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

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

Technical Report MS-CIS-04-15
Department of Computer and Information Science
University of Pennsylvania

August 7, 2004

Supercedes MS-CIS-03-08

Abstract

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

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

1 Introduction

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

This paper addresses a specific instance of the view update problem that arises in a larger project called Harmony [38]. 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 that is 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 disparate data formats we define a single common abstract view and a collection of *lenses* that transform each concrete format into this abstract view. For example, we can synchronize a Mozilla bookmark file with an Internet Explorer bookmark file by transforming each into an *abstract bookmark structure* and synchronizing the results. Having done so, we need to take the updated abstract structures and perform the corresponding updates to the concrete structures. Thus, each lens must include not one but *two* functions—one for extracting an abstract view from a concrete one and another for pushing an updated abstract view back into the original concrete view to yield an updated concrete view. We call these the *get* and *put* components, respectively. The intuition is that the mapping from concrete to abstract is commonly some sort of projection, so the *get* direction involves getting the abstract part out of a larger concrete structure, while the *put* direction amounts to putting a new abstract part into an old concrete structure. We present a concrete example 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 *put* direction in such a way that each lens is both *well behaved*, in the sense that its *get* and *put* behaviors fit together in a sensible way, and *total*, in the sense that its *get* and *put* functions are guaranteed to be defined on all the inputs to which they may be applied.

To reconcile this tension, any approach to the view update problem must be carefully designed with a particular application domain in mind. The approach described here is tuned to the kinds of projection-and-rearrangement transformations on trees and lists that we have found useful for implementing Harmony instances. It does not directly address some well-known difficulties with view update in the classical setting of relational databases—such as the difficulty of "inverting" queries involving joins—though we hope that our work may suggest new attacks on these problems.

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

The first step in our formal development, in 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 abstract level, before we fix the domain of structures being transformed or the syntax for writing down transformations. First, we must phrase our basic definitions to allow the underlying functions in lenses to be partial, since there will in general be structures to which a given lens cannot sensibly be applied. The

sets of structures to which we *do* intend to apply a given lens is specified by associating it with a type of the form $C \rightleftharpoons A$, where C is a set of concrete "source structures" and A is a set of abstract "target structures." Second, we define a notion of well-behavedness that captures our intuitions about how the *get* and *put* parts of a lens should behave in concert. (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 *put* part to push the very same a back into c, we should get c back.) Third, we use 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 (which we need 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 (which can happen, for example, when new records are added to a database in the abstract view), we show how to adjoin a special "missing" element to the structures manipulated by lenses and establish suitable conventions for how it is treated.

With these semantic foundations in place, we proceed to syntax. We first (Section 4), present a group of generic lens combinators (identities, composition, and constants), which can work with any kind of data. Next (Section 5) we focus attention on tree-structured data and present several more combinators that perform various manipulations on trees (hoisting, splitting, mapping, etc.) and show how to assemble these primitives, along with the generic combinators from before, to yield some useful derived forms. Section 6 introduces another set of generic combinators implementing various sorts of bi-directional conditionals (we defer these to a separate section for the sake of getting to concrete examples early, and because they are among our trickier primitives). 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; our main example is a bi-directional `list_filter` lens whose *put* direction must perform a rather intricate "weaving" operation to recombine a potentially updated abstract list with the concrete list elements that were filtered away by the *get*. 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 a variety of related work and states a precise correspondence (amplified in [37]) between our well-behaved lenses and the closely related idea of "update translation under a constant complement" from databases. Section 11 sketches directions for future research.

2 A Small Example

Suppose our concrete tree c is a small address book:

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

(We draw trees sideways to save space. Each set of hollow curly braces corresponds to a tree node, and each "`X` $\mapsto$ ..." inside 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 simplified format, where each

name is associated directly with a phone number.

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

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

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

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

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

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

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

3 Semantic Foundations

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

3.1 Basic Structures

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

[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 [39].

[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.

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

The intuition behind the notations $l\nearrow$ and $l\searrow$ is that the *get* part of a lens "lifts" an abstract view out of a concrete one, while the *put* part "pushes down" a new abstract view into an existing concrete view. We often say "put a into c [using l]" instead of "apply the *put* function [of l] to (a, c)."

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

$$l\nearrow(C) \subseteq A \qquad \text{(GET)}$$
$$l\searrow(A \times C) \subseteq C \qquad \text{(PUT)}$$

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

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

We call C the *source* and A the *target* in $C \rightleftharpoons A$. 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 $\emptyset$. Conversely, 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 *put* a (with no modifications) back into c, we must get back exactly c (if both operations are defined). PUTGET, on the other hand, demands that the *put* function must capture all of the information contained in the abstract view: if putting a view a into a concrete view c yields a view c', then the abstract view obtained from c' is exactly a.

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

$$\begin{aligned} l\nearrow(s, n) &= s \\ l\searrow(s',\, (s, n)) &= (s', 0) \end{aligned}$$

Then $l\searrow(l\nearrow(s, 1),\, (s, 1)) = (s, 0) \neq (s, 1)$. Intuitively, the law fails because the *put* function has "side effects": it modifies information from the concrete view that is not reflected in the abstract view.

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

$$\begin{aligned} l\nearrow s &= (s, 0) \\ l\searrow((s', n),\, s) &= s' \end{aligned}$$

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

The GETPUT and PUTGET laws reflect fundamental expectations about the behavior of lenses; removing either law significantly weakens the semantic foundation. We may also optionally consider a 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 *put*s is (modulo undefinedness) just the effect of the second: the first gets completely overwritten. We say that a well-behaved lens that also satisfies PUTPUT is *very well behaved*. Both well-behaved and very well behaved lenses correspond to well-known classes of "update translators" from the classical database literature (see Section 10).

The PUTPUT law intuitively states that a series of changes to an abstract view may be applied either incrementally or all at once, resulting in the same final concrete view. This is a natural constraint, and the

foundational development in this section is valid for both well-behaved and very well behaved variants of lenses. However, when we come to defining our lens combinators for tree transformations in Section 5, we will not require PUTPUT because one of our most important lens combinators, `map`, fails to satisfy it for reasons that seem to us pragmatically unavoidable (see Section 5.4).

For now, a very simple example of a lens that is well behaved but not very well behaved can be constructed 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.

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

The *get* function of l projects away the version number and yields just the "data part." The *put* 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 $l\searrow(s,\,l\searrow(s',\,(c,n))) = (s,n+2) \neq (s,n+1) = l\searrow(s,\,(c,n))$.

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

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

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

This is consistent with the pragmatic intuition that we always want our lenses to be defined on the whole of the domains where we intend to use them: the *get* direction should be defined for any structure in the concrete set, and the *put* direction should be capable of putting back any possible updated version from the abstract set. (Since we intend to use lenses to build synchronizers, the updated structures here will be the results of synchronization. But 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 of synchronization will be to propagate all these changes to the other replica. This implies that the *put* function in the lens associated with the other replica must be prepared to accept any value from the abstract domain.[3]) However, totality of lenses—like totality of ordinary recursive functions or termination of while loops—is more difficult to reason about than simple well-behavedness, requiring invention of global termination measures, in contrast to the purely local reasoning used to show well-behavedness. This is why we formulate it as a separate condition rather than building it into the definition of well-behavedness. We expect that, in practice, programmers (or, someday, a mechanical type checker) will always prove that their lenses are well behaved—i.e., that they may diverge but will never terminate and yield wrong results—but that totality will be dealt with in a more rough and ready way (as it is in most real-world functional programming) by a combination of intuition, informal proofs, and testing.

Note, too, that totality is trivial if we do not care about the size of the source and target sets: in particular, *every* lens is total if we take the source and target to be empty. It becomes interesting only when we have larger domains in mind.

3.2 Basic Properties

We now explore some simple but useful consequences of the lens laws.

3.2.1 Definition: Let f be a partial function from $A \times C$ to C and $P \subseteq A \times C$. We say that f is *injective on* P if it is injective (in the standard sense) in the first component of arguments drawn from P—i.e., if,

[3] In other settings, different notions of totality may be appropriate. For example, Hu, Mu, and Takeichi [21] 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 $l\nearrow c$.

for all views a, a', 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 $f(a,c) \neq f(a',c)$.

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

Proof: Let $P = \{(a,c) \mid (a,c) \in A \times C \ \wedge \ l\nearrow(l\searrow(a,c)) \downarrow\}$, and choose $(a,c) \in P$ and $(a',c) \in P$ with $a' \neq a$. Suppose, for a contradiction, that $l\searrow(a,c) = l\searrow(a',c)$. Then, by the definition of P and rule PUTGET, we have $a = l\nearrow l\searrow(a,c) = l\nearrow l\searrow(a',c) = a'$; hence $a = a'$, a contradiction. □

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

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

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

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

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

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

Proof: Straightforward. □

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

Proof: If $C = \emptyset$, then, because l is total, A is also empty and $l\nearrow$ is trivially bijective. If C is non-empty, then we can choose an arbitrary $c \in C$ and define the inverse of $l\nearrow$ as $f = \lambda a.\ l\searrow(a,c)$. The fact that $(l\nearrow) \circ f = id$ follows directly from PUTGET. The fact that $f \circ (l\nearrow) = id$ follows because $f(l\nearrow c') = l\searrow(\nearrow c', c) = l\searrow(\nearrow c', c')$ (by obliviousness) $= c'$ (by GETPUT). □

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

3.3 Recursion

Since our lens framework will be instantiated for the universe of trees, and since trees in many interesting application domains may have unbounded depth (e.g., a bookmark item can be either a link or a folder containing a list of bookmark items), we will need to define lenses by recursion. Our next task in this foundational section is to set up the necessary structure for interpreting such definitions.

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

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

3.3.1 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 $(a,c) \in \mathsf{dom}(l\searrow)$.

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

Proof: Straightforward from the definitions. □

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 as $\bigsqcup_n l_n$) 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, $\bot$ is the lens whose *get* and *put* functions are everywhere undefined.

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

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

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

Proof: We first check that l is a lens, i.e., that both $l\searrow$ and $l\nearrow$ are functions. This is easy since, by definition of the ordering on lenses, we have $l_i\searrow(a,\,c) = v \implies \forall j \geq i.\ l_j\searrow(a,\,c) = v$, and the same for $l\nearrow$. Moreover, $\mathsf{dom}(l\nearrow) = \bigcup_i \mathsf{dom}(l_i\nearrow)$ and $\mathsf{dom}(l\searrow) = \bigcup_i \mathsf{dom}(l_i\searrow)$.

We now show that l is a least upper bound. First, it is clearly an upper bound. To show it is least, let l' be another upper bound. Then, for all i, we have $\mathsf{dom}(l_i\nearrow) \subseteq \mathsf{dom}(l'\nearrow)$ and $\mathsf{dom}(l_i\searrow) \subseteq \mathsf{dom}(l'\searrow)$; hence $\mathsf{dom}(l\nearrow) \subseteq \mathsf{dom}(l'\nearrow)$ and $\mathsf{dom}(l\searrow) \subseteq \mathsf{dom}(l'\searrow)$. Moreover, if $c \in \mathsf{dom}(l\nearrow)$, then there is some i such that $l_i\nearrow c\downarrow$ and $l\nearrow c = l_i\nearrow c$; thus (as l' is an upper bound), we have $l'\nearrow c = l_i\nearrow c = l\nearrow c$. The same property holds for the *put* function, so $l \prec l'$ and l is indeed a least upper bound. □

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

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

3.3.5 Lemma: Let $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ be an increasing chain of lenses, and let $C_0 \subseteq C_1 \subseteq \ldots$ and $A_0 \subseteq A_1 \subseteq \ldots$ be increasing chains of subsets of $\mathcal{U}$. Then:

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

Proof: Let $l = \bigsqcup_n l_n$, let $C = \bigcup_i C_i$, and let $A = \bigcup_i A_i$.

We rely on the following property (which we call $\star_g$): if $l\nearrow c$ is defined for some $c \in C$, then there is some i such that $c \in C_i$ and $l\nearrow c = l_i\nearrow c$. To see this, let $c \in C$; then there is some j such that $\forall k \geq j.\ c \in C_k$. Moreover, by Corollary 3.3.4, there exist some j' such that $l\nearrow c = l_{j'}\nearrow c$. Let i be the max of j and j'; then we have (by definition of $\prec$) $l_i\nearrow c = l_{j'}\nearrow c = l\nearrow c$ and $c \in C_i$.

Similarly, we have the property $\star_p$: if $l\searrow(a,\,c)$ is defined for some $a \in A$ and $c \in C$, then there is some i such that $a \in A_i$, $c \in C_i$, and $l\searrow(a,\,c) = l_i\searrow(a,\,c)$. To see this, let $a \in A$ and $c \in C$; then there are some j and j' such that $\forall k \geq j.\ a \in A_k$ and $\forall k \geq j'.\ c \in C_k$. Moreover, by Corollary 3.3.4, there exists some j'' such that $l\searrow(a,\,c) = l_{j''}\searrow(a,\,c)$. Let i be the max of j, j', and j''; then we have (by definition of $\prec$) $l_i\searrow(a,\,c) = l_{j''}\searrow(a,\,c) = l\searrow(a,\,c)$, with $a \in A_i$ and $c \in C_i$.

We can now show that l satisfies the typing conditions (GET and PUT) of well-behaved lenses. Choose $c \in C$. If $l\nearrow c$ is defined, then by $\star_g$ there is some i such that $c \in C_i$ and $l\nearrow c = l_i\nearrow c$. As l_i is in $A_i \rightleftharpoons C_i$, we have $l_i\nearrow c \in A_i \subseteq A$. Conversely, let $(a, c) \in A \times C$; then if $l\searrow(a,\,c)$ is defined, then by $\star_p$ there is some i such that $(a, c) \in A_i \times C_i$ and $l\searrow(a,\,c) = l_i\searrow(a,\,c)$. As $l_i \in A_i \rightleftharpoons C_i$, we have $l_i\searrow(a,\,c) \in C_i \subseteq C$.

We next show that l satisfies GETPUT and PUTGET. Using $\star_g$ and $\star_p$, we calculate as follows:

GetPut: Suppose $c \in C$. If $l \searrow (l \nearrow c,\, c) = \bot$, then we are done. Otherwise there is some i such that $c \in C_i$ and $l_i \nearrow c = l \nearrow c = a \in A_i \subseteq A$. Hence there is some j such that $a \in A_j$ and $l_j \searrow (a,\, c) = c'$. Let k be the max of i and j, so we have $a \in A_k$ and $c \in C_k$. By definition of $\prec$, we have $l_k \nearrow c = a$ and $l_k \searrow (a,\, c) = c'$. As GetPut holds for l_k, we have $c' = c$, hence GetPut holds for l.

PutGet: Suppose $a \in A$ and $c \in C$. If $l \nearrow l \searrow (a,\, c) = \bot$, then we are done. Otherwise there is some i such that $a \in A_i$, $c \in C_i$, and $l_i \searrow (a,\, c) = l \searrow (a,\, c) = c' \in C_i \subseteq C$. Hence there is some j such that $c' \in C_j$ and $l_j \nearrow c' = a'$. Let k be the max of i and j, so we have $a \in A_k$ and $c \in C_k$. By definition of $\prec$, we have $l_k \searrow (a,\, c) = c'$ and $l_k \nearrow c' = a'$. As PutGet holds for l_k, we have $a' = a$, hence PutGet holds for l.

Finally, we show that l is total if all the l_i are. If $c \in C$, then there is some i such that $c \in C_i$, hence $l_i \nearrow c$ is defined, hence $l \nearrow c$ is defined. If $a \in A$ and $c \in C$, then there is some i such that $a \in A_i$ and $c \in C_i$, hence $l_i \searrow (a,\, c)$ is defined, thus $l \searrow (a,\, c)$ is defined. □

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

Proof: First, the lens that is undefined everywhere is well behaved (it trivially satisfies all equations) and is obviously the smallest lens. We write this lens $\bot_l$. Second, if $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ is an increasing chain of well-behaved lenses, then by Lemma 3.3.5, it has a least upper bound that is well behaved. □

When defining lenses, we will make heavy use of the following standard theorem from domain theory (e.g., [46]). 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 $\mathit{fix}(f) = f(\mathit{fix}(f))$.

3.3.7 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 of f.

Theorem 3.3.6 tells us that we can apply Theorem 3.3.7 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.

3.3.8 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 $\mathcal{U}$. 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 $\mathit{fix}(f) \in (\bigcup_i C_i) \Longleftrightarrow (\bigcup_i A_i)$.

Proof:

1. First recall that $f^0(\bot_l) = \bot_l \in C \rightleftharpoons A$ for any C and A. From this, a simple induction on i (using the given implication at each step) yields $f^i(\bot_l) \in C \rightleftharpoons A$. By 3.3.5(1), $(\bigsqcup_i f^i(\bot_l)) \in C \rightleftharpoons A$. By 3.3.7, $\mathit{fix}(f) \in C \rightleftharpoons A$.
2. First note that, since $C_0 = A_0 = \emptyset$, we have $f^0(\bot_l) = \bot_l \in C_0 \Longleftrightarrow A_0$. From this, a simple induction on i (using the given implication at each step) yields $f^i(\bot_l) \in C_i \Longleftrightarrow A_i$. By 3.3.5(2), $(\bigsqcup_i f^i(\bot_l)) \in (\bigcup_i C_i) \Longleftrightarrow (\bigcup_i A_i)$. By 3.3.7, $\mathit{fix}(f) \in (\bigcup_i C_i) \Longleftrightarrow (\bigcup_i A_i)$. □

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 3.3.8(2).

We specify a *total type* by a pair (C, A) of subsets of $\mathcal{U}$, and say that a lens l has this type, written $l \in (C, A)$ iff $l \in C \iff A$. We use the variable τ for 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 $(C \cup C', A \cup A')$.

3.3.9 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 for all i we have $\tau_i \in \mathbb{T}_i$.

Note that $\mathbb{T}_0, \mathbb{T}_1, \ldots$ is an arbitrary sequence of total types, here—there is no requirement that the sequence 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 Lemma 3.3.5(2) to the case where lenses may be given multiple types.

3.3.10 Lemma: Let $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ be an increasing chain of lenses, and let $\mathbb{T}_0, \mathbb{T}_1, \ldots$ be a sequence of sets of total types, such that for all $\tau_i \in \mathbb{T}_i$ we have $l_i \in \tau_i$. Then for any increasing instance $\tau_0 \subseteq \tau_1 \subseteq \ldots$ of $\mathbb{T}_0, \mathbb{T}_1, \ldots$, we have $\bigsqcup_n l_n \in \bigcup_i \tau_i$.

Proof: Let $\tau = \bigcup_i \tau_i$; then, by definition, $\tau_0 \subseteq \tau_1 \subseteq \ldots$ and $\tau = \bigcup_i \tau_i$. By hypothesis, we have $l_i \in \tau_i$ for all τ_i, hence by Lemma 3.3.5 we have $\bigsqcup_n l_n \in \tau$. □

Similarly, we generalize Corollary 3.3.8(2) to increasing instances of sequences of sets of total types.

3.3.11 Corollary: 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 the following property is satisfied for all l and i,

$$(\forall \tau \in \mathbb{T}_i.\ \ l \in \tau) \implies (\forall \tau \in \mathbb{T}_{i+1}.\ \ f(l) \in \tau),$$

then $\mathit{fix}(f) \in \bigcup_i \tau_i$ for all increasing instances $\tau_0, \tau_1, \ldots$ of $\mathbb{T}_0, \mathbb{T}_1, \ldots$.

Proof: First note that, since $\mathbb{T}_0 = \{(\emptyset, \emptyset)\}$, we have $f^0(\bot_l) = \bot_l \in \tau$ for all $\tau \in \mathbb{T}_0$. From this, a simple induction on i (using the given implication at each step) yields $f^i(\bot_l) \in \tau$ for all $\tau \in \mathbb{T}_i$. By 3.3.10, for any increasing instance $\tau_0, \tau_1, \ldots$ of $\mathbb{T}_0, \mathbb{T}_1, \ldots$, we have $\bigsqcup_n f^n(\bot) \in \bigcup_i \tau_i$. By 3.3.7, $\mathit{fix}(f) \in \bigcup_i \tau_i$. □

To support totality proofs for *mutually* recursive lens definitions (e.g., our `list_filter` example in Section 7), we need to generalize the above argument yet one step further, to *tuples* of total types (and, accordingly, tuples of sets of total types, etc.). To avoid too much notation, we show just the special case where the tuples are pairs.

3.3.12 Definition: The increasing chain $(\tau_0, \tau_0') \subseteq (\tau_1, \tau_1') \subseteq \ldots$ of pairs of total types is an *increasing instance* of the sequence $(\mathbb{T}_0, \mathbb{T}_0'), (\mathbb{T}_1, \mathbb{T}_1'), \ldots$ iff for all i we have $\tau_i \in \mathbb{T}_i$ and $\tau_i' \in \mathbb{T}_i'$.

3.3.13 Lemma: Let $(l_0, l_0') \prec (l_1, l_1') \prec \ldots$ be an increasing chain of pairs of lenses, and let $(\mathbb{T}_0, \mathbb{T}_0'), (\mathbb{T}_1, \mathbb{T}_1'), \ldots$ be a sequence of pairs of sets of total types, such that for all $\tau_i \in \mathbb{T}_i$ we have $l_i \in \tau_i$ and for all $\tau_i' \in \mathbb{T}_i'$ we have $l_i' \in \tau_i'$. Then for any increasing instance $(\tau_0, \tau_0') \subseteq (\tau_1, \tau_1') \subseteq \ldots$ of $(\mathbb{T}_0, \mathbb{T}_0'), (\mathbb{T}_1, \mathbb{T}_1'), \ldots$, we have $\bigsqcup_n l_n \in \bigcup_i \tau_i$ and $\bigsqcup_n l_n' \in \bigcup_i \tau_i'$.

Proof: Immediate consequence of Lemma 3.3.10 (just apply 3.3.10 to the first components of all the pairs and then again to the second components). □

3.3.14 Corollary: Suppose f is a continuous function from pairs of lenses to pairs of lenses and that $(\mathbb{T}_0, \mathbb{T}_0'), (\mathbb{T}_1, \mathbb{T}_1'), \ldots$ is a sequence of pairs of sets of total types with $\mathbb{T}_0 = \mathbb{T}_0' = \{(\emptyset, \emptyset)\}$. If the following two implications hold for all l, l', and i:

1. from $(\forall \tau \in \mathbb{T}_i.\ l \in \tau)$ and $(\forall \tau' \in \mathbb{T}_i'.\ l' \in \tau')$ it follows that $(\forall \tau \in \mathbb{T}_{i+1}.\ \pi_1(f(l, l')) \in \tau)$
2. from $(\forall \tau \in \mathbb{T}_{i+1}.\ l \in \tau)$ and $(\forall \tau' \in \mathbb{T}_i'.\ l' \in \tau')$ it follows that $(\forall \tau' \in \mathbb{T}_{i+1}'.\ \pi_2(f(l, l')) \in \tau')$

then $\mathit{fix}(f) \in (\bigcup_i \tau_i, \bigcup_i \tau_i')$ for all increasing instances $(\tau_0, \tau_0') \subseteq (\tau_1, \tau_1') \subseteq \ldots$ of $(\mathbb{T}_0, \mathbb{T}_0'), (\mathbb{T}_1, \mathbb{T}_1'), \ldots$.

Proof: We first define an auxiliary continuous function g from pairs of lenses to pairs of lenses such that $\mathit{fix}(f) = \mathit{fix}(g)$, then show that $g^i(\bot_l, \bot_l)$ has every pair of total types (τ_i, τ_i') in $\mathbb{T}_i \times \mathbb{T}_i'$, and conclude by Lemma 3.3.13.

Let $f_1 = \pi_1 \circ f$ and $f_2 = \pi_2 \circ f$. As f is continuous, both f_1 and f_2 are continuous. Let g be the function from pairs of lenses to pairs of lenses defined as $g = \lambda\,(l_1, l_2).(f_1(l_1, l_2), f_2(f_1(l_1, l_2), l_2))$. The function g is continuous from pairs of lenses to pairs of lenses.

We first show that $\mathit{fix}(f) = \mathit{fix}(g)$. Let (l_1, l_2) be a fixed point of f, then we have $l_1 = f_1(l_1, l_2)$ and $l_2 = f_2(l_1, l_2)$. We calculate as follows:

$$\begin{aligned} g(l_1, l_2) &= (f_1(l_1, l_2), f_2(f_1(l_1, l_2), l_2)) \\ &= (l_1, f_2(l_1, l_2)) \\ &= (l_1, l_2) \end{aligned}$$

Hence (l_1, l_2) is a fixed point of g. Conversely, let (l_1, l_2) be a fixed point of g. Then we have $g(l_1, l_2) = (l_1, l_2)$; that is, $f_1(l_1, l_2) = l_1$ and $f_2(f_1(l_1, l_2), l_2) = l_2$, hence $f_2(l_1, l_2) = l_2$. Thus (l_1, l_2) is a fixed point of f. As a pair of lenses is a fixed point of f iff it is a fixed point of g, the smallest fixed point of f is the smallest fixed point of g, hence $\mathit{fix}(f) = \mathit{fix}(g)$.

We show that $g^i(\bot_l, \bot_l)$ has every pair of total types (τ_i, τ_i') in $\mathbb{T}_i \times \mathbb{T}_i'$ for all i, by induction on i. The case where $i = 0$ is immediate: $g^0(\bot_l, \bot_l) = \bot_l \in (\tau_0, \tau_0')$ since $\tau_0 = \tau_0' = (\emptyset, \emptyset)$. We now prove the induction case, showing that $g^{i+1}(\bot_l, \bot_l) \in (\tau_{i+1}, \tau_{i+1}')$ for all $(\tau_{i+1}, \tau_{i+1}') \in \mathbb{T}_{i+1} \times \mathbb{T}_{i+1}$.

Let $\tau_{i+1} \in \mathbb{T}_{i+1}$. By the induction hypothesis, we have $g^i(\bot_l, \bot_l) \in (\tau_i, \tau_i')$ for all $(\tau_i, \tau_i') \in \mathbb{T}_i \times \mathbb{T}_i$. Hence, by definition of g and by the first implication hypothesis, it follows that

$$\pi_1(g^{i+1}(\bot_l, \bot_l)) = f_1(g^i(\bot_l, \bot_l)) = \pi_1(f(g^i(\bot_l, \bot_l))) \in \tau_{i+1}.$$

Let $\tau_{i+1}' \in \mathbb{T}_{i+1}'$. By the previous argument, we have $f_1(g^i(\bot_l, \bot_l)) \in \tau_{i+1}$ for all $\tau_{i+1} \in \mathbb{T}_{i+1}$. By the induction hypothesis we have $\pi_2(g^i(\bot_l, \bot_l)) \in \tau_i'$ for all $\tau_i' \in \mathbb{T}_i'$. Hence, by definition of g and by the second implication hypothesis, it follows that

$$\pi_2(g^{i+1}(\bot_l, \bot_l)) = f_2(f_1(g^i(\bot_l, \bot_l)), \pi_2(g^i(\bot_l, \bot_l))) = \pi_2(f(f_1(g^i(\bot_l, \bot_l)), \pi_2(g^i(\bot_l, \bot_l)))) \in \tau_{i+1}'.$$

Combining these two arguments, we thus have $g^{i+1}(\bot_l, \bot_l) \in (\tau_{i+1}, \tau_{i+1}')$ for all $(\tau_{i+1}, \tau_{i+1}') \in \mathbb{T}_{i+1} \times \mathbb{T}_{i+1}'$.

Let $(\tau_0, \tau_0') \subseteq (\tau_1, \tau_1') \subseteq \ldots$ be an increasing instance of $(\mathbb{T}_0, \mathbb{T}_0'), (\mathbb{T}_1, \mathbb{T}_1'), \ldots$. In what follows, we write l_i for $\pi_1(g^i(\bot_l, \bot_l))$ and l_i' for $\pi_2(g^i(\bot_l, \bot_l))$.

By Lemma 3.3.13, we have $\bigsqcup_i l_i \in \bigcup_i \tau_i$ and $\bigsqcup_i l_i' \in \bigcup_i \tau_i'$.

By continuity of pairing, we conclude that $(\bigsqcup_i l_i, \bigsqcup_i l_i') = \mathit{fix}(g) = \mathit{fix}(f) \in (\bigcup_i \tau_i, \bigcup_{i'} \tau_{i'}')$. □

3.4 Dealing with Creation

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

Intuitively, $l\searrow(a, \Omega)$ means "create a *new* concrete view from the information in the abstract view a." By convention, Ω is only used in an interesting way when it is the second argument to the *put* function: in all of the lenses defined below, we maintain the invariants that (1) $l\nearrow\Omega = \Omega$, (2) $l\searrow(\Omega, c) = \Omega$ for any c, (3) $l\nearrow c \neq \Omega$ for any $c \neq \Omega$, and (4) $l\searrow(a, c) \neq \Omega$ for any $a \neq \Omega$ and any c (including Ω). We write $C \stackrel{\Omega}{\rightleftharpoons} A$ 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. (There are other, formally equivalent, ways of handling missing concrete views. The advantages of this one are discussed in Section 5.4.)

A useful consequence of these conventions is that a lens $l \in C \stackrel{\Omega}{\rightleftharpoons} A$ also has type $C \rightleftharpoons A$.

3.4.1 Lemma: For any lens l and sets of views C and A:

1. $l \in C \stackrel{\Omega}{\rightleftharpoons} A \implies l \in C \rightleftharpoons A$.
2. $l \in C \stackrel{\Omega}{\Longleftrightarrow} A \implies l \in C \Longleftrightarrow A$.

Proof: Let $l \in C \stackrel{\Omega}{\rightleftharpoons} A$.

1. We must prove that for all $c \in C$, $l\nearrow c \in A$. As $l\nearrow c \in A_\Omega$, and since $c \neq \Omega$, by convention we have $l\nearrow c \neq \Omega$. Similarly, let a, c in $A \times C$, then $l\searrow(a, c) \in C$.
2. By convention, $C_\Omega \subseteq \mathsf{dom}(l\nearrow)$ implies $C \subseteq \mathsf{dom}(l\nearrow)$, and $A \times C_\Omega \subseteq \mathsf{dom}(l\searrow)$ implies $A \times C \subseteq \mathsf{dom}(l\searrow)$, as required. □

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 $\mathcal{U}$. 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").

Most of the lens definitions in this and following sections 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.

