to each subtree. Defining `map` first lightens the notational burden in the explanations of several fine points about the behavior and typing of both combinators.) For example, the lens $\texttt{map}\ l$ has the following behavior in the *get* direction when applied to a tree with three children:

$$\left\{\begin{bmatrix} n_1 \mapsto t_1 \\ n_2 \mapsto t_2 \\ n_3 \mapsto t_3 \end{bmatrix}\right\} \quad \text{becomes} \quad \left\{\begin{bmatrix} n_1 \mapsto l\nearrow t_1 \\ n_2 \mapsto l\nearrow t_2 \\ n_3 \mapsto l\nearrow t_3 \end{bmatrix}\right\}$$

The *putback* 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, c':

$$(\texttt{map}\ l)\searrow \left(\left\{\begin{bmatrix} n_1 \mapsto t_1 \\ n_2 \mapsto t_2 \\ n_3 \mapsto t_3 \end{bmatrix}\right\}, \left\{\begin{bmatrix} n_1 \mapsto t'_1 \\ n_2 \mapsto t'_2 \\ n_3 \mapsto t'_3 \end{bmatrix}\right\}\right) = \left\{\begin{bmatrix} n_1 \mapsto l\searrow(t_1,\, t'_1) \\ n_2 \mapsto l\searrow(t_2,\, t'_2) \\ n_3 \mapsto l\searrow(t_3,\, t'_3) \end{bmatrix}\right\}$$

In general, however, the abstract tree a in the *putback* direction need not have the same domain as c (i.e., the edits that produced the new abstract view may have involved adding and deleting children); the behavior of `map` in this case is a little more involved. Observe, first, that the domain of c' is determined by the domain of the abstract argument to *putback*. Since we aim at building total lenses, we may suppose that $(\texttt{map}\ l)\nearrow((\texttt{map}\ l)\searrow(a,\, c))$ is defined, in which case $\mathsf{dom}((\texttt{map}\ l)\nearrow((\texttt{map}\ l)\searrow(a,\, c))) = \mathsf{dom}(a)$ by rule PUTGET, and $\mathsf{dom}((\texttt{map}\ l)\searrow(a,\, c)) = \mathsf{dom}(a)$ as $(\texttt{map}\ l)\nearrow$ does not change the domain of the tree. This means we can simply drop children that occur in $\mathsf{dom}(c)$ but not in $\mathsf{dom}(a)$. Children bearing names that occur both in $\mathsf{dom}(a)$ and $\mathsf{dom}(c)$ are dealt with as described above. This leaves the children that only appear in $\mathsf{dom}(a)$, which need to be passed through l so that they can be included in c'; to do this, we need some concrete argument to pass to $l\searrow$. There is no corresponding child in c, so instead these abstract trees are put into the missing tree Ω—indeed, this case is precisely why we introduced Ω. Formally, the behavior of `map` is defined as follows. (It relies on the convention that $c(n) = \Omega$ if $n \notin \mathsf{dom}(c)$; the type declaration also involves some new notation, explained below.)

$$(\texttt{map}\ l)\nearrow c \;=\; \{\!|\, n \mapsto l\nearrow c(n) \mid n \in \mathsf{dom}(c) \,|\!\}$$

$$(\texttt{map}\ l)\searrow(a,\, c) \;=\; \{\!|\, n \mapsto l\searrow(a(n),\, c(n)) \mid n \in \mathsf{dom}(a) \,|\!\}$$

$\forall C, A \subseteq \mathcal{T}$ *with* $C = C^{\circlearrowleft}$, $A = A^{\circlearrowleft}$, $\mathsf{doms}(C) = \mathsf{doms}(A)$.
$\forall l \in (\bigcap_{n\in\mathcal{N}} .\ C(n) \stackrel{\Omega}{=\!=} A(n))$.
$\quad \texttt{map}\ l \in C \stackrel{\Omega}{=\!=} A$

$\forall C, A \subseteq \mathcal{T}$ *with* $C = C^{\circlearrowleft}$, $A = A^{\circlearrowleft}$, $\mathsf{doms}(C) = \mathsf{doms}(A)$.
$\forall l \in (\bigcap_{n\in\mathcal{N}} .\ C(n) \stackrel{\Omega}{\Longleftrightarrow} A(n))$.
$\quad \texttt{map}\ l \in C \stackrel{\Omega}{\Longleftrightarrow} A$

Because of the way that it takes the tree apart, transforms the pieces, and reassembles them, the typing of `map` is a little subtle. For example, in the *get* direction, `map`

does not modify the names of the immediate children of the concrete tree, and in the *putback* direction, the names of the abstract tree are left unchanged; we might therefore expect a simple typing rule stating that, if $l \in (\bigcap_{n\in\mathcal{N}} C(n) \overset{\Omega}{\rightleftharpoons} A(n))$—i.e., if l is a well-behaved lens from the concrete subtree type $C(n)$ to the abstract subtree type $A(n)$ for each child n—then $\mathtt{map}\ l \in C \overset{\Omega}{\rightleftharpoons} A$. Unfortunately, for arbitrary C and A, the `map` lens is not guaranteed to be well-behaved at this type. In particular, if $\mathsf{doms}(C)$, the set of domains of trees in C, is not equal to $\mathsf{doms}(A)$, then the *putback* function can produce a tree that is not in C, as the following example shows. Consider the sets of trees

$$C = \{\, \{\!|\, \mathtt{x} \mapsto \mathtt{m} \,|\!\}\,,\ \{\!|\, \mathtt{y} \mapsto \mathtt{n} \,|\!\}\,\} \qquad A = C \cup \{\, \{\!|\, \mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{n} \,|\!\}\,\}$$

and observe that with trees

$$a = \{\!|\, \mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{n} \,|\!\} \qquad c = \{\!|\, \mathtt{x} \mapsto \mathtt{m} \,|\!\}$$

we have $\mathtt{map}\ \mathtt{id} \searrow (a,\, c) = a$, a tree that is not in C. This shows that the type of `map` must include the requirement that $\mathsf{doms}(C) = \mathsf{doms}(A)$. (Recall that, for any type T, the set $\mathsf{doms}(T)$ 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 $C(m) \overset{\Omega}{\rightleftharpoons} A(m)$ for each name m, then $\mathtt{map}\ l$ 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 $C(\mathtt{y}) = A(\mathtt{y}) = \{\{\!|\mathtt{p}|\!\}, \{\!|\mathtt{q}|\!\}\}$. The lens `id` has type $C(\mathtt{x}) \overset{\Omega}{\rightleftharpoons} A(\mathtt{x})$ and $C(\mathtt{y}) \overset{\Omega}{\rightleftharpoons} A(\mathtt{y})$ (and $C(z) = \emptyset \overset{\Omega}{\rightleftharpoons} \emptyset = A(z)$ for all other names z). But it is clearly not the case that $\mathtt{map}\ \mathtt{id} \in C \overset{\Omega}{\rightleftharpoons} A$.

To avoid this error, but still give a type for `map` that is precise enough to derive interesting types for lenses defined in terms of `map`, we require that the source and target sets in the type of `map` be closed under the "shuffling" of their children. Formally, if T is a set of trees, then the set of *shufflings* of T, denoted $T^{\circlearrowleft}$, is

$$T^{\circlearrowleft} = \bigcup_{D \in \mathsf{doms}(T)} \{\!|\, n \mapsto T(n) \mid n \in D \,|\!\}$$

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 $T = T^{\circlearrowleft}$. In the example above, $A^{\circlearrowleft} = C^{\circlearrowleft} = C$—i.e., C is shuffle closed, but A is not.

Alternatively, every shuffle-closed set T can be identified with a set of set of names D and a function f from names to types, such that $t \in T$ iff $\mathsf{dom}(t) \in D$ and $t(n) \in f(n)$ for every name $n \in \mathsf{dom}(t)$. Formally, the shuffle closed set T is defined as follows:

$$T = \bigcup_{d \in D} \{\!|\, n \mapsto f(n) \mid n \in d \,|\!\}$$

In the situations where `map` is used, shuffle closure is typically easy to check. For example, the restriction on tree grammars embodied by W3C Schema implies shuffle closure (informally, the restriction on W3C Schema is analogous to imposing shuffle closure on the schemas along every path, not just at the root). Additionally, any set of trees whose elements each have singleton domains is shuffle closed. Also, for every set of trees T, the encoding introduced in Section 7 of lists with elements in T is shuffle closed, which justifies using `map` (with recursion) to implement operations on lists. Furthermore, types of the form $\{\!|n \mapsto T \mid n \in \mathcal{N}|\!\}$ with infinite domain but with the same structure under each edge, which are heavily used in database examples (where the top-level names are keys and the structures under them are records) are shuffle closed.

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

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

whereas

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

Intuitively, there is a difference between, on the one hand, modifying a child n and, on the other, removing it and then adding it back: in the first case, any information in the concrete view that is "projected away" in the abstract view will be carried along to the new concrete view; in the second, such information will be replaced with default values. This difference seems pragmatically reasonable, so we prefer to keep `map` and lose PutPut.[6]

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

[6] Alternatively, we could use a refinement of the type system to track when PutPut does hold, annotating some of the lens combinators with extra type information recording the fact that they are oblivious, and then give `map` two types: the one we gave here plus another saying "when `map` is applied to an oblivious lens, the result is very well behaved."

information. This led us to the present design, where only the `const` lens (along with other lenses defined from it, such as `focus`) expects a default tree d. This approach is much more convenient to program with than the others we tried, since one only provides defaults at the exact points where information is discarded.

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

$$(\texttt{wmap}\ m)\nearrow c\ =\ \{\!|n \mapsto m(n)\nearrow c(n) \mid n \in \mathsf{dom}(c)|\!\}$$

$$(\texttt{wmap}\ m)\searrow(a,\, c)\ =\ \{\!|n \mapsto m(n)\searrow(a(n),\, c(n)) \mid n \in \mathsf{dom}(a)|\!\}$$

$\forall C, A \subseteq \mathcal{T}$ *with* $C = C^{\circlearrowleft}$, $A = A^{\circlearrowleft}$, $\mathsf{doms}(C) = \mathsf{doms}(A)$.
$\forall m \in (\Pi n \in \mathcal{N}.\ C(n) \overset{\Omega}{\rightleftharpoons} A(n))$.
$\texttt{wmap}\ m \in C \overset{\Omega}{\rightleftharpoons} A$

$\forall C, A \subseteq \mathcal{T}$ *with* $C = C^{\circlearrowleft}$, $A = A^{\circlearrowleft}$, $\mathsf{doms}(C) = \mathsf{doms}(A)$.
$\forall m \in (\Pi n \in \mathcal{N}.\ C(n) \overset{\Omega}{\Longleftrightarrow} A(n))$.
$\texttt{wmap}\ m \in C \overset{\Omega}{\Longleftrightarrow} A$

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

Since the typing of `wmap` is rather subtle, it is worth stating its well-behavedness lemma explicitly (and, in the appendix, giving the proof).

5.1 Lemma [Well-behavedness]:
$\forall C, A \subseteq \mathcal{T}$ *with* $C = C^{\circlearrowleft}$, $A = A^{\circlearrowleft}$, $\mathsf{doms}(C) = \mathsf{doms}(A)$.
$\forall m \in (\Pi n \in \mathcal{N}.\ C(n) \overset{\Omega}{\rightleftharpoons} A(n))$.
$\texttt{wmap}\ m \in C \overset{\Omega}{\rightleftharpoons} A$

Copying and Merging

We next consider two lenses that duplicate information in one direction and re-integrate (by performing equality checks) in the other.

A view of some underlying data structure may sometimes require that two distinct subtrees maintain a relationship, such as equality. For example, under the subtree representing a manager, Alice, an employee-manager database may list the name and ID number of every employee in Alice's group. If Bob is managed by Alice, then Bob's employee record will also list his name and ID number (as well as other information including a pointer to Alice, as his manager). If Bob's name changes at a later date, then we expect that it will be updated (identically) under both his record and Alice's record. If the concrete representation contains his name in only a single location, we need to duplicate the information in the *get* direction. To do this we need a lens that copies a subtree and then allows us to transform the copy into the shape that we want.

In the *get* direction, (`copy` m n) takes a tree, c, that has no child labeled n. If $c(m)$ exists, then (`copy` m n) duplicates $c(m)$ by setting both $a(m)$ and $a(n)$ equal to $c(m)$. In the *putback* direction, `copy` simply discards $a(n)$. The type of `copy` ensures that no information is lost, because $a(m) = a(n)$.

$$
\begin{array}{l}
\quad (\texttt{copy}\ m\ n) \nearrow c \ = \ c \cdot \{\!|n \mapsto c(m)|\!\} \\
(\texttt{copy}\ m\ n) \searrow (a,\, c) \ = \ a\backslash_n \\
\hline
\forall m, n \in \mathcal{N}.\ \forall C \subseteq \mathcal{T}\backslash_{\{m,n\}}.\ \forall D \subseteq \mathcal{T}. \\
\quad \texttt{copy}\ m\ n \in (C \cdot \{\!|m \mapsto D_\Omega|\!\}) \overset{\Omega}{\Longleftrightarrow} (C \cdot \{\, \{\!|m \mapsto d,\ n \mapsto d|\!\} \mid d \in D_\Omega \})
\end{array}
$$

Because we want `copy` to be a total lens, the equality constraint in the abstract type of `copy` is essential to ensure well-behavedness. To see why, consider what would happen if the *putback* function were defined even when $a(m)$ and $a(n)$ were not equal and `copy`$\searrow$ removed either $a(m)$ or $a(n)$. Then there would be no way for a subequent application of the *get* function to restore the discarded information. Consequently, PutGet would be violated.

Unfortunately, because of this constraint, the set of lenses that can be validly composed to the right of a `copy` is also restricted—the composed lenses must respect the equality. As an example of what can go wrong, consider (`copy a b`; `prune b` $\{\!|\}\!|$) and suppose that we want to assign it a lens typing with concrete component $\{\!|\texttt{a} \mapsto D|\!\}$. A simple calculation shows that *get* function behaves like `id`: the lens first copies `a` to `b` and then prunes away `b`. We run into problems, however, if we evaluate (`copy a b`; `prune b` $\{\!|\}\!|$) $\searrow \left(\{\!|\texttt{a} \mapsto d_1|\!\}, \{\!|\texttt{a} \mapsto d_2|\!\}\right)$ with $d_1 \neq d_2$. Unwinding the composition, we evaluate (`copy a b`)$\searrow$ with an abstract argument $\{\!|\texttt{a} \mapsto d_1,\ \texttt{b} \mapsto d_2|\!\}$. As argued above, the `copy` lens cannot be both defined and well-behaved on such an abstract argument because the copied data is not identical. As the example demonstrates, the lenses composed after a `copy` must preserve the equality of the copied data. Otherwise we cannot ensure that the type requirement $a(m) = a(n)$ will be satisfied.

In our intended application, using lenses to build synchronizers for tree-structured data, we have not found a need for `copy`. This is not surprising, because if a concrete representation demands that some invariant hold within the data structure, we assume that (1) each application will locally maintain the invariants in its own representation, and (2) the function of a synchronizer is to simply propagate changes from one well-formed replica to another. Moreover, if one field in a concrete representation is derivable from another (or a set of other fields), then we need not expose *both* fields in the abstract view. Instead, we can *merge* the fields (see below). Any change to the merged field will be pushed back down to all the derived fields in the concrete view. Thus, `merge`, the inverse of `copy` makes more sense for the views manipulated by a data synchronizer.

