A Language for Bi-Directional Tree Transformations

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

Technical Report MS-CIS-03-08
Department of Computer and Information Science
University of Pennsylvania

August 5, 2003
Revised May 15, 2004

Abstract

We present a semantic foundation and a collection of program combinators for bi-directional transformations on tree-structured data. In one direction, these transformations map a "concrete" tree into a simplified "abstract" one; in the other, they map a modified abstract tree, together with the original concrete tree, to a correspondingly modified concrete tree. The challenge of understanding and designing these transformations—called *lenses*—arises from their asymmetric nature: information is discarded when mapping from concrete to abstract, and must be restored on the way back.

We identify a natural mathematical space of well-behaved lenses, in which the two components are constrained to fit together in a sensible way. We 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 our semantic framework in the form of a collection of *lens combinators* that can be assembled to describe powerful transformations on trees and whose well-behavedness follows from simple, local checks. These combinators include familiar constructs from functional programming (composition, mapping, projection, recursion) together with some novel primitives for manipulating trees (splitting, pruning, pivoting, etc.). 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 paper addresses a specific instance of "updating through a view" that arises in a larger project called Harmony [26]. Harmony is a generic framework for synchronizing tree-structured data—a tool for propagating updates between different copies, possibly stored in different formats, of tree-shaped data structures. 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 is to provide a platform on which a Harmony programmer can quickly assemble a high-quality synchronizer for a new type of tree-structured data stored in a standard low-level format such as XML. Other Harmony instances currently used daily or under development include synchronizers for calendars (Palm DateBook, ical, and iCalendar formats), address books, Keynote 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 as well as *lenses* that transform each concrete format into this abstract view. For example, we can synchronize a Mozilla bookmark file with an Explorer bookmark file by using appropriate lenses to transform each into an *abstract bookmark structure* and synchronizing the results. However, we are not done: we then need to take the updated abstract structures resulting from synchronization and transform them back into correspondingly updated concrete structures. To achieve this, a 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.

Naturally, the tricky aspects of constructing lenses arise in the *put* direction. If the *get* part of a lens is a projection—i.e., information is suppressed when moving from concrete to abstract—then the *put* part must restore this information in some appropriate way. (We will see a concrete example shortly.) The difficulty is that there may, in general, be many ways of doing so.

Our approach to this problem has been to design a set of program combinators—a small domain-specific language—with the property that every expression built from them simultaneously specifies both a *get* function and the corresponding *put*. All the atomic combinators denote lenses whose *get* and *put* functions fit together in a suitable sense, and all the combining forms (e.g., composition, which is parameterized on two lenses) preserve this property.

We begin by identifying a natural mathematical space of well-behaved lenses. There is bit of interesting work to do at this level, before we fix the domain of structures being transformed (trees) or the syntax for writing down transformations. First, we must phrase our basic definitions to allow lenses to be partial—i.e., to capture the fact that there may be structures to which a given lens cannot sensibly be applied. Second, we need some laws that express 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, then we should get back to the original c. Third, we must deal with the fact that we will later want to define lenses by recursion (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.). This raises familiar issues of monotonicity and continuity.

With these semantic foundations in place, we develop syntactic forms denoting lenses on the specific domain of trees. These include *atomic lenses* for basic tree transformations and lens *combinators* (composition, mapping, etc.) that allow complex lenses to be built up from simpler ones. From these basic constructs, we can build a rich variety of derived forms—e.g., lenses for manipulating list-structured data encoded as trees. The checks required to guarantee that expressions denote well-behaved lenses are simple and local—a kind of "type system for well-behavedness." In this paper, we perform these checks manually, leaving mechanical typechecking for future work.

We begin in Section 2 with a small example illustrating the fundamental ideas. Section 3 develops the semantic foundations of lenses in a general setting, addressing issues of partiality and continuity. Section 4 instantiates this generic framework with a set of combinators for our specific application domain of lenses over trees. Section 5 illustrates the use of the combinators in real-world lens programming by walking through a substantial example derived from the Harmony bookmark synchronizer. Section 6 surveys a variety of related work from both the programming languages and the database literature and states a precise correspondence (amplified in [25]) between our well-behaved lenses and the closely related idea of "update translation under a constant complement" from databases. Section 7 sketches some directions for future research.

2 A Small Example

Suppose our concrete data source c is a small address book, represented as the following tree:

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

We draw trees sideways to save space. Each set of hollow curly braces denotes a node, and each "$\texttt{X} \mapsto$ ..." inside denotes a child labeled `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 sometimes separate children with commas for easier reading.

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

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

Why would we want this? Perhaps because the edits are going to be performed by synchronizing this abstract tree with another replica of the same address book in which no URL information is recorded, or perhaps there is no synchronizer involved but the edits are going to be performed by a human who is only interested in phone information and whose screen should not be cluttered with URLs. Whatever the reason, we are going to make our changes to the abstract tree a, yielding a new abstract tree a' of the same form but with modified content.

For example, let us change `Pat`'s phone number, drop `Chris`, and add a new friend, `Jo`.

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

Note that we are only interested in the final tree a', not the actual sequence of edit operations that may have been used to transform a into a'. This design choice arises from the fact that synchronization in Harmony is based on the current states of the replicas, rather than on a trace of modifications (the tradeoffs between state-based and trace-based synchronizers are discussed in [27]).

Finally, 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`) should have their values carried over from c.

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

We also need to "fill in" appropriate values for the parts of c' (in particular, `Jo`'s `URL`) that were created in a' and for which c therefore contains no information. Here, we simply set the `URL` to a constant default, but in more complex situations 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 syntactic forms that can be combined to describe a wide variety of lenses in a concise, natural, and mathematically coherent manner.

3 Semantic Foundations

While our combinators are specialized for dealing with tree transformations, their semantic underpinnings can be presented in an abstract setting that is parameterized by the data structures ("views") manipulated by lenses.[1] In this section, we simply assume we are given some set $\mathcal{U}$ of views; in Section 4 we will choose $\mathcal{U}$ to be the set of trees.

3.1 Basic Structure

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 application to be strict, i.e., $f(g(x))\downarrow$ implies $g(x)\downarrow$.

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*.

We often say "put a into c" instead of "apply the *put* function to (a, c)." 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. (Abstract views, being smaller and lighter than concrete ones, naturally float upwards.)

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 (1) it maps arguments in C to results in A and vice versa—$l\nearrow(C) \subseteq A$ and $l\searrow(A\times C)\subseteq C$—and (2) its *get* and *put* functions obey the following laws:

$$\begin{array}{lll} l\searrow(l\nearrow c,\ c) \sqsubseteq c & \text{for all } c \in C & (\textsc{GetPut}) \\ l\nearrow(l\searrow(a,\ c)) \sqsubseteq a & \text{for all } (a,c) \in A\times C & (\textsc{PutGet}) \end{array}$$

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.

The GetPut law states that, if some abstract view obtained from a concrete view c is unmodified, putting it back into c must yield the same concrete view. PutGet states 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. Let $C = \texttt{string}\times\texttt{int}$ and $A = \texttt{string}$, and define l as:

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

[1]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.

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

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

PutGet fails in this case 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 are essential, reflecting 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. 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 6).

The PutPut law intuitively states that a series of changes to an abstract view may be applied incrementally or all at once, resulting in the same final concrete view in both cases. This is a natural and intuitive 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 4, we will not require PutPut because one of our most important lens combinators, `map`, fails to satisfy it for reasons that seem pragmatically unavoidable (see Section 4.2).

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.

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

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

We want lenses to be total: 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.) 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 us to invent global termination measures, in contrast to the purely local reasoning used to show well-behavedness. This is why we have formulated it as a separate condition rather than making it part of the definition of well-behavedness. We expect that, in practice, programmers (or, better yet, a type checker) will prove that all 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.

3.2 Basic Properties

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

Let f be a partial function from $A \times C$ to C. We will say that f is *injective* if it is injective in its first argument—i.e., if, for all views a, a', and c such that $f(a,c)\downarrow$ and $f(a',c)\downarrow$, we have $a \neq a' \implies f(a,c) \neq f(a',c)$.

We now state some properties well-behaved lenses have.

3.2.1 Lemma: Let l in $C \rightleftharpoons A$. Then the function $l\searrow$ is injective on the set $\{(a,c) \mid l\nearrow l\searrow(a,\, c)\downarrow \;\wedge\; (a,c) \in A \times C\}$.