We prove that every lens we define is well behaved (i.e., that the type declaration accompanying its definition is a theorem) and total, and that every lens that takes other lenses as parameters is continuous in these parameters. Indeed, nearly all of the lenses are *very* well behaved (if their lens arguments are), the only exceptions being `map` and `flatten`; 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 *put* direction.

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

Having defined `id`, we must now prove that it is well behaved and total—i.e., that its type declaration is a theorem. Since we will need similar arguments for every lens we define, some shorthand is useful. By our conventions on the treatment of Ω, the GET condition in Definition 3.1.2 need only be checked for C (not C_Ω) and PUT need only be checked for $A \times C_\Omega$. Similarly, GETPUT need only be checked for $c \in C$, and PUTGET for $a \in A$ and $c \in C_\Omega$.

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

Proof:
GET: $\texttt{id}\nearrow c = c \in C$.
PUT: $\texttt{id}\searrow(a,\,c) = a \in C$.
GETPUT: $\texttt{id}\searrow(\texttt{id}\nearrow c,\,c) = \texttt{id}\searrow(c,\,c) = c$.
PUTGET: $\texttt{id}\nearrow\texttt{id}\searrow(a,\,c) = \texttt{id}\nearrow a = a$. □

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

Proof: Immediate: both the *get* and *put* directions of `id` are total functions. □

For each lens definition, the totality lemma will be almost identical to the well-behavedness lemma, 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, since well-behavedness is part of the definition of totality. However, when we come to lens definitions that are parameterized on other lenses (like composition, just below), the totality of the compound lens will depend on the totality (not just well-behavedness) of its argument lenses; if all we know is that the arguments are well behaved, then we cannot use the combined lemma to establish the well-behavedness of the compound lens. 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 prefer to state the two lemmas separately.

Composition

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

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

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 *put* direction, the two *put* functions are applied in turn: first, the *put* function of k is used to put a into the concrete view that the *get* of k was applied to, i.e., $l\nearrow c$; the result of this *put* is then put into c using the *put* function of l. (If the concrete view c is Ω, then, $l\nearrow c$ will also be Ω by our conventions on the treatment of Ω, so the effect of $(l;k)\searrow(a,\,\Omega)$ will be to use k to put a into Ω and then l to put the result into Ω.) Note that we record two different type declarations for composition: one for the case where the parameter lenses l and k are only known to be well behaved, and another for the case where they are also known to be total.

To aid in checking well-behavedness, we will sometimes annotate uses of the composition operator with a suitable "cut type," writing $l \;;_B\; k$ instead of just $l;k$. We will maintain the invariant that, whenever we are interested in checking the well-behavedness of a composite lens $l \;;_B\; k$, the source and target types C and A will be determined by the context; the annotation B allows us to propagate this invariant to l and k. We sometimes annotate C and A explicitly by 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. In particular, we use it heavily in the bookmark lenses in Section 8.)

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

Proof:

GET: If $k \nearrow l \nearrow c = (l;k) \nearrow c$ is defined, then $l \nearrow c \in B$ by GET for l, so $(l;k) \nearrow c \in A$ by GET for k.

PUT: If $l \searrow (k \searrow (a,\, l \nearrow c),\, c) = (l;k) \searrow (a,\, c)$ is defined, then $l \nearrow c \in B_\Omega$ by GET for l and our convention on treatment of Ω by *get* functions, so $k \searrow (a,\, l \nearrow c) \in B$ by PUT for k, so $l \searrow (k \searrow (a,\, l \nearrow c),\, c) \in C$ by PUT for l.

GETPUT: Assume that $(l;k) \nearrow c$ is defined. Then:

$$
\begin{array}{lll}
 & (l;k) \searrow \Big(\underline{(l;k) \nearrow c},\, c\Big) & \\
= & \underline{(l;k) \searrow (k \nearrow l \nearrow c,\, c)} & \text{by definition (of the underlined expression)} \\
= & l \searrow \Big(\underline{k \searrow (k \nearrow l \nearrow c,\, l \nearrow c)},\, c\Big) & \text{by definition} \\
\sqsubseteq & \underline{l \searrow (l \nearrow c,\, c)} & \text{GETPUT for } k \\
\sqsubseteq & c & \text{GETPUT for } l
\end{array}
$$

PUTGET: Assume that $(l;k) \searrow (a,\, c)$ is defined. Then:

$$
\begin{array}{lll}
 & (l;k) \nearrow \underline{(l;k) \searrow (a,\, c)} & \\
= & \underline{(l;k) \nearrow l \searrow (k \searrow (a,\, l \nearrow c),\, c)} & \text{by definition} \\
= & \underline{k \nearrow l \nearrow l \searrow (k \searrow (a,\, l \nearrow c),\, c)} & \text{by definition} \\
\sqsubseteq & \underline{k \nearrow k \searrow (a,\, l \nearrow c)} & \text{PUTGET for } l \\
\sqsubseteq & a & \text{PUTGET for } k \qquad \square
\end{array}
$$

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

Proof: Let $c \in C$; then $l \nearrow c$ is defined (by totality of l) and is in B, hence $k \nearrow l \nearrow c = (l;k) \nearrow c$ is defined (by totality of k). Conversely, let $a \in A$ and $c \in C_\Omega$; then $l \nearrow c$ is defined and is in B_Ω. Thus, $k \searrow (a,\, l \nearrow c)$ is defined and is in B, and so $l \searrow (k \searrow (a,\, l \nearrow c),\, c) = (l;k) \searrow (a,\, c)$ is defined. $\square$

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 3.3.7, which requires that the function $\lambda l.\ l_1;l_2$ be continuous in l. According to the following lemma, this will be the case whenever l_1 and l_2 are continuous in l. We will prove 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.

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.

Proof: We first argue that $\lambda l.\ (F(l);G(l))$ is monotone. Let l and l' be two lenses with $l \prec l'$. We must show that $F(l);G(l) \prec F(l');G(l')$. For the *get* direction, let $c \in \mathcal{U}$, and assume that $(F(l);G(l)) \nearrow c$ is defined. We have:

$$
\begin{array}{lll}
 & (F(l);G(l)) \nearrow c & \\
= & G(l) \nearrow F(l) \nearrow c & \\
= & G(l) \nearrow F(l') \nearrow c & \text{by } F(l) \prec F(l'), \text{ since } F(l) \nearrow c \text{ is defined} \\
= & G(l') \nearrow F(l') \nearrow c & \text{by } G(l) \prec G(l') \\
= & (F(l');G(l')) \nearrow c. &
\end{array}
$$

For the *put* direction, let $(a, c) \in \mathcal{U} \times \mathcal{U}_\Omega$, assume that $(F(l); G(l)) \searrow (a,\ c)$ is defined, and calculate as follows:

$$
\begin{array}{lll}
 & (F(l); G(l)) \searrow (a,\ c) & \\
= & F(l) \searrow (G(l) \searrow (a,\ F(l) \nearrow c),\ c) & \\
= & F(l) \searrow (G(l) \searrow (a,\ F(l') \nearrow c),\ c) & \text{by } F(l) \prec F(l') \\
= & F(l) \searrow (G(l') \searrow (a,\ F(l') \nearrow c),\ c) & \text{by } G(l) \prec G(l') \\
= & F(l') \searrow (G(l') \searrow (a,\ F(l') \nearrow c),\ c) & \text{by } F(l) \prec F(l') \\
= & (F(l'); G(l')) \searrow (a,\ c). &
\end{array}
$$

Thus $\lambda l.\ (F(l); G(l))$ is monotone. We must now prove that it is continuous.

Let $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ be an increasing chain of well-behaved lenses. Let $l = \bigsqcup_i l_i$. We have, for $c \in \mathcal{U}$,

$$
\begin{array}{lll}
 & (F(l); G(l)) \nearrow c = v & \\
\iff & G(l) \nearrow F(l) \nearrow c = v & \text{by definition of ;} \\
\iff & G(l) \nearrow F(\bigsqcup_i l_i) \nearrow c = v & \text{by definition of } l \\
\iff & G(l) \nearrow (\bigsqcup_i F(l_i)) \nearrow c = v & \text{by continuity of } F \\
\iff & \exists i_1. G(l) \nearrow F(l_{i_1}) \nearrow c = v & \text{by Corollary 3.3.4 (GET)} \\
\iff & \exists i_1. G(\bigsqcup_i l_i) \nearrow F(l_{i_1}) \nearrow c = v & \text{by definition of } l \\
\iff & \exists i_1. (\bigsqcup_i G(l_i)) \nearrow F(l_{i_1}) \nearrow c = v & \text{by continuity of } G \\
\iff & \exists i_2, i_1. G(l_{i_2}) \nearrow F(l_{i_1}) \nearrow c = v & \text{by Corollary 3.3.4 (GET)} \\
\iff & \exists i. G(l_i) \nearrow F(l_i) \nearrow c = v & \text{by } \begin{cases} \text{letting } i = \mathtt{max}(i_1, i_2) \\ \text{monotonicity of } F \text{ and } G \end{cases} \\
\iff & \exists i. (F(l_i); G(l_i)) \nearrow c = v & \text{by definition of ;} \\
\iff & (\bigsqcup_i (F(l_i); G(l_i))) \nearrow c = v & \text{by Corollary 3.3.4 (GET)}
\end{array}
$$

and

$$
\begin{array}{lll}
 & (F(l); G(l)) \searrow (a,\ c) = v & \\
\iff & F(l) \searrow (G(l) \searrow (a,\ F(l) \nearrow c),\ c) = v & \text{by definition of ;} \\
\iff & F(l) \searrow (G(l) \searrow (a,\ F(\bigsqcup_i l_i) \nearrow c),\ c) = v & \text{by definition of } l \\
\iff & F(l) \searrow (G(l) \searrow (a,\ (\bigsqcup_i F(l_i)) \nearrow c),\ c) = v & \text{by continuity of } F \\
\iff & \exists i_1. F(l) \searrow (G(l) \searrow (a,\ F(l_{i_1}) \nearrow c),\ c) = v & \text{by Corollary 3.3.4 (GET)} \\
\iff & \exists i_1. F(l) \searrow (G(\bigsqcup_i l_i) \searrow (a,\ F(l_{i_1}) \nearrow c),\ c) = v & \text{by definition of } l \\
\iff & \exists i_1. F(l) \searrow ((\bigsqcup_i G(l_i)) \searrow (a,\ F(l_{i_1}) \nearrow c),\ c) = v & \text{by continuity of } G \\
\iff & \exists i_2, i_1. F(l) \searrow (G(l_{i_2}) \searrow (a,\ F(l_{i_1}) \nearrow c),\ c) = v & \text{by Corollary 3.3.4 (PUT)} \\
\iff & \exists i_2, i_1. F(\bigsqcup_i li) \searrow (G(l_{i_2}) \searrow (a,\ F(l_{i_1}) \nearrow c),\ c) = v & \text{by definition of } l \\
\iff & \exists i_2, i_1. (\bigsqcup_i F(l_i)) \searrow (G(l_{i_2}) \searrow (a,\ F(l_{i_1}) \nearrow c),\ c) = v & \text{by continuity of } F \\
\iff & \exists i_3, i_2, i_1. F(l_{i_3}) \searrow (G(l_{i_2}) \searrow (a,\ F(l_{i_1}) \nearrow c),\ c) = v & \text{by Corollary 3.3.4 (PUT)} \\
\iff & \exists i. F(l_i) \searrow (G(l_i) \searrow (a,\ F(l_i) \nearrow c),\ c) = v & \text{by } \begin{cases} \text{letting } i = \mathtt{max}(i_1, i_2, i_3) \\ \text{monotonicity of } F \text{ and } G \end{cases} \\
\iff & \exists i. (F(l_i); G(l_i)) \searrow (a,\ c) = v & \text{by definition of ;} \\
\iff & (\bigsqcup_i (F(l_i); G(l_i))) \searrow (a,\ c) = v & \text{by Corollary 3.3.4 (PUT).}
\end{array}
$$

Hence the lenses $\bigsqcup_i (F(l_i); G(l_i))$ and $F(l); G(l)$ are equal. □

Constant

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

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

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

We can define a similar lens, `const` v, that is identical to the standard version except that the *put* function is undefined when the concrete view is Ω. This lens has type $C \Longleftrightarrow \{v\}$ (note that this type does not mention Ω). Later (in Section 6) we will see how to use conditional combinators to wrap lenses like `const` v to produce a lens whose *put* function is extended to handle missing concrete views.

4.6 Lemma [Well-behavedness]: $\forall C{\subseteq}\mathcal{U}.\ \forall v{\in}\mathcal{U}.\ \forall d{\in}C.\ \texttt{const}\ v\ d \in C \overset{\Omega}{\rightleftharpoons} \{v\}$.

Proof:
GET: $(\texttt{const}\ v\ d)\nearrow c = v \in \{v\}$.

PUT: $(\texttt{const}\ v\ d)\searrow (v,\, c) \in \{c, d\} \subseteq C$.

GETPUT: $(\texttt{const}\ v\ d)\searrow((\texttt{const}\ v\ d)\nearrow c,\, c) = (\texttt{const}\ v\ d)\searrow(v,\, c) = c$.

PUTGET: If $c \neq \Omega$, then $(\texttt{const}\ v\ d)\nearrow((\texttt{const}\ v\ d)\searrow(v,\, c)) = (\texttt{const}\ v\ d)\nearrow c = v$. Otherwise, $(\texttt{const}\ v\ d)\nearrow((\texttt{const}\ v\ d)\searrow(v,\, \Omega)) = (\texttt{const}\ v\ d)\nearrow d = v$. □

4.7 Lemma [Totality]: $\forall C{\subseteq}\mathcal{U}.\ \forall v{\in}\mathcal{U}.\ \forall d{\in}C.\ \texttt{const}\ v\ d \in C \overset{\Omega}{\Longleftrightarrow} \{v\}$.

Proof: Immediate: both the *get* and *put* directions of $(\texttt{const}\ v\ d)$ are total functions for every v and d. □

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

5 Lenses for Trees

To keep our lens definitions as straightforward as possible, we work with an extremely simple form of trees: unordered, edge-labeled trees with no repeated labels. This does not give us—primitively—all the structure we need for some applications; in particular, we will need to deal with ordered data such as lists and XML documents via an encoding (shown in Section 8) instead of primitively. Experience has shown that the reduction in the complexity of the lens *definitions* that we obtain in this way far outweighs the increase in complexity of lens *programs* due to manipulating ordered data in encoded form.

5.1 Notation

From this point forward, we will choose the universe $\mathcal{U}$ to be the set $\mathcal{T}$ of finite, unordered, edge-labeled trees, with labels drawn from some infinite set $\mathcal{N}$ of *names*—e.g., character strings—and with the children of a given node all labeled with distinct names. Trees of this form are sometimes called *feature trees* (e.g., [34]). The variables a, c, d, and t range over $\mathcal{T}$; by convention, we use a for trees that are thought of as abstract and c or d for concrete trees.

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

Tree values are written using hollow curly braces. The empty tree is written $\{\}$. (Note that $\{\}$, the tree with no children, is different from Ω.) We often describe trees by comprehension, writing

$\{\!|n \mapsto F(n) \mid n \in N|\!\}$, where F is some function from $\mathcal{N}$ to $\mathcal{T}_\Omega$ and $N \subseteq \mathcal{N}$ is some set of names. When t and t' have disjoint domains, we write $t \cdot t'$ or $\{\!|t\ t'|\!\}$ (the latter especially in multi-line displays) for the tree mapping n to $t(n)$ for $n \in \mathsf{dom}(t)$, to $t'(n)$ for $n \in \mathsf{dom}(t')$, and to Ω otherwise.

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

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

For writing down types,[4] we extend these tree notations "pointwise" to sets of trees. If $T \subseteq \mathcal{T}$ and $n \in \mathcal{N}$, then $\{\!|n \mapsto T|\!\}$ denotes the set of singleton trees $\{\{\!|n \mapsto t|\!\} \mid t \in T\}$. If $T \subseteq \mathcal{T}$ and $N \subseteq \mathcal{N}$, then $\{\!|N \mapsto T|\!\}$ denotes the set of trees $\{t \mid \mathsf{dom}(t) = N \text{ and } \forall n \in N.\ t(n) \in T\}$ and $\left\{\!\middle|N \overset{?}{\mapsto} T\middle|\!\right\}$ denotes the set of trees $\{t \mid \mathsf{dom}(t) \subseteq N \text{ and } \forall n \in N.\ t(n) \in T_\Omega\}$. We write $T_1 \cdot T_2$ for $\{t_1 \cdot t_2 \mid t_1 \in T_1,\ t_2 \in T_2\}$ and $T(n)$ for $\{t(n) \mid t \in T\} \setminus \{\Omega\}$. If $T \subseteq \mathcal{T}$, then $\mathsf{dom}(T) = \{\mathsf{dom}(t) \mid t \in T\}$. Note that $\mathsf{dom}(T)$ 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.

5.2 Hoisting and Plunging

Let's warm up with some combinators that perform simple structural transformations on trees of very simple shapes. We will see in Section 5.3 how to combine these with a powerful "forking" operator to perform related operations on more general sorts of trees.

Hoist

The lens `hoist` n is used to "shorten" a tree by removing an edge at the top. In the *get* direction, it expects a tree that has exactly one child, named n. It returns this child, removing the edge n. In the *put* direction, the value of the old concrete tree is ignored and a new concrete tree is created, with a single edge n pointing to the given abstract tree. (In Section 5.3, we will meet a derived form, `hoist_nonunique`, that works on bushier trees.)

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

5.2.1 Lemma [Well-behavedness]: $\forall C{\subseteq}\mathcal{T}.\ \forall n{\in}\mathcal{N}.\ \texttt{hoist}\ n \in \{\!|n \mapsto C|\!\} \overset{\Omega}{=\!=} C$.

Proof:
GET: $(\texttt{hoist}\ n)\nearrow \{\!|n \mapsto c|\!\} = c \in C$

PUT: $(\texttt{hoist}\ n)\searrow(a,\, c) = \{\!|n \mapsto a|\!\} \in \{\!|n \mapsto C|\!\}$

GETPUT: $(\texttt{hoist}\ n)\searrow\big((\texttt{hoist}\ n)\nearrow \{\!|n \mapsto t|\!\}),\ \{\!|n \mapsto t|\!\}\big) = (\texttt{hoist}\ n)\searrow\big(t,\ \{\!|n \mapsto t|\!\}\big) = \{\!|n \mapsto t|\!\}$.

PUTGET: $(\texttt{hoist}\ n)\nearrow((\texttt{hoist}\ n)\searrow(a,\, c)) = (\texttt{hoist}\ n)\nearrow \{\!|n \mapsto a|\!\} = a$. □

5.2.2 Lemma [Totality]: $\forall C{\subseteq}\mathcal{T}.\ \forall n{\in}\mathcal{N}.\ \texttt{hoist}\ n \in \{\!|n \mapsto C|\!\} \overset{\Omega}{\Longleftrightarrow} C$.

Proof: Straightforward: the *put* direction is a total function, and the *get* direction is clearly defined for every tree in the source type $\{\!|n \mapsto C|\!\}$. □

[4] 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.

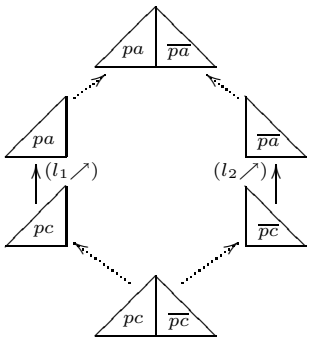


Figure 1: The *get* direction of `xfork`

Plunge

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

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

5.2.3 Lemma [Well-behavedness]: $\forall C{\subseteq}\mathcal{T}.\ \forall n{\in}\mathcal{N}.\ \texttt{plunge}\ n \in C \overset{\Omega}{=\!=} \{\!|n \mapsto C|\!\}$.

Proof:
GET: $(\texttt{plunge}\ n)\nearrow c = \{\!|n \mapsto c|\!\} \in \{\!|n \mapsto C|\!\}$.
PUT: $(\texttt{plunge}\ n)\searrow(\{\!|n \mapsto t|\!\},\, c) = t \in C$.
GETPUT: $(\texttt{plunge}\ n)\searrow((\texttt{plunge}\ n)\nearrow c,\, c) = (\texttt{plunge}\ n)\searrow(\{\!|n \mapsto c|\!\},\, c) = c$.
PUTGET: $(\texttt{plunge}\ n)\nearrow((\texttt{plunge}\ n)\searrow(\{\!|n \mapsto t|\!\},\, c)) = (\texttt{plunge}\ n)\nearrow t = \{\!|n \mapsto t|\!\}$. □

5.2.4 Lemma [Totality]: $\forall C{\subseteq}\mathcal{T}.\ \forall n{\in}\mathcal{N}.\ \texttt{plunge}\ n \in C \overset{\Omega}{\Longleftrightarrow} \{\!|n \mapsto C|\!\}$.

Proof: Straightforward: the *get* direction is a total function, and the *put* direction is defined for every pair consisting of a tree in the target type $\{\!|n \mapsto C|\!\}$ and any concrete tree whatsoever (or Ω). □

5.3 Forking

The lens combinator `xfork` applies different lenses to different parts of a tree: it splits the tree into two parts according to the names of its immediate children, applies a different lens to each, and concatenates the results. Formally, `xfork` takes as arguments two sets of names and two lenses. The *get* direction of `xfork` pc pa l_1 l_2 can be visualized as in Figure 1 (the concrete tree is at the bottom). The triangles labeled pc denote trees whose immediate child edges have labels in pc; dotted arrows represent splitting or concatenating trees. The result of applying $l_1\nearrow$ to $c|_{pc}$ (the tree formed by dropping the immediate children of c whose names are not in pc) must be a tree whose top-level labels are in the set pa, and, similarly the result of applying $l_2\nearrow$ to $c\backslash_{pc}$ must be in $\overline{pa}$. That is, the lenses l_1 and l_2 are allowed to change the sets of names in the trees they are given, but each must map from its own part of pc to its own part of pa. Conversely, in the *put* direction, l_1 must map from pa to pc and l_2 from $\overline{pa}$ to $\overline{pc}$. Here is the full definition:

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

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

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

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

5.3.1 Lemma [Well-behavedness]: $\forall pc, pa \subseteq \mathcal{N}.\ \forall C_1 \subseteq \mathcal{T}|_{pc}.\ \forall A_1 \subseteq \mathcal{T}|_{pa}.\ \ \forall C_2 \subseteq \mathcal{T}\backslash_{pc}.\ \forall A_2 \subseteq \mathcal{T}\backslash_{pa}.\ \ \forall l_1 \in C_1 \overset{\Omega}{\rightleftharpoons} A_1.\ \forall l_2 \in C_2 \overset{\Omega}{\rightleftharpoons} A_2.\ \ \texttt{xfork}\ pc\ pa\ l_1\ l_2 \in (C_1 \cdot C_2) \overset{\Omega}{\rightleftharpoons} (A_1 \cdot A_2)$.

Proof:

GET: If $c \in C_1 \cdot C_2$, then $c|_{pc} \in C_1$ and $c\backslash_{pc} \in C_2$. Hence $l_1\nearrow c|_{pc} \in A_1$ and $l_2\nearrow c\backslash_{pc} \in A_2$, and so we have $(\texttt{xfork}\ pc\ pa\ l_1\ l_2)\nearrow c \in A_1 \cdot A_2$.

PUT: Similarly, $l_1 \searrow (a|_{pa},\ c|_{pc}) \in C_1$ and $l_2 \searrow (a\backslash_{pa},\ c\backslash_{pc}) \in C_2$, hence $(\texttt{xfork}\ pc\ pa\ l_1\ l_2)\searrow (a,\ c) \in C_1 \cdot C_2$.

GETPUT: Suppose that $(\texttt{xfork}\ pc\ pa\ l_1\ l_2)\nearrow c$ is defined. Then $l_1\nearrow c|_{pc} \cdot l_2\nearrow c\backslash_{pc}$ is defined and

$$
\begin{array}{l}
(l_1\nearrow c|_{pc} \cdot l_2\nearrow c\backslash_{pc})|_{pa} = l_1\nearrow c|_{pc} \\
(l_1\nearrow c|_{pc} \cdot l_2\nearrow c\backslash_{pc})\backslash_{pa} = l_2\nearrow c\backslash_{pc}.
\end{array}
$$

Thus,

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

by GETPUT for l_1. Similarly,

$$l_2 \searrow ((l_1\nearrow c|_{pc} \cdot l_2\nearrow c\backslash_{pc})\backslash_{pa},\ c\backslash_{pc}) \sqsubseteq c\backslash_{pc}$$

by GETPUT for l_2. Assembling these pieces, we have

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

PUTGET: Suppose that $(\texttt{xfork}\ pc\ pa\ l_1\ l_2)\searrow (a,\ c)$ is defined. Then $l_1 \searrow (a|_{pa},\ c|_{pc}) \cdot l_2 \searrow (a\backslash_{pa},\ c\backslash_{pc})$ is defined, with

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

and

$$\big(l_1 \searrow (a|_{pa},\ c|_{pc}) \cdot l_2 \searrow (a\backslash_{pa},\ c\backslash_{pc})\big)\backslash_{pc} = l_2 \searrow (a\backslash_{pa},\ c\backslash_{pc}).$$

By PUTGET for l_1,

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

and by PUTGET for l_2,

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

Assembling these pieces, we have

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

□

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

Proof: Suppose $c \in C_1 \cdot C_2$. Then we have $c|_{pc} \in C_1$ and $c\backslash_{pc} \in C_2$. By the totality of l_1 and l_2, we know that $l_1{\nearrow}\, c|_{pc}$ is defined and is in A_1 and $l_2{\nearrow}\, c\backslash_{pc}$ is defined and is in A_2. As these two views have disjoint domains, $l_1{\nearrow}\, c|_{pc} \cdot l_2{\nearrow}\, c\backslash_{pc} = (\texttt{xfork}\ pc\ pa\ l_1\ l_2){\nearrow}\, c$ is defined.

Let $a \in A_1 \cdot A_2$ and $C \in (C_1 \cdot C_2)_\Omega$. We have:

- $a|_{pa} \in A_1$;
- $c|_{pc} \in C_1 \cup \{\Omega\}$;
- $a\backslash_{pa} \in A_2$;
- $c\backslash_{pc} \in C_2 \cup \{\Omega\}$.

Hence:

- $l_1 \searrow (a|_{pa},\, c|_{pc}) = c_1$ is defined and in C_1, and
- $l_2 \searrow (a\backslash_{pa},\, c\backslash_{pc}) = c_2$ is defined and in C_2.

As c_1 and c_2 have disjoint domains, $c_1 \cdot c_2 = (\texttt{xfork}\ pc\ pa\ l_1\ l_2) \searrow (a,\, c)$ is defined. □

5.3.3 Lemma [Continuity]: Let F and G be continuous functions from lenses to lenses. Then the function $\lambda l.\ \texttt{xfork}\ pc\ pa\ F(l)\ G(l)$ is continuous.

Proof: Begin with monotonicity. Let l and l' be two lenses with $l \prec l'$. We must show that $\texttt{xfork}\ pc\ pa\ F(l)\ G(l) \prec \texttt{xfork}\ pc\ pa\ F(l')\ G(l')$. Choose $c \in \mathcal{T}$ such that $\texttt{xfork}\ pc\ pa\ F(l)\ G(l){\nearrow}\, c$ is defined. Then

$$\begin{array}{ll} & (\texttt{xfork}\ pc\ pa\ F(l)\ G(l)){\nearrow}\, c \\ = & (F(l){\nearrow}\, c|_{pc}) \cdot (G(l){\nearrow}\, c\backslash_{pc}) \\ = & (F(l'){\nearrow}\, c|_{pc}) \cdot (G(l'){\nearrow}\, c\backslash_{pc}) \qquad \text{since } F(l) \prec F(l') \text{ and } G(l) \prec G(l') \\ = & (\texttt{xfork}\ pc\ pa\ F(l')\ G(l')){\nearrow}\, c. \end{array}$$

Now choose $(a, c) \in \mathcal{T} \times \mathcal{T}_\Omega$ with $\texttt{xfork}\ pc\ pa\ F(l)\ G(l) \searrow (a,\, c)$ is defined. We have:

$$\begin{array}{ll} & (\texttt{xfork}\ pc\ pa\ F(l)\ G(l)) \searrow (a,\, c) \\ = & (F(l) \searrow (a|_{pa},\, c|_{pc})) \cdot (G(l) \searrow (a\backslash_{pa},\, c\backslash_{pc})) \\ = & (F(l') \searrow (a|_{pa},\, c|_{pc})) \cdot (G(l') \searrow (a\backslash_{pa},\, c\backslash_{pc})) \quad \text{since } F(l) \prec F(l') \text{ and } G(l) \prec G(l') \\ = & (\texttt{xfork}\ pc\ pa\ F(l')\ G(l')) \searrow (a,\, c). \end{array}$$

Thus $\lambda l.\ \texttt{xfork}\ pc\ pa\ F(l)\ G(l)$ is monotone. We next prove it is continuous.