By contrast, some have argued for the need for *more powerful* forms of `copy` in settings such as editing a user-friendly view of a structured document [Hu et al. 2004; Mu et al. 2004a]. Consider a situation where a user edits a view of a document in which a table of contents is automatically generated from the section headings appearing in the source text (i.e., the concrete view is just some structured text, while the abstract view contains the text plus the table of contents). One might

feel that adding a new section to the text in the abstract view should cause an entry to be added to the table of contents, and similarly that adding an entry to the table of contents should create an empty section in the text. Such functionality is not consistent with our PutGet law: both adding a section heading and adding an entry in the table of contents will result in the same concrete document after a *putback*; such a *putback* function is not injective and cannot participate in a lens in our sense. However, in contexts where this kind of behavior is a primary goal, system designers may be willing to weaken the promises they make to programmers by guaranteeing weaker properties than PutGet. For example, Mu et al [2004a] only require their bidirectional transformations to obey a PutGetPut law. PutGetPut is weaker than PutGet in two ways. First, it does not require that $l\nearrow(l\searrow(a,c))$ equals a. Rather, it requires that, if $c' = l\searrow(a,c)$ and $a' = l\nearrow(c')$, then a' should "contain the same information as a," in the sense that $l\searrow(a',c') = c'$. Second, PutGetPut allows *get* to be undefined over parts of the range of *putback*—PutGetPut is only required to hold when the *get* is defined, but no requirements are made on how broadly *get* must be defined. (Given that their setting is interactive, it is reasonable to say, as they do, that if *get* after some *putback* is undefined, then the system can signal the user that the modification to a was illegal and cancel it). Hu et al [2004] go a step further and weaken *both* PutGet and GetPut by only requiring PutGet when a is $l\nearrow(c)$ and by only requiring GetPut when c is $l\searrow(a,c')$ for some a and c'.

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

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

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

Unlike `copy`, `merge` turns out to be quite useful in our synchronization framework. For example, our bookmark synchronizer must deal with the fact that the XML representation of Apple Safari bookmark files includes the URL data for every link

twice. By merging the appropriate children, we record this dependency and ensure that updates to the URL fields are consistently propagated to both locations.

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 tree(s) through one or the other of its sub-lenses.

The requirement that makes conditionals tricky is totality: we want to be able to take a concrete view, put it through a conditional lens to obtain some abstract view, and then take *any* other abstract view of suitable type and push it back down. But this will only work if either (1) we somehow ensure that the abstract view is guaranteed to be sent to the same sub-lens on the way down as we took on the way up, or else (2) the two sub-lenses are constrained to behave coherently. Since we want reasoning about well-behavedness and totality to be compositional in the absence of recursion (i.e., we want the well-behavedness and totality of composite lenses to follow just from the well-behavedness and totality of their sub-lenses, not from special facts about the behavior of the sub-lenses), the second is unacceptable.

Interestingly, once we adopt the first approach, we can give a *complete* characterization of all possible conditional lenses: we argue 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.

Concrete Conditional

Our first conditional, `ccond`, is parameterized on a predicate C_1 on views and two lenses, l_1 and l_2. In the *get* direction, it tests the concrete view c and applies the *get* of l_1 if c satisfies the predicate and l_2 otherwise. In the *putback* direction, `ccond` again examines the concrete view, and applies the *putback* of l_1 if it satisfies the predicate and the *putback* of l_2 otherwise. This is arguably the simplest possible way to define a conditional: it fixes all of its decisions in the *get* direction, so the only constraint on l_1 and l_2 is that they have the same target. (Since we are interested in using `ccond` to define total lenses, this condition can actually be rather hard to achieve in practice.)

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

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

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

$$\forall C, C_1, A{\subseteq}\mathcal{V}.\ \forall l_1 \in C{\cap}C_1 \overset{\Omega}{\Longleftrightarrow} A.\ \forall l_2 \in C{\setminus}C_1 \overset{\Omega}{\Longleftrightarrow} A.\ \texttt{ccond}\ C_1\ l_1\ l_2 \in C \overset{\Omega}{\Longleftrightarrow} A$$

One subtlety in the definition is worth noting: we arbitrarily choose to *putback* Ω using l_2 (because $\Omega \notin C_1$ for any $C_1 \subseteq \mathcal{V}$). We could equally well arrange the

definition so as to send Ω through l_1. In fact, l_1 need not be well-behaved (or even defined) on Ω; we can construct a well-behaved, total lens using `ccond` when $l_1 \in C \cap C_1 \Longleftrightarrow A$ and $l_2 \in C \setminus C_1 \overset{\Omega}{\Longleftrightarrow} A$.

Abstract Conditional

A quite different way of defining a conditional lens is to make it ignore its *concrete* argument in the *putback* direction, basing its decision whether to use $l_1\searrow$ or $l_2\searrow$ entirely on its abstract argument. This obliviousness to the concrete argument removes the need for any side conditions relating the behavior of l_1 and l_2—everything works fine if we *putback* using the opposite lens from the one that we used to *get*—as long as, when we *immediately* put the result of *get*, we use the same lens that we used for the *get*. Requiring that the sources and targets of l_1 and l_2 be disjoint guarantees this.

$$(\texttt{acond}\ C_1\ A_1\ l_1\ l_2)\nearrow c\ =\ \begin{cases} l_1\nearrow c & \text{if } c \in C_1 \\ l_2\nearrow c & \text{if } c \notin C_1 \end{cases}$$

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

$$\forall C, A, C_1, A_1 \subseteq \mathcal{V}.\ \forall l_1 \in C \cap C_1 \overset{\Omega}{\rightleftharpoons} A \cap A_1.\ \forall l_2 \in (C \setminus C_1) \overset{\Omega}{\rightleftharpoons} (A \setminus A_1).\ \texttt{acond}\ C_1\ A_1\ l_1\ l_2 \in C \overset{\Omega}{\rightleftharpoons} A$$

$$\forall C, A, C_1, A_1 \subseteq \mathcal{V}.\ \forall l_1 \in C \cap C_1 \overset{\Omega}{\Longleftrightarrow} A \cap A_1.\ \forall l_2 \in (C \setminus C_1) \overset{\Omega}{\Longleftrightarrow} (A \setminus A_1).\ \texttt{acond}\ C_1\ A_1\ l_1\ l_2 \in C \overset{\Omega}{\Longleftrightarrow} A$$

In Section 5, we defined the lens `rename` m n, whose type demands that each concrete tree have a child named m and that every abstract tree have a child named n. Using this conditional, we can write a more permissive lens that renames a child if it is present and otherwise behaves like the identity.

$$\texttt{rename\_if\_present}\ m\ n\ =\ \texttt{acond}\ (\{\!|m \mapsto \mathcal{T}|\!\} \cdot \mathcal{T}\backslash_{\{m,n\}})\ (\{\!|n \mapsto \mathcal{T}|\!\} \cdot \mathcal{T}\backslash_{\{m,n\}})\ (\texttt{rename}\ m\ n)\ \texttt{id}$$

$$\forall n, m \in \mathcal{N}.\ \forall C \subseteq \mathcal{T}.\ \forall D, E \subseteq (\mathcal{T}\backslash_{\{m,n\}}).\ \texttt{rename\_if\_present}\ m\ n \in (\{\!|m \mapsto C|\!\} \cdot D) \cup E \overset{\Omega}{\Longleftrightarrow} (\{\!|n \mapsto C|\!\} \cdot D) \cup E$$

General Conditional

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. Here, we let them overlap arbitrarily, behaving like `ccond` in the region where they do overlap (i.e., for arguments (a, c) to *putback* where a is in the intersection of the targets) and like `acond` in the regions where the abstract argument to *putback* belongs to just one of the targets. To this we can add one additional observation: that the use of Ω in the definition of `acond` is actually arbitrary. All that is required is that, when we use the *putback* of l_1, the concrete argument should come

from $(C_1)_\Omega$, so that l_1 is guaranteed to do something reasonable with it. These considerations lead us to the following definition.

$$(\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2)\nearrow c = \begin{cases} l_1\nearrow c & \text{if } c \in C_1 \\ l_2\nearrow c & \text{if } c \notin C_1 \end{cases}$$

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

$$\forall C, C_1, A_1, A_2 \subseteq \mathcal{V}.\ \forall l_1 \in (C \cap C_1) \xrightleftharpoons{\Omega} A_1.\ \forall l_2 \in (C \backslash C_1) \xrightleftharpoons{\Omega} A_2.$$
$$\forall f_{21} \in (C \backslash C_1) \to (C \cap C_1)_\Omega.\ \forall f_{12} \in (C \cap C_1) \to (C \backslash C_1)_\Omega.$$
$$\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2 \in C \xrightleftharpoons{\Omega} (A_1 \cup A_2)$$

$$\forall C, C_1, A_1, A_2 \subseteq \mathcal{V}.\ \forall l_1 \in (C \cap C_1) \xLeftrightarrow{\Omega} A_1.\ \forall l_2 \in (C \backslash C_1) \xLeftrightarrow{\Omega} A_2.$$
$$\forall f_{21} \in (C \backslash C_1) \to (C \cap C_1)_\Omega.\ \forall f_{12} \in (C \cap C_1) \to (C \backslash C_1)_\Omega.$$
$$\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2 \in C \xLeftrightarrow{\Omega} (A_1 \cup A_2)$$

When a is in the targets of both l_1 and l_2, $\texttt{cond}\searrow$ chooses between them based solely on c (as does `ccond`, whose targets always overlap). If a lies in the range of only l_1 or l_2, then `cond`'s choice of lens for *putback* is predetermined (as with `acond`, whose targets are disjoint). Once $l\searrow$ is chosen to be either $l_1\searrow$ or $l_2\searrow$, if the old value of c is not in $\mathsf{ran}(l\searrow)_\Omega$, 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 in general we can compute a more interesting value, as we will see in the `list_filter` lens, defined in Section 7.

We argued above that `cond` captures all the power of `ccond` and `acond`—indeed, because of the fixup functions f_{12} and f_{21}, it captures even more. We now argue, informally, that this is the maximum generality possible—i.e., that any well-behaved and total lens combinator that behaves like a binary conditional can be obtained as a special case of `cond`. Of course, the argument hinges on what we mean when we say "l behaves like a conditional." We would like to capture the intuition that l should, in each direction, "test its input(s) and decide whether to behave like l_1 or l_2." In the *get* direction, there is little choice about how to say this: since there is just one argument, the test just amounts to testing membership in a set (predicate) C_1. In the *putback* direction, there is some apparent flexibility, since the test might investigate both arguments. However, the requirements of well-behavedness (and the feeling that a conditional lens should be "parametric" in l_1 and l_2, in the sense that the choice between l_1 and l_2 should not be made by investigating their behavior) actually eliminate most of this flexibility. If, for example, the abstract input a falls in $a \in A_1 \cap A_2$, then the choice of whether to apply $l_1\searrow$ or $l_2\searrow$ is fully determined by c: if $c \in C_1$, then it may be that $a = l_1\nearrow c$; in this case, using $l_1\searrow$ guarantees that $l\searrow(a,\ c) = c$, as required by GetPut, whereas $l_2\searrow$ gives us no such guarantee; conversely, if $c \in C \backslash C_1$, we must use l_2.

Similarly, if $a \in A_1 \backslash A_2$, then we have no choice but to use l_1, since l_2's type does not promise that applying it to an argument of this type will yield a result in C_1.

Similarly, if $a \in A_2 \backslash A_1$, then we must use l_2. However, here we do have a little genuine freedom: if $a \in A_1 \backslash A_2$ while $c \in C \backslash C_1$, then, by the type of l_2, there is no danger that $a = l_2 \nearrow c$. In order to apply l_1, we need *some* element of $(C_1)_\Omega$ to use as the concrete argument, but it does not matter which one we pick; and conversely for l_2. The fixup functions f_{21} and f_{12} cover all possible (deterministic) ways of making this choice based on the given c. It is possible to be slightly more general by making f_{21} and f_{12} take both a and c as arguments, but pragmatically there seems little point in doing this, since either $l_1 \searrow$ or $l_2 \searrow$ is going to be called on their result, and these functions can just as well take a into account.

7. DERIVED LENSES FOR LISTS

XML and many other concrete data formats make heavy use of ordered lists. We describe in this section how we can represent lists as trees, using a standard cons-cell encoding, and introduce some derived lenses to manipulate them. We begin with very simple lenses for projecting the head and tail of a list. We then define recursive lenses implementing some more complex operations on lists: mapping, reversal, grouping, concatenating, and filtering. We give the proofs of the well-behavedness and totality lemmas (in Appendix A) for these recursive lenses to demonstrate how the reasoning principles developed in Section 3 can be applied to practical examples.

Encoding

7.1 Definition: A tree t is said to be a *list* iff either it is empty or it has exactly two children, one named `*h` and another named `*t`, and $t($`*t`$)$ is also a list. We use the lighter notation $[t_1 \ldots t_n]$ for the tree

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

In types, we write `[]` for the set $\{\{\!|\}\!\}$ containing only the empty list, $C :: D$ for the set $\left\{\!\left|\texttt{*h} \mapsto C,\ \texttt{*t} \mapsto D\right|\!\right\}$ 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 the set of lists of Ds whose length is at least i and at most j, and writing $[D^i]$ for the set of lists whose length is exactly i (i.e., $[D^{i..i}]$). Given two list values, l_1 and l_2, the set of lists denoted by the *interleaving* $l_1 \& l_2$ consists of all the lists formed by interleaving the elements of l_1 with the elements of l_2 in an arbitrary fashion. For example, `[a, b]&[c]` is the set {`[a, b, c]`, `[a, c, b]`, `[c, a, b]`}. We lift the interleaving operator to list types in the obvious way: the interleaving of two list types, $[B]$ and $[C]$, is the union of all the interleavings of lists belonging to $[B]$ with lists belonging to $[C]$. Similarly, we lift the usual append operator, written `++`, to list types: $[C]$`++`$[D]$ denotes the set of lists obtained by appending any element of $[C]$ to any element of $[D]$.

Head and Tail Projections

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

$$\frac{\texttt{hd}\ d\ =\ \texttt{focus *h}\ \left\{\texttt{*t} \mapsto d\right\}}{\forall C, D{\subseteq}\mathcal{T}.\ \forall d{\in}D. \quad \texttt{hd}\ d \in (C :: D) \overset{\Omega}{\Longleftrightarrow} C}$$

$$\frac{\texttt{tl}\ d\ =\ \texttt{focus *t}\ \left\{\texttt{*h} \mapsto d\right\}}{\forall C, D{\subseteq}\mathcal{T}.\ \forall d{\in}C. \quad \texttt{tl}\ d \in (C :: D) \overset{\Omega}{\Longleftrightarrow} D}$$

The lens `hd` expects a default tree, which it uses in the *putback* direction as the tail of the created tree when the concrete tree is missing; in the *get* direction, it returns the tree under `*h`. The lens `tl` works analogously. Note that the types of these lenses apply to both homogeneous lists (the type of `hd` implies $\forall C{\subseteq}\mathcal{T}.\ \forall d{\in}\texttt{[}C\texttt{]}.\ \texttt{hd}\ d \in \texttt{[}C\texttt{]} \overset{\Omega}{\Longleftrightarrow} C$) as well as cons cells whose head and tail have unrelated types; both possibilities are used in the type of the `bookmark` lens in Section 8. The types of `hd` and `tl` follow from the type of `focus`.