Proof: Let D be the set $\{(a,c) \mid l\nearrow l\searrow(a,\, c)\downarrow \;\wedge\; (a,c) \in A \times C\}$, $(a,c) \in D$, and $(a',c) \in D$, such that $a' \neq a$. We prove by contradiction that $l\searrow(a,\, c) \neq l\searrow(a',\, c)$. Assume that $l\searrow(a,\, c) = l\searrow(a',\, c)$, then by definition of D and by rule PutGet, we have $a = l\nearrow l\searrow(a,\, c) = l\nearrow l\searrow(a',\, c) = a'$, hence $a = a'$, a contradiction. □

The following corollary 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.

3.2.2 Corollary: Let l in $C \Longleftrightarrow A$, then the function $l\searrow$ is injective on $A \times C$.

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 collection of bookmark items), we will need to define lenses by recursion. Our final task for 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, giving us interpretations of a variety of common syntactic forms from programming languages—in particular, functional abstraction and application (i.e., "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: Let l and l' be two lenses. We say that $l \prec l'$ 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 $\bigsqcup_{n\in\omega} l_n$ 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, this 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

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

and undefined everywhere else is a least upper bound for the chain.

Proof: We first prove 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, we have $\mathsf{dom}(l\nearrow) = \bigcup_i \mathsf{dom}(l_i\nearrow)$ and $\mathsf{dom}(l\searrow) = \bigcup_i \mathsf{dom}(l_i\searrow)$.

We now prove that l is a least upper bound. By definition, it is an upper bound. 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, let $c \in \mathsf{dom}(l\nearrow)$, 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, hence we have $l \prec l'$, so l is 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:

GET $(\bigsqcup_{n\in\omega} l_n) \nearrow c = v \iff \exists i.\; l_i \nearrow c = v$

PUT $(\bigsqcup_{n\in\omega} l_n) \searrow (a,\, c) = v \iff \exists i.\; l_i \searrow (a,\, c) = v$

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 $C_0 \subseteq C_1 \subseteq \cdots \subseteq C_n \subseteq \ldots$ and $A_0 \subseteq A_1 \subseteq \cdots \subseteq A_n \subseteq \ldots$ be two increasing chains of subsets of $\mathcal{U}$. We have:

1. Well-behavedness commutes with limits: $(\forall i \in \omega.\; l_i \in C_i \rightleftharpoons A_i) \implies (\bigsqcup_{n\in\omega} l_n) \in (\bigcup_i C_i) \rightleftharpoons (\bigcup_i A_i)$;
2. Totality commutes with limits: $(\forall i \in \omega.\; l_i \in C_i \Longleftrightarrow A_i) \implies (\bigsqcup_{n\in\omega} l_n) \in (\bigcup_i C_i) \Longleftrightarrow (\bigcup_i A_i)$.

Proof: In the following we write l for $\bigsqcup_{n\in\omega} l_n$, C for $\bigcup_i C_i$, and A for $\bigcup_i A_i$.

We rely on the following property $(\star_g)$: 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 $\prec$) $l_i \searrow (a,\, c) = l_{j''} \searrow (a,\, c) = l \searrow (a,\, c)$, $a \in A_i$, and $c \in C_i$.

We first prove that l satisfies the typing condition of well-behaved lenses. Let $c \in C$, then 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 $l_i \nearrow c \in A_i \subseteq A$. 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 is in $A_i \rightleftharpoons C_i$, we have $l_i \searrow (a,\, c) \in C_i \subseteq C$.

We now prove that $\bigsqcup_{n\in\omega} l_n$ satisfies GETPUT and PUTGET.

Using $\star_g$ and $\star_p$, we now 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.

We now prove totality of l if all the l_i are total. 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 $\perp_l$. 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 in the next section, we will make heavy use of the following standard theorem from domain theory (e.g., [33]). 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 $f(\bigsqcup_{n\in w} l_n) = \bigsqcup_{n\in\omega} f(l_n)$. 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

$$\mathit{fix}(f) = \bigsqcup_{n\in\omega} f^n(\perp_l)$$

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. More generally, we can 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: $(f, g) = \mathit{fix}(\lambda(x, y).(F(x, y), G(x, y)))$.

4 A Language for Transforming Trees

We now describe our combinators for tree transformations. We first introduce some notations for trees and operations on them. We then present a number of atomic lenses and lens combinators, which we assemble to create several derived lenses. We finally describe an encoding of lists as trees and introduce some specialized derived lenses for manipulating them. We give small examples along the way; an extended example using most of the lenses together appears in Section 5. The (open) problem of characterizing the formal expressive power of these combinators is discussed in Section 7.

4.1 Trees

From now on, we will be working with 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. (Trees of this sort are sometimes called *feature trees*—e.g., [22].) Each tree can be viewed as a partial function from names to other trees. We write $\mathsf{dom}(t)$ for the domain of a tree t—i.e., the set of the names of its immediate children. 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. We write $t(n)$ for the tree associated to name n in t.

We sometimes define trees by extension. For instance, let t be a tree and p be a set of names such that $p \subseteq \mathsf{dom}(t)$; we may define a tree w as $w = \{\!|n \mapsto t(n) \mid n \in p|\!\}$.

When p is a set of names, we write $\overline{p}$ for $\mathcal{N} \setminus p$, 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 similarly $t|_{\overline{p}}$ for $\{\!|n \mapsto t(n) \mid n \in \mathsf{dom}(t) \setminus p|\!\}$.

We will also need a notation for *merging* trees. When t and t' have disjoint domains, we write $t + t'$ or $\{\!|t\ t'|\!\}$ (the latter especially in displays) for the tree mapping n to $t(n)$ for $n \in \mathsf{dom}(t)$ and to $t'(n)$ for $n \in \mathsf{dom}(t')$.

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

There are cases where we need to apply a *put* function, but where no old concrete tree is available (as we saw with `Jo`'s `URL` in Section 2). To deal with these cases, we adjoin to the set of trees a special placeholder Ω, pronounced "missing." Intuitively, $l \searrow (a,\, \Omega)$ means "create a *new* concrete tree from the information in the abstract tree a." We write $\mathcal{T}_\Omega$ for $\mathcal{T} \cup \{\Omega\}$ and take the universe $\mathcal{U}$ in the definition of lenses to be $\mathcal{T}_\Omega$. By convention, Ω is only used in an interesting way when it is the second argument to the *put* function: in all of the lenses defined below, we maintain the invariants that (1) $l \nearrow \Omega = \Omega$, (2) $l \searrow (\Omega,\, c) = \Omega$ for any c, (3) $l \nearrow c \neq \Omega$ for any $c \neq \Omega$, and (4) $l \searrow (a,\, c) \neq \Omega$ for any $a \neq \Omega$ and any c (including Ω). We write $C \overset{\Omega}{\rightleftharpoons} A$ for the set of lenses from C to A obeying these conventions. (There are other, formally equivalent, ways of handling missing concrete trees. The advantages of this one are discussed below, after the definition of the `map` combinator.) For brevity, in the lens definitions below, we 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. To shorten some definitions below, we adopt the conventions that $\mathsf{dom}(\Omega) = \emptyset$, and that $\Omega|_p = \Omega$ for any p.

The Harmony system targets data stored in XML (among other formats). However, the form of trees that we use internally is much simpler than XML, which associates each node with both unordered children (attributes) and ordered ones (sub-elements). We show in Section 5 how XML trees can be encoded into ours. We chose unordered trees for engineering reasons: experience has shown that the resulting reduction in the complexity of the lens *definitions* far outweighs the modest increase in complexity of lens *programs* due to manipulating XML via this encoding instead of primitively.

Each lens primitive defined below will be accompanied by a type declaration asserting its well-behavedness under certain conditions (e.g., "the identity lens belongs to $C \overset{\Omega}{\rightleftharpoons} C$ for any C"). These declarations are not quite a type *system*, in the sense that we do not have an algorithm that will check the conditions automatically (yet—see Section 7), but they are close to one in that the required checks are local and generally easy. For writing down type declarations, it is convenient to extend some of the tree notations to sets of trees. If $T \subseteq \mathcal{T}$, then $\mathsf{dom}(T) = \{\mathsf{dom}(t) \mid t \in T\}$. If $T \subseteq \mathcal{T}$ and $n \in \mathcal{N}$ is a name, then $\{\!|n \mapsto T|\!\}$ denotes the set of trees $\{\{\!|n \mapsto t|\!\} \mid t \in T\}$. If $K \subseteq \mathcal{N}$ is a set of names, then and $\{\!|n \mapsto K|\!\}$ means $\{\{\!|n \mapsto k|\!\} \mid k \in K\}$—i.e., $\{\{\!|n \mapsto \{\!|k \mapsto \{\!||\!\}|\!\}|\!\} \mid k \in K\}$. We write $T_1 + T_2$ for $\{t_1 + t_2 \mid t_1 \in T_1,\, t_2 \in T_2\}$ and $T(n)$ for $\{t(n) \mid t \in T \wedge n \in \mathsf{dom}(t)\}$.