Let $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ be an increasing chain of well-behaved lenses, and let $l = \bigsqcup_i l_i$. We have:

$$
\begin{array}{lll}
 & (\texttt{xfork}\ pc\ pa\ F(l)\ G(l))\nearrow c = t & \\
\iff & (F(l)\nearrow c|_{pc}) \cdot (G(l)\nearrow c\backslash_{pc}) = t & \text{by definition} \\
\iff & (F(\bigsqcup_i l_i)\nearrow c|_{pc}) \cdot (G(\bigsqcup_i l_i)\nearrow c\backslash_{pc}) = t & \text{by definition} \\
\iff & ((\bigsqcup_i F(l_i))\nearrow c|_{pc}) \cdot ((\bigsqcup_i G(l_i))\nearrow c\backslash_{pc}) = t & \text{by continuity of } F \text{ and } G \\
\iff & \exists i_1, i_2.(F(l_{i_1})\nearrow c|_{pc}) \cdot (G(l_{i_2})\nearrow c\backslash_{pc}) = t & \text{by Corollary 3.3.4 (GET) twice} \\
\iff & \exists i.(F(l_i)\nearrow c|_{pc}) \cdot (G(l_i)\nearrow c\backslash_{pc}) = t & \text{by } \begin{cases} i = \texttt{max}(i_1, i_2) \\ \text{monotonicity of } F \text{ and } G \end{cases} \\
\iff & \exists i.(\texttt{xfork}\ pc\ pa\ F(l_i)\ G(l_i))\nearrow c = t & \text{by definition} \\
\iff & (\bigsqcup_i \texttt{xfork}\ pc\ pa\ F(l_i)\ G(l_i))\nearrow c = t & \text{by corollary 3.3.4 (GET)}
\end{array}
$$

and

$$
\begin{array}{lll}
 & (\texttt{xfork}\ pc\ pa\ F(l)\ G(l))\searrow (a,\, c) = t & \\
\iff & (F(l)\searrow (a|_{pa},\, c|_{pc})) \cdot (G(l)\searrow (a\backslash_{pa},\, c\backslash_{pc})) = t & \text{by definition} \\
\iff & (F(\bigsqcup_i l_i)\searrow (a|_{pa},\, c|_{pc})) \cdot (G(\bigsqcup_i l_i)\searrow (a\backslash_{pa},\, c\backslash_{pc})) = t & \text{by definition} \\
\iff & ((\bigsqcup_i F(l_i))\searrow (a|_{pa},\, c|_{pc})) \cdot ((\bigsqcup_i G(l_i))\searrow (a\backslash_{pa},\, c\backslash_{pc})) = t & \text{by continuity of } F \text{ and } G \\
\iff & \exists i_1, i_2.(F(l_{i_1})\searrow (a|_{pa},\, c|_{pc})) \cdot (G(l_{i_2})\searrow (a\backslash_{pa},\, c\backslash_{pc})) = t & \text{by Corollary 3.3.4 (PUT) twice} \\
\iff & \exists i.(F(l_i)\searrow (a|_{pa},\, c|_{pc})) \cdot (G(l_i)\searrow (a\backslash_{pa},\, c\backslash_{pc})) = t & \text{by } \begin{cases} i = \texttt{max}(i_1, i_2) \\ \text{monotonicity of } F \text{ and } G \end{cases} \\
\iff & \exists i.(\texttt{xfork}\ pc\ pa\ F(l_i)\ G(l_i))\searrow (a,\, c) = t & \text{by definition} \\
\iff & (\bigsqcup_i \texttt{xfork}\ pc\ pa\ F(l_i)\ G(l_i))\searrow (a,\, c) = t & \text{by corollary 3.3.4 (PUT).}
\end{array}
$$

□

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

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

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

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

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

$$\texttt{filter}\ p\ d \quad = \quad \texttt{fork}\ p\ \texttt{id}\ (\texttt{const}\ \{\!\}\ d)$$

$$\forall C{\subseteq}\mathcal{T}.\ \forall p{\subseteq}\mathcal{N}.\ \forall d \in C\backslash_p.$$
$$\texttt{filter}\ p\ d \in (C|_p \cdot C\backslash_p) \overset{\Omega}{\Longleftrightarrow} C|_p$$

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

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

$$\texttt{prune}\ n\ d \quad = \quad \texttt{fork}\ \{n\}\ \left(\texttt{const}\ \{\!\}\ \left\{\!\left|n \mapsto d\right|\!\right\}\right)\ \texttt{id}$$

$$\forall C{\subseteq}\mathcal{T}.\ \forall n{\in}\mathcal{N}.\ \forall d{\in}C(n).\quad \texttt{prune}\ n\ d \in (C|_n \cdot C\backslash_n) \overset{\Omega}{\Longleftrightarrow} C\backslash_n$$

This lens is similar to `filter`, except that (1) the name given is the child to be removed, and (2) the default tree is the one to go under n if the concrete tree is Ω. Just like `filter`, we can define a variant of `prune` that does not require a default view as $\texttt{prune}\ n = \texttt{fork}\ \{n\}\ (\texttt{const}\ \{\!\})\ id$, with type $(C|_n \cdot C\backslash_n) \Longleftrightarrow C\backslash_n$.

Conversely, we can grow a tree in the *get* direction by explicitly adding a child, which is dropped in the *put* direction. The type annotation disallows changes in the subtree below this child.

$$\texttt{add}\ n\ t \quad = \quad \texttt{xfork}\ \{\}\ \{n\}\ (\texttt{const}\ t\ \{\!\};\ \texttt{plunge}\ n)\ (\texttt{id})$$

$$\forall n{\in}\mathcal{N}.\ \forall C{\subseteq}\mathcal{T}\backslash_n.\ \forall t \in \mathcal{T}.\quad \texttt{add}\ n\ t \in C \overset{\Omega}{\Longleftrightarrow} \left\{\!\left|n \mapsto \{t\}\right|\!\right\} \cdot C$$

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

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

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

In the *get* direction, `focus` filters away all other children, then removes the edge n and yields n's subtree. As usual, the default tree is only used in the case of creation, where it is the default for children that have been filtered away. Again the type of `focus` follows from the types of the lenses from which it is defined, observing that $\texttt{filter}\ \{n\}\ d \in (C \cdot \{\!|n \mapsto D|\!\}) \overset{\Omega}{\Longleftrightarrow} \{\!|n \mapsto D|\!\}$ and that $\texttt{hoist}\ \texttt{n} \in \{\!|n \mapsto D|\!\} \overset{\Omega}{\Longleftrightarrow} D$. We can also define a version of `focus` that does not require a default tree as $\texttt{focus}\ n = \texttt{filter}\ \{n\};\ (\texttt{hoist}\ n)$, with type $\texttt{focus}\ n \in (C \cdot \left\{\!\left|n \mapsto D\right|\!\right\}) \Longleftrightarrow D$.

The `hoist` primitive defined in Section 5.2 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.

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

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

Our next derived lens renames a single child.

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

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

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 *put* direction splits the abstract view into a tree with a single child `n`, and the rest of the tree. The tree under `n` is put back using the lens $(\texttt{hoist}\ m;\ \texttt{plunge}\ n)$, 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 astract view has 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.

5.4 Mapping

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

The `map` combinator is parameterized on a single lens l. In the *get* direction, `map` applies $l{\nearrow}$ to each *subtree* of the root and combines the results together into a new tree. (Later in the section, we will define a more general combinator, called `wmap`, that applies a different lens to each subtree. 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 `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 *put* direction of `map` is more interesting. In the simple case where a and c have equal domains, its behavior is straightforward: it uses $l{\searrow}$ to combine concrete and abstract subtrees with identical names and assembles the results into a new concrete tree:

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

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

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

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

Because of the way that it takes tree apart, transforms the pieces, and reassembles them, the typing of `map` is a little subtle. For example, in the *get* direction, `map` does not modify the names of the immediate children of the concrete tree and in the *put* direction, the names of the abstract tree are left unchanged; we might therefore expect a simple typing rule stating that, if $l \in (\bigcap_{n\in\mathcal{N}} C(n) \stackrel{\Omega}{\rightleftharpoons} A(n))$—i.e., if l is a well-behaved lens from the concrete subtree type $C(n)$ to the abstract subtree type $A(n)$ for each child n—then

$\texttt{map}\ l \in C \overset{\Omega}{\rightleftharpoons} A$. Unfortunately, for arbitrary C and A, the `map` lens is not guaranteed to be well-behaved at this type. In particular, if $\mathsf{dom}(C)$, the set of domains of trees in C, is not equal to $\mathsf{dom}(A)$, then the *put* function can produce a tree that is not in C, as the following example shows. Consider the sets of trees

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

and observe that with trees

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

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

A related problem arises when the sets of trees A and C have dependencies between the names of children and the trees that may appear under those names. Again, one might naively expect that, if l has type $C(n) \overset{\Omega}{\rightleftharpoons} A(m)$ for each name m, then $\texttt{map}\ l$ would have type $C \overset{\Omega}{\rightleftharpoons} A$. Consider, however, the set

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

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

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

where both `m` and `n` appear with both `p` and `q`. When we consider just the projections of C and A at specific names, we obtain the same sets of subtrees: $C(\texttt{x}) = A(\texttt{x}) = \{\{\!|\texttt{m}|\!\}, \{\!|\texttt{n}|\!\}\}$ and $C(\texttt{y}) = A(\texttt{y}) = \{\{\!|\texttt{p}|\!\}, \{\!|\texttt{q}|\!\}\}$, and the lens `id` has type $C(\texttt{x}) \overset{\Omega}{\rightleftharpoons} A(\texttt{x})$ and $C(\texttt{y}) \overset{\Omega}{\rightleftharpoons} A(\texttt{y})$ (and $C(z) = \emptyset \overset{\Omega}{\rightleftharpoons} \emptyset = A(z)$ for all other names z). But it is clearly not the case that $\texttt{map}\ \texttt{id} \in C \overset{\Omega}{\rightleftharpoons} A$. To avoid this error (but still give a type for `map` that is precise enough to derive interesting types for lenses defined in terms of `map`), we require that the source and target sets in the type of `map` be closed under the "shuffling" of their children. Formally, if T is a set of trees, then the set of *shufflings* of T, denoted $T^{\circlearrowleft}$, is

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

where $\{\!|\, n \mapsto T(n) \mid n \in D \,|\!\}$ is the set of trees with domain D whose children under n are taken from the set $T(n)$. We say that T is *shuffle closed* iff $T = T^{\circlearrowleft}$. For instance, in the example above, $A^{\circlearrowleft} = C^{\circlearrowleft} = C$.

In the situations where `map` is used, shuffle closure is typically very easy to check. For example, any set of trees whose elements each have singleton domains is shuffle closed. Also, for every set of trees T, the encoding introduced in Section 7 of lists with elements in T is shuffle closed, which justifies using `map` (with recursion) to implement operations on lists.

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

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

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.

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

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

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

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

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

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 `wmap` $\{x \mapsto$ `plunge a`$\}$ maps `plunge a` to trees under x and `id` to the subtrees of every other child.

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

Proof:

GET: Suppose $c \in C$ and $m(n) \nearrow c(n)$ is defined for each $n \in \mathsf{dom}(c)$. Then, by the (dependent) type of m, we have $m(n) \nearrow c(n) \in A(n)$ for each n. Since $\mathsf{dom}(A) = \mathsf{dom}(C)$, there exists a non-empty subset of A whose elements all have domain $D = \mathsf{dom}(c)$. Also, the tree $\{\!|\, n \mapsto m(n) \nearrow c(n) \mid n \in \mathsf{dom}(c) \,|\!\}$ is an element of the set $\{\!|\, n \mapsto A(n) \mid n \in D \,|\!\}$, which is itself a subset of A since A is shuffle closed. Hence, $(\texttt{wmap}\ m) \nearrow c \in A$.

PUT: Let $a \in A$ and $c \in C$. For all $n \in \mathsf{dom}(a)$, we have $m(n) \searrow (a(n), c(n)) \in C(n)$ (with $c(n)$ possibly being Ω). Hence, by a similar argument as above, since $\mathsf{dom}(A) = \mathsf{dom}(C)$ and $C = C^{\circlearrowleft}$, we have $(\texttt{wmap}\ m) \searrow (a, c) \in C$.

GETPUT: Assume that $(\texttt{wmap}\ m) \nearrow c$ is defined. Then

$$
\begin{array}{rll}
 & (\texttt{wmap}\ m) \searrow ((\texttt{wmap}\ m) \nearrow c,\, c) & \\
= & (\texttt{wmap}\ m) \searrow (\{\!|\, n \mapsto m(n) \nearrow c(n) \mid n \in \mathsf{dom}(c) \,|\!\},\, c) & \\
= & \{\!|\, n \mapsto m(n) \searrow (m(n) \nearrow c(n),\, c(n)) \mid n \in \mathsf{dom}(c) \,|\!\} & \\
\sqsubseteq & \{\!|\, n \mapsto c(n) \mid n \in \mathsf{dom}(c) \,|\!\} & \text{by GETPUT for each } m(n) \\
= & c. &
\end{array}
$$

PUTGET: Assume that $(\texttt{wmap}\ m) \searrow (a,\, c)$ is defined. Then

$$\begin{array}{ll} & (\texttt{wmap}\ m)\nearrow((\texttt{wmap}\ m) \searrow (a,\, c)) \\ = & (\texttt{wmap}\ m)\nearrow \{\!|n \mapsto m(n) \searrow (a(n),\, c(n)) \mid n \in \mathsf{dom}(a)|\!\} \\ = & \{\!|n \mapsto m(n)\nearrow(l \searrow (a(n),\, c(n))) \mid n \in \mathsf{dom}(a)|\!\} \\ \sqsubseteq & \{\!|n \mapsto a(n) \mid n \in \mathsf{dom}(a)|\!\} \qquad \text{by PUTGET for } m(n) \text{ on each child} \\ = & a. \end{array}$$ □

5.4.2 Lemma [Totality]: $\forall C, A \subseteq \mathcal{T}$ *with* $C = C^{\circlearrowleft}$, $A = A^{\circlearrowleft}$, *and* $\mathsf{dom}(C) = \mathsf{dom}(A)$. $\forall m \in (\Pi n \in \mathcal{N}.\ C(n) \overset{\Omega}{\Longleftrightarrow} A(n))$. $\texttt{wmap}\ m \in C \overset{\Omega}{\Longleftrightarrow} A$.

Proof: Suppose $c \in C$ and $m(n)$ is a total function for each n. Then for any $n \in \mathsf{dom}(c)$, we have $c(n) \in C(n)$; hence, $m(n)\nearrow c(n)$ is defined for each n, i.e., $(\texttt{wmap}\ m)\nearrow c$ is defined. Conversely, suppose $a \in A$ and $c \in C_\Omega$. For any n in $\mathsf{dom}(a)$, we have $a(n) \in A(n)$ and $c(n) \in C(n)_\Omega$; hence $l \searrow (a(n),\, c(n))$ is defined. Thus, $(\texttt{wmap}\ m) \searrow (a,\, c)$ is defined. □

5.4.3 Lemma [Continuity]: For each name n, let F_n be a continuous function from lenses to lenses. Then the function $\lambda l.\ \texttt{wmap}\ (\lambda n.\ F_n(l))$ is continuous.

Proof: To show monotonicity, let l and l' be lenses with $l \prec l'$. We must show that $\texttt{wmap}\ (\lambda n.\ F_n(l)) \prec \texttt{wmap}\ (\lambda n.\ F_n(l'))$. Let $c \in \mathcal{T}$, and suppose that $(\texttt{wmap}\ (\lambda n.\ F_n(l)))\nearrow c$ is defined. We have

$$\begin{array}{ll} & (\texttt{wmap}\ (\lambda n.\ F_n(l)))\nearrow c \\ = & \{\!|n \mapsto F_n(l)\nearrow c(n) \mid n \in \mathsf{dom}(c)|\!\} \\ = & \{\!|n \mapsto F_n(l')\nearrow c(n) \mid n \in \mathsf{dom}(c)|\!\} \quad \text{since } l \prec l' \text{ and each } F_n \text{ is monotone} \\ = & (\texttt{wmap}\ (\lambda n.\ F_n(l')))\nearrow c. \end{array}$$

Conversely, suppose that $(a, c) \in \mathcal{T} \times \mathcal{T}_\Omega$ and that $(\texttt{wmap}\ (\lambda n.\ F_n(l))) \searrow (a,\, c)$ is defined. Then

$$\begin{array}{ll} & (\texttt{wmap}\ (\lambda n.\ F_n(l))) \searrow (a,\, c) \\ = & \{\!|n \mapsto F_n(l) \searrow (a(n),\, c(n)) \mid n \in \mathsf{dom}(a)|\!\} \\ = & \{\!|n \mapsto F_n(l') \searrow (a(n),\, c(n)) \mid n \in \mathsf{dom}(a)|\!\} \quad \text{since } l \prec l' \text{ and each } F_n \text{ is monotone} \\ = & (\texttt{wmap}\ (\lambda n.\ F_n(l'))) \searrow (a,\, c). \end{array}$$

Thus $\lambda l.\ \texttt{wmap}\ (\lambda n.\ F_n(l))$ is monotone. We now show that it is continuous.

Let $l_0 \prec l_1 \prec \ldots \prec l_n \prec \ldots$ be an increasing chain of lenses and $l = \bigsqcup_i l_i$. Let $c \in \mathcal{T}$. For notational convenience, we assume some total ordering on the names of the children of c and write $\texttt{f}(c)$ and $\texttt{l}(c)$ for the first and last names of c, respectively. We have

$$\begin{array}{lll} & t = (\texttt{wmap}\ (\lambda n.\ F_n(l)))\nearrow c & \\ \Longleftrightarrow & t = \{\!|n \mapsto F_n(l)\nearrow c(n) \mid n \in \mathsf{dom}(c)|\!\} & \\ \Longleftrightarrow & t = \{\!|n \mapsto F_n(\bigsqcup_i l_i)\nearrow c(n) \mid n \in \mathsf{dom}(c)|\!\} & \\ \Longleftrightarrow & t = \{\!|n \mapsto (\bigsqcup_i F_n(l_i))\nearrow c(n) \mid n \in \mathsf{dom}(c)|\!\} & \text{by continuity of each } F_n \\ \Longleftrightarrow & \exists i_{\texttt{f}(c)}, \ldots, i_{\texttt{l}(c)}. & \\ & t = \{\!|n \mapsto (F_n(l_{i_n}))\nearrow c(n) \mid n \in \mathsf{dom}(c)|\!\} & \text{by 3.3.4 for GET, } |\mathsf{dom}(c)| \text{ times} \\ \Longleftrightarrow & \exists i.\ t = \{\!|n \mapsto (F_n(l_i))\nearrow c(n) \mid n \in \mathsf{dom}(c)|\!\} & \text{by monotonicity of each } F_n \\ & & \text{with } i = \texttt{max}(i_{\texttt{f}(c)}, \ldots, i_{\texttt{l}(c)}) \\ \Longleftrightarrow & \exists i.\ t = (\texttt{wmap}\ (\lambda n.\ F_n(l_i)))\nearrow c & \\ \Longleftrightarrow & t = (\bigsqcup_i(\texttt{wmap}\ (\lambda n.\ F_n(l_i))))\nearrow c & \text{by 3.3.4 for GET.} \end{array}$$

Conversely, let $(a, c) \in \mathcal{T} \times \mathcal{T}_\Omega$. We assume an ordering on the names of the children of a, and write $\mathtt{f}(a)$ and $\mathtt{l}(a)$ for the first and last names of a, respectively. We have

$$
\begin{array}{lll}
 & t = (\mathtt{wmap}\ (\lambda n.\ F_n(l))) \searrow (a,\ c) & \\
\iff & t = \{\!|n \mapsto F_n(l) \searrow (a(n),\ c(n)) \mid n \in \mathsf{dom}(a)|\!\} & \\
\iff & t = \{\!|n \mapsto F_n(\bigsqcup_i l_i) \searrow (a(n),\ c(n)) \mid n \in \mathsf{dom}(a)|\!\} & \\
\iff & t = \{\!|n \mapsto (\bigsqcup_i F_n(l_i)) \searrow (a(n),\ c(n)) \mid n \in \mathsf{dom}(a)|\!\} & \text{by continuity of each } F_n \\
\iff & \exists i_{\mathtt{f}(a)}, \ldots, i_{\mathtt{l}(a)}. & \\
 & \quad t = \{\!|n \mapsto (F_n(l_{i_n})) \searrow (a(n),\ c(n)) \mid n \in \mathsf{dom}(a)|\!\} & \text{by 3.3.4 for PUT, } |\mathsf{dom}(a)| \text{ times} \\
\iff & \exists i.\ t = \{\!|n \mapsto (F_n(l_i)) \searrow (a(n),\ c(n)) \mid n \in \mathsf{dom}(a)|\!\} & \text{by monotonicity of each } F_n \text{ with } i = \mathtt{max}(i_{\mathtt{f}(a)}, \ldots, i_{\mathtt{l}(a)}) \\
\iff & \exists i.\ t = (\mathtt{wmap}\ (\lambda n.\ F_n(l_i))) \searrow (a,\ c) & \\
\iff & \quad t = (\bigsqcup_i (\mathtt{wmap}\ (\lambda n.\ F_n(l_i)))) \searrow (a,\ c) & \text{by 3.3.4 for PUT.}
\end{array}
$$

Note the use here of the fact that all trees have finite domain. This is not just a technicality: if trees are allowed to have infinitely many children, continuity fails in general. □

Having defined `wmap`, we can easily define `map` as a derived form:

$$
\begin{array}{|l|}
\hline
\mathtt{map}\ l \quad = \quad \mathtt{wmap}\ (\lambda x{\in}\mathcal{N}.\ l) \\
\hline
\forall C, A{\subseteq}\mathcal{T}\ \textit{with}\ C = C^{\circlearrowleft},\ A = A^{\circlearrowleft},\ \textit{and}\ \mathsf{dom}(C) = \mathsf{dom}(A). \\
\forall l \in (\bigcap_{n\in\mathcal{N}}.\ C(n) \stackrel{\Omega}{=\!=} A(n)). \\
\quad \mathtt{map}\ l \in C \stackrel{\Omega}{=\!=} A \\
\forall C, A{\subseteq}\mathcal{T}\ \textit{with}\ C = C^{\circlearrowleft},\ A = A^{\circlearrowleft},\ \textit{and}\ \mathsf{dom}(C) = \mathsf{dom}(A). \\
\forall l \in (\bigcap_{n\in\mathcal{N}}.\ C(n) \stackrel{\Omega}{\Longleftrightarrow} A(n)). \\
\quad \mathtt{map}\ l \in C \stackrel{\Omega}{\Longleftrightarrow} A \\
\hline
\end{array}
$$

5.5 Copying and Merging

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

Copy

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 under 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 *put* direction, `copy` simply discards $a(n)$. The type of `copy` ensures that no information is lost, because $a(m) = a(n)$.

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

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

5.5.1 Lemma [Well-behavedness]: $\forall m, n \in \mathcal{N}.\ \forall C \subseteq \mathcal{T}\backslash_{\{m,n\}}.\ \forall D \subseteq \mathcal{T}.\ \ \texttt{copy}\ m\ n \in \ \ (C \cdot \{\!|m \mapsto D_\Omega|\!\}) \overset{\Omega}{=\!=} (C \cdot \{\{\!|m \mapsto d,\ n \mapsto d|\!\} \mid d \in D_\Omega\})$.

Proof:
GET: Immediate. $(\texttt{copy}\ m\ n)\nearrow c$ unconditionally copies $c(m)$ to $a(n)$ (even when $c(m) = \Omega$), guaranteeing that $a(m) = a(n)$. Because $c(m) \in D_\Omega$, and both $a(m)$ and $a(n) = c(m)$, we have $(\texttt{copy}\ m\ n)\nearrow c \in (C \cdot \{\{\!|m \mapsto d,\ n \mapsto d|\!\} \mid d \in D_\Omega\})$.

PUT: Immediate: restricting n from the target set yields the source set.

GETPUT: Suppose $(\texttt{copy}\ m\ n)\searrow((\texttt{copy}\ m\ n)\nearrow c,\, c)$ is defined. Then $(\texttt{copy}\ m\ n)\searrow((\texttt{copy}\ m\ n)\nearrow c,\, c) = (c \cdot \{\!|n \mapsto c(m)|\!\})\backslash_n = c$.

PUTGET: Suppose $(\texttt{copy}\ m\ n)\nearrow((\texttt{copy}\ m\ n)\searrow(a,\, c))$ is defined. Then, since $a \in C \cdot \{\{\!|m \mapsto d,\ n \mapsto d|\!\} \mid d \in D_\Omega\}$, we can write a as $c' \cdot \{\!|m \mapsto d,\ n \mapsto d|\!\}$ for some $d \in D_\Omega$. Then $(\texttt{copy}\ m\ n)\nearrow((\texttt{copy}\ m\ n)\searrow(a,\, c)) = (\texttt{copy}\ m\ n)\nearrow((\texttt{copy}\ m\ n)\searrow((c' \cdot \{\!|m \mapsto d,\ n \mapsto d|\!\}),\, c)) = (\texttt{copy}\ m\ n)\nearrow(c' \cdot \{\!|m \mapsto d|\!\}) = c' \cdot \{\!|m \mapsto d,\ n \mapsto d|\!\} = a$. □

5.5.2 Lemma [Totality]: $\forall m, n \in \mathcal{N}.\ \forall C \subseteq \mathcal{T}\backslash_{\{m,n\}}.\ \forall D \subseteq \mathcal{T}.\ \ \texttt{copy}\ m\ n \in \ \ (C \cdot \{\!|m \mapsto D_\Omega|\!\}) \overset{\Omega}{\Longleftrightarrow} (C \cdot \{\{\!|m \mapsto d,\ n \mapsto d|\!\} \mid d \in D_\Omega\})$.

Proof: The *get* direction of `copy` will be defined as long as the input c lacks the name n; this is guaranteed by its type. The *put* direction is a total function. □

Readers may note that `copy` with the type given here is not very useful. The PUTGET law imposes strict constraints on the lenses that subsequently operate on $a(m)$ and $a(n)$. In particular, let a_1 and a_2 denote the results of applying $l_1\nearrow$ and $l_2\nearrow$ to $a(m)$ and $a(n)$, respectively. Suppose we guarantee that all updates made to a_1 and a_2 are "consistent" with each other — that any information maintained in common by a_1 and a_2 will be updated in an identical manner. Behaviorally, this is all we desire. However, in the *put* direction our type annotations require us to ensure $l_1\searrow(a_1',\, a(m)) = l_2\searrow(a_2',\, a(n))$. But this is impossible to ensure unless l_1 and l_2 preserve exactly the same information. For example, consider a case where c is a record in an address book. Applying $l_1\nearrow$ transforms c to an abstract view in which only names and phone numbers are recorded, and l_2 transforms a copy of c to an abstract view that includes, in one form or another, the entire contents of c. Now suppose we edit the address in a_2. The *put* direction of l_2 will push the new address back in to $a(n)$. However, the *put* direction of l_1 can only try to restore the address from the old value stored in $c(m)$. So unless l_1 and l_2 preserve exactly the same set of information, there is no way to satisfy the type requirement that $a(m) = a(n)$. However, if l_1 and l_2 preserve exactly the same information, no more, no less, then there are very few useful or interesting lens that can be applied after the `copy`.

An alternative is to remove the constraint that $a(m) = a(n)$. However, a more permissive type for `copy` raises problems with respect to totality and well-behavedness. If we remove the equality constraint, then the *put* direction of copy must be defined even when $a(m)$ and $a(n)$ are unequal. If `copy` removes $a(n)$ in the *put* direction, then there is no way to restore the information in $a(n)$ in the *get* direction, and consequently PUTGET will not hold.

In our use of lenses to synchronize tree-structured data we have not experienced 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 (a) each application will locally maintain the invariants in its own representation, and (b) the function of Harmony is simply to propagate changes from one well-formed replica to another.

We can assume that the synchronizer will always be presented with abstract views in which the duplicated information is consistent, and so will only ever create such views. 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 is thus guaranteed to preserve the invariants of the concrete representation when the change to the single field in the abstract view is pushed back down to all the derived fields in the concrete view. In our setting, `merge`, the inverse of `copy`, makes far more sense than `copy`. Fortunately, because of the asymmetry of *get* and *put*, the problematic interaction with PUTGET does not arise when merging two equal subtrees in a concrete view, as we show in the next subsection.

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 [21, 32]. For example, consider editing a WYSIWYG view of a document in which the table of contents is automatically generated from the section headings in the text. One might feel that adding a new section should add an entry to the table of contents, and similarly that adding an entry to the table of contents should create an empty section in the text with an appropriate section title. 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 *put*; such a *put* function is not injective and cannot participate in a lens in our sense. In contexts where such functionality 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 [32] only require their bidirectional transformations to obey a PUTGETPUT law. PUTGETPUT is weaker than PUTGET in two ways. First, they do not require $l\nearrow(l\searrow(a,c))$ to equal a. Rather, they require 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, they allow *get* to be undefined over parts of the range of *put* — PUTGETPUT is only required to hold when it 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* of a *put* is undefined, then the system can signal the user that the modification to a was illegal and must be withdrawn). Hu et al [21] support `copy` functionality in a different way. They weaken *both* PUTGET and GETPUT by only requiring PUTGET to hold when a is already $l\nearrow(c)$, and by only requiring GETPUT to hold when c is $l\searrow(a,c')$ for some a and c'.

Merge

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

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

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

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

Note that `merge`, unlike `copy`, can be usefully given a more permissive type that removes the equality

constraint on the type of `merge`'s concrete view. We can define the behavior (`merge` $m\ n)\nearrow c$ even when the subtrees under m and n are unequal, so that `merge` is still total. Even though *get* may discard the subtree under n, we can restore it in the *put* direction, even if it were unequal to $c(m)$. We can preserve well-behavedness in this case, because the old value of c is passed back in the *put* direction.

5.5.3 Lemma [Well-behavedness]: $\forall m, n{\in}\mathcal{N}.\ \forall C{\subseteq}\mathcal{T}\backslash_{\{m,n\}}.\ \forall D{\subseteq}\mathcal{T}.$ `merge` $m\ n\ \in (C \cdot \{\!|m \mapsto D_\Omega,\ n \mapsto D_\Omega|\!\}) \stackrel{\Omega}{=} (C \cdot \{\!|m \mapsto D_\Omega|\!\})$.

Proof:
GET: Immediate: $(C \cdot \{\!|m \mapsto D_\Omega,\ n \mapsto D_\Omega|\!\})\backslash_n = C \cdot \{\!|m \mapsto D_\Omega|\!\}$.

PUT: By the form of the definition of the *put* direction of `merge`, there are two cases to consider: First, if $c(m) = c(n)$ (i.e., either both m and n are missing or both are present and their subtrees are equal, or c itself is Ω), then (`merge` $m\ n)\searrow(a,\ c) = a \cdot \{\!|n \mapsto a(m)|\!\}$. But this belongs to $C \cdot \{\!|m \mapsto D_\Omega,\ n \mapsto D_\Omega|\!\}$, since $a \in C \cdot \{\!|m \mapsto D_\Omega|\!\}$ and $a(m) \in D_\Omega$. Second, if $c(m) \neq c(n)$ (i.e., either one of m and n is missing and the other is not, or both are present but they lead to different subtrees), then (`merge` $m\ n)\searrow(a,\ c) = a \cdot \{\!|m \mapsto c(n)|\!\}$. But this again belongs to $C \cdot \{\!|m \mapsto D_\Omega,\ n \mapsto D_\Omega|\!\}$, since $a \in C \cdot \{\!|m \mapsto D_\Omega|\!\}$ and $c(n) \in D_\Omega$.