List Map

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

$$\frac{\texttt{list\_map}\ l\ =\ \texttt{wmap}\ \{\texttt{*h} \mapsto l,\ \texttt{*t} \mapsto \texttt{list\_map}\ l\}}{\begin{array}{l}\forall C, A{\subseteq}\mathcal{T}.\ \forall l \in C \overset{\Omega}{=\!=} A. \quad \texttt{list\_map}\ l \in \texttt{[}C\texttt{]} \overset{\Omega}{=\!=} \texttt{[}A\texttt{]} \\ \forall C, A{\subseteq}\mathcal{T}.\ \forall l \in C \overset{\Omega}{\Longleftrightarrow} A. \quad \texttt{list\_map}\ l \in \texttt{[}C\texttt{]} \overset{\Omega}{\Longleftrightarrow} \texttt{[}A\texttt{]}\end{array}}$$

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

Since `list_map` is our first recursive lens, it is worth noting how recursive calls are made in each direction. The *get* function of 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 *putback* direction, `wmap` applies l to the head of the abstract tree and either the head of the concrete tree (if it is present) or Ω, and it applies `list_map` l to the tail of the abstract tree and the tail of the concrete tree (if it is present) or Ω. In both directions, the recursive calls continue until the entire tree—concrete (for the *get*) or abstract (for the *putback*)—has been traversed. (The recursion is controlled by the abstract argument in the *putback* direction because the `map` combinator uses the children of the abstract tree to determine how many times to call its argument lens.)

Because `list_map` is defined recursively, proving it is well behaved requires just a little more work than than for non-recursive derived lenses: we need to show that it has a particular type *assuming* that the recursive use of `list_map` has the same type. This is no surprise: exactly the same reasoning process is used in typing recursive functional programs.

Recall that the type of `wmap` requires that both sets of trees in its type be shuffle closed. To prove that `list_map` is well-behaved and total, we will need a lemma showing that cons-cell and list types are shuffle closed.

7.2 Lemma: Let $S, T \subseteq \mathcal{T}$. Then

(1) $(S :: T) = (S :: T)^{\circlearrowleft}$
(2) $\texttt{[}T\texttt{]} = \texttt{[}T\texttt{]}^{\circlearrowleft}$.

With these pieces in hand, the well-behavedness lemma follows by a straightforward calculation using the type of `wmap`.

7.3 Lemma [Well-behavedness]:
$\forall C, A \subseteq \mathcal{T}.\ \forall l \in C \overset{\Omega}{\rightleftharpoons} A. \quad \texttt{list\_map}\ l \in \texttt{[}C\texttt{]} \overset{\Omega}{\rightleftharpoons} \texttt{[}A\texttt{]}$

The proof of totality for `list_map` is more interesting. We use Corollary 3.17(2), which requires that we (1) identify two chains of types, $\emptyset = C_0 \subseteq C_1 \subseteq \ldots$ and $\emptyset = A_0 \subseteq A_1 \subseteq \ldots$, and (2) from $k \in C_i \overset{\Omega}{\Longleftrightarrow} A_i$, prove that $f(k) \in C_{i+1} \overset{\Omega}{\Longleftrightarrow} A_{i+1}$ for all i. We can then conclude that $\mathit{fix}(f) \in \bigcup_i C_i \overset{\Omega}{\Longleftrightarrow} \bigcup_i A_i$. For `list_map` we choose increasing chains of types as follows:

$$C_i = \emptyset \subseteq \texttt{[]} \subseteq C :: \texttt{[]} \subseteq C :: C :: \texttt{[]} \subseteq \ldots$$
$$A_i = \emptyset \subseteq \texttt{[]} \subseteq A :: \texttt{[]} \subseteq A :: A :: \texttt{[]} \subseteq \ldots$$

The full argument is given in the proof of Lemma 7.4 in Appendix A.

7.4 Lemma [Totality]: $\forall C, A \subseteq \mathcal{T}.\ \forall l \in C \overset{\Omega}{\Longleftrightarrow} A.\ \texttt{list\_map}\ l \in \texttt{[}C\texttt{]} \overset{\Omega}{\Longleftrightarrow} \texttt{[}A\texttt{]}$

Reverse

Our next lens reverses the elements of a list. The algorithm we use to implement list reversal runs in quadratic time—we reverse the tail of the list and then use an auxiliary lens to rotate the head to the end of the reversed tail. Before presenting the `list_reverse` lens, we describe this auxiliary lens, called `rotate`.

```
rotate = acond ([] ∪ (D :: [])) ([] ∪ (D :: []))
           id
           (rename *h tmp;
            hoist_nonunique *t {*h, *t};
            fork {*h} id (rename tmp *h; rotate; plunge *t))
```

$$\forall D \subseteq \mathcal{T}. \quad \texttt{rotate} \in \texttt{[}D\texttt{]} \overset{\Omega}{\Longleftrightarrow} \texttt{[}D\texttt{]}$$

In the *get* direction, `rotate` has two cases. If the list is empty or a singleton, the `acond` applies `id`, which returns the original empty or singleton list unmodified. Otherwise, it (1) renames the head to `tmp`; (2) hoists up the tail, which yields children `*h` and `*t` since the list is neither empty nor a singleton; and (3) splits the tree in two using `fork`, applying the `id` lens to the part of the tree consisting of the single child `*h` (i.e., the second element in the original list), and puts the `tmp` element at the end of the list. To do this, it first renames `tmp` back to `*h`, yielding a list whose head is the head of the original list and whose tail is the tail of the tail of the original list. The recursive call to `rotate` puts the head of this list to the end of the list, yielding the original list with two differences: the first element is at

the end and the second element not present. Finally, the resulting list is plunged under `*t`, and (after the `fork`) the result is concatenated with the original second element.

The *putback* direction also has two cases, corresponding to the two arms of the `acond` lens. It first checks whether the abstract view is the empty list or a singleton list. If so, then it applies the `id` lens, which returns the abstract list unchanged. Otherwise, it applies the three steps given above in reverse order: it first splits the abstract and concrete lists as in the *get* direction, passing the head through the `id` lens and partially rotating the tail. To do this, it hoists the tail tag, recursively applies rotate (bringing the last element to the head of this list), and renames `*h` to `tmp`. The result after the fork is the original list (under the names `*h` and `*t`) without its original last element concatenated with the last element under the name `tmp`. Next the lens `hoist_nonunique` plunges the `*h` and `*t` children under `*t`. Finally, `tmp` is renamed back to `*h`. This has the effect of bringing the last element of the abstract list to the head of the result and shifting the position of every other element by one.

The well-behavedness proof is a simple calculation, using Corollary 3.17(1) and the types of the lenses that make up `rotate`.

7.5 Lemma [Well-behavedness]: $\forall D{\subseteq}\mathcal{T}.\ \texttt{rotate} \in \texttt{[}D\texttt{]} \stackrel{\Omega}{\rightleftharpoons} \texttt{[}D\texttt{]}$

The totality lemma is proved using Corollary 3.17(2), after establishing, by induction on i, that $\texttt{rotate} \in \texttt{[}D^i\texttt{]} \stackrel{\Omega}{\Longleftrightarrow} \texttt{[}D^i\texttt{]}$.

7.6 Lemma [Totality]: $\forall D{\subseteq}\mathcal{T}.\ \texttt{rotate} \in \texttt{[}D\texttt{]} \stackrel{\Omega}{\Longleftrightarrow} \texttt{[}D\texttt{]}$

Using `rotate`, the definition of `list_reverse` is straightforward:

$$\frac{\texttt{list\_reverse} \;=\; \texttt{wmap}\ \{\texttt{*t} \mapsto \texttt{list\_reverse}\};\ \texttt{rotate}}{\forall D{\subseteq}\mathcal{T}.\quad \texttt{list\_reverse} \in \texttt{[}D\texttt{]} \stackrel{\Omega}{\Longleftrightarrow} \texttt{[}D\texttt{]}}$$

In the *get* direction, we simply reverse the tail and rotate the head element to the end of the list. In the *putback* direction, we perform these steps in reverse order, first rotating the last element of the list to the head and then reversing the tail. Note also that `list_reverse` behaves like the identity when it is applied to the empty list, i.e., $\{\!|\}$, since the *get* and *putback* components of `wmap` and `rotate` each map $\{\!|\}$ to $\{\!|\}$.

The algorithm for computing the reversal of a list shown here runs in quadratic time. Interestingly, we have not been able to code the familiar, linear-time algorithm as a derived lens (of course, we could introduce a primitive lens for reversing lists that uses the efficient implementation internally, but it is more interesting to try to write the efficient version using our combinators). One difficulty arises if we use an accumulator to store the result: the *putback* function of such a transformation would be non-injective and so could not satisfy PutGet. To see this, consider putting the tree containing `[c]` under the accumulator child and `[b a]` as the rest of the list. This will yield the same result, `[a b c]`, as putting back a tree containing `[]` under the accumulator child and `[a b c]` as the rest of the list.

The well-behavedness lemma follows straightforwardly from the types of `wmap` and `rotate`, using Corollary 3.17(1).

7.7 Lemma [Well-behavedness]: $\forall D{\subseteq}\mathcal{T}.\ \texttt{list\_reverse} \in \texttt{[}D\texttt{]} \stackrel{\Omega}{\rightleftharpoons} \texttt{[}D\texttt{]}$

For the totality lemma, we use Corollary 3.17(2), after proving, by induction on i, that $\texttt{list\_reverse} \in \texttt{[}D^i\texttt{]} \stackrel{\Omega}{\Longleftrightarrow} \texttt{[}D^i\texttt{]}$ for all i.

7.8 Lemma [Totality]: $\forall D{\subseteq}\mathcal{T}.\ \texttt{list\_reverse} \in \texttt{[}D\texttt{]} \stackrel{\Omega}{\Longleftrightarrow} \texttt{[}D\texttt{]}$

Grouping

Next we give the definition of a "grouping" lens that, in the *get* direction, takes a list of Ds and produces a list of lists of Ds where the elements have been grouped in pairs. It is used in our bookmark synchronizer as part of a transformation that takes dictionaries of user preferences stored in the particular XML format used by Apple's Safari browser and yields trees in a simplified abstract format. When the concrete list has an even number of elements, the behavior `group` lens is simple—e.g., it maps $\texttt{[d}_1\texttt{, d}_2\texttt{, d}_3\texttt{, d}_4\texttt{, d}_5\texttt{, d}_6\texttt{]}$ to $\texttt{[[d}_1\texttt{, d}_2\texttt{], [d}_3\texttt{, d}_4\texttt{], [d}_5\texttt{, d}_6\texttt{]]}$. When there are an odd number of elements in the list, `group` places the final odd element in a singleton list–e.g., it maps $\texttt{[d}_1\texttt{, d}_2\texttt{, d}_3\texttt{]}$ to $\texttt{[[d}_1\texttt{, d}_2\texttt{], [d}_3\texttt{]]}$. The typing for `group`, given below, describes both the odd and even case.

Because it explicitly destroys and builds up cons cells, the definition of `group` is a little bit longer than the lenses we have seen so far. We explain the behavior of each part of the lens in detail below.

```
group =
  acond [] []
    id
    (acond (D :: []) ((D :: []) :: [])
      (plunge *h; add *t [])
      (rename *h tmp;
       hoist_nonunique *t {*h, *t};
       fork {*t}
         (map group)
         (xfork {*h } {*t} (add *t {|}; plunge *t) (rename tmp *h);
          plunge *h)))
```

$$\forall D{\subseteq}\mathcal{T} \quad \texttt{group} \in \texttt{[}D\texttt{]} \stackrel{\Omega}{\Longleftrightarrow} \texttt{[}D :: D :: \texttt{[]]++([]} \cup ((D :: \texttt{[]}) :: \texttt{[]}))$$

The *get* component of `group` has two cases, one for each branch of the two `acond` conditionals. If the concrete list is empty, then `group` behaves like the first branch, which is the identity. Otherwise, if the concrete list is a singleton, then `group` behaves like the second branch, which plunges the singleton list under `*h` and adds a child `*t` leading to the empty list. That is, it transforms singleton lists c into the singleton list containing c, $\{\!|\texttt{*h} \mapsto c,\ \texttt{*t} \mapsto \{\!|\}\!|\}$. Otherwise, if neither of the two previous cases apply, then `group` behaves like the third branch. There are three steps. First, it renames the head element, storing it away under a child named `tmp`. Next, it hoists up the tail of the list, yielding a tree with children `tmp`, `*h`, and `*t` (since the list is neither empty nor a singleton). In the third step, it recursively groups the tail, massages the other tree into a list of length two, and yields the cons cell made up of these trees as the result.

More specifically, in the third step of the final case, `group` splits the tree into a tree with a single child `*t` and a tree containing the `*h` and `tmp` children. It then

recursively groups the tail using (`map group`). The other tree is split yet again, into `*h` and `tmp`. The tree with `*h` is made into a singleton list by adding a child `*t` leading to the empty view, and then plunged under `*t`. The tree containing `tmp` is turned into the head of a cons cell by renaming `tmp` back to `*h`. After the `xfork`, these two trees are plunged under `*h`. Thus, $\{\!|\texttt{tmp} \mapsto \texttt{d}_i, \texttt{*h} \mapsto \texttt{d}_j|\!\}$ is transformed into the tree $\{\!|\texttt{*h} \mapsto \texttt{[d}_i\texttt{, d}_j\texttt{]}|\!\}$. The final result is obtained by merging the grouped tail with this head element.

Since each lens used in `group` is oblivious,[7] the *putback* function is symmetric, with three cases corresponding to the branches of the `acond`. Its behavior can be calculated by evaluating the compositions in reverse order.

The well-behavedness of `group` follows from Corollary 3.17(1) and a simple, compositional argument using the types of each lens appearing in its definition.

7.9 Lemma [Well-behavedness]:
$\forall D{\subseteq}\mathcal{T} \quad \texttt{group} \in \texttt{[}D\texttt{]} \stackrel{\Omega}{=} \texttt{[}D\texttt{::}D\texttt{::[]]++([]} \cup (\texttt{(}D\texttt{::[])::[]}))$

We prove the totality lemma using Corollary 3.17(2), using the increasing chains of types:

$$\begin{aligned} C_i &= \emptyset \subseteq \texttt{[]} \subseteq \quad D\texttt{::[]} \quad \subseteq \quad D\texttt{::(}D\texttt{::[])} \quad \subseteq \ldots \\ A_i &= \emptyset \subseteq \texttt{[]} \subseteq \texttt{(}D\texttt{::[])::[]} \subseteq \texttt{(}D\texttt{::}D\texttt{::[])::[]} \subseteq \ldots \end{aligned}$$

whose limit is the total type we want to show for `group`.

7.10 Lemma [Totality]:
$\forall D{\subseteq}\mathcal{T} \quad \texttt{group} \in \texttt{[}D\texttt{]} \stackrel{\Omega}{\Longleftrightarrow} \texttt{[}D\texttt{::}D\texttt{::[]]++([]} \cup (\texttt{(}D\texttt{::[])::[]}))$

Concatenation

The `concat` lens takes a tree t as an argument. It transforms lists containing two sublists of Ds and concatenates them into a single list using a single element t to track the position where the first list ends and the second begins. For example, given the tree `[[C, h, r, i, s], [S, m, i, t, h]]`, the *get* component of (`concat` $\{\!|$`" "` $\mapsto \{\!|\{\}|\!\}|\!\}$) produces the single list `[C, h, r, i, s, " ", S, m, i, t, h]`. Conversely, the *putback* function takes a list containing exactly one t and splits the list in two, producing lists containing the elements to the left and right of t respectively. The definition is as follows.

$$\begin{array}{l} \texttt{concat}\ t\ =\ \texttt{acond}\ (\texttt{[]}\texttt{::[}D\texttt{]}\texttt{::[]})\ (t\texttt{::[}D\texttt{]}) \\ \qquad\qquad (\texttt{wmap}\ \{\texttt{*h} \mapsto \texttt{const}\ t\ \texttt{[]}, \texttt{*t} \mapsto \texttt{hd []}\}) \\ \qquad\qquad (\texttt{fork}\ \{\texttt{*t}\}\ \texttt{id}\ (\texttt{hoist *h; rename *t tmp}); \\ \qquad\qquad\ \texttt{fork}\ \{\texttt{*h}\}\ \texttt{id}\ (\texttt{rename tmp *h; concat}\ t\texttt{; plunge *t})) \\ \hline \forall D{\subseteq}\mathcal{T}, t \in \mathcal{T}.\ \textit{with}\ t \notin D. \quad \texttt{concat}\ t \in \texttt{[}D\texttt{]::[}D\texttt{]::[]} \stackrel{\Omega}{\Longleftrightarrow} \texttt{[}D\texttt{]++(}t\texttt{::[}D\texttt{])} \end{array}$$

[7] Although `group` uses the `const` lens indirectly, via `add`, it is semantically oblivious. Recall that (`add` n {}) expands into (`xfork` {}{n} (`const` {} {}; `plunge` n) `id`). The type annotation on `add` ensures that the *putback* function is only ever applied to abstract trees that have a child `n` leading to {}. From this, a simple argument shows that both arguments to `const`$\searrow$ are always {}. As a result, in this case, the behavior of `const`$\searrow$ does not depend on its concrete argument—the lens is oblivious.