4.2 Primitives

In this section we define several atomic lenses and lens combinators (we will often just say "lenses" for both). We begin with a few generic lenses that do not depend on the universe being trees: the identity lens, the constant lens, and sequential composition of lenses. We then introduce several lenses that inspect and manipulate tree structures—four atomic lenses (`rename`, `hoist`, `plunge`, and `pivot`) and two lens combinators (`xfork` and `map`).

All lenses introduced in this section, with the exception of the `const` lens, preserve all information when building the abstract tree in the *get* direction. The two lens combinators also preserve all information when applied to information-preserving lenses. Most lenses thus do not need to use the concrete tree in the *put* direction.

We show that every atomic lens is well behaved, and every lens combinator is continuous and preserves well-behavedness. (Indeed, we conjecture that the primitive lenses are all *very* well behaved and that almost all the lens combinators preserve very-well-behavedness, the single exception being `map`.)

Identity

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

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

Having defined `id`, we must now prove that it is well behaved—i.e., that the type declaration at the bottom of the box is a theorem. Since we will need a similar argument for every lens we define, some shorthand will be useful. First, we use the labels GET and PUT to refer to the conditions in Definition 3.1.2 on the types of $l\nearrow$ and $l\searrow$—i.e., that $l\nearrow$ maps arguments in C_Ω to results in A_Ω and that $l\searrow$ maps arguments in $A_\Omega \times C_\Omega$ to results in C_Ω. Second, by our conventions on the treatment of Ω, the GET condition 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 $c \in C_\Omega$.

4.2.1 Lemma: $\forall C\subseteq\mathcal{T}.\ \texttt{id} \in C \overset{\Omega}{\rightleftharpoons} C$.

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: $\texttt{id}\nearrow\texttt{id}\searrow(a,\, c) = \texttt{id}\nearrow a = a$. □

We now prove that `id` is total. In totality proofs, we do not consider the case $l\searrow(\Omega,\, c)$ as this case is always defined (and equal to Ω).

4.2.2 Lemma: `id` is total.

Proof: Immediate: `id` is defined everywhere. □

Constant

Another simple lens is the constant lens, $\texttt{const}\ t\ d$, which transforms any tree into the provided constant t in the *get* direction. In the *put* direction, it is defined iff the abstract tree is equal to t.[2] In this case, the *put* function of `const` simply restores the old concrete tree if it is available; if the old concrete tree is Ω, the *put* function returns a default tree d.

$$\boxed{\begin{array}{rcll} (\texttt{const}\ t\ d)\nearrow c & = & t & \\ (\texttt{const}\ t\ d)\searrow(a,\, c) & = & c & \text{if } c \neq \Omega \\ & & d & \text{if } c = \Omega \\ \hline \forall C\subseteq\mathcal{T}.\ \forall t\in\mathcal{T}.\ \forall d\in C. & & \multicolumn{2}{l}{\texttt{const}\ t\ d \in C \overset{\Omega}{\rightleftharpoons} \{t\}} \end{array}}$$

4.2.3 Lemma: $\forall C\subseteq\mathcal{T}.\ \forall t\in\mathcal{T}.\ \forall d\in C.\ \texttt{const}\ t\ d \in C \overset{\Omega}{\rightleftharpoons} \{t\}$.

Proof:

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

[2]By the PUTGET law, this is the only possible definition: if $\texttt{const}\ t\ d\nearrow(\texttt{const}\ t\ d\searrow(a,\, c))\ \downarrow$, then $\texttt{const}\ t\ d\nearrow(\texttt{const}\ t\ d\searrow(a,\, c)) = a = t$.

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

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

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

4.2.4 Lemma: $(\texttt{const}\ t\ d)$ is total.

Proof: $\mathsf{dom}((\texttt{const}\ t\ d)\nearrow) = \mathcal{T}_\Omega$ and $\mathsf{dom}((\texttt{const}\ t\ d)\searrow) = \{t\} \times \mathcal{T}_\Omega$. □

Composition

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

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

The *get* direction applies the *get* function of l to yield a first abstract tree, 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 tree 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 tree 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.)

4.2.5 Lemma: $\forall A, B, C \subseteq \mathcal{T}.\ \forall l \in C \stackrel{\Omega}{\rightleftharpoons} B.\ \forall k \in B \stackrel{\Omega}{\rightleftharpoons} A.\ \ l;k \in C \stackrel{\Omega}{\rightleftharpoons} A$.

Proof:

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

PUT: If $l \searrow (k \searrow (a,\ l \nearrow c),\ c) = (l;k) \searrow (a,\ c)$ defined, then $l \nearrow c \in B_\Omega$ by GET for l and the 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:

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

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

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

□

Besides well-behavedness, we must also show that lens composition preserves continuity. 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. 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—i.e., the continuity of every lens expression will follow from the continuity of its immediate constituents.

4.2.6 Lemma: Let F and G be two 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 $c \in \mathcal{T}$, 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 $(a, c) \in \mathcal{T} \times \mathcal{T}_\Omega$, 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 $c \in \mathcal{T}$,

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

and

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

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

Finally, to support compositional reasoning in proofs of lens totality, we must check that composition preserves totality.

4.2.7 Lemma: If l is total from C to B and k total from B to A, then $(l;k)$ is total from C to 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. □

Rename

The `rename` lens changes the names of the immediate children of a tree according to some bijection b on names. In the type, we write $b \in \mathit{Bij}(\mathcal{N})$ to mean that b is a bijective function on names and write $b(C)$ for the set of trees formed by taking each tree in C and renaming its top-level children according to b.

$$\boxed{\begin{array}{rcl}
(\texttt{rename}\ b)\nearrow c &=& \{\!|b(n)\mapsto c(n)|\!\}\\
(\texttt{rename}\ b)\searrow(a,\,c) &=& \{\!|b^{-1}(n)\mapsto a(n)|\!\}\\
\hline
\multicolumn{3}{c}{\forall C\subseteq\mathcal{T}.\ \forall b\in \mathit{Bij}(\mathcal{N}).\quad \texttt{rename}\ b\in C \overset{\Omega}{\rightleftharpoons} b(C)}
\end{array}}$$

4.2.8 Lemma: $\forall C\subseteq\mathcal{T}.\ \forall b\in \mathit{Bij}(\mathcal{N}).\ \texttt{rename}\ b\in C \overset{\Omega}{\rightleftharpoons} b(C)$.

Proof:

Get: $(\texttt{rename}\ b)\nearrow c \in b(C)$

Put: $(\texttt{rename}\ b)\searrow(a,\,c) \in b^{-1}(b(C)) = C$

GetPut: $(\texttt{rename}\ b)\searrow((\texttt{rename}\ b)\nearrow c,\, c) = (\texttt{rename}\ b)\searrow\left(\{\!|b(n)\mapsto c(n)|\!\},\, c\right) = \{\!|b^{-1}(b(n))\mapsto c(n)|\!\} = \{\!|n\mapsto c(n)|\!\} = c.$

PutGet: $(\texttt{rename}\ b)\nearrow(\texttt{rename}\ b)\searrow(a,\,c) = (\texttt{rename}\ b)\nearrow \{\!|b^{-1}(n)\mapsto a(n)|\!\} = \{\!|b(b^{-1}(n))\mapsto a(n)|\!\} = \{\!|n\mapsto a(n)|\!\} = a.$ □

4.2.9 Lemma: $(\texttt{rename}\ b)$ is total.

Proof: Immediate, as (`rename` b) is defined everywhere. $\square$

In examples, we use the notation `{h3 = name, dl = contents}` for the bijection that maps `h3` to `name`, `name` to `h3`, `dl` to `contents`, and `contents` to `dl`.

In practice, we also sometimes use a "deep rename" lens that changes all the names in a tree, rather than just the immediate children of the root; this can be defined from `rename` using `map` and recursion.

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 concrete tree is ignored and a new tree is created, with a single edge n pointing to the given abstract tree.