GETPUT: Suppose (`merge` $m\ n)\searrow((\texttt{merge}\ m\ n)\nearrow c,\ c)$ is defined. There are again two cases to consider. If $c(m) = c(n)$, then (`merge` $m\ n)\searrow((\texttt{merge}\ m\ n)\nearrow c,\ c) = (c\backslash_n) \cdot \{\!|n \mapsto (c\backslash_n)(m)|\!\} = (c\backslash_n) \cdot \{\!|n \mapsto c(n)|\!\} = c$. On the other hand, if $c(m) \neq c(n)$, then (`merge` $m\ n)\searrow((\texttt{merge}\ m\ n)\nearrow c,\ c) = (c\backslash_n) \cdot \{\!|n \mapsto c(n)|\!\} = c$.

PUTGET: Suppose (`merge` $m\ n)\nearrow((\texttt{merge}\ m\ n)\searrow(a,\ c))$ is defined. There are again two cases to consider. If $c(m) = c(n)$, then (`merge` $m\ n)\nearrow((\texttt{merge}\ m\ n)\searrow(a,\ c)) = (a \cdot \{\!|n \mapsto a(m)|\!\})\backslash_n = a$, since $n \notin \mathsf{dom}(a)$. On the other hand, if $c(m) \neq c(n)$, then (`merge` $m\ n)\nearrow((\texttt{merge}\ m\ n)\searrow(a,\ c)) = (a \cdot \{\!|n \mapsto c(n)|\!\})\backslash_n = a$. □

5.5.4 Lemma [Totality]: $\forall m, n{\in}\mathcal{N}.\ \forall C{\subseteq}\mathcal{T}\backslash_{\{m,n\}}.\ \forall D{\subseteq}\mathcal{T}.$ `merge` $m\ n\ \in (C \cdot \{\!|m \mapsto D_\Omega,\ n \mapsto D_\Omega|\!\}) \stackrel{\Omega}{\Longleftrightarrow} (C \cdot \{\!|m \mapsto D_\Omega|\!\})$.

Proof: The *get* direction of `merge` is a total function. In the *put* direction, the definedness of the $\cdot$ operation is guaranteed by the fact that $a \in (C \cdot \{\!|m \mapsto D_\Omega|\!\}) \subseteq \mathcal{T}\backslash_{\{n\}}$. □

6 Conditionals

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

The requirement that makes conditionals tricky is totality: we want to be able to take a concrete view, put it through our conditional lens to obtain some abstract view, and then take *any* other abstract view of suitable type and push it back down. But this will only work if either (1) we somehow ensure that the abstract view is guaranteed to be sent to the same sub-lens on the way down as we took on the way up, or else (2) the two sub-lenses are constrained to behave coherently. Since we want reasoning about well-behavedness and totality to be compositional in the absence of recursion (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), (2) 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 B on views and two lenses, l_1 and l_2. In the *get* direction, it tests the concrete view, c, and applies the *get* of l_1 if c satisfies the predicate and l_2 otherwise. In the *put* direction, `ccond` again examines the concrete view and applies the *put* of l_1 if it satisfies the predicate and l_2 otherwise. 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. (However, if we are interested in using `ccond` to define total lenses, this is actually a rather strong condition.)

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

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

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

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

One subtlety in the definition is worth noting: we arbitrarily choose to *put* Ω using l_2 (because $\Omega \notin C_1$ for any $C_1 \subseteq \mathcal{U}$). 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 *put* 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 *put* 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.

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

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

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

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

In Section 5.3, 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.

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

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

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

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

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

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

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

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

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

6.1 Lemma [Well-behavedness]: $\forall C, C_1, A_1, A_2 \subseteq \mathcal{U}.\ \forall l_1 \in (C \cap C_1) \stackrel{\Omega}{\rightleftharpoons} A_1.\ \forall l_2 \in (C \backslash C_1) \stackrel{\Omega}{\rightleftharpoons} A_2.\ \forall f_{21} \in (C \backslash C_1) \to (C \cap C_1)_\Omega.\ \forall f_{12} \in (C \cap C_1) \to (C \backslash C_1)_\Omega.\ \ \texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2 \in C \stackrel{\Omega}{\rightleftharpoons} (A_1 \cup A_2)$.

Proof:

GET: Suppose $c \in C$ and $l\nearrow c$ is defined, where, for brevity here and in the other proofs for `cond`, we write l for $(\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2)$. If $c \in C_1$, then $l\nearrow c = l_1\nearrow c \in A_1 \subseteq A_1 \cup A_2$ by the type of l_1. Otherwise, $l\nearrow c = l_2\nearrow c \in A_2 \subseteq A_1 \cup A_2$ by the type of l_2.

PUT: Suppose $(a, c) \in (A_1 \cup A_2) \times C_\Omega$ and $l \searrow (a,\ c)$ is defined. There are six cases to consider, one for each clause in the definition, and the result in each case is immediate from the typing of l_1 or l_2, as the case may be. Note, in particular, that the range of f_{21} falls within the source of l_1 in the fourth clause, and similarly for f_{12} and l_2 in the sixth clause.

GetPut: Suppose $c \in C$ and $l \searrow (l \nearrow c,\, c)$ is defined. If $c \in C_1$, then $l \nearrow c = l_1 \nearrow c$, which, by the type of l_1, belongs to A_1. So $l \searrow (l_1 \nearrow c,\, c) = l_1 \searrow (l_1 \nearrow c,\, c)$ by either the first or the third clause in the definition of $l \searrow$. This, in turn, is equal to c by GetPut for l_1. On the other hand, if $c \notin C_1$, then $l \nearrow c = l_2 \nearrow c$, which, by the type of l_2, belongs to A_2. So $l \searrow (l_2 \nearrow c,\, c) = l_2 \searrow (l_2 \nearrow c,\, c)$ by either the second or the fourth clause in the definition of $l \searrow$. This is equal to c by GetPut for l_2.

PutGet: Suppose $(a, c) \in (A_1 \cup A_2) \times C_\Omega$ and $l \nearrow (l \searrow (a,\, c))$ is defined. There are again six cases to consider:

1. If $a \in A_1 \cap A_2$ and $c \in C_1$, then $l \nearrow (l \searrow (a,\, c)) = l \nearrow (l_1 \searrow (a,\, c))$. But $l_1 \searrow (a,\, c) \in C_1$ by the type of l_1, so $l \nearrow (l_1 \searrow (a,\, c)) = l_1 \nearrow (l_1 \searrow (a,\, c)) = a$ by PutGet for l_1.

2. If $a \in A_1 \cap A_2$ and $c \notin C_1$, then $l \nearrow (l \searrow (a,\, c)) = l \nearrow (l_2 \searrow (a,\, c))$. But $l_2 \searrow (a,\, c) \in C_2$ by the type of l_2, so $l \nearrow (l_2 \searrow (a,\, c)) = l_2 \nearrow (l_2 \searrow (a,\, c)) = a$ by PutGet for l_2.

3. If $a \in A_1 \setminus A_2$ and $c \in (C_1)_\Omega$, then $l \nearrow (l \searrow (a,\, c)) = l \nearrow (l_1 \searrow (a,\, c))$. But $l_1 \searrow (a,\, c) \in C_1$ by the type of l_1, so $l \nearrow (l_1 \searrow (a,\, c)) = l_1 \nearrow (l_1 \searrow (a,\, c)) = a$ by PutGet for l_1.

4. If $a \in A_1 \setminus A_2$ and $c \notin (C_1)_\Omega$, then $l \nearrow (l \searrow (a,\, c)) = l \nearrow (l_1 \searrow (a,\, f_{21}(c)))$. But $l_1 \searrow (a,\, f_{21}(c)) \in C_1$ by the types of f_{21} and l_1, so $l \nearrow (l_1 \searrow (a,\, f_{21}(c))) = l_1 \nearrow (l_1 \searrow (a,\, f_{21}(c))) = a$ by PutGet for l_1.

5. If $a \in A_2 \setminus A_1$ and $c \notin C_1$, then $l \nearrow (l \searrow (a,\, c)) = l \nearrow (l_2 \searrow (a,\, c))$. But $l_2 \searrow (a,\, c) \in C_2$ by the type of l_2, so $l \nearrow (l_2 \searrow (a,\, c)) = l_2 \nearrow (l_2 \searrow (a,\, c)) = a$ by PutGet for l_2.

6. If $a \in A_2 \setminus A_1$ and $c \in C_1$, then $l \nearrow (l \searrow (a,\, c)) = l \nearrow (l_2 \searrow (a,\, f_{12}(c)))$. But $l_2 \searrow (a,\, f_{12}(c)) \in C_2$ by the types of f_{12} and l_2, so $l \nearrow (l_2 \searrow (a,\, f_{12}(c))) = l_2 \nearrow (l_2 \searrow (a,\, f_{12}(c))) = a$ by PutGet for l_2. □

6.2 Lemma [Totality]: $\forall C, C_1, A_1, A_2 \subseteq \mathcal{U}.\ \forall l_1 \in (C \cap C_1) \overset{\Omega}{\Longleftrightarrow} A_1.\ \forall l_2 \in (C \setminus C_1) \overset{\Omega}{\Longleftrightarrow} A_2.\ \forall f_{21} \in (C \setminus C_1) \to (C \cap C_1)_\Omega.\ \forall f_{12} \in (C \cap C_1) \to (C \setminus C_1)_\Omega.$ `cond` $C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2 \in C \overset{\Omega}{\Longleftrightarrow} (A_1 \cup A_2)$.

Proof: Straightforward: each clause in the definitions of $l \nearrow$ and $l \searrow$ directly invokes the corresponding part of either l_1 or l_2, from whose type the definedness of the result then follows. □

6.3 Lemma [Continuity]: Let F_1 and F_2 be continuous functions from lenses to lenses. Then the function $\lambda l.$ `cond` $C_1\ A_1\ A_2\ f_{21}\ f_{12}\ F_1(l)\ F_2(l)$ is continuous.

Proof: Details omitted—the argument is similar to other continuity proofs above. □

Before we introduced `cond`, we argued that it captured all the power of `ccond` and `acond`, and (because of the fixup functions f_{12} and f_{21}), more besides. We now argue 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 *put* 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 if $a \in A_1 \cap A_2$, 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 $c \in C \setminus C_1$, we must use l_2.

Similarly if $a \in A_1 \setminus A_2$, 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 $a \in A_2 \setminus A_1$, then we must use l_2. However, here we do have a little genuine freedom: if $a \in A_1 \setminus A_2$ while $c \in C \setminus C_1$, 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.)

Special Types for Conditional Lenses

In this section, we record some additional types that our conditional lenses inhabit, which we need for our proof that `list_filter`, defined in Section 7, is total. This material can be skimmed on a first reading.

The first theorem presents an alternate total type for `cond` where the target sets in the types of l_1, l_2 and the entire `cond` lens are intersected with an arbitrary set, A. Recall that the standard type for `ccond` takes two lenses with type $C \cap C_1 \overset{\Omega}{\Longleftrightarrow} A_1$ and $C \setminus C_1 \overset{\Omega}{\Longleftrightarrow} A_2$ (as well as conversion functions f_{21} and f_{12}) and produces a lens with type $C \overset{\Omega}{\Longleftrightarrow} A_1 \cup A_2$. This type is usually the type that we want. However, in some situations (when reasoning about totality), we need to show a *fixed* instance of `cond` has many different types. The abtract components of some of these types may be smaller than $(A_1 \cup A_2)$, where A_1 and A_2 appear literally in the *syntax* of the `ccond` instance. The new type presented here allows us to simplify some of these cases by only considering the lens type that is intersected with the abstract type we want, reducing the proof burden.

6.4 Theorem: The `cond` lens has the following types:

1. $\forall C, C_1, A, A_1, A_2 \subseteq \mathcal{U}.\ \forall l_1 \in (C \cap C_1) \overset{\Omega}{\rightleftharpoons} (A \cap A_1).\ \ \forall l_2 \in (C \setminus C_1) \overset{\Omega}{\rightleftharpoons} (A \cap A_2).\ \ \forall f_{21} \in (C \setminus C_1) \to (C \cap C_1)_\Omega.\ \ \forall f_{12} \in (C \cap C_1) \to (C \setminus C_1)_\Omega.\ \ \texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2 \in C \overset{\Omega}{\rightleftharpoons} (A \cap (A_1 \cup A_2))$.
2. $\forall C, C_1, A, A_1, A_2 \subseteq \mathcal{U}.\ \forall l_1 \in (C \cap C_1) \overset{\Omega}{\Longleftrightarrow} (A \cap A_1).\ \ \forall l_2 \in (C \setminus C_1) \overset{\Omega}{\Longleftrightarrow} (A \cap A_2).\ \ \forall f_{21} \in (C \setminus C_1) \to (C \cap C_1)_\Omega.\ \ \forall f_{12} \in (C \cap C_1) \to (C \setminus C_1)_\Omega.\ \ \texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2 \in C \overset{\Omega}{\Longleftrightarrow} (A \cap (A_1 \cup A_2))$.

Proof: We prove (1) by showing that the `cond` lens is well-behaved at $C \overset{\Omega}{\rightleftharpoons} (A \cap (A_1 \cup A_2))$, and then prove (2) by showing that that the lens is also total if both l_1 and l_2 are total.

GET: Suppose $c \in C$ and $l \nearrow c$ is defined. (Again, for brevity, we write l for $(\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2)$). If $c \in C_1$, then $l \nearrow c = l_1 \nearrow c \in (A \cap A_1) \subseteq (A \cap (A_1 \cup A_2))$ by the type of l_1. Otherwise, $l \nearrow c = l_2 \nearrow c \in (A \cap A_2) \subseteq (A \cap (A_1 \cup A_2))$ by the type of l_2.

PUT: Suppose $(a, c) \in (A \cap (A_1 \cup A_2)) \times C_\Omega$ and $l \searrow (a, c)$ is defined. There are six cases to consider, one for each clause in the definition, and the result in each case is immediate from the typing of l_1 or l_2, as the case may be. Note, in particular, that the range of f_{21} falls within the source of l_1 in the fourth clause, and similarly for f_{12} and l_2 in the sixth clause.

GETPUT: Suppose $c \in C$ and $l \searrow (l \nearrow c, c)$ is defined. If $c \in C_1$, then $l \nearrow c = l_1 \nearrow c$, which, by the type of l_1, belongs to $(A \cap A_1)$. So $l \searrow (l_1 \nearrow c, c) = l_1 \searrow (l_1 \nearrow c, c)$ by either the first or the third clause in the definition of $l \searrow$. This, in turn, is equal to c by GETPUT for l_1. On the other hand, if $c \notin C_1$, then $l \nearrow c = l_2 \nearrow c$, which, by the type of l_2, belongs to $(A \cap A_2)$. So $l \searrow (l_2 \nearrow c, c) = l_2 \searrow (l_2 \nearrow c, c)$ by either the second or the fourth clause in the definition of $l \searrow$. This is equal to c by GETPUT for l_2.

PUTGET Suppose $(a, c) \in (A \cap (A_1 \cup A_2)) \times C_\Omega$ and $l \nearrow (l \searrow (a, c))$ is defined. There are again six cases to consider:

1. If $a \in (A \cap (A_1 \cap A_2))$ and $c \in C_1$, then $l \nearrow (l \searrow (a, c)) = l \nearrow (l_1 \searrow (a, c))$. But $l_1 \searrow (a, c) \in C_1$ by the type of l_1, so $l \nearrow (l_1 \searrow (a, c)) = l_1 \nearrow (l_1 \searrow (a, c)) = a$ by PUTGET for l_1.
2. If $a \in (A \cap (A_1 \cap A_2))$ and $c \notin C_1$, then $l \nearrow (l \searrow (a, c)) = l \nearrow (l_2 \searrow (a, c))$. But $l_2 \searrow (a, c) \in C_2$ by the type of l_2, so $l \nearrow (l_2 \searrow (a, c)) = l_2 \nearrow (l_2 \searrow (a, c)) = a$ by PUTGET for l_2.
3. If $a \in (A \cap (A_1 \setminus A_2))$ and $c \in (C_1)_\Omega$, then $l \nearrow (l \searrow (a, c)) = l \nearrow (l_1 \searrow (a, c))$. But $l_1 \searrow (a, c) \in C_1$ by the type of l_1, so $l \nearrow (l_1 \searrow (a, c)) = l_1 \nearrow (l_1 \searrow (a, c)) = a$ by PUTGET for l_1.

4. If $a \in (A \cap (A_1 \backslash A_2))$ and $c \notin (C_1)_\Omega$, then $l\nearrow(l\searrow(a,\, c)) = l\nearrow(l_1\searrow(a,\, f_{21}(a,c)))$. But $l_1\searrow(a,\, f_{21}(a,c)) \in C_1$ by the types of f_{21} and l_1, so $l\nearrow(l_1\searrow(a,\, f_{21}(a,c))) = l_1\nearrow(l_1\searrow(a,\, f_{21}(a,c))) = a$ by PUTGET for l_1.

5. If $a \in (A \cap (A_2 \backslash A_1))$ and $c \notin C_1$, then $l\nearrow(l\searrow(a,\, c)) = l\nearrow(l_2\searrow(a,\, c))$. But $l_2\searrow(a,\, c) \in C_2$ by the type of l_2, so $l\nearrow(l_2\searrow(a,\, c)) = l_2\nearrow(l_2\searrow(a,\, c)) = a$ by PUTGET for l_2.

6. If $a \in (A \cap (A_2 \backslash A_1))$ and $c \in C_1$, then $l\nearrow(l\searrow(a,\, c)) = l\nearrow(l_2\searrow(a,\, f_{12}(a,c)))$. But $l_2\searrow(a,\, f_{12}(a,c)) \in C_2$ by the types of f_{12} and l_2, so $l\nearrow(l_2\searrow(a,\, f_{12}(a,c))) = l_2\nearrow(l_2\searrow(a,\, f_{12}(a,c))) = a$ by PUTGET for l_2. □

Hence, $l \in C \overset{\Omega}{\leftrightharpoons} A \cap (A_1 \cup A_2)$. Next we prove that l is total at that type if l_1 and l_2 are total, by showing that its *get* and *put* functions are totally defined on their domains.

We first show that the *get* function is totally defined on C. Pick $c \in C$. If $c \in C_1$ then $l\nearrow c = l_1\nearrow c$. As $l_1 \in C \cap C_1 \overset{\Omega}{\Longleftrightarrow} A \cap A_1$, it follows that $l_1\nearrow c$ is defined. Similarly, if $c \in (C \setminus C_1)$, then $l\nearrow c = l_2\nearrow c$. As $l_2 \in C \setminus C_1 \overset{\Omega}{\Longleftrightarrow} A \cap A_2$, it follows that $l_2\nearrow c$ is defined. Hence, $l\nearrow$ is a total function.

Second, we prove that the *put* function is totally defined on $(A \cap (A_1 \cup A_2)) \times C_\Omega$. There are six cases, corresponding to the six cases in the definition of the *put* function:

1. If $a \in (A \cap (A_1 \cap A_2))$ and $c \in C_1$, then $l\searrow(a,\, c) = l_1\searrow(a,\, c)$ is defined as $l_1\searrow$ is total on $(A \cap A_1) \times (C \cap C_1)_\Omega$.

2. If $a \in (A \cap (A_1 \cap A_2))$ and $c \notin C_1$, then $l\searrow(a,\, c) = l_2\searrow(a,\, c)$ is defined as $l_2\searrow$ is total on $(A \cap A_2) \times (C \setminus C_1)_\Omega$.

3. If $a \in (A \cap (A_1 \backslash A_2))$ and $c \in (C_1)_\Omega$, then $l\searrow(a,\, c) = l_1\searrow(a,\, c)$ is defined as $l_1\searrow$ is total on $(A \cap A_1) \times (C \cap C_1)_\Omega$.

4. If $a \in (A \cap (A_1 \backslash A_2))$ and $c \notin (C_1)_\Omega$, then $l\searrow(a,\, c) = l_1\searrow(a,\, f_{21}(c))$ is defined as f_{21} is a totally defined function with type: $(C \setminus C_1) \to (C \cap C_1)_\Omega$ and $l_1\searrow$ is total on $(A \cap A_1) \times (C \cap C_1)_\Omega$.

5. If $a \in (A \cap (A_2 \backslash A_1))$ and $c \notin C_1$, then then $l\searrow(a,\, c) = l_2\searrow(a,\, c)$ is defined as $l_2\searrow$ is total on $(A \cap A_2) \times (C \setminus C_1)_\Omega$.

6. If $a \in (A \cap (A_2 \backslash A_1))$ and $c \in C_1$, then $l\searrow(a,\, c) = l_2\searrow(a,\, f_{12}(c))$ is defined as f_{12} is a totally defined function with type: $(C \cap C_1) \to (C \setminus C_1)_\Omega$ and $l_2\searrow$ is total on $(A \cap A_2) \times (C \setminus C_1)_\Omega$.

Hence, $l\searrow$ is a total function.

We conclude that $(\texttt{cond}\ C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2) \in C \overset{\Omega}{\Longleftrightarrow} (A \cap (A_1 \cup A_2))$.

The next two theorems record types for conditional lenses in special cases where the conditional *always* selects one lens or the other (in both directions). In these situations, we can use a more flexible typing rule that makes no assumptions about the branch that is never used. The first describes `ccond` instances where the second branch is always taken.

6.5 Theorem [Always-False `ccond`]: $\forall C, C_1, A \subseteq \mathcal{U}$. *with* $C \cap C_1 = \emptyset$. $\forall l_2 \in C \setminus C_1 \overset{\Omega}{\Longleftrightarrow} A$. $\texttt{ccond}\ C_1\ l_1\ l_2 \in C \overset{\Omega}{\Longleftrightarrow} A$.

Proof: First we argue that $(\texttt{ccond}\ C_1\ l_1\ l_2) = l_2$ by showing that their respective *get* and *put* functions are identical. For any $c \in C$, we must have $c \notin (C_1 \cap C)$ (because it is empty) and so $c \in (C \setminus C_1)$. Hence, $(\texttt{ccond}\ C_1\ l_1\ l_2)\nearrow c = l_2\nearrow c$. Similarly, for any (a, c) in $A \times C_\Omega$, we must have $c \notin (C \cap C_1)$. By definition, $(\texttt{ccond}\ C_1\ l_1\ l_2)\searrow(a,\, c) = l_2\searrow(a,\, c)$.

Since $(\texttt{ccond}\ C_1\ l_1\ l_2) = l_2$, the well-behavedness and totality of the `ccond` lens follow from the well-behavedness and totality of l_2. In particular, since l_1 is never used, we do not need any assumptions about it. □

Note that there is no corresponding *always-true* rule for `ccond`. Even if $C \setminus C_1 = \emptyset$, in the *put* direction, the Ω tree still gets sent through l_2. However, for the generic conditional, `cond`, we can prove an *always-true* rule.

6.6 Theorem [Always-True `cond`]: $\forall C, C_1, A_1, A_2 \subseteq \mathcal{U}$. *with* $C \cap C_1 \neq \emptyset$ *and* $C \setminus C_1 = \emptyset$. $\forall l_1 \in C \cap C_1 \overset{\Omega}{\Longleftrightarrow} A_1$. `cond` $C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2 \in C \overset{\Omega}{\Longleftrightarrow} A_1$.

Proof: First we argue that (`cond` $C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2) = l_1$ by showing that their respective *get* and *put* functions are identical. For any $c \in C$ since $(C \cap C_1) \neq \emptyset$ and $(C \setminus C_1) = \emptyset$ we must have $c \in (C \cap C_1)$. Thus, by definition, (`cond` $C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2) \nearrow c = l_1 \nearrow c$. Similarly, for any (a, c) in $A_1 \times C_\Omega$, either $c = \Omega$ or $c \in (C \cap C_1)$; hence, by definition, (`cond` $C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2) \searrow (a,\, c) = l_1 \searrow (a,\, c)$.

Since (`cond` $C_1\ A_1\ A_2\ f_{21}\ f_{12}\ l_1\ l_2) = l_1$, the well-behavedness and totality of the `cond` lens follow from the well-behavedness and totality of l_1. In particular, since l_2 and the conversion functions f_{21} and f_{12} are never used, we do not need any assumptions about them. □

7 Derived Lenses for Lists

XML and many other concrete data formats make heavy use of ordered lists. We describe in this section how we can represent lists as trees, using a standard cons cell encoding, and introduce some derived lenses to manipulate them. We begin with some very simple lenses for projecting the head and tail of a list encoded as a cons cell. We then define some recursive lenses implementing more complex operations on lists: mapping, reversal, and filtering. The simplest of these lenses, `list_map`, uses `wmap` and recursion to apply a lens to every element of a list. The next lens reverses the order of elements in a list. We conclude with a quite intricate derived form, `list_filter`, that uses the general conditional, `cond`, to filter lists according to some predicate.

Other list-processing derived forms that we have implemented (but do not show here) include a "grouping" lens that, in the *get* direction, takes a list whose elements alternate between elements of D and elements of E and returns a list of pairs of Ds and Es—e.g., it maps `[d1 e1 d2 e2 d3 e3]` to `[[d1 e1] [d2 e2] [d3 e3]]`.

Encoding

7.1 Definition: A tree t is said to be a *list* iff either it is empty (no children) or it has exactly two children, one named `*h` and another named `*t`, with t(`*t`) also a list. In the following, we use the lighter notation $[t_1 \ldots t_n]$ for the tree:

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

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 lists of Ds whose length is at least i and at most j. The interleaving of a list of type `[`$B^{i..j}$`]` and a list of type `[`$C^{n..m}$`]`, taking elements from the first list and elements from the second in an arbitrary fashion but maintaining the relative order of each, is written `[`$B^{i..j}$`]`&`[`$C^{m..n}$`]`.

Head and Tail Projections

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

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

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

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

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

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

Our next lens, `hoist_hd`, takes a list and "flattens" its first cell using `hoist_nonunique`. It is annotated with a set of names p specifying the possible domain of the tree at the head of the list. We will need this operation for one step of the HTML processing in the example in Section 8.

$$\texttt{hoist\_hd}\ p \quad = \quad \texttt{hoist\_nonunique}\ \texttt{*h}\ p;\ \texttt{hoist\_nonunique}\ \texttt{*t}\ \overline{p}$$

$$\forall p \subseteq (\mathcal{N} \backslash \{\texttt{*t}\}).\ \forall C \subseteq (\mathcal{T}|_p).\ \forall D \subseteq (\mathcal{T}\backslash_p). \quad \texttt{hoist\_hd}\ p \in (C :: D) \overset{\Omega}{\Longleftrightarrow} (C \cdot D)$$

Observe that, by assumption, the concrete view has type $C :: D$ where $C \in \mathcal{T}|_p$ and $D \in \mathcal{T}\backslash_p$. Then

$$\texttt{hoist\_nonunique}\ \texttt{*h}\ p \in C :: D \overset{\Omega}{=\!=} C \cdot \left\{\!\left|\texttt{*t} \mapsto D\right|\!\right\}$$

and also

$$\texttt{hoist\_nonunique}\ \texttt{*t}\ \overline{p} \in C \cdot \left\{\!\left|\texttt{*t} \mapsto D\right|\!\right\} \overset{\Omega}{=\!=} C \cdot D$$

yielding the desired result for the composition.

List Map

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

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

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

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

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

It is worth noting how the recursive calls in `list_map` terminate. In the *get* direction, the `wmap` lens simply applies l to the head and `list_map` l to the tail until it reaches a tree with no children. Similarly, in the *put* direction, the lens is applied at each level of the abstract tree, using the corresponding part of the concrete tree, if it is present, and Ω otherwise. In either case, the recursive calls continue until the entire tree has been traversed.

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

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

7.2 Lemma: $\forall S, T \subseteq \mathcal{T}.(S :: T) = (S :: T)^{\circlearrowleft}$.

Proof: We calculate $(S :: T)^{\circlearrowleft}$ directly. From the definition of cons cells, the set $\mathsf{dom}(S :: T)$ of possible domains of trees in $(S :: T)$ is $\{\{\texttt{*h}, \texttt{*t}\}\}$. We then calculate $(S :: T)^{\circlearrowleft}$ as:

$$\begin{array}{rcl}(S :: T)^{\circlearrowleft} & = & \bigcup_{D \in \mathsf{dom}(S::T)} \{\!|n \mapsto (S :: T)(n) \mid n \in D|\!\} \\ & = & \{\!|\texttt{*h} \mapsto S,\ \texttt{*t} \mapsto T|\!\} \\ & = & S :: T.\end{array}$$

□

7.3 Lemma: $\forall T \subseteq \mathcal{T}.[\![T]\!] = [\![T]\!]^{\circlearrowleft}$.

Proof: We calculate $[\![T]\!]^{\circlearrowleft}$ directly. From the definition of lists, the set $\mathsf{dom}([\![T]\!])$ of domains of trees in $[\![T]\!]$ is $\{\emptyset, \{\texttt{*h}, \texttt{*t}\}\}$. We then calculate $[\![T]\!]^{\circlearrowleft}$ as:

$$\begin{array}{rcl}[\![T]\!]^{\circlearrowleft} & = & \bigcup_{D \in \mathsf{dom}([\![T]\!])} \{\!|n \mapsto [\![T]\!](n) \mid n \in D|\!\} \\ & = & \{\!|\}\!\} \cup \{\!|\texttt{*h} \mapsto T,\ \texttt{*t} \mapsto [\![T]\!]|\!\} \\ & = & [\![T]\!].\end{array}$$

□

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

Proof: Note that `list_map` l is the fixed point of the function: $f = \lambda k.\ \texttt{wmap}\ \{\texttt{*h} \mapsto l,\ \texttt{*t} \mapsto k\}$. We use Corollary 3.3.8 (1), which states that if, assuming that $k \in [\![C]\!] \stackrel{\Omega}{\rightleftharpoons} [\![A]\!]$, we can prove $f(k) \in [\![C]\!] \stackrel{\Omega}{\rightleftharpoons} [\![A]\!]$, then $\mathit{fix}(f) \in [\![C]\!] \stackrel{\Omega}{\rightleftharpoons} [\![A]\!]$.