In the *get* direction, there are two cases, one for each branch of the `acond`. If the concrete list is of the form $(\texttt{[]} :: l :: \texttt{[]})$, where $l \in \texttt{[}D\texttt{]}$, then `concat` t produces the result (t`++`l) by applying (`const` t `[]`) to the head and (`hd []`) to extract l from the tail. Otherwise, the first element of the concrete list is non-empty and the `acond` selects the second branch. The first `fork` splits the outermost cons cell according to {`*t`}. The `id` lens is applied to the tail component, which has the form $\{\!|\texttt{*t} \mapsto (l_2 :: \texttt{[]})|\!\}$. The other component has the form $\{\!|\texttt{*h} \mapsto \{\!|\texttt{*h} \mapsto d,\ \texttt{*t} \mapsto l_1|\!\}|\!\}$. The edge labeled `*h` is clipped out using `hoist`, yielding children `*h` and `*t` (i.e., the head and tail of the first sublist) and the `*t` child is renamed to `tmp`. These two steps yield a tree $\{\!|\texttt{*h} \mapsto d,\ \texttt{tmp} \mapsto l_1|\!\}$. The second `fork` splits the tree according to {`*h`}. The `id` lens is applied to the tree $\{\!|\texttt{*h} \mapsto d|\!\}$. The other part of the tree is $\{\!|\texttt{tmp} \mapsto l_1,\ \texttt{*t} \mapsto (l_2 :: \texttt{[]})|\!\}$. By renaming `tmp` to `*h`, recursively concatenating, and plunging the result under `*t`, we obtain the tree $\{\!|\texttt{*t} \mapsto (l_1\texttt{++}(t :: l_2))|\!\}$. Combining these two results into a single tree, we obtain the list $(d :: l_1)\texttt{++}(t :: l_2)$.

The *putback* function is oblivious; its behavior is symmetric to the *get* function.

Once again, the well-behavedness lemma for `concat` t follows by a simple, compositional calculation, using Corollary 3.17(1).

7.11 Lemma [Well-behavedness]:
$\forall D \subseteq \mathcal{T}, t \in \mathcal{T}.\ \textit{with}\ t \notin D.\quad \texttt{concat}\ t \in \texttt{[}D\texttt{]} :: \texttt{[}D\texttt{]} :: \texttt{[]} \stackrel{\Omega}{=\!=} \texttt{[}D\texttt{]++}(t :: \texttt{[}D\texttt{]})$

The totality lemma follows from Corollary 3.17(2), using the increasing chains of types:

$$C_i \;=\; \emptyset \subseteq\ \texttt{[]} :: \texttt{[}D\texttt{]} :: \texttt{[]}\ \subseteq\ (D :: \texttt{[]}) :: \texttt{[}D\texttt{]} :: \texttt{[]}\ \subseteq\ (D :: D :: \texttt{[]}) :: \texttt{[}D\texttt{]} :: \texttt{[]}\ \subseteq \ldots$$
$$A_i \;=\; \emptyset \subseteq \texttt{[]++}(t :: \texttt{[}D\texttt{]}) \subseteq (D :: \texttt{[]})\texttt{++}(t :: \texttt{[}D\texttt{]}) \subseteq (D :: D :: \texttt{[]})\texttt{++}(t :: \texttt{[}D\texttt{]}) \subseteq \ldots$$

whose limit is the total type we want to show for `concat` t.

7.12 Lemma [Totality]:
$\forall D \subseteq \mathcal{T}, t \in \mathcal{T}.\ \textit{with}\ t \notin D.\quad \texttt{concat}\ t \in \texttt{[}D\texttt{]} :: \texttt{[}D\texttt{]} :: \texttt{[]} \stackrel{\Omega}{\Longleftrightarrow} \texttt{[}D\texttt{]++}(t :: \texttt{[}D\texttt{]})$

Filter

Our most interesting derived list processing lens, `list_filter`, is parameterized on two sets of views, D and E, which we assume to be disjoint and non-empty. In the *get* direction, it takes a list whose elements belong to either D or E and projects away those that belong to E, leaving an abstract list containing only Ds; in the *putback* direction, it restores the projected-away Es from the concrete list. Its definition utilizes our most complex lens combinators—`wmap` and two forms of conditionals—and 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 *putback* direction, things are more interesting because there are many ways that we could restore projected elements from the concrete list. The lens laws impose some constraints on the behavior of $l\!\searrow$. The GetPut law forces the *putback* function to restore each of the filtered elements when the abstract list is put into the original concrete list. For example (letting `d` and `e` be elements of D and E) we must have $l\!\searrow$(`[d]`, `[e d]`) = `[e d]`. The PutGet law forces the *putback* function to include every element of the abstract list in the

resulting concrete list in the same order, and these elements must be the only Ds in the result; there is, however, no restriction on the Es when the abstract tree is not the filtered concrete tree.

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 $l\searrow$ 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 $l\searrow$ appends the rest of the elements of a at the end of c. If a runs out first, then $l\searrow$ restores the remaining Es from the end of c and discards any remaining Ds in c (as it must to satisfy PutGet).

These choices lead us to the following specification for a single step of the *putback* part of a recursively defined lens implementing l. If the abstract list a is empty, then we restore all the Es from c. If c is empty and a is not empty, then we return a. If a and c are both cons cells whose heads are in D, then we return a cons cell whose head is the head of a and whose tail is the result obtained by recursing on the tails of both a and c. Otherwise (i.e., c has type $E::([D]\&[E])$) we restore the head of c and recurse on a and the tail of c. Translating this into lens combinators leads to the definition below of a recursive lens `inner_filter`, which filters lists containing at least one D, and a top-level lens `list_filter` that handles arbitrary lists of Ds and Es.

```
inner_filter D E =
    ccond (E::([D^{1..ω}]&[E]))
      (tl any_E; inner_filter D E)
      (wmap {*h ↦ id,
             *t ↦ (cond [E] [] [D^{1..ω}] fltr_E (λc. c++[any_D])
                    (const [] [])
                    (inner_filter D E))})

 list_filter D E =
    cond [E] [] [D^{1..ω}] fltr_E (λc. c++[any_D])
        (const [] [])
        (inner_filter D E)}
```

$$\forall D, E \subseteq \mathcal{T}.\ \textit{with}\ D \cap E = \emptyset\ \textit{and}\ D \neq \emptyset\ \textit{and}\ E \neq \emptyset.$$
$$\texttt{inner\_filter}\ D\ E \in [D^{1..\omega}]\&[E] \overset{\Omega}{\Longleftrightarrow} [D^{1..\omega}]$$
$$\texttt{list\_filter}\ D\ E \in [D]\&[E] \overset{\Omega}{\Longleftrightarrow} [D]$$

The "choice operator" any_D denotes an arbitrary element of the (non-empty) set D.[8] The function $fltr_E$ is the usual list-filtering *function*, which for present purposes

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

we simply assume has been defined as a primitive. (In our actual implementation, we use `list_filter`↗; but for expository purposes, and to simplify the totality proofs, we avoid this extra bit of recursiveness.) Finally, the function $\lambda c.\ c$`++[`any_D`]` appends some arbitrary element of D to the right-hand end of a list c. These "fixup functions" are applied in the *putback* direction by the `cond` lens.

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

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

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

$$
\begin{aligned}
&\underline{l \searrow (a,\, c)} = \underline{i \searrow (a,\, c)} \\
&\qquad \text{by the definition of } \texttt{cond}, \text{ as } a \in \texttt{[}D^{1..\omega}\texttt{]} \text{ and } c \in (\texttt{[}D\texttt{]\&[}E\texttt{]}) \setminus \texttt{[}E\texttt{]} \\
&= \underline{(\texttt{tl}\ \mathit{any}_E;\ i) \searrow (a,\, c)} \\
&\qquad \text{by the definition of } \texttt{ccond}, \text{ as } c \in E :: (\texttt{[}D^{1..\omega}\texttt{]\&[}E\texttt{]}) \\
&= (\texttt{tl}\ \mathit{any}_E) \searrow \Big(i \searrow \Big(a,\, \underline{(\texttt{tl}\ \mathit{any}_E) \nearrow c}\Big),\, c\Big) \\
&\qquad \text{by the definition of composition} \\
&= (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\underline{i \searrow (a,\, \texttt{[d}_2\ \texttt{e}_2\texttt{]})},\, c\Big) \\
&\qquad \text{reducing } (\texttt{tl}\ \mathit{any}_E) \nearrow c \\
&= (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\underline{\texttt{wmap}\ \{\texttt{*h} \mapsto \texttt{id}, \texttt{*t} \mapsto l\} \searrow (a,\, \texttt{[d}_2\ \texttt{e}_2\texttt{]})},\, c\Big) \\
&\qquad \text{by the definition of } \texttt{ccond}, \text{ as } \texttt{[d}_2\ \texttt{e}_2\texttt{]} \notin E :: (\texttt{[}D^{1..\omega}\texttt{]\&[}E\texttt{]}) \\
&= (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\texttt{d}_1 :: (\underline{l \searrow (\texttt{[]},\, \texttt{[e}_2\texttt{]})}),\, c\Big) \\
&\qquad \text{by the definition of } \texttt{wmap} \text{ with } \texttt{id} \searrow (\texttt{d}_1,\, \texttt{d}_2) = \texttt{d}_1 \\
&= (\texttt{tl}\ \mathit{any}_E) \searrow \Big(\texttt{d}_1 :: (\underline{(\texttt{const [] []}) \searrow (\texttt{[]},\, \texttt{[e}_2\texttt{]})}),\, c\Big) \\
&\qquad \text{by the definition of } \texttt{cond}, \text{ as } \texttt{[]} \in \texttt{[]} \text{ and } \texttt{[e}_2\texttt{]} \in \texttt{[}E\texttt{]} \\
&= \underline{(\texttt{tl}\ \mathit{any}_E) \searrow (\texttt{d}_1 :: \texttt{[e}_2\texttt{]},\, c)} \\
&\qquad \text{by the definition of } \texttt{const} \\
&= \texttt{[e}_1\ \texttt{d}_1\ \texttt{e}_2\texttt{]} \quad \text{by the definition of } \texttt{tl}.
\end{aligned}
$$

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

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

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

$$\begin{array}{ll} & \underline{l \searrow (a,\, c)} = i \searrow \Big(a,\, \underline{(\lambda c.\ c\mathtt{++}[\mathit{any}_D])\ c}\Big) \\ & \qquad \text{by the definition of } \mathtt{cond}, \text{ as } a \in [D^{1..\omega}] \text{ and } c \in [E] \\ = & \underline{i \searrow (a,\, [\mathtt{e}_1\ \mathtt{d}_0])} \\ & \qquad \text{by the definition of } (\lambda c.\ c\mathtt{++}[\mathit{any}_D]) \\ = & \underline{(\mathtt{tl}\ \mathit{any}_E;\ i) \searrow (a,\, [\mathtt{e}_1\ \mathtt{d}_0])} \\ & \qquad \text{by the definition of } \mathtt{ccond}, \text{ as } [\mathtt{e}_1\ \mathtt{d}_0] \in E :: ([D^{1..\omega}] D \& [E]) \\ = & \ (\mathtt{tl}\ \mathit{any}_E) \searrow \Big(i \searrow \Big(a,\, \underline{(\mathtt{tl}\ \mathit{any}_E) \nearrow [\mathtt{e}_1\ \mathtt{d}_0]}\Big),\, [\mathtt{e}_1\ \mathtt{d}_0]\Big) \\ & \qquad \text{by the definition of composition} \\ = & \ (\mathtt{tl}\ \mathit{any}_E) \searrow \Big(\underline{i \searrow (a,\, [\mathtt{d}_0])},\, [\mathtt{e}_1\ \mathtt{d}_0]\Big) \\ & \qquad \text{reducing } (\mathtt{tl}\ \mathit{any}_E) \nearrow [\mathtt{e}_1\ \mathtt{d}_0] \\ = & \ (\mathtt{tl}\ \mathit{any}_E) \\ & \qquad \searrow \Big(\underline{\mathtt{wmap}\ \{\mathtt{*h} \mapsto \mathtt{id}, \mathtt{*t} \mapsto l\} \searrow (a,\, [\mathtt{d}_0])},\, [\mathtt{e}_1\ \mathtt{d}_0]\Big) \\ & \qquad \text{by the definition of } \mathtt{ccond}, \text{ as } [\mathtt{d}_0] \notin E :: ([D^{1..\omega}] \& [E]) \\ = & \ (\mathtt{tl}\ \mathit{any}_E) \searrow \Big(\mathtt{d}_1 :: (\underline{l \searrow ([\,],\, [\,])}),\, [\mathtt{e}_1\ \mathtt{d}_0]\Big) \\ & \qquad \text{by the definition of } \mathtt{wmap} \text{ with } \mathtt{id} \searrow (\mathtt{d}_1,\, \mathtt{d}_0) = \mathtt{d}_1 \\ = & \ (\mathtt{tl}\ \mathit{any}_E) \searrow \Big(\mathtt{d}_1 :: (\underline{(\mathtt{const}\ [\,]\ [\,]) \searrow ([\,],\, [\,])}),\, [\mathtt{e}_1\ \mathtt{d}_0]\Big) \\ & \qquad \text{by the definition of } \mathtt{cond}, \text{ as } [\,] \in [\,] \text{ and } [\,] \in [E] \\ = & \ \underline{(\mathtt{tl}\ \mathit{any}_E) \searrow (\mathtt{d}_1 :: [\,],\, [\mathtt{e}_1\ \mathtt{d}_0])} \\ & \qquad \text{by the definition of } \mathtt{const} \\ = & \ [\mathtt{e}_1\ \mathtt{d}_1] \quad \text{by the definition of } \mathtt{tl}. \end{array}$$

The well-behavedness proof for `inner_filter` is straightforward: we simply decide on a type for the recursive use of `inner_filter` and then show that, under this assumption, the body of the lens has this type. Since `list_filter` is not recursive, both its well-behavedness and totality lemmas both follow straightforwardly from the types of the lenses that are used in its definition.