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

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

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

4.2.11 Lemma: (`hoist` n) is total.

Proof: Immediate. $\square$

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 abstract tree. In the *put* direction, the value of the concrete tree is ignored and the abstract tree is required to have exactly one child, named n. It returns this child, removing the edge n.

$$\boxed{\begin{array}{rcl} (\texttt{plunge}\ n)\nearrow c & = & \{\!|n \mapsto c|\!\} \\ (\texttt{plunge}\ n)\searrow (a,\, c) & = & t \qquad \text{if } a = \{\!|n \mapsto t|\!\} \\ \hline \multicolumn{3}{c}{\forall C\subseteq\mathcal{T}.\ \forall n\in\mathcal{N}. \quad \texttt{plunge}\ n \in C \overset{\Omega}{\rightleftharpoons} \{\!|n \mapsto C|\!\}} \end{array}}$$

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

Proof:

Get: $(\texttt{plunge}\ n)\nearrow c = \{\!|n \mapsto c|\!\} \in \{\!|n \mapsto C|\!\}.$

Put: $(\texttt{plunge}\ n)\searrow \big(\{\!|n \mapsto t|\!\}\,,\, c\big) = t \in C.$

GetPut: $(\texttt{plunge}\ n)\searrow ((\texttt{plunge}\ n)\nearrow c,\, c) = (\texttt{plunge}\ n)\searrow \big(\{\!|n \mapsto c|\!\}\,,\, c\big) = c.$

PutGet: $(\texttt{plunge}\ n)\nearrow ((\texttt{plunge}\ n)\searrow \big(\{\!|n \mapsto t|\!\}\,,\, c\big)) = (\texttt{plunge}\ n)\nearrow t = \{\!|n \mapsto t|\!\}.$ $\square$

4.2.13 Lemma: (`plunge` n) is total.

Proof: Immediate. $\square$

Pivot

The lens `pivot` n rearranges the structure at the top of a tree, transforming

$$\left\{\!\!\left|\begin{array}{l} n \mapsto k \\ t \end{array}\right|\!\!\right\} \quad \text{to} \quad \{\!|k \mapsto t|\!\}\,.$$

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.

$$\boxed{\begin{array}{c}\begin{array}{rcl} (\texttt{pivot}\ n) \nearrow c & = & \{\!|k \mapsto t|\!\} \quad \text{if } c = \left\{\!\!\left|\begin{array}{l} n \mapsto k \\ t \end{array}\right|\!\!\right\} \\ (\texttt{pivot}\ n) \searrow (a,\, c) & = & \left\{\!\!\left|\begin{array}{l} n \mapsto k \\ t \end{array}\right|\!\!\right\} \quad \text{if } a = \{\!|k \mapsto t|\!\} \end{array} \\ \hline \forall n{\in}\mathcal{N}.\ \forall K{\subseteq}\mathcal{N}.\ \forall C{\subseteq}(\mathcal{T}|_{\overline{n}}). \qquad \texttt{pivot}\ n \in (\{\!|n \mapsto K|\!\} + C) \overset{\Omega}{\rightleftharpoons} \{\!|K \mapsto C|\!\} \end{array}}$$

4.2.14 Lemma: $\forall n{\in}\mathcal{N}.\ \forall K{\subseteq}\mathcal{N}.\ \forall C{\subseteq}(\mathcal{T}|_{\overline{n}}).\ \texttt{pivot}\ n \in (\{\!|n \mapsto K|\!\} + C) \overset{\Omega}{\rightleftharpoons} \{\!|K \mapsto C|\!\}$.

Proof:

Get: $(\texttt{pivot}\ n) \nearrow \left\{\!\!\left|\begin{array}{l} n \mapsto k \\ t \end{array}\right|\!\!\right\} = \{\!|k \mapsto t|\!\} \in \{\!|K \mapsto C|\!\}$

Put: $(\texttt{pivot}\ n) \searrow \big(\{\!|k \mapsto t|\!\}\,,\, c\big) = \left(\left\{\!\!\left|\begin{array}{l} n \mapsto k \\ t \end{array}\right|\!\!\right\}\right) \in (\{\!|n \mapsto K|\!\} + C)$

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

$$\begin{array}{rl} & (\texttt{pivot}\ n) \searrow \left((\texttt{pivot}\ n) \nearrow \left\{\!\!\left|\begin{array}{l} n \mapsto k \\ t \end{array}\right|\!\!\right\},\ \left\{\!\!\left|\begin{array}{l} n \mapsto k \\ t \end{array}\right|\!\!\right\}\right) \\ = & (\texttt{pivot}\ n) \searrow \left(\{\!|k \mapsto t|\!\}\,,\ \left\{\!\!\left|\begin{array}{l} n \mapsto k \\ t \end{array}\right|\!\!\right\}\right) \\ = & \left\{\!\!\left|\begin{array}{l} n \mapsto k \\ t \end{array}\right|\!\!\right\} \end{array}$$

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

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

4.2.15 Lemma: $(\texttt{pivot}\ n)$ is total.

Proof: Because $\mathsf{dom}((\texttt{pivot}\ n)\nearrow) = \bigcup_{K\in\mathcal{N}}\{(\{\!|n \mapsto K|\!\} + \mathcal{T}|_{\overline{n}})_\Omega\}$ and $\mathsf{dom}((\texttt{pivot}\ n)\searrow) = \bigcup_{K\in\mathcal{N}} \{\!|K \mapsto \mathcal{T}|_{\overline{n}}|\!\} \times \mathcal{T}_\Omega$. $\square$

Xfork

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 predicates on 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

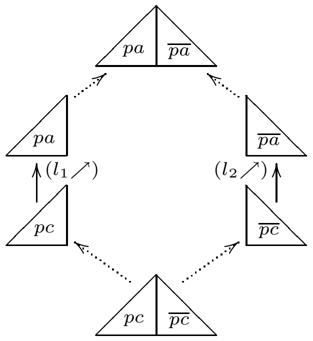


Figure 1: The *get* direction of `xfork`

pc denote trees whose immediate child edges have labels satisfying *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 do not satisfy *pc*) must be a tree whose top-level labels satisfy the predicate *pa*, and, similarly the result of applying $l_2\nearrow$ to $c|_{\overline{pc}}$ must satisfy $\overline{pa}$. 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 $\overline{pc}$. Formally:

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

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

Proof:

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

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

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

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

Thus,

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

by GETPUT for l_1. Similarly,

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

by GETPUT for l_2. Thus,

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

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

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

and

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

By PUTGET for l_1,

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

and by PUTGET for l_2,

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

Thus,

$$\begin{array}{rl} & (\texttt{xfork}\ pc\ pa\ l_1\ l_2) \nearrow (l_1 \searrow (a|_{pa},\ c|_{pc}) + l_2 \searrow (a|_{\overline{pa}},\ c|_{\overline{pc}})) \\ = & \big(l_1 \nearrow (l_1 \searrow (a|_{pa},\ c|_{pc}) + l_2 \searrow (a|_{\overline{pa}},\ c|_{\overline{pc}}))|_{pc}\big) + \big(l_2 \nearrow (l_1 \searrow (a|_{pa},\ c|_{pc}) + l_2 \searrow (a|_{\overline{pa}},\ c|_{\overline{pc}}))|_{\overline{pc}}\big) \\ \sqsubseteq & a|_{pa} + a|_{\overline{pa}} \\ = & a. \end{array}$$

$\square$

4.2.17 Lemma: 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: Let l and l' be two lenses with $l \prec l'$. We must first 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

$$\begin{array}{rll} & (\texttt{xfork}\ pc\ pa\ F(l)\ G(l)) \nearrow c & \\ = & (F(l) \nearrow c|_{pc}) + (G(l) \nearrow c|_{\overline{pc}}) & \\ = & (F(l') \nearrow c|_{pc}) + (G(l') \nearrow c|_{\overline{pc}}) & \text{by } F(l) \prec F(l') \text{ and } G(l) \prec G(l') \\ = & (\texttt{xfork}\ pc\ pa\ F(l')\ G(l')) \nearrow c. & \end{array}$$

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:

$$\begin{array}{rll} & (\texttt{xfork}\ pc\ pa\ F(l)\ G(l)) \searrow (a,\ c) & \\ = & (F(l) \searrow (a|_{pa},\ c|_{pc})) + (G(l) \searrow (a|_{\overline{pa}},\ c|_{\overline{pc}})) & \\ = & (F(l') \searrow (a|_{pa},\ c|_{pc})) + (G(l') \searrow (a|_{\overline{pa}},\ c|_{\overline{pc}})) & \text{by } F(l) \prec F(l') \text{ and } G(l) \prec G(l') \\ = & (\texttt{xfork}\ pc\ pa\ F(l')\ G(l')) \searrow (a,\ c). & \end{array}$$

Thus $\lambda l.\ \texttt{xfork}\ pc\ pa\ F(l)\ G(l)$ is monotonic. We now 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. Let $l = \bigsqcup_i l_i$. We have:

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

and

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

□

4.2.18 Lemma: If l_1 is a total lens from $C_1 \subseteq \mathcal{T}|_{pc}$ to $A_1 \subseteq \mathcal{T}|_{pa}$ and l_2 is a total lens from $C_2 \subseteq \mathcal{T}|_{\overline{pc}}$ to $A_2 \subseteq \mathcal{T}|_{\overline{pa}}$, then $(\texttt{xfork}\ pc\ pa\ l_1\ l_2)$ is total from $C_1 + C_2$ to $A_1 + A_2$.