We assume that $k \in [\![C]\!] \stackrel{\Omega}{\rightleftharpoons} [\![A]\!]$ and show that $f(k)$ has type $[\![C]\!] \stackrel{\Omega}{\rightleftharpoons} [\![A]\!]$ directly, using the type of `wmap`. We write m for the total function from names to lenses described by $\{\texttt{*h} \mapsto l,\ \texttt{*h} \mapsto k\}$; i.e., m maps `*h` to l, `*t` to k, and every other name to `id`. We first show that $m \in (\Pi n \in \mathcal{N}.\ C(n) \stackrel{\Omega}{\rightleftharpoons} A(n))$:

$$\begin{array}{rrcl} & m(\texttt{*h}) = l & \in & [\![C]\!](\texttt{*h}) \stackrel{\Omega}{\rightleftharpoons} [\![A]\!](\texttt{*h}) \\ \text{i.e.,} & m(\texttt{*h}) = l & \in & C \stackrel{\Omega}{\rightleftharpoons} A \\ \multicolumn{4}{l}{\text{by the type of } l;} \end{array}$$

$$\begin{array}{rrcl} & m(\texttt{*t}) = k & \in & [\![C]\!](\texttt{*t}) \stackrel{\Omega}{\rightleftharpoons} [\![A]\!](\texttt{*t}) \\ \text{i.e.,} & m(\texttt{*t}) = k & \in & [\![C]\!] \stackrel{\Omega}{\rightleftharpoons} [\![A]\!] \\ \multicolumn{4}{l}{\text{by assumption;}} \end{array}$$

$$\begin{array}{rrcll} & m(n) = \texttt{id} & \in & [\![C]\!](n) \stackrel{\Omega}{\rightleftharpoons} [\![A]\!](n) & \forall n \notin \{\texttt{*h}, \texttt{*t}\} \\ \text{i.e.,} & m(n) = \texttt{id} & \in & \emptyset \stackrel{\Omega}{\rightleftharpoons} \emptyset & \\ \multicolumn{5}{l}{\text{vacuously.}} \end{array}$$

Hence, m has the correct type. The type of `wmap` also requires that both $[\![C]\!]$ and $[\![A]\!]$ be shuffle closed and that $\mathsf{dom}([\![C]\!]) = \mathsf{dom}([\![A]\!])$. The first condition follows from Lemma 7.3; the second condition is immediate as both $\mathsf{dom}([\![C]\!])$ and $\mathsf{dom}([\![A]\!])$ are the set $\{\{\texttt{*h}, \texttt{*t}\}, \emptyset\}$.

Using the type of `wmap`, we conclude that $f(k) \in [\![C]\!] \stackrel{\Omega}{\rightleftharpoons} [\![A]\!]$ and by Corollary 3.3.8, that $\mathit{fix}(f) =$ `list_map` $l \in [\![C]\!] \stackrel{\Omega}{\rightleftharpoons} [\![A]\!]$. □

The proof of totality for `list_map` is more interesting. We use Corollary 3.3.8 (2), noting again that `list_map` l is the fixed point of the function f defined above. The corollary 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 $k \in C_i \stackrel{\Omega}{\Longleftrightarrow} A_i$, prove that $f(k) \in C_{i+1} \stackrel{\Omega}{\Longleftrightarrow} A_{i+1}$ for all i. We can then conclude that $\mathit{fix}(f) \in \bigcup_i C_i \stackrel{\Omega}{\Longleftrightarrow} \bigcup_i A_i$.

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

Proof: We pick these two chains of types:

$$C_0 = A_0 = \emptyset$$
$$C_{i+1} = \texttt{[}C^{0..i}\texttt{]}$$
$$A_{i+1} = \texttt{[}A^{0..i}\texttt{]}$$

Next, we show that $f(l) \in C_{i+1} \overset{\Omega}{\Longleftrightarrow} A_{i+1}$. The case $i = 0$ is immediate because $C_1 = A_1 = \texttt{[]}$ and $\texttt{list\_map}\ l \in \texttt{[]} \overset{\Omega}{\Longleftrightarrow} \texttt{[]}$. For the case $i > 0$, we calculate the type of $f(l)$ directly from the type of `wmap`. As above, we write m for the function that maps `*h` to l, `*t` to k and every other n to `id`. From the assumption that $k \in C_i \overset{\Omega}{\Longleftrightarrow} A_i$, we have

$$\begin{array}{lrcrcl} & m(\texttt{*h}) = l & \in & \texttt{[}C^{0..i+1}\texttt{]}(\texttt{*h}) & \overset{\Omega}{\Longleftrightarrow} & \texttt{[}A^{0..i+1}\texttt{]}(\texttt{*h}) \\ \text{i.e.,} & m(\texttt{*h}) = l & \in & C & \overset{\Omega}{\Longleftrightarrow} & A \end{array}$$

by $i > 0$ and the type of l;

$$\begin{array}{lrcrcl} & m(\texttt{*t}) = k & \in & \texttt{[}C^{0..i+1}\texttt{]}(\texttt{*t}) & \overset{\Omega}{\Longleftrightarrow} & \texttt{[}A^{0..i+1}\texttt{]}(\texttt{*t}) \\ \text{i.e.,} & m(\texttt{*t}) = k & \in & \texttt{[}C^{0..i}\texttt{]} & \overset{\Omega}{\Longleftrightarrow} & \texttt{[}A^{0..i}\texttt{]} \end{array}$$

by assumption; and

$$\begin{array}{lrcrcll} & m(n) = \texttt{id} & \in & \texttt{[}C^{0..i+1}\texttt{]}(n) & \overset{\Omega}{\Longleftrightarrow} & \texttt{[}A^{0..i+1}\texttt{]}(n) & \forall n \notin \{\texttt{*h}, \texttt{*t}\} \\ \text{i.e.,} & m(n) = \texttt{id} & \in & \emptyset & \overset{\Omega}{\Longleftrightarrow} & \emptyset & \end{array}$$

vacuously.

As above, both $\texttt{[}C^{0..i+1}\texttt{]}$ and $\texttt{[}A^{0..i+1}\texttt{]}$ are shuffle closed and they have equal domains. Using the type of `wmap`, we conclude that $f(k) \in \texttt{[}C^{0..i+1}\texttt{]} \overset{\Omega}{\Longleftrightarrow} \texttt{[}A^{0..i+1}\texttt{]}$, and hence

$$\begin{array}{lrcrcl} & \texttt{list\_map}\ l & \in & \bigcup_i C_i & \overset{\Omega}{\Longleftrightarrow} & \bigcup_i A_i \\ \text{i.e.,} & \texttt{list\_map}\ l & \in & (\emptyset \cup \bigcup_i \texttt{[}C^{0..i}\texttt{]}) & \overset{\Omega}{\Longleftrightarrow} & (\emptyset \cup \bigcup_i \texttt{[}A^{0..i}\texttt{]}) \\ \text{i.e.,} & \texttt{list\_map}\ l & \in & \texttt{[}C\texttt{]} & \overset{\Omega}{\Longleftrightarrow} & \texttt{[}A\texttt{]}, \end{array}$$

which finishes the proof. □

Reverse

Our next lens reverses the elements of a list.[5]

The algorithm we use to implement list reversal is a quadratic-time algorithm—we reverse the tail of the list and then use an auxilary lens to append the old head to the end of the reversed tail. Before presenting the `list_reverse` lens, we describe this auxiliary lens, called `snoc`. The *get* direction of `snoc` m transforms a bush consisting of a child m adjoined to a list (either the empty view or children `*h` and `*t`) into a non-empty list where the tree under m is the last element.

```
snoc m  =  acond {|m ↦ D|} (D :: [])
             (add *t {|}; rename m *h)
             (xfork {m *t} {*t}
               (hoist_nonunique *t {*h *t};
                snoc m;
                plunge *t)
               (id))
```

$$\forall D \subseteq \mathcal{T}. \quad \texttt{snoc}\ m \in (\{|m \mapsto D|\} \cdot \texttt{[}D\texttt{]}) \overset{\Omega}{\Longleftrightarrow} \texttt{[}D^{1..\omega}\texttt{]}$$

[5] Malo Denielou has recently suggested a different way of implementing `reverse` that is arguably somewhat more intuitive, but we have not yet pushed through the full proof that it is total (though it appears to be).

In the *get* direction, `snoc` has two cases. If the tree has a single child m then `snoc` m builds a singleton list by renaming the m to `*h` and adding an empty tail. Otherwise, it moves the child m under the tail tag, `*t`, and recurses, leaving the head in place.

The *put* direction tests whether the abstract view is a singleton list. If it is a singleton, then the lens renames the head of the list to m and uses the *put* of the `add` lens to strip away the empty tail. Otherwise, it uses the *put* of `xfork` to split the abstract and concrete views into children under `*t` and `*h`. The head is then put back using `id`; the tail is passed through the composition in reverse: (1) `plunge *t`, which hoists up the tail and yields a list, (2) a recursive call, which removes the last element of the list from the first step and places it under the child named m, and finally (3) `hoist_nonunique *t *h *t`, which plunges `*h` and `*t` under `*t`, leaving m at the top level of the tree.

7.6 Lemma [Well-behavedness]: $\forall D \subseteq \mathcal{T}.\ \texttt{snoc}\ m \in (\{\!|m \mapsto D|\!\} \cdot \texttt{[}D\texttt{]}) \stackrel{\Omega}{\rightleftharpoons} \texttt{[}D^{1..\omega}\texttt{]}$.

Proof: First, note that `snoc` m is the fixed point of the function:

$$
\begin{array}{l}
f = \lambda l.\ \texttt{acond}\ \{\!|m \mapsto D|\!\}\ (D :: \texttt{[]}) \\
\qquad (\texttt{add *t}\ \{\!|\}\!|;\ \texttt{rename}\ m\ \texttt{*h}) \\
\qquad (\texttt{xfork}\ \{m\ \texttt{*t}\}\ \{\texttt{*t}\} \\
\qquad\quad (\texttt{hoist\_nonunique *t}\ \{\texttt{*h *t}\}; \\
\qquad\quad\ l; \\
\qquad\quad\ \texttt{plunge *t}) \\
\qquad\quad (\texttt{id}))
\end{array}
$$

In the rest of the proof, we use the following abbreviations:

$$
\begin{array}{rcl}
C & = & \{\!|m \mapsto D|\!\} \cdot \texttt{[}D\texttt{]} \\
A & = & \texttt{[}D^{1..\omega}\texttt{]} \\
C_1 & = & \{\!|m \mapsto D|\!\} \\
A_1 & = & D :: \texttt{[]}
\end{array}
$$

The structure of the proof is the same as for the well-behavedness proof for `list_map`. We assume that $l \in C \stackrel{\Omega}{\rightleftharpoons} A$ and prove that $f(l) \in C \stackrel{\Omega}{\rightleftharpoons} A$. Using Corollary 3.3.8 (1), we conclude that $\mathit{fix}(f) = \texttt{snoc}\ m \in C \stackrel{\Omega}{\rightleftharpoons} A$.

We calculate the type of $f(l)$, working top down. The outermost lens is an `acond` instance; we must prove that the first branch has this type:

$$
\begin{array}{rl}
 & (\texttt{add *t}\ \{\!|\}\!|; \\
 & \texttt{rename}\ m\ \texttt{*h}) \quad \in \quad C \cap C_1 \quad \stackrel{\Omega}{\rightleftharpoons} \quad A \cap A_1 \\
\text{i.e.,} & (\texttt{add *t}\ \{\!|\}\!|; \\
 & \texttt{rename}\ m\ \texttt{*h}) \quad \in \quad \{\!|m \mapsto D|\!\} \quad \stackrel{\Omega}{\rightleftharpoons} \quad D :: \texttt{[]}
\end{array}
$$

which follows from the types of `add` and `rename`.

Similarly, we must show that the second branch has this type:

$$
\begin{array}{rl}
 & \texttt{xfork}\ \{m\ \texttt{*t}\}\ \{\texttt{*t}\} \\
 & \quad (\texttt{hoist\_nonunique *t}\ \{\texttt{*h *t}\};\ l;\ \texttt{plunge *t}) \\
 & \quad (\texttt{id}) \qquad\qquad \in \qquad\qquad C \setminus C_1 \quad \stackrel{\Omega}{\rightleftharpoons} \quad A \setminus A_1 \\
\text{i.e.,} & \texttt{xfork}\ \{m\ \texttt{*t}\}\ \{\texttt{*t}\} \\
 & \quad (\texttt{hoist\_nonunique *t}\ \{\texttt{*h *t}\};\ l;\ \texttt{plunge *t}) \\
 & \quad (\texttt{id}) \qquad\qquad \in \quad \{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{1..\omega}\texttt{]} \quad \stackrel{\Omega}{\rightleftharpoons} \quad \texttt{[}D^{2..\omega}\texttt{]}.
\end{array}
$$

To prove this type for the second branch we show that the two arms of the `xfork` have these types,

$$\begin{array}{rcll} k_1 & = & (\texttt{hoist\_nonunique *t } \{\texttt{*h *t}\}; & \\ & & l; & \\ & & \texttt{plunge *t}) & \in \quad \{\!|m \mapsto D,\ \texttt{*t} \mapsto [D]|\!\} \quad \stackrel{\Omega}{=\!=} \quad \{\!|\texttt{*t} \mapsto [D^{1..\omega}]|\!\} \\ \text{i.e.,} & k_1 & = & \in \{\!|m \mapsto D,\ \texttt{*t} \mapsto [D]|\!\} \\ & & \texttt{hoist\_nonunique *t } \{\texttt{*h *t}\}; & : \quad \{\!|m \mapsto D|\!\} \cdot [D] \\ & & l & : \quad [D^{1..\omega}] \\ & & \texttt{plunge *t} & \stackrel{\Omega}{=\!=} \quad \{\!|\texttt{*t} \mapsto [D^{1..\omega}]|\!\} \end{array}$$

which follows from the types of `hoist_nonunique`, l, `plunge` and the composition operator;

and $k_2 = \texttt{id} \in \{\!|\texttt{*h} \mapsto D|\!\} \stackrel{\Omega}{=\!=} \{\!|\texttt{*h} \mapsto D|\!\}$
immediately, by the type of `id`,

and note that

$$\begin{array}{rcl} \{\!|m \mapsto D,\ \texttt{*t} \mapsto [D]|\!\} & \subseteq & \mathcal{T}|_{\{m,\,\texttt{*t}\}} \\ \{\!|\texttt{*t} \mapsto [D^{1..\omega}]|\!\} & \subseteq & \mathcal{T}|_{\texttt{*t}} \end{array}$$

$$\begin{array}{rcl} \{\!|\texttt{*h} \mapsto D|\!\} & \subseteq & \mathcal{T}\backslash_{\{m,\,\texttt{*t}\}} \\ \{\!|\texttt{*h} \mapsto D|\!\} & \subseteq & \mathcal{T}\backslash_{\texttt{*t}} \end{array}$$

$$\begin{array}{rcl} \{\!|m \mapsto D|\!\} \cdot [D^{1..\omega}] & = & (\{\!|m \mapsto D,\ \texttt{*t} \mapsto [D]|\!\}) \cdot (\{\!|\texttt{*h} \mapsto D|\!\}) \\ [D^{2..\omega}] & = & (\{\!|\texttt{*t} \mapsto [D^{1..\omega}]|\!\}) \cdot (\{\!|\texttt{*h} \mapsto D|\!\}). \end{array}$$

We use all of these facts, and the type of `xfork`, to prove:

$$\texttt{xfork } \{m\ \texttt{*t}\}\ \{\texttt{*t}\}\ k_1\ k_2 \in \{\!|m \mapsto D|\!\} \cdot [D^{1..\omega}] \stackrel{\Omega}{=\!=} [D^{2..\omega}].$$

We conclude that the `acond` instance (i.e., $f(l)$) has type $C \stackrel{\Omega}{=\!=} A$, and so, by Corollary 3.3.8 (1), that $fix(f) = \texttt{snoc}\ m$ has the same type. □

7.7 Lemma [Totality]: $\forall D \subseteq \mathcal{T}.\ \ \texttt{snoc}\ m \in (\{\!|m \mapsto D|\!\} \cdot [D]) \stackrel{\Omega}{\Longleftrightarrow} [D^{1..\omega}]$.

Proof: To prove that `snoc` m is total, we use Corollary 3.3.8 (2). Let

$$\begin{array}{l} C_0 = A_0 = \emptyset \\ C_{i+1} = \{\!|m \mapsto D|\!\} \cdot [D^{0..i}] \\ A_{i+1} = [D^{1..i+1}] \end{array}$$

be two chains of types. Again, we note that `snoc` m is the fixed point of the same function f described in the well-behavedness proof. In the rest of this proof, we use the following abbreviations:

$$\begin{array}{rcl} C_1 & = & \{\!|m \mapsto D|\!\} \\ A_1 & = & D :: [] \end{array}$$

We prove, by induction on i, that if $l \in C_i \stackrel{\Omega}{\Longleftrightarrow} A_i$ then $f(l) \in C_{i+1} \stackrel{\Omega}{\Longleftrightarrow} A_{i+1}$. Let $C = C_{i+1} = (\{\!|m \mapsto D|\!\} \cdot [D^{0..i}])$ and $A = A_{i+1} = [D^{1..i+1}]$. To show that $f(l) \in C \stackrel{\Omega}{\Longleftrightarrow} A$, we must show that the `acond` instance also has that type.

We first prove that the each branch has the correct type. The first branch is straightforward:

$$\begin{array}{rcll} & (\texttt{add *t } \{\!|\}; \texttt{ rename } m\ \texttt{*h}) & \in \quad C \cap C_1 & \stackrel{\Omega}{\Longleftrightarrow} \quad A \cap A_1 \\ \text{i.e.,} & (\texttt{add *t } \{\!|\}; \texttt{ rename } m\ \texttt{*h}) & \in \quad \{\!|m \mapsto D|\!\} & \stackrel{\Omega}{\Longleftrightarrow} \quad (D :: []) \end{array}$$

which follows from the type of `add`, `rename` and composition;

The second branch must have type:

$$\begin{array}{lll} & \texttt{xfork}\ \{m\ \texttt{*t}\}\ \{\texttt{*t}\} & \\ & \quad (\texttt{hoist\_nonunique}\ \texttt{*t}\ \{\texttt{*h}\ \texttt{*t}\};\ l;\ \texttt{plunge}\ \texttt{*t}) & \\ & \quad (\texttt{id}) & \in \quad C \setminus C_1 \quad \overset{\Omega}{\Longleftrightarrow} \quad A \setminus A_1 \\ \text{i.e.,} & \texttt{xfork}\ \{m\ \texttt{*t}\}\ \{\texttt{*t}\} & \\ & \quad (\texttt{hoist\_nonunique}\ \texttt{*t}\ \{\texttt{*h}\ \texttt{*t}\};\ l;\ \texttt{plunge}\ \texttt{*t}) & \\ & \quad (\texttt{id}) & \in \quad (\{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{0..i}\texttt{]}) \setminus \{\!|m \mapsto D|\!\} \\ & & \quad \overset{\Omega}{\Longleftrightarrow} \texttt{[}D^{1..i+1}\texttt{]} \setminus (D :: \texttt{[]}) \end{array}$$

There are two cases.

Case $i = 0$ We calculate that the second branch must have concrete and abstract type components:

$$\begin{array}{rcl} (\{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{0..0}\texttt{]}) \setminus \{\!|m \mapsto D|\!\} & = & \emptyset \\ \texttt{[}D^{1..1}\texttt{]} \setminus (D :: \texttt{[]}) & = & \emptyset, \end{array}$$

which vacuously holds.

Case $i > 0$*:* We calculate that the second branch must have concrete and abstract type components:

$$\begin{array}{rcl} (\{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{0..i}\texttt{]}) \setminus \{\!|m \mapsto D|\!\} & = & \{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{1..i}\texttt{]} \\ \texttt{[}D^{1..i+1}\texttt{]} \setminus (D :: \texttt{[]}) & = & \texttt{[}D^{2..i+1}\texttt{]} \end{array}$$

To show that the second branch has this type, we must prove that each arm of the `xfork` lens has the correct type:

$$\begin{array}{lll} k_1 \quad = \quad (\texttt{hoist\_nonunique}\ \texttt{*t}\ \{\texttt{*h}\ \texttt{*t}\}; & & \\ \qquad l; & & \\ \qquad \texttt{plunge}\ \texttt{*t}) & \in \quad \{\!|m \mapsto D, \texttt{*t} \mapsto \texttt{[}D^{0..i-1}\texttt{]}|\!\} \quad \overset{\Omega}{\Longleftrightarrow} \quad \{\!|\texttt{*t} \mapsto \texttt{[}D^{1..i}\texttt{]}|\!\} \\ \text{i.e.,} \quad k_1 \quad = & \in \{\!|m \mapsto D, \texttt{*t} \mapsto \texttt{[}D^{0..i-1}\texttt{]}|\!\} & \\ \qquad \texttt{hoist\_nonunique}\ \texttt{*t}\ \{\texttt{*h}\ \texttt{*t}\}; & : \quad \{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{0..i-1}\texttt{]} & \\ \qquad l & : \quad \texttt{[}D^{1..i}\texttt{]} & \\ \qquad \texttt{plunge}\ \texttt{*t} & \overset{\Omega}{\Longleftrightarrow} \quad \{\!|\texttt{*t} \mapsto \texttt{[}D^{1..i}\texttt{]}|\!\} & \end{array}$$

which follows from the types of `hoist_nonunique`, l (using the induction hypothesis), `plunge`, and the composition operator;

$$k_2 \quad = \quad \texttt{id} \qquad \in \qquad \{\!|\texttt{*h} \mapsto D|\!\} \quad \overset{\Omega}{\Longleftrightarrow} \quad \{\!|\texttt{*h} \mapsto D|\!\}$$

immediately, by the type of `id`,

and observe that

$$\begin{array}{rcl} \{\!|m \mapsto D, \texttt{*t} \mapsto \texttt{[}D^{0..i}\texttt{]}|\!\} & \subseteq & \mathcal{T}|_{\{m,\,\texttt{*t}\}} \\ \{\!|\texttt{*t} \mapsto \texttt{[}D^{1..i}\texttt{]}|\!\} & \subseteq & \mathcal{T}|_{\texttt{*t}} \\ \\ \{\!|\texttt{*h} \mapsto D|\!\} & \subseteq & \mathcal{T}\backslash_{\{m,\,\texttt{*t}\}} \\ \{\!|\texttt{*h} \mapsto D|\!\} & \subseteq & \mathcal{T}\backslash_{\texttt{*t}} \end{array}$$

$$\begin{array}{rcl} \{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{1..i}\texttt{]} & = & (\{\!|m \mapsto D, \texttt{*t} \mapsto \texttt{[}D^{0..i-1}\texttt{]}|\!\}) \cdot (\{\!|\texttt{*h} \mapsto D|\!\}) \\ \texttt{[}D^{2..i+1}\texttt{]} & = & (\{\!|\texttt{*t} \mapsto \texttt{[}D^{1..i}\texttt{]}|\!\}) \cdot (\{\!|\texttt{*h} \mapsto D|\!\}). \end{array}$$

We use all of these facts to prove that `xfork` has the following type:

$$\texttt{xfork}\ \{m\ \texttt{*t}\}\ \{\texttt{*t}\}\ k_1\ k_2 \in \{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{1..i}\texttt{]} \overset{\Omega}{\Longleftrightarrow} \texttt{[}D^{2..i+1}\texttt{]}$$

Then by the type of `acond`, we have $f(l) \in C \overset{\Omega}{\Longleftrightarrow} A$, which finishes the case and the inductive proof. Using Corollary 3.3.8 (2), we conclude that

$$\begin{array}{lll} \mathit{fix}(f) \quad = \quad \texttt{snoc}\ m \quad \in & \bigcup_i C_i & \overset{\Omega}{\Longleftrightarrow} \quad \bigcup_i A_i \\ \text{i.e.,} \quad \texttt{snoc}\ m \quad \in & \emptyset \cup \bigcup_i \{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{0..i}\texttt{]} & \overset{\Omega}{\Longleftrightarrow} \quad \emptyset \cup \bigcup_i \texttt{[}D^{1..i+1}\texttt{]} \\ \text{i.e.,} \quad \texttt{snoc}\ m \quad \in & \{\!|m \mapsto D|\!\} \cdot \texttt{[}D\texttt{]} & \overset{\Omega}{\Longleftrightarrow} \quad \texttt{[}D^{1..\omega}\texttt{]} \end{array}$$

as required. □

7.8 Lemma: To prove that `list_reverse` is total, we will use the following precise total type for `snoc` m: $\forall i.\ \texttt{snoc}\ m \in (\{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{0..i}\texttt{]}) \overset{\Omega}{\Longleftrightarrow} \texttt{[}D^{1..i+1}\texttt{]}$.

Proof: The proof is by induction on i. As above, we use the following abbreviations in the proof:

$$\begin{array}{lcl} C_1 & = & \{\!|m \mapsto D|\!\} \\ A_1 & = & D :: \texttt{[]} \end{array}$$

For the base case, $i = 0$, we must show that $\texttt{snoc}\ m \in C \overset{\Omega}{\Longleftrightarrow} A$ where $C = \{\!|m \mapsto D|\!\} \cdot \texttt{[]}$ and $A = \texttt{[}D^{1..1}\texttt{]}$. The outermost lens is an `acond` instance so we must first prove that the each branch has the correct type. The type of the first branch is straightforward:

$$\begin{array}{lcccl} & (\texttt{add *t}\ \{\!|\}; \texttt{rename}\ m\ \texttt{*h}) & \in & C \cap C_1 \overset{\Omega}{\Longleftrightarrow} A \cap A_1 \\ \text{i.e.,} & (\texttt{add *t}\ \{\!|\}; \texttt{rename}\ m\ \texttt{*h}) & \in & \{\!|m \mapsto D|\!\} \overset{\Omega}{\Longleftrightarrow} (D :: \texttt{[]}) \end{array}$$

by the type of `add`, `rename` and composition.

The second branch must have type:

$$\begin{array}{lll} & \texttt{xfork}\ \{m\ \texttt{*t}\}\ \{\texttt{*t}\} & \\ & \quad (\texttt{hoist\_nonunique *t}\ \{\texttt{*h *t}\};\ l;\ \texttt{plunge *t}) & \\ & \quad (\texttt{id}) & \in\ C \setminus C_1 \overset{\Omega}{\Longleftrightarrow} A \setminus A_1 \\ \text{i.e.,} & \texttt{xfork}\ \{m\ \texttt{*t}\}\ \{\texttt{*t}\} & \\ & \quad (\texttt{hoist\_nonunique *t}\ \{\texttt{*h *t}\};\ l;\ \texttt{plunge *t}) & \\ & \quad (\texttt{id}) & \in\ \emptyset \overset{\Omega}{\Longleftrightarrow} \emptyset \end{array} ,$$

which it does, vacuously.

Otherwise, $i > 0$ and we must show that $\texttt{snoc}\ m \in C \overset{\Omega}{\Longleftrightarrow} A$ where $C = \{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{0..i}\texttt{]}$ and $A = \texttt{[}D^{1..i+1}\texttt{]}$. As above, we must show that the `acond` instance has this type by showing that each of its branches has the correct type. The proof that the first branch has the correct type is identical to the case above. We calculate that the second branch must have concrete and abstract type components:

$$\begin{array}{rcl} (\{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{0..i}\texttt{]}) \setminus \{\!|m \mapsto D|\!\} & = & \{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{1..i}\texttt{]} \\ \texttt{[}D^{1..i+1}\texttt{]} \setminus (D :: \texttt{[]}) & = & \texttt{[}D^{2..i+1}\texttt{]} \end{array}$$

To show that the second branch has this type, we must prove that each arm of the `xfork` lens has the correct type:

$$\begin{array}{lcll} k_1 & = & (\texttt{hoist\_nonunique *t}\ \{\texttt{*h *t}\}; & \\ & & l; & \\ & & \texttt{plunge *t}) & \in\ \{\!|m \mapsto D, \texttt{*t} \mapsto \texttt{[}D^{0..i-1}\texttt{]}|\!\} \overset{\Omega}{\Longleftrightarrow} \{\!|\texttt{*t} \mapsto \texttt{[}D^{1..i}\texttt{]}|\!\} \\ \text{i.e.,} \quad k_1 & = & & \in \{\!|m \mapsto D, \texttt{*t} \mapsto \texttt{[}D^{0..i-1}\texttt{]}|\!\} \\ & & \texttt{hoist\_nonunique *t}\ \{\texttt{*h *t}\}; & : \{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{0..i-1}\texttt{]} \\ & & \texttt{snoc}\ m & : \texttt{[}D^{1..i}\texttt{]} \\ & & \texttt{plunge *t} & \overset{\Omega}{\Longleftrightarrow} \{\!|\texttt{*t} \mapsto \texttt{[}D^{1..i}\texttt{]}|\!\} \end{array}$$

which follows from the types of `hoist_nonunique`, `snoc` m (using the induction hypothesis to show it has type $\{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{0..i-1}\texttt{]} \overset{\Omega}{\Longleftrightarrow} \texttt{[}D^{1..i}\texttt{]}$), `plunge`, and the composition operator;

$$k_2 = \texttt{id} \in \{\!|\texttt{*h} \mapsto D|\!\} \overset{\Omega}{\Longleftrightarrow} \{\!|\texttt{*h} \mapsto D|\!\}$$

immediately, by the type of `id`,

and observe that

$$\begin{array}{rcl}
\{\!|m \mapsto D, \texttt{*t} \mapsto \texttt{[}D^{0..i}\texttt{]}|\!\} & \subseteq & \mathcal{T}|_{\{m,\,\texttt{*t}\}} \\
\{\!|\texttt{*t} \mapsto \texttt{[}D^{1..i}\texttt{]}|\!\} & \subseteq & \mathcal{T}|_{\texttt{*t}} \\
\\
\{\!|\texttt{*h} \mapsto D|\!\} & \subseteq & \mathcal{T}\backslash_{\{m,\,\texttt{*t}\}} \\
\{\!|\texttt{*h} \mapsto D|\!\} & \subseteq & \mathcal{T}\backslash_{\texttt{*t}} \\
\\
\{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{1..i}\texttt{]} & = & (\{\!|m \mapsto D, \texttt{*t} \mapsto \texttt{[}D^{0..i-1}\texttt{]}|\!\}) \cdot (\{\!|\texttt{*h} \mapsto D|\!\}) \\
\texttt{[}D^{2..i+1}\texttt{]} & = & (\{\!|\texttt{*t} \mapsto \texttt{[}D^{1..i}\texttt{]}|\!\}) \cdot (\{\!|\texttt{*h} \mapsto D|\!\}).
\end{array}$$

We use all of these facts to prove that `xfork` has the following type:

$$\texttt{xfork}\ \{m\ \texttt{*t}\}\ \{\texttt{*t}\}\ k_1\ k_2 \in \{\!|m \mapsto D|\!\} \cdot \texttt{[}D^{1..i}\texttt{]} \overset{\Omega}{\Longleftrightarrow} \texttt{[}D^{2..i+1}\texttt{]}$$

Then by the type of `acond`, we have $\texttt{snoc}\ m \in C \overset{\Omega}{\Longleftrightarrow} A$, which finishes the case and the inductive proof. □

Using `snoc`, we can write `list_reverse` as follows:

```
list_reverse  =  acond [] []
                   (id)
                   (rename *h x;
                    hoist_nonunique *t {*h *t};
                    fork {*h *t} (list_reverse) id;
                    snoc x)
```

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

The *get* direction has two cases, corresponding to the two arms of the conditional. The first arm maps the empty list to the empty list via `id`. The second lens, selected when the concrete tree is not empty, is the composition of the following sequence: (1) a lens that renames the head of the list to `x`, (2) one that hoists the tail up one level yielding a list, (3) a recursive call, and (4) `snoc x`, which moves the child under x to the end of the (now reversed) tail.