7.13 Lemma [Well-behavedness]:
$\forall D, E \subseteq \mathcal{T}$. *with* $D \cap E = \emptyset$ *and* $D \neq \emptyset$ *and* $E \neq \emptyset$.
$\texttt{inner\_filter}\ D\ E \in [D^{1..\omega}] \& [E] \stackrel{\Omega}{\rightleftharpoons} [D^{1..\omega}]$
$\texttt{list\_filter}\ D\ E \in [D] \& [E] \stackrel{\Omega}{\rightleftharpoons} [D]$

The totality proof for `inner_filter`, on the other hand, is somewhat challenging, involving detailed reasoning about the behavior of particular subterms under particular conditions. The proof uses Lemma 3.19, with sequences of sets of total types

$$\begin{aligned} \mathbb{T}_0 &= \{(\emptyset, \emptyset)\} \\ \mathbb{T}_{i+1} &= \{([D^{1..x}] \& [E^{0..y}], [D^{1..x}]) \mid x + y = i\}. \end{aligned}$$

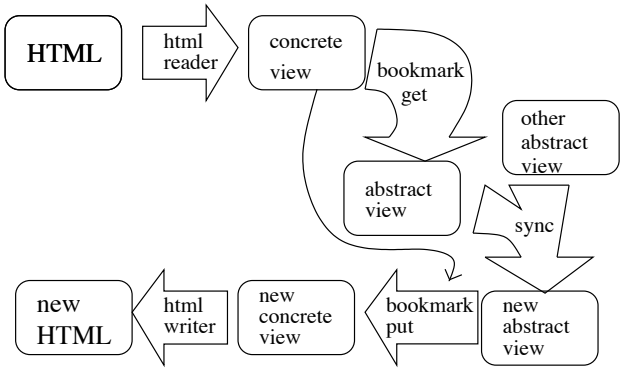
The complete argument is given in electronic Appendix A.

7.14 Lemma [Totality]:
$\forall D, E \subseteq \mathcal{T}$. *with* $D \cap E = \emptyset$ *and* $D \neq \emptyset$ *and* $E \neq \emptyset$.
$\texttt{inner\_filter}\ D\ E \in [D^{1..\omega}] \& [E] \stackrel{\Omega}{\Longleftrightarrow} [D^{1..\omega}]$
$\texttt{list\_filter}\ D\ E \in [D] \& [E] \stackrel{\Omega}{\Longleftrightarrow} [D]$

8. EXTENDED EXAMPLE: A BOOKMARK LENS

In this section, we develop a larger and more realistic example of programming with our lens combinators. The example comes from a demo application of our data synchronization framework, Harmony, in which bookmark information from diverse browsers, including Internet Explorer, Mozilla, Safari, Camino, and OmniWeb, is synchronized by transforming each format from its concrete native representation into a common abstract form. We show here a slightly simplified form of the Mozilla lens, which handles the HTML-based bookmark format used by Netscape and its descendants.

The overall path taken by the bookmark data through the Harmony system can be pictured as follows.



We first use a generic HTML reader to transform the HTML bookmark file into an isomorphic concrete tree. This concrete tree is then transformed, using the *get* direction of the `bookmark` lens, into an abstract "generic bookmark tree." The abstract tree is synchronized with the abstract bookmark tree obtained from some other bookmark file, yielding a new abstract tree, which is transformed into a new concrete tree by passing it back through the *putback* direction of the `bookmark`

$$
\begin{array}{lcl}
ALink_1 & = & \{\texttt{name} \mapsto Val \;\; \texttt{url} \mapsto Val\} \\
ALink & = & \{\texttt{link} \mapsto ALink_1\} \\
AFolder_1 & = & \{\texttt{name} \mapsto Val \;\; \texttt{contents} \mapsto AContents\} \\
AFolder & = & \{\texttt{folder} \mapsto AFolder_1\} \\
AContents & = & [AItem] \\
AItem & = & ALink \cup AFolder
\end{array}
$$

Fig. 2. Abstract Bookmark Types

lens (supplying the original concrete tree as the second argument). Finally, the new concrete tree is written back out to the filesystem as an HTML file. We now discuss these transformations in detail.

Abstractly, the type of bookmark data is a `name` pointing to a value and a `contents`, which is a list of items. An *item* is either a *link*, with a `name` and a `url`, or a *folder*, which has the same type as bookmark data. Figure 2 formalizes these types.

Concretely, in HTML (see Figure 3), a bookmark item is represented by a `<dt>` element containing an `<a>` element whose `href` attribute gives the link's url and whose content defines the name. The `<a>` element also includes an `add_date` attribute, which we have chosen not to reflect in the abstract form because it is not supported by all browsers. A bookmark folder is represented by a `<dd>` element containing an `<h3>` header (giving the folder's name) followed by a `<dl>` list containing the sequence of items in the folder. The whole HTML bookmark file follows the standard `<head>`/`<body>` form, where the contents of the `<body>` have the format of a bookmark folder, without the enclosing `<dd>` tag. (HTML experts will note that the use of the `<dl>`, `<dt>`, and `<dd>` tags here is not actually legal HTML. This is unfortunate, but the conventions established by early versions of Netscape have become a de-facto standard.)

The generic HTML reader and writer know nothing about the specifics of the bookmark format; they simply transform between HTML syntax and trees in a mechanical way, mapping an HTML element named `tag`, with attributes `attr1` to `attrm` and sub-elements `subelt1` to `subeltn`,

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

into a tree of this form:

$$
\left\{\left|\begin{array}{l}
\texttt{tag} \mapsto \left\{\left|\begin{array}{ll}
\texttt{attr1} \mapsto \texttt{val1} \\
\quad\vdots \\
\texttt{attrm} \mapsto \texttt{valm} \\
* \mapsto \left[\begin{array}{l}\texttt{subelt1} \\ \quad\vdots \\ \texttt{subeltn}\end{array}\right]
\end{array}\right|\right\}
\end{array}\right|\right\}
$$

Note that the sub-elements are placed in a *list* under a distinguished child named `*`. This preserves their ordering from the original HTML file. (The ordering of sub-elements is sometimes important—e.g., in the present example, it is important

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

Fig. 3. Sample Bookmarks (HTML)

```
{html -> {* ->
 [{head -> {* -> [{title -> {* ->
                    [{PCDATA -> Bookmarks}]}}]}}
  {body -> {* ->
   [{h3 -> {* -> [{PCDATA -> Bookmarks Folder}]}}
    {dl -> {* ->
     [{dt -> {* ->
       [{a -> {* -> [{PCDATA -> Google}]
              add_date -> 1032458036
              href -> www.google.com}}]}}
      {dd -> {* ->
       [{h3 -> {* -> [{PCDATA ->
                      Conferences Folder}]}}
        {dl -> {* ->
         [{dt -> {* ->
            [{a ->
              {* -> [{PCDATA -> ICFP}]
               add_date -> 1032528670
               href -> www.cs.luc.edu/icfp
              }}]}}]}}]}}]}}]}}]}}
```

Fig. 4. Sample Bookmarks (concrete tree)

$$
\begin{array}{lcl}
\mathit{Val} & = & \{\mathcal{N}\} \\
\mathit{PCDATA} & = & \{\texttt{PCDATA} \mapsto \mathit{Val}\} \\
\\
\mathit{CLink} & = & \texttt{<dt>}\ \mathit{CLink}_1 :: [] \ \texttt{</dt>} \\
\mathit{CLink}_1 & = & \texttt{<a add\_date href>}\ \mathit{PCDATA} :: [] \ \texttt{} \\
\\
\mathit{CFolder} & = & \texttt{<dd>}\ \mathit{CContents}\ \texttt{</dd>} \\
\\
\mathit{CContents} & = & \mathit{CContents}_1 :: \mathit{CContents}_2 :: [] \\
\mathit{CContents}_1 & = & \texttt{<h3>}\ \mathit{PCDATA} :: [] \ \texttt{</h3>} \\
\mathit{CContents}_2 & = & \texttt{<dl>}\ [\mathit{CItem}]\ \texttt{</dl>} \\
\\
\mathit{CItem} & = & \mathit{CLink} \cup \mathit{CFolder} \\
\\
\mathit{CBookmarks} & = & \texttt{<html>}\ \mathit{CBookmarks}_1 :: \mathit{CBookmarks}_2 :: [] \ \texttt{</html>} \\
\mathit{CBookmarks}_1 & = & \texttt{<head>}\ (\texttt{<title>}\ \mathit{PCDATA}\ \texttt{</title>} :: []) \ \texttt{</head>} \\
\mathit{CBookmarks}_2 & = & \texttt{<body>}\ \mathit{CContents}\ \texttt{</body>}
\end{array}
$$

Fig. 5. Concrete Bookmark Types

```
{name -> Bookmarks Folder
 contents ->
  [{link -> {name -> Google
             url -> www.google.com}}
   {folder ->
     {name -> Conferences Folder
      contents ->
       [{link ->
         {name -> ICFP
          url -> www.cs.luc.edu/icfp}}]}}]}
```

Fig. 6. Sample Bookmarks (abstract tree)

to maintain the ordering of the items within a bookmark folder. Since the HTML reader and writer are generic, they *always* record the ordering from the the original HTML in the tree, leaving it up to whatever lens is applied to the tree to throw away ordering information where it is not needed.) A leaf of the HTML document—i.e., a "parsed character data" element containing a text string `str`—is converted to a tree of the form `{PCDATA -> str}`. Passing the HTML bookmark file shown in Figure 3 through the generic reader yields the tree in Figure 4.

Figure 5 shows the type *CBookmarks* of concrete bookmark structures. For readability, the type relies on a notational shorthand that reflects the structure of the encoding of HTML as trees. We write $\texttt{<tag attr1}\ldots\texttt{attrn>}\ C\ \texttt{</tag>}$ for $\{\texttt{tag} \mapsto \{\texttt{attr1} \mapsto \mathit{Val}\ \ldots\texttt{attrn} \mapsto \mathit{Val}\ \texttt{*} \mapsto C\}\}$. Recall that *Val* is the set of all values (trees with a single childless child). For elements with no attributes, this degenerates to simply $\texttt{<tag>}\ C\ \texttt{</tag>} = \{\texttt{tag} \mapsto \{\texttt{*} \mapsto C\}\}$.

The transformation between this concrete tree and the abstract bookmark tree shown in Figure 6 is implemented by means of the collection of lenses shown in Figure 7. Most of the work of these lenses (in the *get* direction) involves stripping out various extraneous structure and then renaming certain branches to have the desired "field names." Conversely, the *putback* direction restores the original names and rebuilds the necessary structure.

To aid in checking well-behavedness, we annotate each lens with its source and target type, writing $\in C\ l \stackrel{\Omega}{\rightleftharpoons} A$. (This infix notation—where l is written between its source and target types, instead of the more conventional $l \in C \stackrel{\Omega}{\rightleftharpoons} A$—looks strange in-line, but it works well for multi-line displays such as Figure 7.) and annotate each composition with a suitable "cut type," writing $l\ ;:B\ k$ instead of just $l; k$.

It is then straightforward to check, using the type annotations supplied, that `bookmarks` $\in$ $CBookmarks \stackrel{\Omega}{\rightleftharpoons} AFolder_1$. (We omit the proof of totality, since we have already seen more intricate totality arguments in Section 7).

In practice, composite lenses are developed incrementally, gradually massaging the trees into the correct shape. Figure 8 shows the process of developing the `link` lens by transforming the representation of the HTML under a `<dt>` element containing a link into the desired abstract form. At each level, tree branches are relabeled with `rename`, undesired structure is removed with `prune`, `hoist`, and/or `hd`, and then work is continued deeper in the tree via `wmap`.

The *putback* direction of the `link` lens restores original names and structure automatically, by composing the *putback* directions of the constituent lenses of `link` in turn. For example, Figure 9 shows an update to the abstract tree of the link in Figure 8. The concrete tree beneath the update shows the result of applying *putback* to the updated abstract tree. The *putback* direction of the `hoist PCDATA` lens, corresponding to moving from step *viii* to step *vii* in Figure 8, puts the updated string in the abstract tree back into a more concrete tree by replacing `Search-Engine` with {| `PCDATA -> Search-Engine` |}. In the transition from step *vi* to step *v*, the *putback* direction of `prune add_date` {| `$today` |} utilizes the concrete tree to restore the value, `add_date -> 1032458036`, projected away in the abstract tree. If the concrete tree had been Ω—i.e., in the case of a new bookmark added in the new abstract tree—then the default argument {| `$today` |} would have been used to fill in today's date. (Formally, the whole set of lenses is parameterized on the variable `$today`, which ranges over names.)

The *get* direction of the `folder` lens separates out the folder name and its contents, stripping out undesired structure where necessary. Finally, we use `wmap` to iterate over the contents.

The `item` lens processes one element of a folder's contents; this element might be a link or another folder, so we want to either apply the `link` lens or the `folder` lens. Fortunately, we can distinguish them by whether they are contained within a `<dd>` element or a `<dt>` element; we use the `wmap` operator to wrap the call to the correct sublens. Finally, we rename `dd` to `folder` and `dt` to `link`.

The main lens is `bookmarks`, which (in the *get* direction) takes a whole concrete bookmark tree, strips off the boilerplate header information using a combination of `hoist`, `hd`, and `tl`, and then invokes `folder` to deal with the rest. The huge default tree supplied to the `tl` lens corresponds to the head tag of the HTML document, which is filtered away in the abstract bookmark format. This default tree would be used to recreate a well-formed head tag if it was missing in the original concrete tree.

```
link =                                  ∈ {*↦Clink1 :: []}
  hoist *;                                : CLink1 :: []
  hd [];                                  : CLink1
  hoist a;                                : {*↦PCDATA :: [], add_date↦Val,
                                             href↦Val}
  rename * name;                          : {name↦PCDATA :: [], add_date↦Val,
                                             href↦Val}
  rename href url;                        : {name↦PCDATA :: [], add_date↦Val,
                                             url↦Val}
  prune add_date {$today};                : {name↦PCDATA :: [], url↦Val}
  wmap {name -> (hd [];                   : PCDATA
  hoist PCDATA)}                        ⇌Ω {name↦Val, url↦Val} = ALink1

folder =                                ∈ {*↦CContents}
  hoist *;                                : CContents
  xfork {*h} {name}
    (hoist *h;                                  : {h3↦{*↦PCDATA :: []}}
     rename h3 name)
    (hoist *t;                                  : CContents2 :: []
     hd [];                                     : {dl↦{*↦[CItem]}}
     rename dl contents)                  : {name↦{*↦PCDATA :: []},
                                             contents↦{*↦[CItem]}}
  wmap {name -> (hoist *;                       : PCDATA :: []
                 hd [];                         : PCDATA
                 hoist PCDATA)
       contents -> (hoist *;                    : [CItem]
                    list_map item)}
                                        ⇌Ω {name↦Val,
                                            contents↦[AItem]} = AFolder1

item =                                  ∈ CItem
  wmap {dd -> folder, dt -> link };       : {dd↦AFolder1} ∪ {dt↦ALink1}
  rename_if_present dd folder;            : {folder↦AFolder1} ∪ {dt↦ALink1}
  rename_if_present dt link               : ⇌Ω AFolder ∪ ALink = AItem

bookmarks =                             ∈ CBookmarks
  hoist html;                             : {*↦CBookmarks1 :: CBookmarks2 :: []}
  hoist *;                                : CBookmarks1 :: CBookmarks2 :: []
  tl {|head ↦ {|* ↦ [{|title ↦ {|* ↦
      [{|PCDATA ↦ Bookmarks|}]|}|}]|}|};  : CBookmarks2 :: []
  hd [];                                  : CBookmarks2
  hoist body;                             : {*↦CContents}
  folder                                ⇌Ω AFolder1
```

Fig. 7. Bookmark lenses

9. LENSES FOR RELATIONAL DATA

We close our technical development by presenting a few additional lenses that we use in Harmony to deal with preparing relational data—trees (or portions of trees) consisting of "lists of records"—for synchronization. These lenses do not constitute a full treatment of view update for relational data, but may be regarded as a small step in that direction. (A later and more comprehensive proposal is reported

Step	Lens expression	Resulting abstract tree (from 'get')

i:	`id`	`{* ->`
		`  [{a -> {* -> [{PCDATA -> Google}]`
		`  add_date -> 1032458036`
		`  href -> www.google.com}}]}}`

ii:	`hoist *`	`[{a -> {* -> [{PCDATA -> Google}]`
		`  add_date -> 1032458036`
		`  href -> www.google.com}}]`

iii:	`hoist *;`
	`hd []`
		`{a -> {* -> [{PCDATA -> Google}]`
		`  add_date -> 1032458036`
		`  href -> www.google.com}}`

iv:	`hoist *;`
	`hd [];`
	`hoist a;`
		`{* -> [{PCDATA -> Google}]`
		`  add_date -> 1032458036`
		`  href -> www.google.com}`

v:	`hoist *;`
	`hd [];`
	`hoist a;`
	`rename * name;`
	`rename href url`
		`{name -> [{PCDATA -> Google}]`
		`  add_date -> 1032458036`
		`  url -> www.google.com}`

vi:	`hoist *;`
	`hd [];`
	`hoist a;`
	`rename * name;`
	`rename href url;`
	`prune add_date {$today}`
		`{name -> [{PCDATA -> Google}]`
		`  url -> www.google.com}`

vii:	`hoist *;`
	`hd [];`
	`hoist a;`
	`rename * name;`
	`rename href url;`
	`prune add_date {$today};`
	`wmap { name -> (hd {}) }`
		`{name -> {PCDATA -> Google}`
		`  url -> www.google.com}`

viii:	`hoist *;`
	`hd [];`
	`hoist a;`
	`rename * name;`
	`rename href url;`
	`prune add_date {$today};`
	`wmap { name -> (hd {}; hoist PCDATA) }`
		`{name -> Google`
		`  url -> www.google.com}`

Fig. 8. Building up a link lens incrementally.

```
{link -> {name -> Google
          url -> www.google.com}}
```

updated to...

```
{link -> {name -> Search-Engine
          url -> www.google.com}}
```

yields (after *putback*)...

```
{dt -> {* ->
        [{a -> {* -> [{PCDATA -> Search-Engine}]
                add_date -> 1032458036
                href -> www.google.com}}]}}
```

Fig. 9. Update of abstract tree, and resulting concrete tree

in [Bohannon et al. 2006]). In particular, the `join` lens performs a transformation related to the *outer join* operation in database query languages.