Proof: Let $c \in C_1 + C_2$, then we have $c|_{pc} \in C_1$, and $c|_{\overline{pc}} \in C_2$. By totality for l_1 and l_2, $l_1 \nearrow c|_{pc}$ is defined and is in A_1, and $l_2 \nearrow c|_{\overline{pc}}$ is defined and is in A_2. As these two views have disjoint domains, $l_1 \nearrow c|_{pc} + l_2 \nearrow c|_{\overline{pc}} = (\texttt{xfork}\ pc\ pa\ l_1\ l_2) \nearrow c$ is defined.

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

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

Hence we have:

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

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

Map

Our most complex lens combinator, `map`, is parameterized on a total function m from names to lenses. In the *get* direction, `map` applies the lens $m(n)$ to each child n, at one level deeper in the tree, leaving the top of the tree intact. Concretely, when we write a lens using `map`, we typically describe only part of the function and adopt the convention that the function gives `id` at every other name. For example, the lens `map` $\{n_1 \mapsto l_1,\ n_2 \mapsto l_2\}$ has the following behavior in the *get* direction:

$$\left\{\begin{vmatrix} n_1 \mapsto t_1 \\ n_2 \mapsto t_2 \\ n_3 \mapsto t_3 \end{vmatrix}\right\} \quad \text{becomes} \quad \left\{\begin{vmatrix} n_1 \mapsto l_1 \nearrow t_1 \\ n_2 \mapsto l_2 \nearrow t_2 \\ n_3 \mapsto id \nearrow t_3 \end{vmatrix}\right\}$$

The *put* direction of `map` is more interesting. In the simple case where a and c have equal domains, its behavior is straightforward:

$$(\texttt{map}\ \{n_1 \mapsto l_1,\ n_2 \mapsto l_2\} l) \searrow \left(\left\{\begin{vmatrix} n_1 \mapsto t_1 \\ n_2 \mapsto t_2 \\ n_3 \mapsto t_3 \end{vmatrix}\right\}, \left\{\begin{vmatrix} n_1 \mapsto t_1' \\ n_1 \mapsto t_2' \\ n_3 \mapsto t_3' \end{vmatrix}\right\} \right) = \left\{\begin{vmatrix} n_1 \mapsto l_1 \searrow (t_1,\, t_1') \\ n_1 \mapsto l_2 \searrow (t_1,\, t_1') \\ n_k \mapsto id \searrow (t_3,\, t_3') \end{vmatrix}\right\}$$

The general case is a bit more involved. If $(\texttt{map}\ m) \searrow (a,\, c)$ is defined, then, by rule PutGet, we should have $(\texttt{map}\ m) \nearrow ((\texttt{map}\ m) \searrow (a,\, c)) \sqsubseteq a$. Thus we necessarily have $\mathsf{dom}((\texttt{map}\ m) \searrow (a,\, c)) = \mathsf{dom}(a)$ if it is defined. Children bearing names that occur both in $\mathsf{dom}(a)$ and $\mathsf{dom}(c)$ are dealt with as described above. Children that only occur in $\mathsf{dom}(c)$ are dropped. Children that only appear in $\mathsf{dom}(a)$ need to be passed through $m(l)$ so that they can be included in the result; to do this, we need to *put* them into some concrete tree. There is no corresponding child in c, so instead these abstract trees are put into the missing tree Ω.

The typing of `map` is a little subtle. In the *get* direction, `map` does not modify the names of the immediate children of the concrete tree. We might therefore expect that if, for each $n \in \mathcal{N}$, we have $m(n) \in C(n) \overset{\Omega}{\rightleftharpoons} A(n)$, then `map` $m \in C \overset{\Omega}{\rightleftharpoons} A$. Unfortunately, this is not the case, as the following example shows. Consider the set of trees

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

and the lens `map` $\{\mathtt{x} \mapsto (\mathtt{rename}\ \{\mathtt{m} = \mathtt{p}\})\}$. By the reasoning above, we might expect this lens to have type $S \overset{\Omega}{\rightleftharpoons} S$ because $\mathtt{rename}\ \{\mathtt{m} = \mathtt{p}\} \in S(\mathtt{x}) \overset{\Omega}{\rightleftharpoons} S(\mathtt{x})$ and $\mathtt{id} \in S(\mathtt{y}) \overset{\Omega}{\rightleftharpoons} S(\mathtt{y})$. Yet when we apply the *get* direction of the lens to a tree in S

$$\mathtt{map}\ \{\mathtt{x} \mapsto (\mathtt{rename}\ \{\mathtt{m} = \mathtt{p}\})\} \nearrow \{\!|\mathtt{x} \mapsto \mathtt{m},\ \mathtt{y} \mapsto \mathtt{n}|\!\} = \{\!|\mathtt{x} \mapsto \mathtt{p}, \mathtt{y} \mapsto \mathtt{n}|\!\}$$

we get a tree that is not in S. To remedy this problem (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 sets of concrete and abstract trees 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 of domain d whose children under n are taken from the set $T(n)$. We say that T is *shuffle closed* iff $T = T^{\circlearrowleft}$, and we write $\mathcal{T}^{\circlearrowleft}$ for the set of all shuffle-closed sets of trees.

We are now ready to state the definition of `map` m and its type:

$$\boxed{\begin{array}{rcl} (\mathtt{map}\ \mathrm{m}) \nearrow c & = & \left\{\!\left| n \mapsto m(n) \nearrow c(n) \mid n \in \mathsf{dom}(c) \right|\!\right\} \\ (\mathtt{map}\ \mathrm{m}) \searrow (a,\ c) & = & \left\{\!\left| \begin{array}{ll} n \mapsto m(n) \searrow (a(n),\ c(n)) & \mid n \in \mathsf{dom}(a) \cap \mathsf{dom}(c) \\ n \mapsto m(n) \searrow (a(n),\ \Omega) & \mid n \in \mathsf{dom}(a) \setminus \mathsf{dom}(c) \end{array} \right|\!\right\} \\ \hline \multicolumn{3}{l}{\forall C, A \subseteq \mathcal{T}^{\circlearrowleft}\ \mathit{with}\ \mathsf{dom}(C) = \mathsf{dom}(A).\ \forall m \in (\Pi n.\ C(n) \overset{\Omega}{\rightleftharpoons} A(n)) \qquad \mathtt{map}\ m \in C \overset{\Omega}{\rightleftharpoons} A} \end{array}}$$

In the type annotation, we use the dependent type notation $\Pi n.\ C(n) \overset{\Omega}{\rightleftharpoons} A(n)$ to mean that m is a total function mapping each name n to a well-behaved lens from $C(n)$ to $A(n)$.

4.2.19 Lemma: $\forall C, A \subseteq \mathcal{T}^{\circlearrowleft}$ *with* $\mathsf{dom}(C) = \mathsf{dom}(A)$. $\forall m \in (\Pi n.\ C(n) \overset{\Omega}{\rightleftharpoons} A(n))$ `map` $m \in C \overset{\Omega}{\rightleftharpoons} A$.

Proof:

Get: Let $c \in C$ such that $m(n) \nearrow c(n)$ is defined. For each $n \in \mathsf{dom}(c)$, we have $m(n) \nearrow c(n) \in A(n)$; hence, as $A = A^{\circlearrowleft}$, we have $(\mathtt{map}\ \mathrm{m}) \nearrow c \in A$.