The *put* direction also has two cases. Again, the first arm of the conditional maps the empty list to the empty list. The other composite lens runs the sequence described above in reverse, to obtain a concrete tree equivalent to the reversed abstract tree as follows. First, the *put* of `snoc x` takes the (non-empty) abstract list and produces a tree where the last element of the list is removed and placed under `x`. Next, this abstract view, consisting of a child `x` and a list is *put* back through the `fork` lens, which reverses the list part of the tree and leaves the child named `x` unchanged. Third, the *put* of `hoist_nonunique *t {*h, *t}` plunges the head and tail under `*t`. Finally, the child named `x` is renamed to `*h`, yielding a well-formed list.

The algorithm for computing the reversal of a list used 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 lens combinators plus recursion). One difficulty arises if we use an accumulator to store the result: the *put* 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.

7.9 Lemma [Well-behavedness]: $\forall D \subseteq \mathcal{T}.\ \texttt{list\_reverse} \in \texttt{[}D\texttt{]} \overset{\Omega}{=\!=} \texttt{[}D\texttt{]}$.

Proof: First, note that `list_reverse` is the fixed point of the function:

$f = \lambda l.$ `acond [] []`
  (`id`)
    (`rename *h x;`
     `hoist_nonunique *t {*h *t};`
     `fork {*h *t}` l `id;`
     `snoc x`)

In the rest of the proof, we use the following type abbreviations:

$$\begin{array}{lllll} C & = & A & = & [D] \\ C_1 & = & A_1 & = & [] \end{array}$$

In outline, the proof proceeds as follows. We assume that $l \in C \stackrel{\Omega}{=\!\!\Rightarrow} A$ and prove that $f(l) \in C \stackrel{\Omega}{=\!\!\Rightarrow} A$. Using Corollary 3.3.8 (1), we conclude that $\mathit{fix}(f) = \texttt{list\_reverse} \in C \stackrel{\Omega}{=\!\!\Rightarrow} A$.

We calculate the type of $f(l)$, working top down. The outermost lens is an `acond` instance; we must prove that the first branch has the correct type:

$$\begin{array}{lllll} & \texttt{id} & \in & C \cap C_1 & \stackrel{\Omega}{=\!\!\Rightarrow} \quad A \cap A_1 \\ \text{i.e.,} & \texttt{id} & \in & [] & \stackrel{\Omega}{=\!\!\Rightarrow} \quad [] \end{array}$$

which follows from the type of `id`;

and that the second branch has the correct type:

$$\begin{array}{lll} & \texttt{rename *h x;} & \\ & \texttt{hoist\_nonunique *t \{*h *t\};} & \\ & \texttt{fork \{*h *t\}} \; l \; \texttt{id;} & \\ & \texttt{snoc x} & \in \quad C \setminus C_1 \quad \stackrel{\Omega}{=\!\!\Rightarrow} \quad A \setminus A_1 \\ \text{i.e.,} & \texttt{rename *h x;} & \\ & \texttt{hoist\_nonunique *t \{*h *t\};} & \\ & \texttt{fork \{*h *t\}} \; l \; \texttt{id;} & \\ & \texttt{snoc x} & \in \quad [D^{1..\omega}] \quad \stackrel{\Omega}{=\!\!\Rightarrow} \quad [D^{1..\omega}] \end{array}$$

To prove this type for the second branch we show:

$$\begin{array}{lllll} k_1 & = & & \in & [D^{1..\omega}] \\ & & \texttt{rename *h x;} & : & \left\{\!\!\left\{ \texttt{x} \mapsto D, \texttt{*t} \mapsto [D] \right\}\!\!\right\} \\ & & \texttt{hoist\_nonunique *t \{*h, *t\};} & : & \left\{\!\!\left\{ \texttt{x} \mapsto D \right\}\!\!\right\} \cdot [D] \\ & & \texttt{fork \{*h *t\}} \; l \; \texttt{id;} & : & \left\{\!\!\left\{ \texttt{x} \mapsto D \right\}\!\!\right\} \cdot [D] \\ & & \texttt{snoc x} & \stackrel{\Omega}{=\!\!\Rightarrow} [D^{1..\omega}]. & \end{array}$$

(Note that the second to last step follows from the hypothesis about the type of l.) We conclude that $f(l) \in C \stackrel{\Omega}{=\!\!\Rightarrow} A$ and by Corollary 3.3.8 (1), that `list_reverse` has the same type, $[D] \stackrel{\Omega}{=\!\!\Rightarrow} [D]$. □

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

Proof: The proof, in outline, is as follows. We first note that `list_reverse` is the fixed point of the function f, defined in the well-behavedness proof above. We then prove for all i, that $f(l) \in C_{i+1} \stackrel{\Omega}{\Longleftrightarrow} A_{i+1}$ assuming that $l \in C_i \stackrel{\Omega}{\Longleftrightarrow} A_i$. By Corollary 3.3.8 (2), we conclude that $\mathit{fix}(f) \in \bigcup_i C_i \stackrel{\Omega}{\Longleftrightarrow} \bigcup_i A_i$.

Define two chains of types:

$C_0 = A_0 = \emptyset$
$C_{i+1} = A_{i+1} = [D^{0..i}]$.

also used as abbreviations for the types in the `acond` lens:

$$C_1 \quad = \quad A_1 \quad = \quad [].$$

We will show that $l \in C_i \overset{\Omega}{\Longleftrightarrow} A_i$ implies $f(l) \in C_{i+1} \overset{\Omega}{\Longleftrightarrow} A_{i+1}$ by induction on i. Let $C = C_{i+1} = \texttt{[}D^{0..i}\texttt{]}$ and $A = A_{i+1} = \texttt{[}D^{0..i}\texttt{]}$. To show that $f(l) \in C \overset{\Omega}{\Longleftrightarrow} A$, we must show that the outermost `acond` lens has that type.

By the type of `acond`, we must prove that both branches have the correct type. The type of the first branch is easy to calculate and verify:

$$\begin{array}{rrcccc} & \texttt{id} & \in & C \cap C_1 & \overset{\Omega}{\Longleftrightarrow} & A \cap A_1 \\ \text{i.e.,} & \texttt{id} & \in & \texttt{[]} & \overset{\Omega}{\Longleftrightarrow} & \texttt{[]} \end{array}$$

by the type of `id`.

Showing that the second branch has the correct type, calculated as:

$$\begin{array}{l} \texttt{rename *h x;} \\ \texttt{hoist\_nonunique *t \{*h, *t\};} \\ \texttt{fork \{*h, *t\}}\ l\ \texttt{id;} \\ \texttt{snoc x} \qquad\qquad \in \quad C \setminus C_1 \quad \overset{\Omega}{\Longleftrightarrow} \quad A \setminus A_1, \end{array}$$

requires a little more work. There are two cases. If $i = 0$ then

$$\begin{array}{rcccc} C \setminus C_1 & = & \texttt{[]} \setminus \texttt{[]} & = & \emptyset \\ A \setminus A_1 & = & \texttt{[]} \setminus \texttt{[]} & = & \emptyset \end{array}$$

and the second branch has type $\emptyset \overset{\Omega}{\Longleftrightarrow} \emptyset$ vacuously. Otherwise $i > 0$ and

$$\begin{array}{rcccc} C \setminus C_1 & = & \texttt{[}D^{0..i}\texttt{]} \setminus \texttt{[]} & = & \texttt{[}D^{1..i}\texttt{]} \\ A \setminus A_1 & = & \texttt{[}D^{0..i}\texttt{]} \setminus \texttt{[]} & = & \texttt{[}D^{1..i}\texttt{]} \end{array}$$

Thus, we must show that the second branch has this type.

$$\begin{array}{rrcl} & \texttt{rename *h x;} & & \\ & \texttt{hoist\_nonunique *t \{*h, *t\};} & & \\ & \texttt{fork \{*h, *t\}}l\texttt{id;} & & \\ & \texttt{snoc x} & \in & \texttt{[}D^{1..i}\texttt{]} \quad \overset{\Omega}{\Longleftrightarrow} \quad \texttt{[}D^{1..i}\texttt{]} \\ \text{i.e.,}\quad k \quad = & & \in & \texttt{[}D^{1..i}\texttt{]} \\ & \texttt{rename *h x;} & : & \left\{\texttt{x} \mapsto D,\ \texttt{*t} \mapsto \texttt{[}D^{0..i-1}\texttt{]}\right\} \\ & \texttt{hoist\_nonunique *t \{*h, *t\};} & : & \left\{\texttt{x} \mapsto D\right\} \cdot \texttt{[}D^{0..i-1}\texttt{]} \\ & \texttt{fork \{*h, *t\}}\ l\ \texttt{id;} & : & \left\{\texttt{x} \mapsto D\right\} \cdot \texttt{[}D^{0..i-1}\texttt{]} \\ & \texttt{snoc x} & \overset{\Omega}{\Longleftrightarrow} & \texttt{[}D^{1..i}\texttt{]} \end{array}$$

(The last step follows from Lemma 7.8.) This finishes the case and the inductive proof. Using Corollary 3.3.8 (2), we conclude that

$$\begin{array}{rrcccc} \mathit{fix}(f) & = & \texttt{list\_reverse} & \in & \bigcup_i C_i & \overset{\Omega}{\Longleftrightarrow} & \bigcup_i A_i \\ & \text{i.e.,} & \texttt{list\_reverse} & \in & \emptyset \cup \bigcup_i \texttt{[}D^{0..i}\texttt{]} & \overset{\Omega}{\Longleftrightarrow} & \emptyset \cup \bigcup_i \texttt{[}D^{0..i}\texttt{]} \\ & \text{i.e.,} & \texttt{list\_reverse} & \in & \texttt{[}D\texttt{]} & \overset{\Omega}{\Longleftrightarrow} & \texttt{[}D\texttt{]} \end{array}$$

as required. □

Filter

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

In the *get* direction, the desired behavior of `list_filter` D E (for brevity, let us call it l) is clear. In the *put* direction, things are more interesting. To begin with, the lens laws impose some key constraints on the behavior of $l\!\searrow$. The GetPut law forces the *put* function to restore each of the filtered elements when the abstract list is put into the original concrete list. For example (letting d and e be elements of D and E) we must have $l\searrow(\texttt{[}d\texttt{]}, \texttt{[}e\ d\texttt{]}) = \texttt{[}e\ d\texttt{]}$. The PutGet law forces the *put* function to include every element of the abstract list in the resulting concrete list and to only take Es (not Ds) from the concrete list.

In the general case, where the abstract list a is different from the filtered concrete list $l\nearrow c$, there is some freedom in how $l\!\searrow$ 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. If a runs out first, then $l\!\searrow$ keeps any remaining Es that may be left at the end of c (discarding any remaining Ds in c, as it must to satisfy PutGet). For example, $l\searrow(\texttt{[]}, \texttt{[}d\ e\texttt{]})$ yields `[`e`]`, not `[]`, and $l\searrow(\texttt{[}d\texttt{]}, \texttt{[}e\texttt{]})$ is `[`$e\ d$`]`, not `[`$d\ e$`]`.

These choices lead us to the following specification for a single step, in the *put* direction, of a recursively defined lens implementing l. If the abstract list a and concrete list c are both cons cells whose heads are in D, then it yields the head of a and recurses on both tails. If c begins with an E (i.e., c has type $E :: \texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]}$), then it restores the head of c and recurses on a and the tail of c. If a is empty and c begins with a D (c has type $D :: \texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]}$), then it restores all the remaining Es from c and returns. Translating this into the lens combinators defined above leads (modulo a little new notation and a few additional technicalities, explained below) to the definition below of `list_filter` and a helper lens, `inner_filter`, by mutual recursion. The singly recursive variant with `inner_filter` inlined has the same behavior as the version presented here. We split out `inner_filter` so that we can give it a more precise type, facilitating reasoning about well-behavedness and totality: in the *get* direction it maps lists containing at least one D to $D :: \texttt{[}D\texttt{]}$; the corresponding types for `list_filter` include empty lists.

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

inner_filter D E =
    ccond (E :: ([D^{1..ω}]&[E]))
        (tl any_E; inner_filter D E)
        (wmap {*t ↦ list_filter D E})
```

$\forall D, E \subseteq \mathcal{T}$. *with* $D \cap E = \emptyset$ *and* $D \neq \emptyset$ *and* $E \neq \emptyset$.
`list_filter` $D\ E \in \texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]} \overset{\Omega}{\Longleftrightarrow} \texttt{[}D\texttt{]}$ *and*
`inner_filter` $D\ E \in \texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]} \overset{\Omega}{\Longleftrightarrow} \texttt{[}D^{1..\omega}\texttt{]}$

The "choice operator" any_D denotes an arbitrary element of a non-empty set D.[6] The function $fltr_E$ is used by the `cond` to strip out any Ds from the tail of c remaining when the a argument becomes empty; this is the usual list-filtering *function*, which for present purposes we simply assume has been defined as a primitive. (In our implementation, we actually use `list_filter`$\nearrow$ here; but for expository purposes we prefer to avoid this extra bit of recursiveness.) Finally, the function $\lambda c.\ c@\texttt{[}any_D\texttt{]}$ appends some arbitrary element of D to the right-hand end of a list c. It is used by `cond` for the case where a non-empty a is being *put* into a list c that does not contain any Ds; by adding a dummy d at the end of c, it produces a concrete list that can

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

validly be passed to `inner_filter`, which expects at least one D in its concrete argument marking the point where the head of a should be placed.

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

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

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

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

We now argue that `list_filter` is well behaved and total. As before, the well-behavedness proof is straightforward: we simply decide on types for recursive uses of both `list_filter` and `inner_filter` and then show that, under this assumption, the bodies of both lenses have these same types.

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

Proof: We use corollary 3.3.8 (1), assuming

$$
\begin{aligned}
\texttt{list\_filter}\ D\ E \quad &\in \quad \texttt{[}D\texttt{]\&[}E\texttt{]} \overset{\Omega}{\rightleftharpoons} \texttt{[}D\texttt{]} \\
\texttt{inner\_filter}\ D\ E \quad &\in \quad \texttt{[}D^{1..\omega}\texttt{]\&[}E\texttt{]} \overset{\Omega}{\rightleftharpoons} \texttt{[}D^{1..\omega}\texttt{]}
\end{aligned}
$$

and deriving the expected types for `list_filter` and `inner_filter` from their recursive definitions.

We first derive the type for `list_filter` D E. The outermost combinator is a `cond` lens with concrete predicate $C_1 = \texttt{[}E\texttt{]}$ and abstract predicates $A_1 = \texttt{[]}$ and $A_2 = \texttt{[}D^{1..\omega}\texttt{]}$. We must show that

$$\begin{array}{rrcl} & \texttt{const [] []} & \in & C \cap C_1 \stackrel{\Omega}{=\!=} A_1 \\ & \texttt{const [] []} & \in & (\texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]}) \cap \texttt{[}E\texttt{]} \stackrel{\Omega}{=\!=} \texttt{[]} \\ \text{i.e.,} & \texttt{const [] []} & \in & \texttt{[}E\texttt{]} \stackrel{\Omega}{=\!=} \texttt{[]} \end{array}$$

and

$$\begin{array}{rrcl} & \texttt{inner\_filter}\ D\ E & \in & C \setminus C_1 \stackrel{\Omega}{=\!=} A_2 \\ & \texttt{inner\_filter}\ D\ E & \in & (\texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]}) \setminus \texttt{[}E\texttt{]} \stackrel{\Omega}{=\!=} \texttt{[}D^{1..\omega}\texttt{]} \\ \text{i.e.,} & \texttt{inner\_filter}\ D\ E & \in & \texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]} \stackrel{\Omega}{=\!=} \texttt{[}D^{1..\omega}\texttt{]}. \end{array}$$

The first fact follows from the type of `const`; the second is immediate by hypothesis. Next we prove that the functions fltr_E and $(\lambda c.\ c@\texttt{[}\mathit{any}_D\texttt{]})$ have the correct types:

$$\begin{array}{rcl} \mathit{fltr}_E & \in & (\texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]}) \to (\texttt{[}E\texttt{]})_\Omega \\ \lambda c.\ c@\texttt{[}\mathit{any}_D\texttt{]} & \in & (\texttt{[}E\texttt{]}) \to (\texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]})_\Omega \end{array}$$

Both are immediate. Hence, using the type of `cond`, we conclude that `list_filter` $D\ E \in (\texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]}) \stackrel{\Omega}{=\!=} (\texttt{[]} \cup \texttt{[}D^{1..\omega}\texttt{]})$—i.e., `list_filter` $D\ E \in (\texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]}) \stackrel{\Omega}{=\!=} \texttt{[}D\texttt{]}$—as required.

Next we derive the type for `inner_filter` D E, working top down. The outermost lens is a `ccond` combinator. We must show that each branch has the correct type.

$$\begin{array}{rrcl} & (\texttt{tl}\ \mathit{any}_E;\ \texttt{inner\_filter}\ D\ E) & \in & (\texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]}) \cap (E :: (\texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]})) \stackrel{\Omega}{=\!=} \texttt{[}D^{1..\omega}\texttt{]} \\ \text{i.e.,} & (\texttt{tl}\ \mathit{any}_E;\ \texttt{inner\_filter}\ D\ E) & \in & (E :: (\texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]})) \stackrel{\Omega}{=\!=} \texttt{[}D^{1..\omega}\texttt{]} \end{array}$$

$$\begin{array}{rrcl} & \texttt{wmap}\ \{\texttt{*t} \mapsto \texttt{list\_filter}\ D\ E\} & \in & (\texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]}) \setminus (E :: (\texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]})) \stackrel{\Omega}{=\!=} \texttt{[}D^{1..\omega}\texttt{]} \\ \text{i.e.,} & \texttt{wmap}\ \{\texttt{*t} \mapsto \texttt{list\_filter}\ D\ E\} & \in & D :: (\texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]}) \stackrel{\Omega}{=\!=} \texttt{[}D^{1..\omega}\texttt{]} \end{array}$$

The first fact follows from the type of `tl` with $\mathit{any}_E \in E$, the composition operator, and the hypothesis about the type of `inner_filter`. The second follows from the type of `wmap`, the observation that $\texttt{dom}(D :: (\texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]})) = \texttt{dom}(\texttt{[}D^{1..\omega}\texttt{]})$, our hypothesis about the type of `list_filter`, and Lemma 7.2, which states that cons cell types are shuffle closed. Using the type of `ccond`, we conclude that `inner_filter` $\in \texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]} \stackrel{\Omega}{=\!=} \texttt{[}D^{1..\omega}\texttt{]}$ as required. □

The totality proof for `list_filter`, on the other hand, is somewhat challenging, involving detailed reasoning about the behavior of particular subterms under particular conditions. This is not too surprising, given the well-known difficulties of reasoning about totality of ordinary recursive functional programs. We do not imagine that, in practice, detailed proofs of totality will be undertaken for very many lenses—most lens programmers will probably be satisfied with the assurance of (easier) proofs of well-behavedness plus informal reasoning about totality, just as most working functional programmers are reasonably happy with typechecking plus informal totality arguments for their functions. Still, it is interesting to work through a few non-trivial totality proofs in detail, to see what sorts of reasoning techniques are required.

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

Proof: To start, note that the pair (`inner_filter` $D\ E$, `list_filter` $D\ E$) is the fixed point of the following function from pairs of lenses to pairs of lenses:

$$\begin{array}{rcl} f & = & \lambda(l, l').\ (\texttt{ccond}\ E :: (\texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]}) \\ & & \quad (\texttt{tl}\ \mathit{any}_E;\ l) \\ & & \quad (\texttt{wmap}\ \{\texttt{*t} \mapsto l'\}), \\ & & \texttt{cond [}E\texttt{] [] [}D^{1..\omega}\texttt{]}\ \mathit{fltr}_E\ (\lambda c.\ c@\texttt{[}\mathit{any}_D\texttt{]}) \\ & & \quad (\texttt{const [] []}) \\ & & \quad l) \end{array}$$

Note that the order of `inner_filter` and `list_filter` is swapped here with respect to the original definition. We need to take them in this order because the totality of `list_filter` at each stage of the induction is going to depend on the totality of `inner_filter` at the *same* stage (plus the totality of `list_filter` at the previous stage), while the totality of `inner_filter` will depend only on the totality of `inner_filter` and `list_filter` at the previous stage.

In outline, the proof goes as follows. We start by choosing a sequence of pairs of total type sets $(\mathbb{T}_0, \mathbb{T}'_0), (\mathbb{T}_1, \mathbb{T}'_1), \ldots$. (Note that each $\mathbb{T}_i$ and $\mathbb{T}'_i$ here is a set of total types and a total type is itself a pair (C, A).) Next, we prove a key property of f: that, when we apply it to a pair of lenses possessing all the types in some $(\mathbb{T}_i, \mathbb{T}'_i)$, the result is a pair of lenses possessing all the types in $(\mathbb{T}_{i+1}, \mathbb{T}'_{i+1})$. To match the form of Corollary 3.3.14, we do this in two steps: first, we assume that l has every total type in $\mathbb{T}_i$ and l' has every total type in $\mathbb{T}'_i$ and prove that $\pi_1(f(l, l'))$ has every total type in $\mathbb{T}_{i+1}$; second, we assume that l has every total type in $\mathbb{T}_{i+1}$ and l' has every total type in $\mathbb{T}'_i$ and prove that $\pi_2(f(l, l'))$ has every total type in $\mathbb{T}'_{i+1}$. Next we choose an increasing instance of the sequence—i.e., a chain $(\tau_0, \tau'_0) \subseteq (\tau_1, \tau'_1) \subseteq (\tau_2, \tau'_2) \subseteq \ldots$ where each $\tau_i \in \mathbb{T}_i$ and $\tau'_i \in \mathbb{T}'_i$. We argue that the limit of this increasing instance, $(\bigcup_i \tau_i, \bigcup_i \tau'_i)$, is the pair of total types we want—i.e.,

$$((\texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]}, \texttt{[}D^{1..\omega}\texttt{]}),\ (\texttt{[}D\texttt{]}\&\texttt{[}E\texttt{]}, \texttt{[}D\texttt{]})).$$

We conclude by 3.3.14 that the fixed point of f—i.e., the pair (`inner_filter` D E, `list_filter` D E)—has this type, finishing the proof. We now proceed to the details.

We first define the sequence of pairs of total type sets:

$$\begin{array}{rcl}
\mathbb{T}_0 & = & \{(\emptyset, \emptyset)\} \\
\mathbb{T}'_0 & = & \{(\emptyset, \emptyset)\} \\
\mathbb{T}_{i+1} & = & \{(\texttt{[}D^{1..x}\texttt{]}\&\texttt{[}E^{0..y}\texttt{]}, \texttt{[}D^{1..x}\texttt{]}) \mid x + y = i\} \\
\mathbb{T}'_{i+1} & = & \{(\texttt{[}D^{0..x}\texttt{]}\&\texttt{[}E^{0..y}\texttt{]}, \texttt{[}D^{0..x}\texttt{]}) \mid x + y = i\}
\end{array}$$

To make the construction of this sequence clear, let us calculate its first few elements explicitly:

$$\begin{array}{rcl}
\mathbb{T}_1 & = & \{(\emptyset, \emptyset)\} \\
\mathbb{T}'_1 & = & \{(\texttt{[]}, \texttt{[]})\} \\
\mathbb{T}_2 & = & \{(\texttt{[}D^{1..1}\texttt{]}, \texttt{[}D^{1..1}\texttt{]})\} \\
\mathbb{T}'_2 & = & \{(\texttt{[}D^{0..1}\texttt{]}, \texttt{[}D^{0..1}\texttt{]}), (\texttt{[}E^{0..1}\texttt{]}, \texttt{[]})\} \\
\mathbb{T}_3 & = & \{(\texttt{[}D^{1..2}\texttt{]}, \texttt{[}D^{1..2}\texttt{]}), (\texttt{[}D^{1..1}\texttt{]}\&\texttt{[}E^{0..1}\texttt{]}, \texttt{[}D^{1..1}\texttt{]})\} \\
\mathbb{T}'_3 & = & \{(\texttt{[}D^{0..2}\texttt{]}, \texttt{[}D^{0..2}\texttt{]}), (\texttt{[}D^{0..1}\texttt{]}\&\texttt{[}E^{0..1}\texttt{]}, \texttt{[}D^{0..1}\texttt{]}), (\texttt{[}E^{0..2}\texttt{]}, \texttt{[]})\}
\end{array}$$

In the proof, we use some abbreviations to lighten the presentation. We abbreviate the type argument to the `ccond` lens appearing in the first component of the body of F as follows:

$$C_1 = E :: (\texttt{[}D^{1..\omega}\texttt{]}\&\texttt{[}E\texttt{]})$$

Similarly, we abbreviate the type arguments to the `cond` in the second component:

$$\begin{array}{rcl}
C'_1 & = & \texttt{[}E\texttt{]} \\
A'_1 & = & \texttt{[]} \\
A'_2 & = & \texttt{[}D^{1..\omega}\texttt{]}
\end{array}$$

In each case of the inductive proof below, we will define C and A to be the source and target of the type we are trying to establish for the given lens.

We now prove, by induction on i, the facts about f needed to apply Corollary 3.3.14: first, that if l has every total type in $\mathbb{T}_i$ and l' has every total type in $\mathbb{T}'_i$, then $\pi_1(f(l, l'))$ has every total type in $\mathbb{T}_{i+1}$; and second, that if l has every total type in $\mathbb{T}_{i+1}$ and l' has every total type in $\mathbb{T}'_i$, then $\pi_2(f(l, l'))$ has every total type in $\mathbb{T}'_{i+1}$.

For the base case ($i = 0$), we must first show that $\pi_1(f(l, l'))$ has every total type in the singleton set $\mathbb{T}_1 = \{(\emptyset, \emptyset)\}$. This is immediate, since every lens is total at this type. Second, we must show that

$\pi_2(f(l,l'))$ has every total type in the singleton set $\mathbb{T}'_1 = \{(\texttt{[]}, \texttt{[]})\}$. We let $C = \texttt{[]}$ and $A = \texttt{[]}$ and show that $\pi_2(f(l,l')) \in C \overset{\Omega}{\Longleftrightarrow} A$. Recall that the second component of $f(l,l')$ is defined as

$$\begin{array}{l}\texttt{cond}\ \texttt{[}E\texttt{]}\ \texttt{[]}\ \texttt{[}D^{1..\omega}\texttt{]}\ \mathit{fltr}_E\ (\lambda c.\ c@\texttt{[}\mathit{any}_D\texttt{]}) \\ \quad (\texttt{const}\ \texttt{[]}\ \texttt{[]}) \\ \quad l.\end{array}$$

Observe that, as $(C \cap C'_1) = (\texttt{[]} \cap \texttt{[}E\texttt{]}) = \texttt{[]}$ is not empty but $(C \setminus C'_1) = (\texttt{[]} \setminus \texttt{[}E\texttt{]}) = \emptyset$, by Theorem 6.6 we can use the *always-true* rule for `cond`. Thus, to show that the whole conditional has type $C \overset{\Omega}{\Longleftrightarrow} A'_1$ (which is what we want, since $A'_1 = A = \texttt{[]}$) it suffices to show that the first branch, `const [] []` has type $C \cap C'_1 \overset{\Omega}{\Longleftrightarrow} A'_1$,

$$\begin{array}{rlrrcl} & \texttt{const}\ \texttt{[]}\ \texttt{[]} & \in & C \cap C'_1 & \overset{\Omega}{\Longleftrightarrow} & A'_1 \\ \text{i.e.,} & \texttt{const}\ \texttt{[]}\ \texttt{[]} & \in & \texttt{[]} \cap \texttt{[}E\texttt{]} & \overset{\Omega}{\Longleftrightarrow} & \texttt{[]} \\ \text{i.e.,} & \texttt{const}\ \texttt{[]}\ \texttt{[]} & \in & \texttt{[]} & \overset{\Omega}{\Longleftrightarrow} & \texttt{[]}, \end{array}$$

which follows from the type of `const`.

For the induction step ($i > 0$), we first prove that $\pi_1(f(l,l'))$ has every total type in $\mathbb{T}_{i+1}$, assuming that l has every total type in $\mathbb{T}_i$ and l' has every total type in $\mathbb{T}'_i$. Pick an arbitrary total type τ from $\mathbb{T}_{i+1}$. We break the argument into three sub-cases.

Case $x > 0$ and $y > 0$: Here τ has the form (C, A) with $C = (\texttt{[}D^{1..x}\texttt{]} \& \texttt{[}E^{0..y}\texttt{]})$ and $A = \texttt{[}D^{1..x}\texttt{]}$. Recall that the first component of $f(l,l')$ is

$$\begin{array}{l}\texttt{ccond}\ (E :: (\texttt{[}D^{1..\omega}\texttt{]} \& \texttt{[}E\texttt{]})) \\ \quad (\texttt{tl}\ \mathit{any}_E;\ l) \\ \quad (\texttt{wmap}\ \{\texttt{*t} \mapsto l'\}).\end{array}$$

The typing rule for `ccond` requires that we prove that the branches have the following types:

$$\begin{array}{rlrrcl} & (\texttt{tl}\ \mathit{any}_E;\ l) & \in & C \cap C_1 & \overset{\Omega}{\Longleftrightarrow} & A \\ \text{i.e.,} & (\texttt{tl}\ \mathit{any}_E;\ l) & \in & (\texttt{[}D^{1..x}\texttt{]} \& \texttt{[}E^{0..y}\texttt{]}) \cap (E :: (\texttt{[}D^{1..\omega}\texttt{]} \& \texttt{[}E\texttt{]})) & \overset{\Omega}{\Longleftrightarrow} & \texttt{[}D^{1..x}\texttt{]} \\ \text{i.e., as } y > 0, & (\texttt{tl}\ \mathit{any}_E;\ l) & \in & E :: (\texttt{[}D^{1..x}\texttt{]} \& \texttt{[}E^{0..y-1}\texttt{]}) & \overset{\Omega}{\Longleftrightarrow} & \texttt{[}D^{1..x}\texttt{]} \end{array}$$

which follows from type of `tl` and the induction hypothesis;

$$\begin{array}{rlrrcl} & \texttt{wmap}\ \{\texttt{*t} \mapsto l'\} & \in & C \setminus C_1 & \overset{\Omega}{\Longleftrightarrow} & A \\ \text{i.e.,} & \texttt{wmap}\ \{\texttt{*t} \mapsto l'\} & \in & (\texttt{[}D^{1..x}\texttt{]} \& \texttt{[}E^{0..y}\texttt{]}) \setminus (E :: (\texttt{[}D^{1..\omega}\texttt{]} \& \texttt{[}E\texttt{]})) & \overset{\Omega}{\Longleftrightarrow} & \texttt{[}D^{1..x}\texttt{]} \\ \text{i.e., as } x > 0, & \texttt{wmap}\ \{\texttt{*t} \mapsto l'\} & \in & D :: (\texttt{[}D^{0..x-1}\texttt{]} \& \texttt{[}E^{0..y}\texttt{]}) & \overset{\Omega}{\Longleftrightarrow} & \texttt{[}D^{1..x}\texttt{]} \end{array}$$

which follows from type of `wmap`—by Lemma 7.2, both $D :: (\texttt{[}D^{0..x-1}\texttt{]} \& \texttt{[}E^{0..y}\texttt{]})$ and $\texttt{[}D^{1..x}\texttt{]}$, i.e., $D :: \texttt{[}D^{0..x-1}\texttt{]}$, are shuffle closed; also $\mathsf{dom}(D :: (\texttt{[}D^{0..x-1}\texttt{]} \& \texttt{[}E^{0..y}\texttt{]})) = \mathsf{dom}(\texttt{[}D^{1..x}\texttt{]})$—and the induction hypothesis.

Using the type of `ccond`, we conclude that $\pi_1(f(l,l')) \in C \overset{\Omega}{\Longleftrightarrow} A$, finishing the case.

Case $x = 0$: Recall that the set $\mathbb{T}_{i+1}$ is $\{(\texttt{[}D^{1..x}\texttt{]} \& \texttt{[}E^{0..y}\texttt{]}, \texttt{[}D^{1..x}\texttt{]}) \mid x + y = i\}$. The only element τ in this set with $x = 0$ is the empty total type:

$$\begin{array}{rl} & (\texttt{[}D^{1..0}\texttt{]} \& \texttt{[}E^{0..y}\texttt{]}, \texttt{[}D^{1..0}\texttt{]}) \\ = & (\emptyset \& \texttt{[}E^{0..y}\texttt{]}, \emptyset) \\ = & (\emptyset, \emptyset). \end{array}$$

Immediately, the lens $\pi_1(f(l,l'))$ has type $\emptyset \overset{\Omega}{\Longleftrightarrow} \emptyset$, finishing the case.

Case $y = 0$ and $x > 0$: By construction, τ is (C, A) with $C = \texttt{[}D^{1..x}\texttt{]}$ and $A = \texttt{[}D^{1..x}\texttt{]}$. To verify the type of the `ccond`, we first observe that $C \cap C_1 = \texttt{[}D^{1..x}\texttt{]} \cap (E :: (\texttt{[}D^{1..\omega}\texttt{]} \& \texttt{[}E\texttt{]})) = \emptyset$, so the `ccond` always

selects the second branch in both the *get* and *put* directions. By Theorem 6.5, it suffices to show that the second branch has type $C \setminus C_1 \overset{\Omega}{\Longleftrightarrow} A$ (or just $C \overset{\Omega}{\Longleftrightarrow} A$ since $C \setminus C_1 = C$):

$$\begin{array}{rlrcl} & \texttt{wmap}\ \{\texttt{*t} \mapsto l'\} & \in & C & \overset{\Omega}{\Longleftrightarrow} \quad A \\ \text{i.e.,} & \texttt{wmap}\ \{\texttt{*t} \mapsto l'\} & \in & [D^{1..x}] & \overset{\Omega}{\Longleftrightarrow} \quad [D^{1..x}] \\ \text{i.e., as } x > 0, & \texttt{wmap}\ \{\texttt{*t} \mapsto l'\} & \in & D :: ([D^{0..x-1}]) & \overset{\Omega}{\Longleftrightarrow} \quad [D^{1..x}] \end{array}$$

which follows from type of `wmap` (with the observation that $D :: [D^{0..x-1}] = [D^{1..x}]$ and Lemma 7.2, which states that cons cell types are shuffle closed) and the induction hypothesis.

Using the *always-false* type of `ccond`, we conclude that $\pi_1(f(l, l')) \in C \overset{\Omega}{\Longleftrightarrow} A$, finishing the case.

We now turn to the second half of the induction step. We must prove that $\pi_2(f(l, l'))$ has every total type in $\mathbb{T}'_{i+1}$, assuming that l has every total type in $\mathbb{T}_{i+1}$ and l' has every total type in $\mathbb{T}'_i$. Pick an arbitrary type τ' from $\mathbb{T}'_{i+1}$. This time we break the argument into two sub-cases.

Case $x > 0$: Here τ' has the form (C, A) with $C = [D^{0..x}] \& [E^{0..y}]$ and $A = [D^{0..x}]$. The outer lens in $\pi_2(f(l, l'))$ is a `cond`. By Theorem 6.4, to show that this lens has the desired type, we must show that the branches have the following types:

$$\begin{array}{rlrcl} & \texttt{const [] []} & \in & C \cap C'_1 & \overset{\Omega}{\Longleftrightarrow} \quad A \cap A'_1 \\ \text{i.e.,} & \texttt{const [] []} & \in & ([D^{0..x}] \& [E^{0..y}]) \cap [E] & \overset{\Omega}{\Longleftrightarrow} \quad [D^{0..x}] \cap [] \\ \text{i.e.,} & \texttt{const [] []} & \in & [E^{0..y}] & \overset{\Omega}{\Longleftrightarrow} \quad [] \end{array}$$

which follows from the type of `const`, as the default tree, `[]`, is in $[E^{0..y}]$, for any y;

$$\begin{array}{rlrcl} & l & \in & C \setminus C'_1 & \overset{\Omega}{\Longleftrightarrow} \quad A \cap A'_2 \\ \text{i.e.,} & l & \in & ([D^{0..x}] \& [E^{0..y}]) \setminus [E] & \overset{\Omega}{\Longleftrightarrow} \quad [D^{0..x}] \cap [D^{1..\omega}] \\ \text{i.e.,} & l & \in & [D^{1..x}] \& [E^{0..y}] & \overset{\Omega}{\Longleftrightarrow} \quad [D^{1..x}] \end{array}$$

which follows by the induction hypothesis, as $([D^{1..x}] \& [E^{0..y}], [D^{1..x}]) \in \mathbb{T}_{i+1}$.

Next, we must verify that the conversion functions have the correct types:

$$\begin{array}{rlrcl} & \mathit{fltr}_E & \in & C \setminus C'_1 & \to \quad (C \cap C'_1)_\Omega \\ \text{i.e.,} & \mathit{fltr}_E & \in & [D^{1..x}] \& [E^{0..y}] & \to \quad ([E^{0..y}])_\Omega \end{array}$$

$$\begin{array}{rlrcl} & \lambda c.\ c@[\mathit{any}_D] & \in & C \cap C'_1 & \to \quad (C \setminus C'_1)_\Omega \\ \text{i.e.,} & \lambda c.\ c@[\mathit{any}_D] & \in & [E^{0..y}] & \to \quad ([D^{1..x}] \& [E^{0..y}])_\Omega \end{array}$$

Both of these facts are immediate. Finally, we calculate the target type:

$$\begin{array}{rl} & A \cap (A'_1 \cup A'_2) \\ = & [D^{0..x}] \cap ([] \cup [D^{1..\omega}]) \\ = & [D^{0..x}] \\ = & A. \end{array}$$

By Theorem 6.4, we conclude that $\pi_2(f(l, l')) \in C \overset{\Omega}{\Longleftrightarrow} A$, finishing the case.

Case $x = 0$: Here τ' must be (C, A) with $C = [E^{0..y}]$ and $A = []$. As $C \cap C'_1$ is not empty and $(C \setminus C'_1)$ is empty, by Theorem 6.6, we may use the *always-true* rule for `cond`. Using this rule, to show that the instance of `cond` has type $C \overset{\Omega}{\Longleftrightarrow} A'_1$ (which is what we want since $A'_1 = A = []$) it suffices to show that `const [] []` has type $C \cap C'_1 \overset{\Omega}{\Longleftrightarrow} A'_1$; i.e., that it has type $[E^{0..y}] \overset{\Omega}{\Longleftrightarrow} []$, as we verified above. We conclude that $\pi_2(f(l, l')) \in C \overset{\Omega}{\Longleftrightarrow} A$, which finishes the case.

To apply Corollary 3.3.11 and finish the totality proof, we must show that

$$(([D^{1..\omega}] \& [E], [D^{1..\omega}]), ([D] \& [E], [D]))$$

is the limit of an increasing instance of elements of $\overline{(\mathbb{T}, \mathbb{T}')}$. Let $(\tau_0, \tau_0') \subseteq (\tau_1, \tau_1') \subseteq \ldots$ be defined as

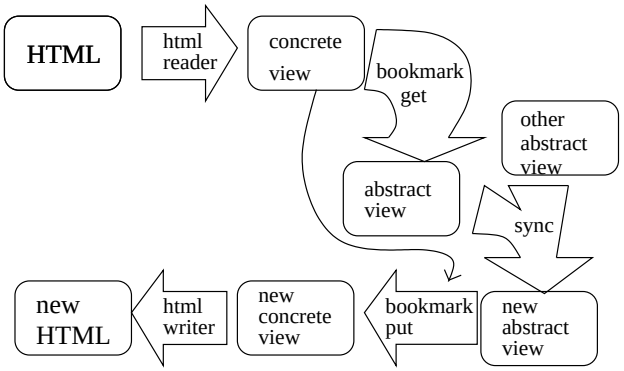
$$\begin{array}{rcll} \tau_0 & = & (\emptyset, \emptyset) & \in \mathbb{T}_0 \\ \tau_0' & = & (\emptyset, \emptyset) & \in \mathbb{T}_0' \\ \vdots & & & \\ \tau_{i+1} & = & (\texttt{[}D^{1..((i+1)/2)}\texttt{]\&[}E^{0..(i/2)}\texttt{]}, \texttt{[}D^{1..((i+1)/2)}\texttt{]}) & \in \mathbb{T}_{i+1} \\ \tau_{i+1}' & = & (\texttt{[}D^{0..((i+1)/2)}\texttt{]\&[}E^{0..(i/2)}\texttt{]}, \texttt{[}D^{0..((i+1)/2)}\texttt{]}) & \in \mathbb{T}_{i+1}', \end{array}$$

where i/n is integer division of i by n. To show that the limit is the pair of total types we want, we prove that each set is contained in the other. First, observe that, for any $c \in (\texttt{[}D^{1..\omega}\texttt{]\&[}E\texttt{]})$ and $a \in \texttt{[}D^{1..\omega}\texttt{]}$, we can find an i such that $(c, a) \in \tau_i$ (lifting $\in$ to pairs of sets in the obvious way) by choosing i so that $i/2$ is greater than the maximum number of elements of D in c, the number of elements of E in c, and the number of elements in a. Similarly, for every $c \in \texttt{[}D\texttt{]\&[}E\texttt{]}$ and $a \in \texttt{[}D\texttt{]}$, we can find a τ_i' such that $(c, a) \in \tau_i'$ by choosing a large enough i. The other inclusion is immediate: every τ_i is a subset of $(\texttt{[}D^{1..\omega}\texttt{]\&[}E\texttt{]}, \texttt{[}D^{1..\omega}\texttt{]})$ (lifting $\subseteq$ to pairs of pairs of sets twice, pointwise), and every τ_i' is a subset of $(\texttt{[}D\texttt{]\&[}E\texttt{]}, \texttt{[}D\texttt{]})$. □

8 Extended Example: A Bookmark Lens

In this section, we develop an 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 *put* direction of the `bookmark` 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>`

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

Figure 2: Abstract Bookmark Types