Flatten

The most critical (and complex) of these lenses is `flatten`, which takes an ordered list of "keyed records" and flattens it into a bush, as in the following example:

$$\texttt{flatten}^{\nearrow}\left[\left\{\texttt{Pat}\mapsto\left\{\begin{array}{l}\texttt{Phone}\mapsto\texttt{333-4444}\\ \texttt{URL}\mapsto\texttt{http://pat.com}\end{array}\right\}\right\}\left\{\texttt{Chris}\mapsto\left\{\begin{array}{l}\texttt{Phone}\mapsto\texttt{888-9999}\\ \texttt{URL}\mapsto\texttt{http://x.org}\end{array}\right\}\right\}\right] = \left\{\begin{array}{l}\texttt{Pat}\mapsto\left[\left\{\begin{array}{l}\texttt{Phone}\mapsto\texttt{333-4444}\\ \texttt{URL}\mapsto\texttt{http://pat.com}\end{array}\right\}\right]\\ \texttt{Chris}\mapsto\left[\left\{\begin{array}{l}\texttt{Phone}\mapsto\texttt{888-9999}\\ \texttt{URL}\mapsto\texttt{http://x.org}\end{array}\right\}\right]\end{array}\right\}$$

The importance of this transformation in the setting of the Harmony system is that it makes the "intended alignment" of the data structurally obvious. This frees Harmony's synchronization algorithm from needing to understand that, although the data is presented in an ordered fashion, order is actually not significant here. Synchronization simply proceeds child-wise—i.e., the record under `Pat` is synchronized with the corresponding record under `Pat` from the other replica, and similarly for `Chris`. If one of the replicas happens to place `Chris` before `Pat` in its concrete, ordered form, exactly the same thing happens.

The `flatten` lens handles concrete lists in which the same key appears more than once by placing all the records with the same key (in the same order as they appear in the concrete view) in the list under that key in the abstract view:

$$\texttt{flatten}^{\nearrow}\left[\begin{array}{l}\left\{\texttt{Pat}\mapsto\left\{\begin{array}{l}\texttt{Phone}\mapsto\texttt{333-4444}\\ \texttt{URL}\mapsto\texttt{http://pat.com}\end{array}\right\}\right\}\\ \left\{\texttt{Chris}\mapsto\left\{\begin{array}{l}\texttt{Phone}\mapsto\texttt{888-9999}\\ \texttt{URL}\mapsto\texttt{http://x.org}\end{array}\right\}\right\}\\ \left\{\texttt{Pat}\mapsto\left\{\begin{array}{l}\texttt{Phone}\mapsto\texttt{123-4321}\\ \texttt{URL}\mapsto\texttt{http://p2.com}\end{array}\right\}\right\}\end{array}\right] = \left\{\begin{array}{l}\texttt{Pat}\mapsto\left[\begin{array}{l}\left\{\begin{array}{l}\texttt{Phone}\mapsto\texttt{333-4444}\\ \texttt{URL}\mapsto\texttt{http://pat.com}\end{array}\right\}\\ \left\{\begin{array}{l}\texttt{Phone}\mapsto\texttt{123-4321}\\ \texttt{URL}\mapsto\texttt{http://p2.com}\end{array}\right\}\end{array}\right]\\ \texttt{Chris}\mapsto\left[\left\{\begin{array}{l}\texttt{Phone}\mapsto\texttt{888-9999}\\ \texttt{URL}\mapsto\texttt{http://x.org}\end{array}\right\}\right]\end{array}\right\}$$

In the *putback* direction, `flatten` distributes elements of each list from the abstract bush into the concrete list, maintaining their original concrete positions. If there are more abstract elements than concrete ones, the extras are simply appended at the end of the resulting concrete list in some arbitrary order, using the auxiliary function *listify*:

$$\begin{array}{lcl} \mathit{listify}(\{\!|\}) & = & \texttt{[]} \\ \mathit{listify}(t) & = & \{\!|k \mapsto tk_1|\!\} :: \cdots :: \{\!|k \mapsto tk_n|\!\} :: \mathit{listify}(t\backslash_k) \\ & & \text{where } k = \mathit{any}_{\mathsf{dom}(t)} \text{ and } t(k) = [tk_1, \ldots, tk_n] \end{array}$$

In the type of `flatten`, we write $\texttt{AList}_K(D)$ for the set of lists of "singleton views" of the form $\{\!|\texttt{k} \mapsto d|\!\}$, where $k \in K$ is a key and $d \in D$ is the value of that key—i.e., $\texttt{AList}_K(D)$ is the smallest set of trees satisfying $\texttt{AList}_K(D) = \texttt{[]} \cup (\{\{\!|\texttt{k} \mapsto D|\!\} \mid k \in K\} :: \texttt{AList}_K(D))$.

$$\begin{array}{l} \texttt{flatten}\nearrow c \quad = \\ \quad \begin{cases} \{\!|\}\!\} & \text{if } c = \texttt{[]} \\ a' + \{\!|k \mapsto d :: \texttt{[]}|\!\} & \text{if } c = \{\!|k \mapsto d|\!\} :: c' \\ & \quad \text{and } \texttt{flatten}\nearrow c' = a' \text{ with } k \notin \mathsf{dom}(a') \\ a' + \{\!|k \mapsto d :: s|\!\} & \text{if } c = \{\!|k \mapsto d|\!\} :: c' \\ & \quad \text{and } \texttt{flatten}\nearrow c' = a' + \{\!|k \mapsto s|\!\} \end{cases} \\ \texttt{flatten}\searrow(a,\, c) \; = \\ \quad \begin{cases} \mathit{listify}(a) & \text{if } c = \texttt{[]} \text{ or } c = \Omega \\ \{\!|k \mapsto d'|\!\} :: r & \text{if } c = \{\!|k \mapsto d|\!\} :: c' \\ & \quad \text{and } a(k) = d' :: \texttt{[]} \\ & \quad \text{and } r = \texttt{flatten}\searrow(a\backslash_k,\, c') \\ \{\!|k \mapsto d'|\!\} :: r & \text{if } c = \{\!|k \mapsto d|\!\} :: c' \\ & \quad \text{and } a(k) = d' :: s \text{ with } s \neq \texttt{[]} \\ & \quad \text{and } r = \texttt{flatten}\searrow\left(a\backslash_k + \{\!|k \mapsto s|\!\},\, c'\right) \\ r & \text{if } c = \{\!|k \mapsto d|\!\} :: c' \\ & \quad \text{and } k \notin \mathsf{dom}(a) \\ & \quad \text{and } r = \texttt{flatten}\searrow(a,\, c') \end{cases} \\ \hline \\ \quad \forall K \subseteq \mathcal{N}.\ \forall D \subseteq \mathcal{T}. \\ \qquad \texttt{flatten} \in \texttt{AList}_K(D) \overset{\Omega}{\Longleftrightarrow} \left\{\!\left|K \overset{?}{\mapsto} \texttt{[}D^{1..\omega}\texttt{]}\right|\!\right\} \end{array}$$

This definition can be simplified if we assume that all the ks in the concrete list are pairwise different—i.e., that they are truly keys. In this case, the abstract view need not be a bush of lists: each k can simply point directly to its associated subtree from the concrete list. In practice, this assumption is often reasonable: the concrete view is a (linearized) database and the ks are taken from a key field in each record. However, the *type* of this "disjoint flatten" becomes more complicated to write down, since it must express the constraint that, in the concrete list, each k occurs at most once. Since we eventually intend to implement a mechanical typechecker for our combinators, we prefer to use the more complex definition with the more elementary type.

An obvious question is whether either variant of `flatten` can be expressed in terms of more primitive combinators plus recursion, as we did for the list mapping,

reversing, and filtering derived forms in Section 7. We feel that this ought to be possible, but we have not yet succeeded in doing it.

One final point about `flatten` is that it does not obey PutPut. Let

$$a_1 = \{\!|\, \mathtt{a} \mapsto \mathtt{[\{\!|\}\!]},\ \mathtt{b} \mapsto \mathtt{[\{\!|\}\!]} \,|\!\}\quad a_2 = \{\!|\, \mathtt{b} \mapsto \mathtt{[\{\!|\}\!]} \,|\!\}\quad c = [\mathtt{a},\ \mathtt{b}]\,.$$

If `flatten` were very well behaved then we would have

$$\mathtt{flatten}\searrow(a_1, \mathtt{flatten}\searrow(a_2, c)) = \mathtt{flatten}\searrow(a_1, c).$$

However, the left hand side of the equality is `[b, a]` but the right hand side is `[a, b]`.

Pivot

The lens `pivot` n rearranges the structure at the top of a tree, transforming $\left\{\!\middle|\begin{matrix} n \mapsto k \\ t \end{matrix}\middle|\!\right\}$ to $\{\!|\, k \mapsto t \,|\!\}$. Intuitively, the value k (i.e., $\{\!|\, k \mapsto \{\!|\}\!| \,|\!\}$) under n represents a *key* k for the rest of the tree t. The *get* function of `pivot` returns a tree where k points directly to t. The *putback* function performs the reverse transformation, ignoring the old concrete tree.

We use `pivot` heavily in Harmony instances where the data being synchronized is relational (sets of records) but its concrete format is ordered (e.g., XML). We first apply `pivot` within each record to bring the key field to the outside. Then we apply `flatten` to smash the list of keyed records into a bush indexed by the keys. As an example, consider the following transformation on a concrete piece of data where $l =$ `list_map (pivot Name)`:

$$l\nearrow \left[\begin{array}{l} \left\{\!\middle|\begin{matrix} \mathtt{Name}\mapsto\mathtt{Pat} \\ \mathtt{Phone}\mapsto\mathtt{333\text{-}4444} \\ \mathtt{URL}\mapsto\mathtt{http://pat.com} \end{matrix}\middle|\!\right\} \\ \left\{\!\middle|\begin{matrix} \mathtt{Name}\mapsto\mathtt{Chris} \\ \mathtt{Phone}\mapsto\mathtt{888\text{-}9999} \\ \mathtt{URL}\mapsto\mathtt{http://x.org} \end{matrix}\middle|\!\right\} \\ \left\{\!\middle|\begin{matrix} \mathtt{Name}\mapsto\mathtt{Pat} \\ \mathtt{Phone}\mapsto\mathtt{123\text{-}4321} \\ \mathtt{URL}\mapsto\mathtt{http://p2.com} \end{matrix}\middle|\!\right\} \end{array}\right] = \left[\begin{array}{l} \left\{\!\middle|\, \mathtt{Pat}\mapsto \left\{\!\middle|\begin{matrix} \mathtt{Phone}\mapsto\mathtt{333\text{-}4444} \\ \mathtt{URL}\mapsto\mathtt{http://pat.com} \end{matrix}\middle|\!\right\} \,\middle|\!\right\} \\ \left\{\!\middle|\, \mathtt{Chris}\mapsto \left\{\!\middle|\begin{matrix} \mathtt{Phone}\mapsto\mathtt{888\text{-}9999} \\ \mathtt{URL}\mapsto\mathtt{http://x.org} \end{matrix}\middle|\!\right\} \,\middle|\!\right\} \\ \left\{\!\middle|\, \mathtt{Pat}\mapsto \left\{\!\middle|\begin{matrix} \mathtt{Phone}\mapsto\mathtt{123\text{-}4321} \\ \mathtt{URL}\mapsto\mathtt{http://p2.com} \end{matrix}\middle|\!\right\} \,\middle|\!\right\} \end{array}\right]$$

which, as we saw above, can then be `flatten`ed into:

$$\left\{\!\middle|\begin{array}{l} \mathtt{Pat}\mapsto \left[\begin{array}{l} \left\{\!\middle|\begin{matrix} \mathtt{Phone}\mapsto\mathtt{333\text{-}4444} \\ \mathtt{URL}\mapsto\mathtt{http://pat.com} \end{matrix}\middle|\!\right\} \\ \left\{\!\middle|\begin{matrix} \mathtt{Phone}\mapsto\mathtt{123\text{-}4321} \\ \mathtt{URL}\mapsto\mathtt{http://p2.com} \end{matrix}\middle|\!\right\} \end{array}\right] \\ \mathtt{Chris}\mapsto \left[\left\{\!\middle|\begin{matrix} \mathtt{Phone}\mapsto\mathtt{888\text{-}9999} \\ \mathtt{URL}\mapsto\mathtt{http://x.org} \end{matrix}\middle|\!\right\}\right] \end{array}\middle|\!\right\}$$

In the type of `pivot`, we extend our conventions about values (i.e., the fact that we write k instead of $\{\!|\, k \mapsto \{\!|\}\!| \,|\!\}$) to types. If $K \subseteq \mathcal{N}$ is a set of names, then $\{\!|\, n \mapsto K \,|\!\}$ means $\{\{\!|\, n \mapsto k \,|\!\} \mid k \in K\}$—i.e., $\{\{\!|\, n \mapsto \{\!|\, k \mapsto \{\!|\}\!| \,|\!\} \,|\!\} \mid k \in K\}$.

$$(\texttt{pivot}\ n)\nearrow c\ =\ \left\{\!\!\left|k \mapsto t\right|\!\!\right\}\ \text{if } c = \left\{\!\!\left|\begin{matrix} n \mapsto k \\ t \end{matrix}\right|\!\!\right\}$$
$$(\texttt{pivot}\ n)\searrow (a,\, c)\ =\ \left\{\!\!\left|\begin{matrix} n \mapsto k \\ t \end{matrix}\right|\!\!\right\}\ \text{if } a = \left\{\!\!\left|k \mapsto t\right|\!\!\right\}$$

$$\forall n{\in}\mathcal{N}.\ \forall K{\subseteq}\mathcal{N}.\ \forall C{\subseteq}(\mathcal{T}\backslash_n).\quad \texttt{pivot}\ n \in (\left\{\!\!\left|n \mapsto K\right|\!\!\right\} \cdot C) \overset{\Omega}{\Longleftrightarrow} \{\left\{\!\!\left|k \mapsto C\right|\!\!\right\}\ |\ k \in K\}$$