Put: Let $a \in A$ and $c \in C$. For all $n \in \mathsf{dom}(a) \cap \mathsf{dom}(c)$, we have $m(n) \searrow (a(n),\ c(n)) \in C(n)$. Moreover, for all $n \in \mathsf{dom}(a) \setminus \mathsf{dom}(c)$, we have $m(n) \searrow (a(n),\ \Omega) \in C(n)$. Hence, as $C = C^{\circlearrowleft}$ and $\mathsf{dom}(A) = \mathsf{dom}(C)$, we have $m(n) \searrow (a,\ c) \in C$.

GetPut: Assume that $(\mathtt{map}\ \mathrm{m}) \nearrow c$ is defined. Then

$$\begin{array}{cll} & (\mathtt{map}\ \mathrm{m}) \searrow ((\mathtt{map}\ \mathrm{m}) \nearrow c,\ c) & \\ = & (\mathtt{map}\ \mathrm{m}) \searrow \left(\left\{\!\left| n \mapsto m(n) \nearrow c(n) \mid n \in \mathsf{dom}(c) \right|\!\right\},\ c\right) & \\ = & \left\{\!\left| n \mapsto m(n) \searrow (m(n) \nearrow c(n),\ c(n)) \mid n \in \mathsf{dom}(c) \right|\!\right\} & \\ \sqsubseteq & \left\{\!\left| n \mapsto c(n) \mid n \in \mathsf{dom}(c) \right|\!\right\} & \text{by GetPut for each } m(n) \\ = & c. & \end{array}$$

PutGet: Assume that $(\texttt{map}\ \mathrm{m})\searrow(a,\ c)$ is defined. Then

$$\begin{array}{ll}
 & (\texttt{map}\ \mathrm{m})\nearrow(\texttt{map}\ \mathrm{m})\searrow(a,\ c)\\
= & (\texttt{map}\ \mathrm{m})\nearrow \left\{\!\!\left|\begin{array}{ll} n\mapsto m(n)\searrow(a(n),\ c(n)) & \mid n\in\mathsf{dom}(a)\cap\mathsf{dom}(c)\\ n\mapsto m(n)\searrow(a(n),\ \Omega) & \mid n\in\mathsf{dom}(a)\setminus\mathsf{dom}(c)\end{array}\right|\!\!\right\}\\
= & \left\{\!\!\left|\begin{array}{ll} n\mapsto m(n)\nearrow m(n)\searrow(a(n),\ c(n)) & \mid n\in\mathsf{dom}(a)\cap\mathsf{dom}(c)\\ n\mapsto m(n)\nearrow m(n)\searrow(a(n),\ \Omega) & \mid n\in\mathsf{dom}(a)\setminus\mathsf{dom}(c)\end{array}\right|\!\!\right\}\\
\sqsubseteq & \left\{\!\!\left|\begin{array}{ll} n\mapsto a(n) & \mid n\in\mathsf{dom}(a)\cap\mathsf{dom}(c)\\ n\mapsto a(n) & \mid n\in\mathsf{dom}(a)\setminus\mathsf{dom}(c)\end{array}\right|\!\!\right\} \qquad \text{by PutGet for each } m(n)\\
= & a.
\end{array}$$

□

4.2.20 Lemma: For each name n, let F_n be a continuous function from lenses to lenses. Then the function $\lambda l.\ \texttt{map}\ \{n\mapsto F_n(l)\}$ is continuous.

Proof: Let l and l' be two lenses with $l\prec l'$. We must show that $\texttt{map}\ \{n\mapsto F_n(l)\mid n\in\mathcal{N}\}\prec\texttt{map}\ \{n\mapsto F_n(l')\mid n\in\mathcal{N}\}$.

Let $c\in\mathcal{T}$ and suppose that $(\texttt{map}\ \{n\mapsto F_n(l)\mid n\in\mathcal{N}\})\nearrow c$ is defined. We have

$$\begin{array}{lll}
 & (\texttt{map}\ \{n\mapsto F_n(l)\mid n\in\mathcal{N}\})\nearrow c & \\
= & \{\!|n\mapsto F_n(l)\nearrow c(n)\mid n\in\mathsf{dom}(c)|\!\} & \\
= & \{\!|n\mapsto F_n(l')\nearrow c(n)\mid n\in\mathsf{dom}(c)|\!\} & \text{by } l\prec l'\\
= & (\texttt{map}\ \{n\mapsto F_n(l')\mid n\in\mathcal{N}\})\nearrow c. &
\end{array}$$

Conversely, let a and c be two trees in $\mathcal{T}\times\mathcal{T}_\Omega$ and suppose that $(\texttt{map}\ \{n\mapsto F_n(l)\mid n\in\mathcal{N}\})\searrow(a,\ c)$ is defined. Then

$$\begin{array}{ll}
 & (\texttt{map}\ \{n\mapsto F_n(l)\mid n\in\mathcal{N}\})\searrow(a,\ c)\\
= & \left\{\!\!\left|\begin{array}{ll} n\mapsto F_n(l)\searrow(a(n),\ c(n)) & \mid n\in\mathsf{dom}(a)\cap\mathsf{dom}(c)\\ n\mapsto F_n(l)\searrow(a(n),\ \Omega) & \mid n\in\mathsf{dom}(a)\setminus\mathsf{dom}(c)\end{array}\right|\!\!\right\}\\
= & \left\{\!\!\left|\begin{array}{ll} n\mapsto F_n(l')\searrow(a(n),\ c(n)) & \mid n\in\mathsf{dom}(a)\cap\mathsf{dom}(c)\\ n\mapsto F_n(l')\searrow(a(n),\ \Omega) & \mid n\in\mathsf{dom}(a)\setminus\mathsf{dom}(c)\end{array}\right|\!\!\right\} \quad \text{by } l\prec l'\\
= & (\texttt{map}\ \{n\mapsto F_n(l')\mid n\in\mathcal{N}\})\searrow(a,\ c).
\end{array}$$

Thus $\lambda l.\ \texttt{map}\ \{n\mapsto F_n(l)\mid n\in\mathcal{N}\}$ is monotonic. We now prove that it is continuous.

Let $l_0\prec l_1\prec\ldots\prec l_n\prec\ldots$ be an increasing chain of lenses with $l=\bigsqcup_i l_i$. Let $c\in\mathcal{T}$. We assume an 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

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

Let $(a,c)\in\mathcal{T}\times\mathcal{T}_\Omega$. We assume an ordering on the names of the children of a, and write $\texttt{f}(a)$ and $\texttt{l}(a)$

for the first and last names of a, respectively. We have

$$\begin{array}{lrll}
 & t = & (\texttt{map}\ \{n \mapsto F_n(l) \mid n \in \mathcal{N}\}) \searrow (a,\, c) & \\
\iff & t = & \left\{\!\!\left|\begin{array}{ll} n \mapsto F_n(l) \searrow (a(n),\, c(n)) & \mid n \in \mathsf{dom}(a) \cap \mathsf{dom}(c) \\ n \mapsto F_n(l) \searrow (a(n),\, \Omega) & \mid n \in \mathsf{dom}(a) \setminus \mathsf{dom}(c) \end{array}\right|\!\!\right\} & \\
\iff & t = & \left\{\!\!\left|\begin{array}{ll} n \mapsto F_n(\bigsqcup_i l_i) \searrow (a(n),\, c(n)) & \mid n \in \mathsf{dom}(a) \cap \mathsf{dom}(c) \\ n \mapsto F_n(\bigsqcup_i l_i) \searrow (a(n),\, \Omega) & \mid n \in \mathsf{dom}(a) \setminus \mathsf{dom}(c) \end{array}\right|\!\!\right\} & \\
\iff & t = & \left\{\!\!\left|\begin{array}{ll} n \mapsto (\bigsqcup_i F_n(l_i)) \searrow (a(n),\, c(n)) & \mid n \in \mathsf{dom}(a) \cap \mathsf{dom}(c) \\ n \mapsto (\bigsqcup_i F_n(l_i)) \searrow (a(n),\, \Omega) & \mid n \in \mathsf{dom}(a) \setminus \mathsf{dom}(c) \end{array}\right|\!\!\right\} & \text{by continuity of } F_n \\
\iff \exists i_{\texttt{f}(a)}, \ldots, i_{\texttt{l}(a)}. & t = & \left\{\!\!\left|\begin{array}{ll} n \mapsto (F_n(l_{i_n})) \searrow (a(n),\, c(n)) & \mid n \in \mathsf{dom}(a) \cap \mathsf{dom}(c) \\ n \mapsto (F_n(l_{i_n})) \searrow (a(n),\, \Omega) & \mid n \in \mathsf{dom}(a) \setminus \mathsf{dom}(c) \end{array}\right|\!\!\right\} & \begin{array}{l}\text{by 3.3.4 for PUT,} \\ |\mathsf{dom}(a)| \text{ times}\end{array} \\
\iff \exists i. & t = & \left\{\!\!\left|\begin{array}{ll} n \mapsto (F_n(l_i)) \searrow (a(n),\, c(n)) & \mid n \in \mathsf{dom}(a) \cap \mathsf{dom}(c) \\ n \mapsto (F_n(l_i)) \searrow (a(n),\, \Omega) & \mid n \in \mathsf{dom}(a) \setminus \mathsf{dom}(c) \end{array}\right|\!\!\right\} & \begin{array}{l}\text{by monotonicity of } F_n \\ \text{with } i = \texttt{max}(i_{\texttt{f}(a)}, \ldots, i_{\texttt{l}(a)})\end{array} \\
\iff \exists i. & t = & (\texttt{map}\ \{n \mapsto F_n(l_i) \mid n \in \mathcal{N}\}) \searrow (a,\, c) & \\
\iff & t = & (\bigsqcup_i (\texttt{map}\ \{n \mapsto F_n(l_i) \mid n \in \mathcal{N}\})) \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 technical point: if trees are allowed to have infinitely many children, continuity fails in the general case.