```
<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>
```

Figure 3: Sample Bookmarks (HTML)

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:

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

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

```
{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
              }}]}}]}}]}}]}}]}}
```

Figure 4: Sample Bookmarks (concrete tree)

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

Figure 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}}]}}]}
```

Figure 6: Sample Bookmarks (abstract tree)

present example, it is important 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 `<tag attr1...attrn>` C `</tag>` for $\{\texttt{tag} \mapsto \{\texttt{attr1} \mapsto \mathit{Val} \ \ldots \texttt{attrn} \mapsto \mathit{Val}\ \texttt{*} \mapsto C\}\}$, where *Val* is the set of all values (trees with a single childless child). For elements with no attributes, this degenerates to simply `<tag>` C `</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 *put* direction restores the original names and rebuilds the necessary structure.

It is straightfoward to check, using the type annotations supplied, that `bookmarks` $\in \mathit{CBookmarks} \overset{\Omega,}{\rightleftharpoons} \mathit{AFolder}_1$. (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 `map`.

The *put* direction of the `link` lens restores original names and structure automatically, by composing the *put* 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 *put* to the updated abstract tree. The *put* 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 *put* 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

```
link =                                  ∈ {|* ↦ Clink1 :: []|}
    hoist *;                              : CLink1 :: []
    hd [];                                : CLink1
    hoist_nonunique a {* add_date href};  : {|* ↦ 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|}
    map {name -> (hd [];                  : PCDATA
                  hoist PCDATA)}          ≙Ω {|name ↦ Val, url ↦ Val|} = ALink1

folder =                                ∈ {|* ↦ CContents|}
    hoist *;                              : CContents
    hoist_hd {h3};                        : {|h3 ↦ {|* ↦ PCDATA :: []|}, CContents2 :: []|}
    fork {h3} (id) (hoist_hd {dl});       : {|h3 ↦ {|* ↦ PCDATA :: []|},
                                              dl ↦ {|* ↦ [CItem]|}|}
    rename h3 name;                       : {|name ↦ {|* ↦ PCDATA :: []|},
                                              dl ↦ {|* ↦ [CItem]|}|}
    rename dl contents;                   : {|name ↦ {|* ↦ PCDATA :: []|},
                                              contents ↦ {|* ↦ [CItem]|}|}
    map {name -> (hoist *;                          : PCDATA :: []
                  hd [];                            : PCDATA
                  hoist PCDATA)
         contents -> (hoist *;                      : [CItem]
                      list_map item)}
                                        ≙Ω {|name ↦ Val, contents ↦ [AItem]|} = AFolder1

item =                                  ∈ CItem
    map { 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
```

Figure 7: Bookmark lenses

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

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

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

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

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

Figure 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 *put*)...

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

Figure 9: Update of abstract tree, and resulting concrete tree

structure where necessary. Note the use of `hoist_hd` to extract the `<h3>` and `<dl>` tags containing the folder name and contents respectively; although the order of these two tags does not matter to us, it matters to Mozilla, so we want to ensure that the *put* direction of the lens puts them to their proper position in case of creation, which `hoist_hd` will ensure. Finally, we use `map` 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 the `map` 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.

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. 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" like

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

and flattens it into a bush like

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

The importance of this transformation is that it makes the "intended alignment" of the data structurally obvious, freeing the synchronization algorithm from having 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.

More generally, `flatten` handles concrete lists in which the same key appears more than once—e.g.,

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

—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:

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

In the *put* 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 $\texttt{AList}_K(D) = \texttt{[]} \cup (\{\{\!\{\texttt{k} \mapsto D\}\!\} \mid k \in K\} :: \texttt{AList}_K(D))$.

$$\texttt{flatten}\nearrow c = \begin{cases} \{\!\{\}\!\} & \text{if } c = \texttt{[]} \\ a' + \{\!\{k \mapsto d :: \texttt{[]}\}\!\} & \text{if } c = \{\!\{k \mapsto d\}\!\} :: c' \\ & \quad \text{and } \texttt{flatten}\nearrow c' = a' \text{ with } k \notin \mathsf{dom}(a') \\ a' + \{\!\{k \mapsto d :: s\}\!\} & \text{if } c = \{\!\{k \mapsto d\}\!\} :: c' \\ & \quad \text{and } \texttt{flatten}\nearrow c' = a' + \{\!\{k \mapsto s\}\!\} \end{cases}$$

$$\texttt{flatten}\searrow(a, c) = \begin{cases} \mathit{listify}(a) & \text{if } c = \texttt{[]} \text{ or } c = \Omega \\ \{\!\{k \mapsto d'\}\!\} :: r & \text{if } c = \{\!\{k \mapsto d\}\!\} :: c' \\ & \quad \text{and } a(k) = d' :: \texttt{[]} \\ & \quad \text{and } r = \texttt{flatten}\searrow(a\backslash_k, c') \\ \{\!\{k \mapsto d'\}\!\} :: r & \text{if } c = \{\!\{k \mapsto d\}\!\} :: c' \\ & \quad \text{and } a(k) = d' :: s \text{ with } s \neq \texttt{[]} \\ & \quad \text{and } r = \texttt{flatten}\searrow\left(a\backslash_k + \{\!\{k \mapsto s\}\!\}, c'\right) \\ r & \text{if } c = \{\!\{k \mapsto d\}\!\} :: c' \\ & \quad \text{and } k \notin \mathsf{dom}(a) \\ & \quad \text{and } r = \texttt{flatten}\searrow(a, c') \end{cases}$$

$$\forall K \subseteq \mathcal{N}.\ \forall D \subseteq \mathcal{T}. \quad \texttt{flatten} \in \texttt{AList}_K(D) \overset{\Omega}{\Longleftrightarrow} \left\{K \overset{?}{\mapsto} \texttt{[}D^{1..\omega}\texttt{]}\right\}$$

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.

9.1 Lemma [Well-behavedness]: $\forall K{\subseteq}\mathcal{N}.\ \forall D{\subseteq}\mathcal{T}.\ \texttt{flatten} \in \texttt{AList}_K(D) \stackrel{\Omega}{=} \left\{\!\!\left\{ K \stackrel{?}{\mapsto} \texttt{[}D^{1..\omega}\texttt{]} \right\}\!\!\right\}$.

Proof:
GET: Suppose $c \in \texttt{AList}_K(D)$ and $\texttt{flatten}{\nearrow}\, c$ is defined. Proceed by induction on the number of list cells in c. If $c = \texttt{[]}$, then the result is immediate. If $c = \{\!\{k \mapsto d\}\!\} :: c'$, then, by induction, $\texttt{flatten}{\nearrow}\, c' \in \left\{\!\!\left\{ K \stackrel{?}{\mapsto} \texttt{[}D^{1..\omega}\texttt{]} \right\}\!\!\right\}$. But then $a' + \{\!\{k \mapsto d :: \texttt{[]}\}\!\}$ (if $\texttt{flatten}{\nearrow}\, c' = a'$ with $k \notin \mathsf{dom}(a')$) and $a' + \{\!\{k \mapsto d :: s\}\!\}$ (if $\texttt{flatten}{\nearrow}\, c' = a' + \{\!\{k \mapsto s\}\!\}$) are also in $\left\{\!\!\left\{ K \stackrel{?}{\mapsto} \texttt{[}D^{1..\omega}\texttt{]} \right\}\!\!\right\}$, as required.

PUT: First, observe that $\mathit{listify}(a) \in \texttt{AList}_K(D)$. To see this, reason by induction on the size of $\mathsf{dom}(a)$. If $\mathsf{dom}(a) = \emptyset$, then $\mathit{listify}(a) = \texttt{[]} \in \texttt{AList}_K(D)$. Otherwise, $\mathit{listify}(a) = \{\!\{k \mapsto tk_1\}\!\} :: \cdots :: \{\!\{k \mapsto tk_n\}\!\} :: \mathit{listify}(t\backslash_k)$ where $k = \mathit{any}_{\mathsf{dom}(a)} \in K$ and and $t(k) = [tk_1, \ldots, tk_n]$, from which the result follows by the induction hypothesis and the definition of $\texttt{AList}_K(D)$.

Now, suppose $c \in (\texttt{AList}_K(D))_\Omega$, $a \in \left\{\!\!\left\{ K \stackrel{?}{\mapsto} \texttt{[}D^{1..\omega}\texttt{]} \right\}\!\!\right\}$, and $\texttt{flatten}{\searrow}\,(a,\, c)$ is defined. If $c = \Omega$, then $\texttt{flatten}{\searrow}\,(a,\, c) = \mathit{listify}(a) \in \texttt{AList}_K(D)$ by the observation above about *listify*. Otherwise, we proceed by induction on the number of list cells in c. If $c = \texttt{[]}$, then the result again follows by the observation about *listify*. If $c = \{\!\{k \mapsto d\}\!\} :: c'$, then there are three cases to consider:

- If $a(k) = d' :: \texttt{[]}$, then $\texttt{flatten}{\searrow}\,(a,\, c) = \{\!\{k \mapsto d'\}\!\} :: r$, with $r = \texttt{flatten}{\searrow}\,(a\backslash_k,\, c')$. By the induction hypothesis, $r \in \texttt{AList}_K(D)$, and the result follows immediately, since $k \in K$ and $d' \in D$ by the type of c.
- If $a(k) = d' :: s$ with $s \neq \texttt{[]}$, then $\texttt{flatten}{\searrow}\,(a,\, c) = \{\!\{k \mapsto d'\}\!\} :: r$, with $r = \texttt{flatten}{\searrow}\left(a\backslash_k + \{\!\{k \mapsto s\}\!\},\, c'\right)$. Again, the induction hypothesis applies (observing that $a\backslash_k + \{\!\{k \mapsto s\}\!\}$ belongs to $\left\{\!\!\left\{ K \stackrel{?}{\mapsto} \texttt{[}D^{1..\omega}\texttt{]} \right\}\!\!\right\}$ because s is assumed to be non-empty), giving us $r \in \texttt{AList}_K(D)$, from which the result follows directly.
- If $k \notin \mathsf{dom}(a)$, then $\texttt{flatten}{\searrow}\,(a,\, c) = \texttt{flatten}{\searrow}\,(a,\, c')$. The induction hypothesis yields $r \in \texttt{AList}_K(D)$ and the result follows directly.

GETPUT: Suppose $c \in \texttt{AList}_K(D)$ and $\texttt{flatten}{\searrow}\,(\texttt{flatten}{\nearrow}\, c,\, c)$ is defined. Proceed by induction on the number of list cells in c. If $c = \texttt{[]}$, then $\texttt{flatten}{\searrow}\,(\texttt{flatten}{\nearrow}\, c,\, c) = \mathit{listify}(\{\!\}) = \texttt{[]}$, as required. If $c = \{\!\{k \mapsto d\}\!\} :: c'$, then there are two cases to consider:

- if $\texttt{flatten}{\nearrow}\, c' = a'$ with $k \notin \mathsf{dom}(a')$, then $\texttt{flatten}{\nearrow}\, c = a' + \{\!\{k \mapsto d :: \texttt{[]}\}\!\}$ and

$$\begin{aligned}
\texttt{flatten}{\searrow}\,(\texttt{flatten}{\nearrow}\, c,\, c) &= \{\!\{k \mapsto d\}\!\} :: \texttt{flatten}{\searrow}\,(a',\, c') \\
&= \{\!\{k \mapsto d\}\!\} :: \texttt{flatten}{\searrow}\,(\texttt{flatten}{\nearrow}\, c',\, c') \\
&= \{\!\{k \mapsto d\}\!\} :: c' && \text{by the induction hypothesis} \\
&= c
\end{aligned}$$

- if $\texttt{flatten}\nearrow c' = a' + \{\!|k \mapsto s|\!\}$, then $\texttt{flatten}\nearrow c = a' + \{\!|k \mapsto d :: s|\!\}$ and

$$\begin{aligned}
\texttt{flatten}\searrow(\texttt{flatten}\nearrow c,\, c) &= \{\!|k \mapsto d|\!\} :: \texttt{flatten}\searrow\left(a' + \{\!|k \mapsto s|\!\},\, c'\right) \\
&= \{\!|k \mapsto d|\!\} :: \texttt{flatten}\searrow(\texttt{flatten}\nearrow c',\, c') \\
&= \{\!|k \mapsto d|\!\} :: c' && \text{by the induction hypothesis} \\
&= c
\end{aligned}$$

PUTGET: Observe, first, that $\texttt{flatten}\nearrow(\mathit{listify}(a)) = a$ for any a. To see this, reason by induction on the size of $\mathsf{dom}(a)$. If $\mathsf{dom}(a) = \emptyset$, then $\texttt{flatten}\nearrow(\mathit{listify}(a)) = \texttt{flatten}\nearrow\texttt{[]} = \{\!|\,|\!\} = a$. Otherwise, $\texttt{flatten}\nearrow(\mathit{listify}(a)) = \texttt{flatten}\nearrow(\{\!|k \mapsto tk_1|\!\} :: \cdots :: \{\!|k \mapsto tk_n|\!\} :: \mathit{listify}(a\backslash_k))$, where $k = \mathit{any}_{\mathsf{dom}(a)}$ and $t(k) = [tk_1, \ldots, tk_n]$. The result then follows by the induction hypothesis and n invocations of the definition of $\texttt{flatten}\nearrow$.

Now, suppose $c \in (\texttt{AList}_K(D))_\Omega$, $a \in \left\{\!\left|K \overset{?}{\mapsto} \texttt{[}D^{1..\omega}\texttt{]}\right|\!\right\}$, and $\texttt{flatten}\nearrow(\texttt{flatten}\searrow(a,\, c))$ is defined. If $c = \Omega$, then $\texttt{flatten}\nearrow(\texttt{flatten}\searrow(a,\, c)) = \texttt{flatten}\nearrow(\mathit{listify}(a)) = a$ by the observation above. Otherwise, we proceed by induction on the number of list cells in c. If $c = \texttt{[]}$, then the result again follows by the observation above. If $c = \{\!|k \mapsto d|\!\} :: c'$, then there are three cases to consider:

- If $a(k) = d' :: \texttt{[]}$, then, by the definition, $\texttt{flatten}\nearrow(\texttt{flatten}\searrow(a,\, c)) = \texttt{flatten}\nearrow(\{\!|k \mapsto d'|\!\} :: \texttt{flatten}\searrow(a\backslash_k,\, c'))$. Now, since $\texttt{flatten}\nearrow(\texttt{flatten}\searrow(a\backslash_k,\, c')) = a\backslash_k$ by the induction hypothesis, and since $k \notin \mathsf{dom}(a\backslash_k)$, the definition of $\texttt{flatten}\nearrow$ gives $\texttt{flatten}\nearrow(\{\!|k \mapsto d'|\!\} :: \texttt{flatten}\searrow(a\backslash_k,\, c')) = (a\backslash_k) + \{\!|k \mapsto d' :: \texttt{[]}|\!\} = a$.
- If $a(k) = d' :: s$ with $s \neq \texttt{[]}$, then, by the definition, $\texttt{flatten}\nearrow(\texttt{flatten}\searrow(a,\, c)) = \texttt{flatten}\nearrow(\{\!|k \mapsto d'|\!\} :: (\texttt{flatten}\searrow\left(a\backslash_k + \{\!|k \mapsto s|\!\},\, c'\right)))$. Now, since $\texttt{flatten}\nearrow(\texttt{flatten}\searrow\left(a\backslash_k + \{\!|k \mapsto s|\!\},\, c'\right)) = a\backslash_k + \{\!|k \mapsto s|\!\}$ by the induction hypothesis, the definition of $\texttt{flatten}\nearrow$ gives $\texttt{flatten}\nearrow(\{\!|k \mapsto d'|\!\} :: (\texttt{flatten}\searrow\left(a\backslash_k + \{\!|k \mapsto s|\!\},\, c'\right))) = a\backslash_k + \{\!|k \mapsto d' :: s|\!\} = a$.
- If $k \notin \mathsf{dom}(a)$, then, by the definition, $\texttt{flatten}\nearrow(\texttt{flatten}\searrow(a,\, c)) = \texttt{flatten}\nearrow(\texttt{flatten}\searrow(a,\, c')) = a$ by the induction hypotheses. □

9.2 Lemma [Totality]: $\forall K \subseteq \mathcal{N}.\ \forall D \subseteq \mathcal{T}.\ \ \texttt{flatten} \in \texttt{AList}_K(D) \overset{\Omega}{\Longleftrightarrow} \left\{\!\left|K \overset{?}{\mapsto} \texttt{[}D^{1..\omega}\texttt{]}\right|\!\right\}$.

Proof: For the *get* direction, suppose $c \in \texttt{AList}_K(D)$. We must show that $\texttt{flatten}\nearrow c$ is defined. Proceed by induction on the number of cells in c. If $c = \texttt{[]}$, then $\texttt{flatten}\nearrow c = \{\!|\,|\!\}$. If $c = \{\!|k \mapsto d|\!\} :: c'$, then, by induction, $\texttt{flatten}\nearrow c'$ is defined and the definedness of $\texttt{flatten}\nearrow c$ follows directly.