Join

Our final lens combinator, based on an idea by Daniel Spoonhower [2004], is inspired by the *full outer join* operator from databases. For example, applying the *get* component of $l = (\texttt{join addr phone})$ to a tree containing a collection of addresses and a collection of phone numbers (both keyed by names) yields a tree where the address and phone information is collected under each name.

$$l\nearrow \left\{\!\!\left|\begin{matrix} \texttt{addr} \mapsto \left\{\!\!\left|\begin{matrix} \texttt{Chris} \mapsto \texttt{Paris} \\ \texttt{Kim} \mapsto \texttt{Palo Alto} \\ \texttt{Pat} \mapsto \texttt{Philadelphia} \end{matrix}\right|\!\!\right\} \\ \texttt{phone} \mapsto \left\{\!\!\left|\begin{matrix} \texttt{Chris} \mapsto \texttt{111-1111} \\ \texttt{Pat} \mapsto \texttt{222-2222} \\ \texttt{Lou} \mapsto \texttt{333-3333} \end{matrix}\right|\!\!\right\} \end{matrix}\right|\!\!\right\} = \left\{\!\!\left|\begin{matrix} \texttt{Chris} \mapsto \left\{\!\!\left|\begin{matrix} \texttt{addr} \mapsto \texttt{Paris} \\ \texttt{phone} \mapsto \texttt{111-2222} \end{matrix}\right|\!\!\right\} \\ \texttt{Kim} \mapsto \left\{\!\!\left|\texttt{addr} \mapsto \texttt{Palo Alto}\right|\!\!\right\} \\ \texttt{Pat} \mapsto \left\{\!\!\left|\begin{matrix} \texttt{addr} \mapsto \texttt{Philadelphia} \\ \texttt{phone} \mapsto \texttt{222-2222} \end{matrix}\right|\!\!\right\} \\ \texttt{Lou} \mapsto \left\{\!\!\left|\texttt{phone} \mapsto \texttt{333-3333}\right|\!\!\right\} \end{matrix}\right|\!\!\right\}.$$

Note that no information is lost in this transformation: names that are missing from either the `addr` or `phone` collection are mapped to views with just a `phone` or `addr` child. In the *putback* direction, `join` performs the reverse transformation, splitting the `addr` and `phone` information associated with each name into separate collections. (The transformation is bijective—since no information is lost by *get*, the *putback* function can ignore its concrete argument.)

$$(\texttt{join}\ m\ n)\nearrow c\ =\ \left\{\!\!\left|k \mapsto \left\{\!\!\left|\begin{matrix} m \mapsto c(m)(k) \\ n \mapsto c(n)(k) \end{matrix}\right|\!\!\right\}\ \middle|\ k \in \mathsf{dom}(c(m)) \cup \mathsf{dom}(c(n))\right|\!\!\right\}$$
$$(\texttt{join}\ m\ n)\searrow (a,\, c)\ =\ \left\{\!\!\left|\begin{matrix} m \mapsto \left\{\!\!\left|k \mapsto a(k)(m)\ |\ k \in \mathsf{dom}(a)\right|\!\!\right\} \\ n \mapsto \left\{\!\!\left|k \mapsto a(k)(n)\ |\ k \in \mathsf{dom}(a)\right|\!\!\right\} \end{matrix}\right|\!\!\right\}$$

$$\forall K{\subseteq}\mathcal{N}.\ \forall T{\subseteq}\mathcal{T}.\quad \texttt{join}\ m\ n \in \left\{\!\!\left|\begin{matrix} m \mapsto \left\{\!\!\left|K \overset{?}{\mapsto} T\right|\!\!\right\} \\ n \mapsto \left\{\!\!\left|K \overset{?}{\mapsto} T\right|\!\!\right\} \end{matrix}\right|\!\!\right\} \overset{\Omega}{\Longleftrightarrow} \left\{\!\!\left|K \overset{?}{\mapsto} \left\{\!\!\left|\begin{matrix} m \mapsto T \\ n \overset{?}{\mapsto} T \end{matrix}\right|\!\!\right\} \cup \left\{\!\!\left|\begin{matrix} m \overset{?}{\mapsto} T \\ n \mapsto T \end{matrix}\right|\!\!\right\}\right|\!\!\right\}$$

10. RELATED WORK

Our lens combinators evolved in the setting of the Harmony data synchronizer. The overall architecture of Harmony and the role of lenses in building synchronizers for various forms of data are described elsewhere [Foster et al. 2006; Pierce et al. 2003], along with a detailed discussion of related work on synchronization.

Our foundational structures—lenses and their laws—are not new: closely related structures have been studied for decades in the database community. However, our treatment of these structures is arguably simpler (transforming states rather than "update functions") and more refined (treating well-behavedness as a form of

type assertion). Our formulation is also novel in addressing the issues of totality, offering programmers a static guarantee that lenses cannot fail at run time, and of continuity, 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 about well-behavedness and totality—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 Section 3. We discuss here the main similarities and differences between our work and these classical approaches to view update—in particular Dayal and Bernstein's notion [1982] of "correct update translation," Bancilhon and Spyratos's [1981] notion of "update translation under a constant complement," Gottlob, Paolini, and Zicari's "dynamic views" [1988], and the basic view update and "relational triggers" mechanisms offered by commercial database systems such as Oracle [Fogel and Lane 2005; Lorentz 2005]

The view update problem concerns translating updates on a view into "reasonable" updates on the underlying database. It is helpful to structure the discussion by breaking this broad problem statement down into more specific questions. First, how is a "reasonable" translation of an update defined? Second, what should we do about the possibility that, for some update, there may be *no* reasonable way of translating its effect to the underlying database? And third, how do we deal with the possibility that there are *many* reasonable translations from which we must choose? We consider these questions in order.

One can imagine many possible ways of assigning a precise meaning to "reasonable update translation," but in fact there is a remarkable degree of agreement in the literature, with most approaches adopting one of two basic positions. The stricter of these is enunciated in Bancilhon and Spyratos's [1981] 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." The constant complement approach has influenced numerous later works in the area, including recent papers by Lechtenbörger [2003] and Hegner [2004].

The other, more permissive, definition of "reasonable" is elegantly formulated by Gottlob, Paolini, and Zicari, who call it "dynamic views" [1988]. They present a general framework and identify two special cases, one being formally equivalent to Bancilhon and Spyratos's constant complement translators and the other—which they advocate on pragmatic grounds—being their own dynamic views.

Our notion of lenses adopts the same, more permissive, attitude towards reasonable behavior of update translation. Indeed, modulo some technical refinements, we

have sketched that the correspondence is tight: the set of all well-behaved lenses is isomorphic to the set of dynamic views in the sense of Gottlob, Paolini, and Zicari. Moreover, the set of very well-behaved lenses is isomorphic to the set of translators under constant complement in the sense of Bancilhon and Spyratos.[9]

Dayal and Bernstein's [1982] seminal theory of "correct update translation" also adopts the more permissive position on "reasonableness." Their notion of "exactly performing an update" corresponds, intuitively, to our PUTGET law.

The pragmatic tradeoffs between these two perspectives on reasonable update translations are discussed by Hegner [1990; 2004], who introduces the term *closed view* for the stricter constant complement approach and *open view* for the looser approach adopted by dynamic views and in the present work. Hegner himself works in a closed-world framework, but notes that both choices may have pragmatic advantages in different situations, open-world being useful when the users are aware that they are actually using a view as a convenient way to edit an underlying database, while closed-world is preferable when users should be isolated from the existence of the underlying database, even at the cost of offering them a more restricted set of possible updates.

Hegner [2004] also formalizes an additional condition on reasonableness (which has also been noted by others—e.g., [Dayal and Bernstein 1982]): *monotonicity* of update translations, in the sense that an update that only adds records from the view should be translated just into additions to the database, and that an update that adds more records to the view should be translated to a larger update to the database (and similarly for deletions).

Commercial databases such as Oracle [Fogel and Lane 2005; Lorentz 2005], SQL Server [Microsoft 2005], and DB2 [International Business Machines Corporation 2004] typically provide two quite different mechanisms for updating through views. First, some very simple views—defined using select, project, and a very restricted form of join (where the key attributes in one relation are a subset of those in the other)—are considered *inherently updatable*. For these, the notion of reasonableness is essentially the constant complement position. Alternatively, programmers can support updates to arbitrary views by adding *relational triggers* that are invoked

[9]To be precise, we need an additional condition regarding partiality. The frameworks of Bacilhon and Spyratos and of Gottlob, Paolini, and Zicari are both formulated in terms of translating *update functions* on A into update functions on C, i.e., their *putback* functions have type $(A \longrightarrow A) \longrightarrow (C \longrightarrow C)$, while our lenses translate abstract *states* into update functions on C, i.e., our *putback* functions have type (isomorphic to) $A \longrightarrow (C \longrightarrow C)$. Moreover, in both of these frameworks, "update translators" (the analog of our *putback* functions) are defined only over some particular chosen set U of abstract update functions, not over all functions from A to A, and these update functions may be composed. Update translators return *total* functions from C to C. Our *putback* functions, on the other hand, are slightly more general as they 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 and the set of dynamic views should already include every abstract update functions that are needed and not rely on composition.

A related observation is that, if we restrict both *get* and *putback* to be total functions (i.e., *putback* must be total with respect to *all* abstract update functions), then our lens laws (including PUTPUT) characterize the set C as isomorphic to $A \times B$ for some B.

whenever an update is attempted on the view and that can execute arbitrary code to update the underlying database. In this case, the notion of reasonableness is left entirely to the programmer.

The second question posed at the beginning of the section was how to deal with the possibility that there are no reasonable translations for some update. The simplest response is just to let the translation of an update fail, if it sees that its effect is going to be unreasonable; this is Dayal and Bernstein's approach, for example. Its advantage is that we can determine reasonableness on a case-by-case basis, allowing translations that usually give reasonable results but that might fail under rare conditions. The disadvantage is that we lose the ability to perform updates to the view offline—we need the concrete database in order to tell whether an update is going to be allowed. Another possibility is to restrict the set of operations to just the ones that can be guaranteed to correspond to reasonable translations; this is the position taken by most papers in the area. A different approach—the one we have taken in this work—is to restrict the view schema so that *arbitrary* (schema-respecting) updates are guaranteed to make sense.

The third question posed above was how to deal with the possibility that there may be multiple reasonable translations for a given update.

One attractive idea is to somehow restrict the set of reasonable translations so that this possibility does not arise—i.e., so that every translatable update has a unique translation. For example, under the constant complement approach, for a particular choice of complement, there will be at most one translation. Hegner's additional condition of monotonicity [2004] ensures that (at least for updates consisting of only inserts or only deletes), the translation of an update is unique, independent of the choice of complement.

Another possibility is to place an ordering on possible translations of a given update and choose one that is minimal in this ordering. This idea plays a central role, for example, in Johnson, Rosebrugh, and Dampney's account of view update in the Sketch Data Model [2001]. Buneman, Khanna, and Tan [2002] have 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.

The key idea in the present work is to allow the programmer to describe the update policy at the same time as the view definition, by enriching the relational primitives with enough annotations to select among a variety of reasonable update policies.

In the literature on programming languages, laws similar to our lens laws (but somewhat simpler, dealing only with total *get* and *putback* functions) appear in Oles' category of "state shapes" [Oles 1985] and in Hofmann and Pierce's work on "positive subtyping" [1995].

Languages for Bi-Directional Transformations

At the level of syntax, different forms of bi-directional programming have been explored across a surprisingly diverse range of communities, including programming languages, databases, program transformation, constraint-based user interfaces, and quantum computing. One useful way of classifying these languages is by

the "shape" of the semantic space in which their transformations live. We identify three major classes:

—*Bi-directional languages*, including ours, form lenses by pairing a *get* function of type $C \to A$ with a *putback* function of type $A \times C \to C$. In general, the *get* function can project away some information from the concrete view, which must then be restored by the *putback* function.

—In *bijective languages*, the *putback* function has the simpler type $A \to C$, being given no concrete argument to refer to. To avoid loss of information, the *get* and *putback* functions must form a (perhaps partial) bijection between C and A.

—*Reversible languages* go a step further, demanding only that the work performed by any function to produce a given output can be undone by applying the function "in reverse" working backwards from this output to produce the original input. Here, there is no separate *putback* function at all: instead, the *get* function itself is constructed so that each step can be run in reverse.

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 [1998]. Meertens's semantic setting is actually even more general: he takes *get* and *putback* to be *relations*, not just functions, and his constraint maintainers are symmetric: *get* relates pairs from $C \times A$ to elements of A and *putback* relates pairs in $A \times C$ to elements of C; 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 [2004a] 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 *putback*. Similarly, they assume that edits to the abstract view mark all modified fields as "updated." These marks are removed when the *putback* 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).

Another paper by Hu, Mu, and Takeichi [2004] applies a bi-directional programming language quite closely related to ours to the design of "programmable editors" for structured documents. As in [Mu et al. 2004a], they support preservation of local invariants in the *putback* direction. Here, instead of annotating the abstract view with modification marks, they assume that a *putback* or a *get* occurs after *every* modification to either view. They use this "only one update" assumption to choose the correct inverse for the lens that copied data in the *get* direction—because only one branch can have been modified at any given time. Consequently, they can *putback* the data from the modified branch and overwrite the unmodified branch. Here, too, the notion of well-behavedness needs to be weakened, as described in Section 5.

The TRIP2 system (e.g., [Matsuoka et al. 1992]) uses bidirectional transformations specified as collections of Prolog rules as a means of implementing direct-manipulation interfaces for application data structures. The *get* and *putback* components of these mappings are written separately by the user.

Languages for Bijective Transformations

An active thread of work in the program transformation community concerns *program inversion* and *inverse computation*—see, for example, Abramov and Glück [2000; 2002] and many other papers cited there. Program inversion [Dijkstra 1979] derives the inverse program from the forward program. Inverse computation [McCarthy 1956] computes a possible input of a program from a particular output. One approach to inverse computation is to design languages that produce easily invertible expressions—for example, languages that can only express injective functions, where every program is trivially invertible.

In the database community, Abiteboul, Cluet, and Milo [1997] 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 [1998] later defined a system for bi-directional transformations based around the concept of *structuring schemas* (parse grammars annotated with semantic information). Thus their *get* functions involved parsing, whereas their *putback*s consisted of unparsing. Again, to avoid ambiguous abstract updates, they restricted themselves to *lossless* grammars that define an isomorphism between concrete and abstract views.

Ohori and Tajima [1994] 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* [Wadler

1987], 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 *putback* functions are defined should be large enough for some given purpose. In particular, we expect that *putback* functions should 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 *putback* 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 [1961] observed that non-injective functions were logically irreversible, and that this irreversibility requires the generation and dissipation of some heat per machine cycle. Bennet [1973] demonstrated that this irreversibility was not inevitable by constructing a *reversible Turing machine*, showing that thermodynamically reversible computers were plausible. Baker [1992] 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. [Mu et al. 2004b]).

Update Translation for Tree Views

There have been many proposals for query languages for trees (e.g., XQuery [XQuery 2005] 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, Heuser, and Vittori [2001], and Braganholo, Davidson, and Heuser [2003] 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 [2001] 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 [1991] restrict the notion of views to queries

that preserve object identity. The view update problem is greatly simplified in this setting, as the objects contained in the view are the objects of the database, and an update on the view is directly an update on objects of the database.