4.2.21 Lemma: If $(\texttt{map}\ \mathrm{m}) \in C \overset{\Omega}{\rightleftharpoons} A$ and $m(n)$ is a total lens from $C(n)$ to $A(n)$ for each n, then $(\texttt{map}\ \mathrm{m})$ is a total lens from C to A.

Proof: Suppose $c \in C$. 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., $(\texttt{map}\ \mathrm{m}) \nearrow c$ is defined. Conversely, suppose $a \in A$ and $c \in C$. For any n in $\mathsf{dom}(a) \cap \mathsf{dom}(c)$, we have $a(n) \in A(n)$ and $c(n) \in C(n)$; hence $m(n) \searrow (a(n),\, c(n))$ is defined. For any $n \in \mathsf{dom}(a)$, we have $a(n) \in A(n)$, so $m(n) \searrow (a(n),\, \Omega)$ is also defined. Thus, $(\texttt{map}\ \mathrm{m}) \searrow (a,\, c)$ is defined. □

Interestingly, the `map` combinator does not obey the PUTPUT law. Consider a constant function from names to lenses $m = \lambda n.l$ and $(a, c) \in \mathsf{dom}(l\searrow)$ such that $l \searrow (a,\, c) \neq l \searrow (a,\, \Omega)$. We have

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

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.

Another point to note 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 the `const` lens 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.

4.3 Derived Lenses

We now derive some useful lenses from the primitive ones of Section 4.2. In each of the derived lenses, the accompanying type declaration can be verified straightforwardly from the types of the primitive lenses from which it is defined. Many of the derived lenses are used in the example of Section 5.

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

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

We may now define a lens that discards all of the children of a tree that do not satisfy some predicate p:

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

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 three primitive lenses: `const` $\{\!\!\}$ d, with type $C|_{\overline{p}} \stackrel{\Omega}{\rightleftharpoons} \{\{\!\!\}\}$, the lens `id`, with type $C|_p \stackrel{\Omega}{\rightleftharpoons} C|_p$, and the `fork` lens (with the observation that $C|_p = C|_p + \{\!\!\}$.)

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

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

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

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

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

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

The `hoist` primitive defined above 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.

$$\boxed{\begin{array}{l} \texttt{hoist\_nonunique}\ n\ p \quad = \quad \texttt{xfork}\ \{n\}\ p\ (\texttt{hoist}\ n)\ \texttt{id} \\ \hline \forall n{\in}\mathcal{N}.\ \forall p{\subseteq}\mathcal{N}.\ \forall D{\subseteq}(\mathcal{T}|_{\overline{n}\cap\overline{p}}).\ \forall C{\subseteq}(\mathcal{T}|_p). \quad \texttt{hoist\_nonunique}\ n\ p \in (\{\!|n \mapsto C|\!\} + D) \overset{\Omega}{\rightleftharpoons} (C + D) \end{array}}$$

The type of hoist_nonunique follows straightforwardly from the types of xfork, hoist, and id.

4.4 Lists

XML and many other concrete data formats make heavy use of ordered lists. We describe in this section how we represent lists, using a standard cons cell encoding, and introduce some derived lenses to manipulate them.

4.4.1 Definition: A tree t is a *list* iff either it is empty or else it has exactly two children, one named *h and another named *t, and $t(\texttt{*t})$ is also a list. In the following, we use the lighter notation $[t_1 \dots t_n]$ for the tree:

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

We write [] for the set $\{\{\!|\,|\!\}\}$ containing only the empty list, $C :: D$ for the set $\{\!|\texttt{*h} \mapsto C\ \texttt{*t} \mapsto D|\!\}$ of "cons cells trees" whose head belongs to C and whose tail belongs to D, and $\texttt{[}C\texttt{]}$ for the set of lists with elements in C—i.e., the smallest set of trees satisfying $\texttt{[}C\texttt{]} = \texttt{[]} \cup (C :: \texttt{[}C\texttt{]})$.

We now define some lenses for manipulating lists. The first two extract the head or tail of the list.

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

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

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, it returns the tree under name *h. The lens tl works analogously. Note that the types of these lenses apply to both homogeneous lists (the type of hd implies $\forall C{\subseteq}\mathcal{T}.\ \forall d{\in}\texttt{[}C\texttt{]}.\ \texttt{hd}\ d \in \texttt{[}C\texttt{]} \overset{\Omega}{\rightleftharpoons} C$) and well as cons cells (pairs) whose head and tail have arbitrary types; both possibilities are used in the type of the bookmark lens in Section 5. The types of hd and tl follow straightforwardly from the type of focus.

The map_list lens iterates over a list, applying its argument to every element of the list:

$$\boxed{\begin{array}{l} \texttt{map\_list}\ l \quad = \quad \texttt{map}\ \{\texttt{*h} \mapsto l,\ \texttt{*t} \mapsto \texttt{map\_list l}\} \\ \hline \forall C, A{\subseteq}\mathcal{T}.\ \forall l \in C \overset{\Omega}{\rightleftharpoons} A. \quad \texttt{map\_list}\ l \in \texttt{[}C\texttt{]} \overset{\Omega}{\rightleftharpoons} \texttt{[}A\texttt{]} \end{array}}$$

This lens simply applies l to every child named *h and recurses on every child named *t. In the *put* direction, $l\searrow$ is used on corresponding pairs from the abstract and concrete lists. The result has the same length as the abstract list; if the concrete list is longer, the tail is thrown away. If it is shorter, the extra elements of the abstract list are put into Ω. The type of map_list follows from the types of map and l, together with Lemma 3.3.5. Recall that the type of map requires that both C and A be closed under shuffling. Accordingly, we must show that $\texttt{[}T\texttt{]} = \texttt{[}T\texttt{]}^{\circlearrowleft}$ for any type T. From the definition of lists, the set of domains of trees in $\texttt{[}T\texttt{]}$ is $D = \{\{\texttt{*h}, \texttt{*t}\}, \emptyset\}$. We can calculate $\texttt{[}T\texttt{]}^{\circlearrowleft}$ directly:

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

A final derived 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.

$$\boxed{\begin{array}{l} \texttt{hoist\_hd}\ p \quad = \quad \texttt{hoist\_nonunique *h}\ p;\ \texttt{hoist\_nonunique *t}\ \overline{p} \\ \hline \forall p{\subseteq}(\mathcal{N}\backslash\{\texttt{*t}\}).\ \forall C{\subseteq}(\mathcal{T}|_p).\ \forall D{\subseteq}(\mathcal{T}|_{\overline{p}}).\qquad \texttt{hoist\_hd}\ p \in (C :: D) \overset{\Omega}{\rightleftharpoons} (C + D) \end{array}}$$

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

$$\texttt{hoist\_nonunique *h}\ p \in C :: D \overset{\Omega}{\rightleftharpoons} C + \texttt{*t} \mapsto D$$

and also

$$\texttt{hoist\_nonunique *t}\ \overline{p} \in C + \texttt{*t} \mapsto D \overset{\Omega}{\rightleftharpoons} C + D$$

which yields the desired result for the composition.