For the *put* direction, suppose $c \in (\texttt{AList}_K(D))_\Omega$ and $a \in \left\{\!\left|K \overset{?}{\mapsto} \texttt{[}D^{1..\omega}\texttt{]}\right|\!\right\}$. We must show that $\texttt{flatten}\searrow(a,\, c)$ is defined. If $c = \Omega$, then $\texttt{flatten}\searrow(a,\, c) = \mathit{listify}(a)$, which is defined because *listify* is defined on all arguments in $\left\{\!\left|K \overset{?}{\mapsto} \texttt{[}D^{1..\omega}\texttt{]}\right|\!\right\}$ (as is easily verified by induction on $|\mathsf{dom}(a)|$). Otherwise, proceed by induction on the number of list cells in c. If $c = \texttt{[]}$, then the result again follows by the definedness of *listify*. If $c = \{\!|k \mapsto d|\!\} :: c'$, then there are three cases to consider:

- If $a(k) = d' :: \texttt{[]}$, then $\texttt{flatten}\searrow(a,\, c) = \{\!|k \mapsto d'|\!\} :: r$, with $r = \texttt{flatten}\searrow(a\backslash_k,\, c')$. By the induction hypothesis, r is defined, and the definedness of $\texttt{flatten}\searrow(a,\, c)$ is immediate.
- If $a(k) = d' :: s$ with $s \neq \texttt{[]}$, then $\texttt{flatten}\searrow(a,\, c) = \{\!|k \mapsto d'|\!\} :: r$, with $r = \texttt{flatten}\searrow\left(a\backslash_k + \{\!|k \mapsto s|\!\},\, c'\right)$. Again, the induction hypothesis tells us that r is defined, and the definedness of $\texttt{flatten}\searrow(a,\, c)$ is immediate.
- If $k \notin \mathsf{dom}(a)$, then $\texttt{flatten}\searrow(a,\, c) = \texttt{flatten}\searrow(a,\, c')$, and the result is immediate by the induction hypothesis. □

Pivot

The lens `pivot` n rearranges the structure at the top of a tree, transforming $\left\{\!\left|\begin{matrix} n \mapsto k \\ t \end{matrix}\right|\!\right\}$ 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 *put* 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. For example, if the concrete presentation of the data looks like this,

$$\left[\begin{array}{l} \left\{\!\left|\begin{array}{l} \texttt{Name} \mapsto \texttt{Pat} \\ \texttt{Phone} \mapsto \texttt{333-4444} \\ \texttt{URL} \mapsto \texttt{http://pat.com} \end{array}\right|\!\right\} \\ \left\{\!\left|\begin{array}{l} \texttt{Name} \mapsto \texttt{Chris} \\ \texttt{Phone} \mapsto \texttt{888-9999} \\ \texttt{URL} \mapsto \texttt{http://chris.org} \end{array}\right|\!\right\} \\ \left\{\!\left|\begin{array}{l} \texttt{Name} \mapsto \texttt{Pat} \\ \texttt{Phone} \mapsto \texttt{123-4321} \\ \texttt{URL} \mapsto \texttt{http://pattoo.com} \end{array}\right|\!\right\} \end{array}\right]$$

then applying (`map_list` (`pivot` `Name`))↗ yields

$$\left[\begin{array}{l} \left\{\!\left|\begin{array}{l} \texttt{Pat} \mapsto \left\{\!\left|\begin{array}{l} \texttt{Phone} \mapsto \texttt{333-4444} \\ \texttt{URL} \mapsto \texttt{http://pat.com} \end{array}\right|\!\right\} \end{array}\right|\!\right\} \\ \left\{\!\left|\begin{array}{l} \texttt{Chris} \mapsto \left\{\!\left|\begin{array}{l} \texttt{Phone} \mapsto \texttt{888-9999} \\ \texttt{URL} \mapsto \texttt{http://chris.org} \end{array}\right|\!\right\} \end{array}\right|\!\right\} \\ \left\{\!\left|\begin{array}{l} \texttt{Pat} \mapsto \left\{\!\left|\begin{array}{l} \texttt{Phone} \mapsto \texttt{123-4321} \\ \texttt{URL} \mapsto \texttt{http://pattoo.com} \end{array}\right|\!\right\} \end{array}\right|\!\right\} \end{array}\right]$$

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

$$\left\{\!\left|\begin{array}{l} \texttt{Pat} \mapsto \left[\begin{array}{l} \left\{\!\left|\begin{array}{l} \texttt{Phone} \mapsto \texttt{333-4444} \\ \texttt{URL} \mapsto \texttt{http://pat.com} \end{array}\right|\!\right\} \\ \left\{\!\left|\begin{array}{l} \texttt{Phone} \mapsto \texttt{123-4321} \\ \texttt{URL} \mapsto \texttt{http://pattoo.com} \end{array}\right|\!\right\} \end{array}\right] \\ \texttt{Chris} \mapsto \left[\begin{array}{l} \left\{\!\left|\begin{array}{l} \texttt{Phone} \mapsto \texttt{888-9999} \\ \texttt{URL} \mapsto \texttt{http://chris.org} \end{array}\right|\!\right\} \end{array}\right] \end{array}\right|\!\right\}.$$

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., $\{\{\!|n \mapsto \{\!|k \mapsto \{\!|\}\!|\}\!|\}\!| \mid k \in K\}$.

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

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

9.3 Lemma [Well-behavedness]: $\forall n{\in}\mathcal{N}.\ \forall K{\subseteq}\mathcal{N}.\ \forall C{\subseteq}(\mathcal{T}\backslash_n).\quad \texttt{pivot}\ n\ \in\ (\{\!|n \mapsto K|\!\}\ \cdot\ C)\ \overset{\Omega}{\rightleftharpoons}\ \{\{\!|k \mapsto C|\!\}\ \mid k \in K\}$.

Proof:

GET: $(\texttt{pivot}\ n)\nearrow \left\{\!\middle|\begin{matrix}n \mapsto k\\ t\end{matrix}\!\middle|\right\} = \{\!|k \mapsto t|\!\} \in \{\{\!|k \mapsto C|\!\}\ \mid k \in K\}$

PUT: $(\texttt{pivot}\ n)\searrow(\{\!|k \mapsto t|\!\}, c) = \left\{\!\middle|\begin{matrix}n \mapsto k\\ t\end{matrix}\!\middle|\right\} \in (\{\!|n \mapsto K|\!\} \cdot C)$

GETPUT: Assume that $(\texttt{pivot}\ n)\nearrow c$ is defined, thus $c = \left\{\!\middle|\begin{matrix}n \mapsto k\\ t\end{matrix}\!\middle|\right\}$. We have:

$$\begin{aligned}
& (\texttt{pivot}\ n)\searrow\left((\texttt{pivot}\ n)\nearrow \left\{\!\middle|\begin{matrix}n \mapsto k\\ t\end{matrix}\!\middle|\right\}, \left\{\!\middle|\begin{matrix}n \mapsto k\\ t\end{matrix}\!\middle|\right\}\right)\\
= \ & (\texttt{pivot}\ n)\searrow\left(\{\!|k \mapsto t|\!\}, \left\{\!\middle|\begin{matrix}n \mapsto k\\ t\end{matrix}\!\middle|\right\}\right)\\
= \ & \left\{\!\middle|\begin{matrix}n \mapsto k\\ t\end{matrix}\!\middle|\right\}
\end{aligned}$$

PUTGET: Assume that $(\texttt{pivot}\ n)\searrow(a, c)$ is defined, thus $a = \{\!|k \mapsto t|\!\}$. We have:

$$(\texttt{pivot}\ n)\nearrow((\texttt{pivot}\ n)\searrow(\{\!|k \mapsto t|\!\}, c)) = (\texttt{pivot}\ n)\nearrow \left\{\!\middle|\begin{matrix}n \mapsto k\\ t\end{matrix}\!\middle|\right\} = \{\!|k \mapsto t|\!\}. \qquad \square$$

9.4 Lemma [Totality]: $\forall n{\in}\mathcal{N}.\ \forall K{\subseteq}\mathcal{N}.\ \forall C{\subseteq}(\mathcal{T}\backslash_n).\ \texttt{pivot}\ n \in (\{\!|n \mapsto K|\!\} \cdot C) \overset{\Omega}{\Longleftrightarrow} \{\{\!|k \mapsto C|\!\}\ \mid k \in K\}$.

Proof: Straightforward from the definition. $\square$

Join

Our final lens combinator, based on an idea by Daniel Spoonhower [42], is inspired by the *full outer join* operator from databases. For example, applying $(\texttt{join addr phone})\nearrow$ to a tree

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

containing a collection of addresses and a collection of phone numbers (both keyed by names) yields a tree

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

where the address and phone information is collected under each name. 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 *put* 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 *put* function can ignore its concrete argument.)

$$
\begin{array}{rcl}
(\texttt{join m n})\nearrow c & = & \left\{k \mapsto \left\{\begin{array}{l} m \mapsto c(m)(k) \\ n \mapsto c(n)(k) \end{array}\right\} \;\middle|\; k \in \mathsf{dom}(c(m)) \cup \mathsf{dom}(c(n))\right\} \\
(\texttt{join m n})\searrow(a,\, c) & = & \left\{\begin{array}{l} m \mapsto \{k \mapsto a(k)(m) \mid k \in \mathsf{dom}(a)\} \\ n \mapsto \{k \mapsto a(k)(n) \mid k \in \mathsf{dom}(a)\} \end{array}\right\} \\
\hline
\multicolumn{3}{c}{\forall K{\subseteq}\mathcal{N}.\ \forall T{\subseteq}\mathcal{T}.\ \texttt{join m n} \in \left\{\begin{array}{l} m \mapsto \left\{K \overset{?}{\mapsto} T\right\} \\ n \mapsto \left\{K \overset{?}{\mapsto} T\right\} \end{array}\right\} \overset{\Omega}{\Longleftrightarrow} \left\{K \overset{?}{\mapsto} \left\{\begin{array}{l} m \mapsto T \\ n \overset{?}{\mapsto} T \end{array}\right\} \cup \left\{\begin{array}{l} m \overset{?}{\mapsto} T \\ n \mapsto T \end{array}\right\}\right\}}
\end{array}
$$

9.5 Lemma [Well-behavedness]: $\forall K{\subseteq}\mathcal{N}.\ \forall T{\subseteq}\mathcal{T}.\ \texttt{join m n} \in \left\{m \mapsto \left\{K \overset{?}{\mapsto} T\right\}, n \mapsto \left\{K \overset{?}{\mapsto} T\right\}\right\} \overset{\Omega}{=\!=} \left\{K \overset{?}{\mapsto} \left\{m \mapsto T, n \overset{?}{\mapsto} T\right\} \cup \left\{m \overset{?}{\mapsto} T, n \mapsto T\right\}\right\}$.

Proof:

GET: Suppose $c \in \left\{m \mapsto \left\{K \overset{?}{\mapsto} T\right\}, n \mapsto \left\{K \overset{?}{\mapsto} T\right\}\right\}$. Suppose that $(\texttt{join m n})\nearrow c$ is defined, and write a' for $(\texttt{join m n})\nearrow c$. For each $k \in K$, we must show that $a'(k) \in \left(\left\{m \mapsto T,\, n \overset{?}{\mapsto} T\right\} \cup \left\{m \overset{?}{\mapsto} T,\, n \mapsto T\right\}\right)_\Omega$. There are three possibilities to consider: First, if $k \in \mathsf{dom}(c(m))$, then $c(m)(k) \in T$ by the type of c. Also, $c(n)(k) \in T_\Omega$, so $a'(k) \in \left\{m \mapsto T,\, n \overset{?}{\mapsto} T\right\}$. Second, if $k \in \mathsf{dom}(c(n))$, then similarly $a'(k) \in \left\{m \overset{?}{\mapsto} T,\, n \mapsto T\right\}$. Finally, if $k \notin \mathsf{dom}(c(m)) \cup \mathsf{dom}(c(n))$, then $k \notin \mathsf{dom}(a')$, which is permitted by the target type.

PUT: Suppose that a has type $\left\{K \overset{?}{\mapsto} \left\{m \mapsto T, n \overset{?}{\mapsto} T\right\} \cup \left\{m \overset{?}{\mapsto} T, n \mapsto T\right\}\right\}$ and that c has type $\left\{m \mapsto \left\{K \overset{?}{\mapsto} T\right\}, n \mapsto \left\{K \overset{?}{\mapsto} T\right\}\right\}$. Suppose that $(\texttt{join m n})\searrow(a,\, c)$ is defined, and write c' for $(\texttt{join m n})\searrow(a,\, c)$. For each $k \in \mathsf{dom}(a)$, note that $a(k)(m) \in T_\Omega$, so $\{k \mapsto a(k)(m) \mid k \in \mathsf{dom}(a)\} \in \left\{K \overset{?}{\mapsto} T\right\}$, and similarly for n.

GETPUT: Suppose $c \in \left\{m \mapsto \left\{K \overset{?}{\mapsto} T\right\}, n \mapsto \left\{K \overset{?}{\mapsto} T\right\}\right\}$ and $(\texttt{join m n})\searrow((\texttt{join m n})\nearrow c,\, c)$ defined. Now calculate as follows, writing a as $(\texttt{join m n})\nearrow c$, and using the fact that $\mathsf{dom}(a) = \mathsf{dom}(c(m)) \cup$

$\mathsf{dom}(c(n))$:

$$
\begin{aligned}
&(\texttt{join m n}) \searrow ((\texttt{join m n}) \nearrow c,\, c)\\
=&(\texttt{join m n}) \searrow \left(\left\{\left[k \mapsto \begin{Bmatrix} m \mapsto c(m)(k) \\ n \mapsto c(n)(k) \end{Bmatrix}\right] \;\middle|\; k \in \mathsf{dom}(c(m)) \cup \mathsf{dom}(c(n))\right\},\, c\right)\\
=&\left\{\begin{bmatrix} m \mapsto \{k' \mapsto a(k')(m) \mid k' \in \mathsf{dom}(a)\} \\ n \mapsto \{k' \mapsto a(k')(n) \mid k' \in \mathsf{dom}(a)\} \end{bmatrix}\right\}\\
=&\left\{\begin{bmatrix} m \mapsto \left\{k' \mapsto \left\{\left[k \mapsto \begin{Bmatrix} m \mapsto c(m)(k) \\ n \mapsto c(n)(k) \end{Bmatrix}\right] \;\middle|\; k \in \mathsf{dom}(c(m)) \cup \mathsf{dom}(c(n))\right\}(k')(m) \mid k' \in \mathsf{dom}(a)\right\} \\ n \mapsto \left\{k' \mapsto \left\{\left[k \mapsto \begin{Bmatrix} m \mapsto c(m)(k) \\ n \mapsto c(n)(k) \end{Bmatrix}\right] \;\middle|\; k \in \mathsf{dom}(c(m)) \cup \mathsf{dom}(c(n))\right\}(k')(n) \mid k' \in \mathsf{dom}(a)\right\} \end{bmatrix}\right\}\\
=&\left\{\begin{bmatrix} m \mapsto \left\{k' \mapsto \left\{\left[k \mapsto \begin{Bmatrix} m \mapsto c(m)(k) \\ n \mapsto c(n)(k) \end{Bmatrix}\right] \;\middle|\; k \in \mathsf{dom}(a)\right\}(k')(m) \mid k' \in \mathsf{dom}(a)\right\} \\ n \mapsto \left\{k' \mapsto \left\{\left[k \mapsto \begin{Bmatrix} m \mapsto c(m)(k) \\ n \mapsto c(n)(k) \end{Bmatrix}\right] \;\middle|\; k \in \mathsf{dom}(a)\right\}(k')(n) \mid k' \in \mathsf{dom}(a)\right\} \end{bmatrix}\right\}\\
=&\left\{\begin{bmatrix} m \mapsto \left\{k' \mapsto \begin{Bmatrix} m \mapsto c(m)(k') \\ n \mapsto c(n)(k') \end{Bmatrix}(m) \mid k' \in \mathsf{dom}(a)\right\} \\ n \mapsto \left\{k' \mapsto \begin{Bmatrix} m \mapsto c(m)(k') \\ n \mapsto c(n)(k') \end{Bmatrix}(n) \mid k' \in \mathsf{dom}(a)\right\} \end{bmatrix}\right\}\\
=&\left\{\begin{bmatrix} m \mapsto \{k' \mapsto c(m)(k') \mid k' \in \mathsf{dom}(a)\} \\ n \mapsto \{k' \mapsto c(n)(k') \mid k' \in \mathsf{dom}(a)\} \end{bmatrix}\right\}\\
=&\left\{\begin{bmatrix} m \mapsto \{k' \mapsto c(m)(k') \mid k' \in \mathsf{dom}(c(m)) \cup \mathsf{dom}(c(n))\} \\ n \mapsto \{k' \mapsto c(n)(k') \mid k' \in \mathsf{dom}(c(m)) \cup \mathsf{dom}(c(n))\} \end{bmatrix}\right\}\\
=&c
\end{aligned}
$$

PutGet: Suppose $a \in \left\{K \overset{?}{\mapsto} \left\{m \mapsto T, n \overset{?}{\mapsto} T\right\} \cup \left\{m \overset{?}{\mapsto} T, n \mapsto T\right\}\right\}$ and $c \in \left\{m \mapsto \left\{K \overset{?}{\mapsto} T\right\}, n \mapsto \left\{K \overset{?}{\mapsto} T\right\}\right\}$. Suppose that $(\texttt{join m n}) \nearrow ((\texttt{join m n}) \searrow (a, c))$ is defined, and write a' for $(\texttt{join m n}) \nearrow ((\texttt{join m n}) \searrow (a, c))$. Consider an arbitrary $k \in K$. If $k \notin \mathsf{dom}(a)$, then, by the definition of the *put* function, $k \notin \mathsf{dom}((\texttt{join m n}) \searrow (a, c)(m))$ and $k \notin \mathsf{dom}((\texttt{join m n}) \searrow (a, c)(n))$; hence, $k \notin \mathsf{dom}(a')$. On the other hand, if $k \in \mathsf{dom}(a)$, then, by the type of a and the definition of the *put* function, either $k \in \mathsf{dom}((\texttt{join m n}) \searrow (a, c))(m))$ or $k \in \mathsf{dom}((\texttt{join m n}) \searrow (a, c))(n))$, so, by the definition of the *get* function, $k \in \mathsf{dom}(a')$, with $a'(k)(m) = a(k)(m)$ and $a'(k)(n) = a(k)(n)$. □

9.6 Lemma [Totality]: $\forall K \subseteq \mathcal{N}.\ \forall T \subseteq \mathcal{T}.\ \texttt{join m n} \in \left\{m \mapsto \left\{K \overset{?}{\mapsto} T\right\}, n \mapsto \left\{K \overset{?}{\mapsto} T\right\}\right\} \overset{\Omega}{\Longleftrightarrow} \left\{K \overset{?}{\mapsto} \left\{m \mapsto T, n \overset{?}{\mapsto} T\right\} \cup \left\{m \overset{?}{\mapsto} T, n \mapsto T\right\}\right\}$.

Proof: The *get* and *put* components are both total functions. □

10 Related Work

The lens combinators described in this paper evolved in the setting of the Harmony data synchronizer. The overall architecture of Harmony and the role of lenses in building synchronizers for various forms of data are described in [38], along with a detailed discussion of related work on synchronization.

Our foundational structures—lenses and their laws—are not new: closely related structures have been studied for decades in the database community. However, our "programming language treatment" of these structures has led us to a formulation that is arguably simpler (transforming states rather than "update

functions") and somewhat more refined (treating well-behavedness as a form of type assertion). Our formulation is also novel in considering the issue of continuity, thus supporting a rich variety of surface language structures including definition by recursion.

The idea of defining programming languages for constructing bi-directional transformations of various sorts has also been explored previously in diverse communities. We appear to be the first to take totality as a primary goal (while connecting the language with a formal semantic foundation, choosing primitives that can be combined into composite lenses whose totality is guaranteed by construction), and the first to emphasize types (i.e., compositional reasoning) as an organizing design principle.

Foundations of View Update

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

Dayal and Bernstein [16] gave a seminal formal account of the theory of "correct update translation." Their notion of "exactly performing an update" corresponds to our PUTGET law. Their "absence of side effects" corresponds to our GETPUT and PUTPUT laws. Their requirement of preservation of semantic consistency corresponds to the partiality of our *put* functions.

Bancilhon and Spyratos [9] developed an elegant semantic characterization of update translation, introducing the notion of *complement* of a view, which must include at least all information missing from the view. When a complement is fixed, there exists at most one update of the database that reflects a given update on the view while leaving the complement unmodified—i.e., that "translates updates under a constant complement." In general, a view may have many complements, each corresponding to a possible strategy for translating view updates to database updates. The problem of translating view updates then becomes a problem of finding, for a given view, a suitable complement.

Gottlob, Paolini, and Zicari [19] offered a more refined theory based on a syntactic translation of view updates. They identified a hierarchy of restricted cases of their framework, the most permissive form being their "dynamic views" and the most restrictive, called "cyclic views with constant complement," being formally equivalent to Bancilhon and Spyratos's update translators.

In a companion report [37], we state a precise correspondence between our lenses and the structures studied by Bancilhon and Spyratos and by Gottlob, Paolini, and Zicari. Briefly, our set of very well behaved lenses is isomorphic to the set of *translators under constant complement* in the sense of Bacilhon and Spyratos, while our set of well-behaved lenses is isomorphic to the set of *dynamic views* in the sense of Gottlob, Paolini, and Zicari. To be precise, both of these results must be qualified by an additional condition regarding partiality. The frameworks of Bacilhon and Spyratos and of Gottlob, Paolini, and Zicari are both formulated in terms of translating *update functions* on A into update functions on C, i.e., their *put* functions have type $(A \longrightarrow A) \longrightarrow (C \longrightarrow C)$, while our lenses translate abstract *states* into update functions on C, i.e., our *put* functions have type (isomorphic to) $A \longrightarrow (C \longrightarrow C)$. Moreover, in both of these frameworks, "update translators" (the analog of our *put* functions) are defined only over some particular chosen set U of abstract update functions, not over all functions from A to A. These update translators return *total* functions from C to C. Our *put* functions, on the other hand, are 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.

A related observation is that, if we restrict both *get* and *put* to be total functions (i.e., *put* 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.

In the literature on programming languages, laws similar to our lens laws (but somewhat simpler, dealing only with total *get* and *put* functions) appear in Oles' category of "state shapes" [36] and in Hofmann and Pierce's work on "positive subtyping" [20].

Recent work by Lechtenbörger [25] establishes that translations of view updates under constant complements are possible precisely if view update effects may be undone using further view updates.

Languages for Bi-Directional Transformations

At the level of syntax, different forms of bi-directional programming have been explored across a surprisingly diverse range of communities, including programming languages, databases, program transformation, constraint-based user interfaces, and quantum computing. One useful way of classifying these languages is by the "shape" of the semantic space in which their transformations live. We identify three major classes:

- *Bi-directional languages*, including ours, form lenses by pairing a *get* function of type $C \to A$ with a put function of type $A \times C \to C$. In general, the *get* function can project away some information from the concrete view, which must then be restored by the *put* function.
- In *bijective languages*, the *put* function has the simpler type $A \to C$—it is given no concrete argument to refer to. To avoid loss of information, the *get* and *put* functions must form a (perhaps partial) bijection between C and A.
- *Reversible languages* go a step further, demanding only that the work performed by any function to produce a given output can be undone by applying the function "in reverse" working backwards from this output to produce the original input. Here, there is no separate *put* function at all: instead, the *get* function itself is constructed so that each step can be run in reverse.

In the first class, the work that is fundamentally most similar to ours is Meertens's formal treatment of *constraint maintainers* for constraint-based user interfaces [30]. Meertens's semantic setting is actually even more general: he takes *get* and *put* to be *relations*, not just functions, and his constraint maintainers are symmetric: *get* relates pairs from $C \times A$ to elements of A and *put* relates pairs in $A \times C$ to elements of C; the idea is that a constraint maintainer forms a connection between two graphical objects on the screen so that, whenever one of the objects is changed by the user, the change can be propagated by the maintainer to the other object such that some desired relationship between the objects is always maintained. Taking the special case where the *get* relation is actually a function (which is important for Meertens because this is the case where composition [in the sense of our ; combinator] is guaranteed to preserve well-behavedness), yields essentially our very well behaved lenses. Meertens proposes a variety of combinators for building constraint maintainers, most of which have analogs among our lenses, but does not directly deal with definition by recursion; also, some of his combinators do not support compositional reasoning about well-behavedness. He considers constraint maintainers for structured data such as lists, as we do for trees, but here adopts a rather different point of view from ours, focusing on constraint maintainers that work with structures not directly but in terms of the "edit scripts" that might have produced them. In the terminology of synchronization, he switches from a state-based to an operation-based treatment at this point.

Recent work of Mu, Hu, and Takeichi on "injective languages" for view-update-based structure editors [32] adopts a similar perspective. Although their transformations obey our GetPut law, their notion of well-behaved transformations is informed by different goals than ours, leading to a weaker form of the PutGet law. A primary concern is using the view-to-view transformations to simultaneously restore invariants *within* the source view as well as update the concrete view. For example, an abstract view may maintain two lists where the name field of each element in one list must match the name field in the corresponding element in the other list. If an element is added to the first list, then not only must the change be propagated to the concrete view, it must also add a new element to the second list in the abstract view. It is easy to see that PutGet cannot hold if the abstract view, itself, is—in this sense—modified by the *put*. Similarly, they assume that edits to the abstract view mark all modified fields as "updated." These marks are removed when the *put* lens computes the modifications to the concrete view—another change to the abstract view that must violate PutGet. Consequently, to support invariant preservation within the abstract view, and to support edit lists, their transformations only obey a much weaker variant of PutGet (described above in Section 5.5).

Another paper by Hu, Mu, and Takeichi [21] applies a bi-directional programming language quite closely related to ours to the design of "programmable editors" for structured documents. As in [32], they support preservation of local invariants in the *put* direction. Here, instead of annotating the abstract view with

modification marks, they assume that a *put* or a *get* occurs after *every* modification to either view. They use this "only one update" assumption to choose the correct inverse for the lens that copied data in the *get* direction — because only one branch can have been modified at any given time. Consequently, they can *put* the data from the modified branch and overwrite the unmodified branch. Here, too, the notion of well-behavedness needs to be weakened, as described in Section 5.5.

The TRIP2 system (e.g. [27]) uses bidirectional transformations specified as collections of Prolog rules as a means of implementing direct-manipulation interfaces for application data structures. The *get* and *put* components of these mappings are written separately by the user.

Languages for Bijective Transformations

An active thread of work in the program transformation community concerns *program inversion* and *inverse computation*—see, for example, [4, 5] and many other papers cited there. Program inversion [18] derives the inverse program from the forward program. Inverse computation [28] computes a possible input of a program from the program and a particular output. One approach to inverse computation is to design languages that produce easily invertible expressions. For example, designing languages that can only express injective functions (in which case every program is trivially invertible). These languages bear some intriguing similarities to ours, but differ in a number of ways, primarily in their focus on the bijective case.

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

Ohori and Tajima [35] 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* [45], extends algebraic pattern matching to abstract data types using programmer-supplied *in* and *out* operators.

Languages for Reversible Transformations

Our work is the first (of which we are aware) in which totality and compositional reasoning about totality are taken as primary design goals. Nevertheless, in all of the languages discussed above there is an expectation that programmers will want their transformations to be "total enough"—i.e., that the sets of inputs for which the *get* and *put* functions are defined should be large enough for some given purpose. In particular, we expect that *put* functions should be accept a suitably large set of abstract inputs for each given concrete input, since the whole point of these languages is to allow editing through a view. A quite different class of languages have been designed to support *reversible* computation, in which the *put* functions are only ever applied to the results of the corresponding *get* functions. While the goals of these languages are quite different from ours—they have nothing to do with view update—there are intriguing similarities in the basic approach.

Landauer [24] observed that non-injective functions were logically irreversible, and that this irreversibility requires the generation and dissipation of some heat per machine cycle. Bennet [11] demonstrated that this irreversibility was not inevitable by constructing a *reversible Turing machine*, showing that thermodynamically reversible computers were plausible. Baker [8] 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. [33]).

Update Translation for Tree Views

There have been many proposals for query languages for trees (e.g., XQuery [44] 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 [17], and Braganholo, Davidson, and Heuser [12] 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 [43] 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 [41] restrict the notion of views to queries that preserve object identity. The view update problem is greatly simplified in this setting, as the objects contained in the view are the objects of the database, and an update on the view is directly an update on objects of the database.

Update Translation for Relational Views

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

Masunaga [26] 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 [22] 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 [23] 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 [10] described a scheme for interactively constructing update translators for object-based views of relational databases.

Medeiros and Tompa [29] 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.* [31] describe Clio, a system for managing heterogenous 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 [7, 6] 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 [14, 15] 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 [13] 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 [40] argued that programmers should specify the translations of abstract updates. However, they did not provide a way to ensure consistency between the *get* and *put* directions of their translations.

Another problem that is sometimes mentioned in connection with view update translation is that of *incremental view maintenance* (e.g., [3])—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 Future Work

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

Naturally, the progress we have made on lens combinators raises a host of further challenges.

Static Analysis

The most urgent of these is automated typechecking. At present, it is the lens programmers' responsibility to check the well-behavedness of the lenses that they write. But the types of the primitive combinators have been designed so that these checks are both local and essentially mechanical. The obvious next step is to

reformulate the type declarations as a type *algebra* and find a mechanical procedure for checking (or, more ambitiously, inferring) types.

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

Implementation

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

Applications

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 are currently developing a number of synchronizers (for calendars, address books, structured text, etc.) as instances of Harmony. This exercise provides valuable stress-testing for both our combinators and their formal foundations.

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 both intricate structural manipulations such as the list transformations in Section 7 as well as 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 transformations on trees (including analogs of database operations such as `join`), or lenses addressing completely different sorts of transformations (e.g., none of our combinators do any significant processing on edge labels, which might include string processing, arithmetic, etc.).

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!

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.

Beyond Trees

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

Acknowledgements

The Harmony project was begun in collaboration with Zhe Yang; Zhe contributed numerous insights whose generic material can be found (generally in much-recombined form) in this paper. Owen Gunden and, more recently, Malo Denielou have also collaborated with us on many aspects of the Harmony design and implementation; in particular, Malo's compiler and programming environment for the combinators described in this paper have contributed enormously. Trevor Jim provided the initial push to start the project by observing that the next step beyond the Unison file synchronizer (of which Trevor was a co-designer) would be synchronizing XML. Conversations with Martin Hofmann, Zack Ives, Nitin Khandelwal, Sanjeev Khanna, William Lovas, Kate Moore, Cyrus Najmabadi, Penny Anderson, and Steve Zdancewic helped us sharpen our ideas. Serge Abiteboul, Zack Ives, Dan Suciu, and Phil Wadler pointed us to related work. We would also like to thank Karthik Bhargavan, Vanessa Braganholo, Peter Buneman, Malo Denielou, Owen Gunden, Michael Hicks, Zack Ives, Trevor Jim, Kate Moore, Wang-Chiew Tan, Stephen Tse, and Zhe Yang, for very helpful comments on earlier drafts of this paper.

The Harmony project is supported by the National Science Foundation under grant ITR-0113226, *Principles and Practice of Synchronization*.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

[14] S. S. Cosmadakis. Translating updates of relational data base views. Master's thesis, Massachusetts Institute of Technology, 1983. MIT-LCS-TR-284.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[40] L. Rowe and K. A. Schoens. Data abstractions, views, and updates in RIGEL. In *SIGMOD'79*, 1979.

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

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

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

[44] W3C. XML Query, 2003. `http://www.w3.org/XML/Query`.

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

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