Update Translation for Relational Views

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

Recent work by Bohannon, Pierce, and Vaughan [2006] extends the framework presented here to obtain lenses that operate natively on relational data. Their lenses are based on the primitives of classical relational algebra, with additional annotations that specify the desired "update policy" in the *putback* direction. They develop a type system, using record predicates and functional dependencies, to aid compositional reasoning about well-behavedness. The chapter on view update in Date's textbook [Date 2003] articulates a similar perspective on translating relational updates.

Masunaga [1984] 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 [1985] 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. These criteria are:

(1) No database side effects: only update tuples in the underlying database that appear somehow in the view.
(2) Only one-step changes: each underlying tuple is updated at most once.
(3) No unnecessary changes: there is no operationally equivalent translation that performs a proper subset of the translated actions.
(4) Replacements cannot be simplified (e.g., to avoid changing the key, or to avoid changing as many attributes).
(5) No delete-insert pairs: for any relation, you have deletions or insertions, but not both (use replacements instead).

These criteria apply to *update* translations on relational databases, whereas our state-based approach means only criteria (1), (3), and (4) might apply to us. Keller later [1986] proposed allowing users to choose an update translator at view definition time by engaging in an interactive dialog with the system and answering questions about potential sources of ambiguity in update translation. Building on this foundation, Barsalou, Siambela, Keller, and Wiederhold [1991] described a

scheme for interactively constructing update translators for object-based views of relational databases.

Medeiros and Tompa [1985] 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. Miller *et al.* [2001] describe Clio, a system for managing heterogeneous transformation and integration. Clio provides a tool for visualizing two schemas, specifying correspondences between fields, defining a mapping between the schemas, and viewing sample query results. They only consider the *get* direction of our lenses, but their system is somewhat mapping-agnostic, so it might eventually be possible to use a framework like Clio as a user interface supporting incremental lens programming like that in Figure 8.

Atzeni and Torlone [1997; 1996] 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 [Cosmadakis 1983; Cosmadakis and Papadimitriou 1984] 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 [2002] 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.

The designers of the RIGEL language [Rowe and Schoens 1979] argued that programmers should specify the translations of abstract updates. However, they did not provide a way to ensure consistency between the *get* and *putback* directions of their translations.

Another problem that is sometimes mentioned in connection with view update translation is that of *incremental view maintenance* (e.g., [Abiteboul et al. 1998])—efficiently recalculating an abstract view after a small update to the underlying concrete view. Although the phrase "view update problem" is sometimes, confusingly, used for work in this domain, there is little technical connection with the problem of translating view updates to updates on an underlying concrete structure.

11. CONCLUSIONS AND ONGOING WORK

We have worked to design a collection of combinators that fit together in a sensible way and that are easy to program with and reason about. Starting with lens laws that define "reasonable behavior," adding type annotations, and proving that each of our lenses is total, has imposed strong constraints on our design of new lenses—constraints that, paradoxically, make the design process easier. In the early stages of the Harmony project, working in an under-constrained design space, we found

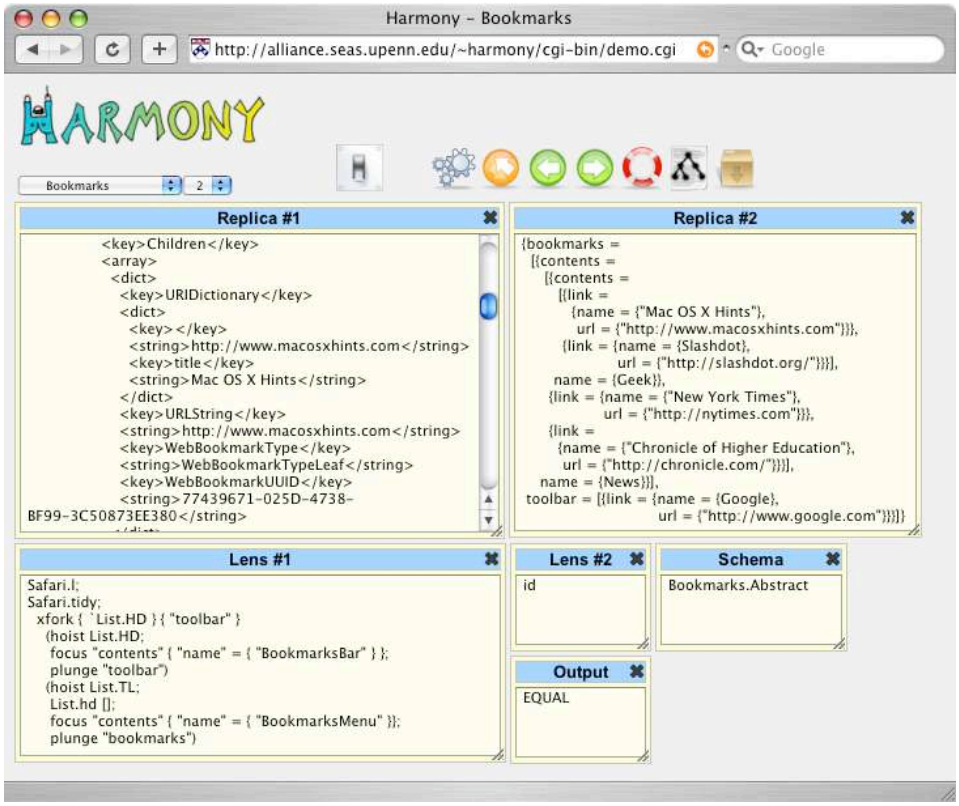


Fig. 10. Web demo of Safari Bookmark lens

it extremely difficult to converge on a useful set of primitive lenses. Later, when we understood how to impose the framework of type declarations and the demand for compositional reasoning, we experienced a huge increase in manageability. The types helped not just in finding programming errors in derived lenses, but in exposing design mistakes in the primitives at an early stage.

Our interest in bi-directional tree transformations arose in the context of the Harmony data synchronization framework. Besides the bookmark synchronizer described in Section 8, we have developed prototype synchronizers for calendars, address books, and structured text, as well as a growing library of lens programs. Building implementations continues to provide valuable stress-testing for both our combinators and their formal foundations. It also gives us confidence that our lenses are practically useful.

The source code for each of these prototypes, along with our lens compiler and synchronization engine, can be found on the Harmony web page [Pierce et al. 2006]. We have also made the system available as an online web demo (a screenshot from the Safari component of our bookmarks portion of this demo is shown in Figure 10).

Static Analysis

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. Our compiler has the ability to perform simple run-time checking and some debugging using probe points and to track stack frames. These simple dynamic techniques have proven helpful in developing and debugging small-to-medium sized lens programs, but we would like to be able to reason statically that a given program is type correct. Fortunately, 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 statically checking (or, more ambitiously, inferring) types.

In the semantic framework of lens types we have developed, the key properties tracked by the types are well-behavedness and totality. However, there are other properties of lenses that one might want to track in a type system including very well behavedness, obliviousness, adherence to the conventions about Ω, etc. Moreover, there is a natural subsumption relation between these different lens types: e.g., every oblivious lens is very well behaved. Once basic mechanized type checking for lenses is in place, the natural next step is to stratify the type system to facilitate reasoning about these more refined properties of lenses.

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.)

Optimization

This algebraic theory will play a crucial role in any serious effort to compile lens programs efficiently. 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 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.

Additional Combinators

Another area for further investigation is the design of additional combinators. While we have found the ones we have described here to be expressive enough to code a large number of examples—both intricate structural manipulations such as the list transformations in Section 7 and more prosaic application transformations such as the ones needed by the bookmark synchronizer in Section 8 —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 transforma-

tions on trees (including analogs of database operations such as `join`), or lenses addressing completely different sorts of transformations (e.g., none of our combinators do any significant processing on edge labels, which might include string processing, arithmetic, etc.). Higher-level combinators embodying more global transformations on trees—perhaps modeled on a familiar tree transformation notation such as XSLT—are another interesting possibility.

Finally, we would also like to investigate recursion combinators that are less powerful than *fix*, but that come equipped with simpler principles for reasoning about totality. We already have one such combinator: `map` iterates over the *width* of the tree. However, we think it should be possible to go further; e.g., one could define lenses by structural recursion on trees.

Expressiveness

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

Recently, we have been exploring bidirectional transformations expressed as word and tree transducers.

Lens Inference

In restricted cases, it may be possible to build lenses in simpler ways than by explicit programming—e.g., by generating them automatically from schemas for concrete and abstract views, or by inference from a set of pairs of inputs and desired outputs ("programming by example"). Such a facility might be used to do part of the work for a programmer wanting to add synchronization support for a new application (where the abstract schema is already known, for example), leaving just a few spots to fill in.

Acknowledgements

The Harmony project was begun in collaboration with Zhe Yang; Zhe contributed numerous insights whose generic material can be found (in much-recombined form) in this paper. Owen Gunden,, Malo Denielou, and Stéphane Lescuyer, 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, Keshav Kunal, 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, Norman Ramsey, Wang-Chiew Tan, Stephen Tse, Zhe Yang, and several anonymous referees for helpful commentson earlier drafts of this paper.

The Harmony project is supported by the National Science Foundation under grant ITR-0113226, *Principles and Practice of Synchronization*. Nathan Foster is also supported by an NSF Graduate Research Fellowship.

REFERENCES

Abiteboul, S., Cluet, S., and Milo, T. 1997. Correspondence and translation for heterogeneous data. In *International Conference on Database Theory (ICDT), Delphi, Greece*.

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

Abiteboul, S., McHugh, J., Rys, M., Vassalos, V., and Wiener, J. L. 1998. Incremental maintenance for materialized views over semistructured data. In *Proc. 24th Int. Conf. Very Large Data Bases (VLDB)*.

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

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

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

Atzeni, P. and Torlone, R. 1997. MDM: a multiple-data model tool for the management of heterogeneous database schemes. In *Proceedings of ACM SIGMOD, Exhibition Section*. 528–531.

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

Bancilhon, F. and Spyratos, N. 1981. Update semantics of relational views. *ACM Transactions on Database Systems 6,* 4 (Dec.), 557–575.

Barsalou, T., Siambela, N., Keller, A. M., and Wiederhold, G. 1991. Updating relational databases through object-based views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, Denver, Colorado*. 248–257.

Bennet, C. H. 1973. Logical reversibility of computation. *IBM Journal of Research and Development 17,* 6, 525–532.

Bohannon, A., Vaughan, J. A., and Pierce, B. C. 2006. Relational lenses: A language for updateable views. In *Principles of Database Systems (PODS)*. Extended version available as University of Pennsylvania technical report MS-CIS-05-27.

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

Buneman, P., Khanna, S., and Tan, W.-C. 2002. On propagation of deletions and annotations through views. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, Madison, Wisconsin*. 150–158.

Cosmadakis, S. S. 1983. Translating updates of relational data base views. M.S. thesis, Massachusetts Institute of Technology. MIT-LCS-TR-284.

Cosmadakis, S. S. and Papadimitriou, C. H. 1984. Updates of relational views. *Journal of the ACM 31,* 4, 742–760.

Date, C. J. 2003. *An Introduction to Database Systems (Eighth Edition)*. Addison Wesley.

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

de Paula Braganholo, V., Heuser, C. A., and Vittori, C. R. M. 2001. Updating relational databases through XML views. In *Proc. 3rd Int. Conf. on Information Integration and Web-based Applications and Services (IIWAS)*.

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

Fogel, S. and Lane, P. 2005. *Oracle Database Administrator's Guide*. Oracle.

Foster, J. N., Greenwald, M. B., Kirkegaard, C., Pierce, B. C., and Schmitt, A. 2006. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*. To appear. Extended abstract in *Database Programming Languages (DBPL)* 2005.

Foster, J. N., Pierce, B. C., and Schmitt, A. 2006. *Harmony Programmer's Manual*. Available from `http://www.seas.upenn.edu/~harmony/`.

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

Hegner, S. J. 1990. Foundations of canonical update support for closed database views. In *International Conference on Database Theory (ICDT), Paris, France*. Springer-Verlag New York, Inc., New York, NY, USA, 422–436.

Hegner, S. J. 2004. An order-based theory of updates for closed database views. *Annals of Mathematics and Artificial Intelligence 40*, 63–125. Summary in *Foundations of Information and Knowledge Systems*, 2002, pp. 230–249.

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

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

International Business Machines Corporation 2004. *IBM DB2 Universal Database Administration Guide: Implementation*. International Business Machines Corporation.

Johnson, M., Rosebrugh, R., and Dampney, C. N. G. 2001. View updates in a semantic data modelling paradigm. In *ADC '01: Proceedings of the 12th Australasian conference on Database technologies*. IEEE Computer Society, Washington, DC, USA, 29–36.

Keller, A. M. 1985. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *ACM SIGACT–SIGMOD Symposium on Principles of Database Systems, Portland, Oregon*.

Keller, A. M. 1986. Choosing a view update translator by dialog at view definition time. In *VLDB'86*.

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

Lechtenbörger, J. 2003. The impact of the constant complement approach towards view updating. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, San Diego, California*. ACM, 49–55.

Lorentz, D. 2005. *Oracle Database SQL Reference*. Oracle.

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

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

McCarthy, J. 1956. The inversion of functions defined by turing machines. In *Automata Studies, Annals of Mathematical Studies*, C. E. Shannon and J. McCarthy, Eds. Number 34. Princeton University Press, 177–181.

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

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

Microsoft 2005. *Creating and Maintaining Databases*. Microsoft.

Miller, R. J., Hernandez, M. A., Haas, L. M., Yan, L., Ho, C. T. H., Fagin, R., and Popa, L. 2001. The clio project: Managing heterogeneity. *30,* 1 (March), 78–83.

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

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

Niehren, J. and Podelski, A. 1993. Feature automata and recognizable sets of feature trees. In *TAPSOFT*. 356–375.

Ohori, A. and Tajima, K. 1994. A polymorphic calculus for views and object sharing. In *ACM SIGACT–SIGMOD–SIGART Symposium on Principles of Database Systems, Minneapolis, Minnesota*.

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

Pierce, B. C. et al. 2006. Harmony: A synchronization framework for heterogeneous tree-structured data. `http://www.seas.upenn.edu/~harmony/`.

Pierce, B. C., Schmitt, A., and Greenwald, M. B. 2003. Bringing Harmony to optimism: A synchronization framework for heterogeneous tree-structured data. Technical Report MS-CIS-03-42, University of Pennsylvania. Superseded by MS-CIS-05-02.

Pierce, B. C. and Vouillon, J. 2004. What's in Unison? A formal specification and reference implementation of a file synchronizer. Tech. Rep. MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania.

Rowe, L. and Schoens, K. A. 1979. Data abstractions, views, and updates in RIGEL. In *ACM SIGMOD Symposium on Management of Data (SIGMOD), Boston, Massachusetts*.

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

Spoonhower, D. 2004. View updates seen through the lens of synchronization. Manuscript.

Tatarinov, I., Ives, Z. G., Halevy, A. Y., and Weld, D. S. 2001. Updating XML. In *ACM SIGMOD Symposium on Management of Data (SIGMOD), Santa Barbara, California*.

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

Winskel, G. 1993. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press.

XQuery 2005. XQuery 1.0: An XML Query Language, W3C Working Draft. `http://www.w3.org/TR/xquery/`.

toplas available only online. You should be able to get the online-only XXXX from the citation page for this article:

YYYY

Alternative instructions on how to obtain online-only appendices are given on the back inside cover of current issues of ACM TOPLAS or on the ACM TOPLAS web page:

http://www.acm.org/toplas YYYY