4.5 Conditional Lenses

In this section, we describe some *conditional* lens combinators, which can be used to selectively apply one lens or another to a tree. Like `xfork` and `map`, these lenses are parameterized by a pair of lenses. Unlike those lenses, which split the input tree and apply a lenses to each part of the tree, the conditional lenses apply different sub-lenses to the entire tree. These lenses are newer and somewhat more experimental than the primitives described above—we do not yet feel we have a good understanding of the design space of conditional constructs for bi-directional programming languages. Neither of these combinators is used in the bookmark example in Section 5, but we have found ways of using each of them in other programming situations.

The requirement that makes conditionals difficult 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. There seem to be several forms of conditional that achieve both well-behavedness and totality, but we have not yet found ones that meet all our needs.

Concrete Conditional

Our first conditional, `ccond`, is parameterized on a predicate on trees and two lenses, l_t and l_f. In the *get* direction, it tests the concrete view, c, and applies the *get* of l_t if c satisfies the predicate and l_f otherwise. In the *put* direction, `ccond` again examines the concrete view and applies the *put* of l_t if it satisfies the predicate and l_f 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_t and l_f 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.)

$$\boxed{\begin{array}{rcl} (\texttt{ccond}\ B\ l_t\ l_f) \nearrow c & = & \begin{cases} l_t \nearrow c & \text{if } c \in B \\ l_f \nearrow c & \text{if } c \notin B \end{cases} \\ (\texttt{ccond}\ B\ l_t\ l_f) \searrow (a,\ c) & = & \begin{cases} l_t \searrow (a,\ c) & \text{if } c \in B \\ l_f \searrow (a,\ c) & \text{if } c \notin B \end{cases} \\ \hline \multicolumn{3}{l}{\forall C, A, B{\subseteq}\mathcal{T}.\ \forall l_t \in C{\cap}B \overset{\Omega}{\rightleftharpoons} A.\ \forall l_f \in C\backslash B \overset{\Omega}{\rightleftharpoons} A.\qquad \texttt{ccond}\ B\ l_t\ l_f \in C \overset{\Omega}{\rightleftharpoons} A} \end{array}}$$

One subtlety in the definition is worth noting: we arbitrarily choose to *put* Ω using l_f (because $\Omega \notin B$ for any $B \subseteq \mathcal{T}$). We could equally well arrange the definition so as to send Ω through l_t.

4.5.2 Lemma: $\forall C, A, B{\subseteq}\mathcal{T}.\ \forall l_t \in C{\cap}B \overset{\Omega}{\rightleftharpoons} A.\ \forall l_f \in C\backslash B \overset{\Omega}{\rightleftharpoons} A.$ `ccond` $B\ l_t\ l_f \in C \overset{\Omega}{\rightleftharpoons} A.$

Proof:

GET: If $c \in B$, then $(\texttt{ccond}\ B\ l_t\ l_f) \nearrow c = l_t \nearrow c \in A$. Otherwise, $c \in C \backslash B$ and so $(\texttt{ccond}\ B\ l_t\ l_f) \nearrow c = l_f \nearrow c \in A$.

PUT: If $c \in B$, then $(\texttt{ccond}\ B\ l_t\ l_f) \searrow (c,\ a) = l_t \searrow (a,\ c) \in C \cap B$. Otherwise $(\texttt{ccond}\ B\ l_t\ l_f) \searrow (c,\ a) = l_f \searrow (a,\ c) \in C \backslash B$. In either case, $(\texttt{ccond}\ B\ l_t\ l_f) \searrow (c,\ a) \in C$.

GETPUT: Assume that $(\texttt{ccond}\ B\ l_t\ l_f) \nearrow c$ is defined. Then

$$\begin{array}{ll} & (\texttt{ccond}\ B\ l_t\ l_f) \searrow ((\texttt{ccond}\ B\ l_t\ l_f) \nearrow c,\ c) \\ = & \begin{cases} (\texttt{ccond}\ B\ l_t\ l_f) \searrow (l_t \nearrow c,\ c) & \text{if } c \in B \\ (\texttt{ccond}\ B\ l_t\ l_f) \searrow (l_t \nearrow c,\ c) & \text{otherwise} \end{cases} \\ = & \begin{cases} l_t \searrow (l_t \nearrow c,\ c) & \text{if } c \in B \\ l_f \searrow (l_f \nearrow c,\ c) & \text{otherwise} \end{cases} \\ = & c \qquad\qquad \text{by GETPUT for } l_t \text{ and } l_f. \end{array}$$

PUTGET: Assume that $(\texttt{ccond}\ B\ l_t\ l_f) \searrow (a,\ c)$ is defined. Then

$$\begin{array}{ll} & (\texttt{ccond}\ B\ l_t\ l_f) \nearrow (\texttt{ccond}\ B\ l_t\ l_f) \searrow (a,\ c) \\ = & \begin{cases} (\texttt{ccond}\ B\ l_t\ l_f) \nearrow l_t \searrow (a,\ c) & \text{if } c \in B \\ (\texttt{ccond}\ B\ l_t\ l_f) \nearrow l_f \searrow (a,\ c) & \text{otherwise} \end{cases} \\ = & \begin{cases} l_t \nearrow l_t \searrow (a,\ c) & \text{if } c \in B \\ l_f \nearrow l_f \searrow (a,\ c) & \text{otherwise} \end{cases} \\ = & a \qquad\qquad \text{by PUTGET for } l_t \text{ and } l_f. \end{array}$$

□

4.5.3 Lemma: Let F and G be continuous functions from lenses to lenses. Then the function $\lambda l.\ \texttt{ccond}\ B\ F(l)\ G(l)$ is continuous.

Proof: Let l and l' be two lenses such that $l \prec l'$. We must show that $\texttt{ccond}\ B\ F(l)\ G(l) \prec \texttt{ccond}\ B\ F(l')\ G(l')$. For monotonicity, first assume that $(\texttt{ccond}\ B\ F(l)\ G(l)) \nearrow c$ is defined. We have:

$$\begin{array}{lll} & (\texttt{ccond}\ B\ F(l)\ G(l)) \nearrow c & \\ = & \begin{cases} F(l) \nearrow c & \text{if } c \in B \\ G(l) \nearrow c & \text{otherwise} \end{cases} & \\ = & \begin{cases} F(l') \nearrow c & \text{if } c \in B \\ G(l') \nearrow c & \text{otherwise} \end{cases} & \text{by } l \prec l' \\ = & (\texttt{ccond}\ B\ F(l)\ G(l)) \nearrow c. & \end{array}$$

Conversely, let a and c be two trees in $\mathcal{T} \times \mathcal{T}_\Omega$ and assume that $(\texttt{ccond}\ B\ F(l)\ G(l)) \searrow (a,\ c)$ is defined. Then

$$\begin{array}{lll} & (\texttt{ccond}\ B\ F(l)\ G(l)) \searrow (a,\ c) & \\ = & \begin{cases} F(l) \searrow (a,\ c) & \text{if } c \in B \\ G(l) \searrow (a,\ c) & \text{otherwise} \end{cases} & \\ = & \begin{cases} F(l') \searrow (a,\ c) & \text{if } c \in B \\ G(l') \searrow (a,\ c) & \text{otherwise} \end{cases} & \text{by } l \prec l' \\ = & (\texttt{ccond}\ B\ F(l')\ G(l')) \searrow (a,\ c). & \end{array}$$

Thus $\lambda l.\ \texttt{ccond}\ B\ F(l)\ G(l)$ is monotonic. We 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 with $l = \bigsqcup_i l_i$. We have

$$\begin{array}{lll}
 & (\texttt{ccond}\ B\ F(l)\ G(l)) \nearrow c = t & \\
\iff & F(l) \nearrow c = t & \text{if } c \in B \\
 & G(l) \nearrow c = t & \text{otherwise} \\
\iff & F(\bigsqcup_i l_i) \nearrow c = t & \\
 & G(\bigsqcup_i l_i) \nearrow c = t & \\
\iff & (\bigsqcup_i F(l_i)) \nearrow c = t & \text{by continuity of } F \text{ and } G \\
 & (\bigsqcup_i G(l_i)) \nearrow c = t & \\
\iff & \exists i_1.(F(l_{i_1})) \nearrow c = t & \text{by Corollary 3.3.4 (G\scriptsize{ET})} \\
 & \exists i_2.(G(l_{i_2})) \nearrow c = t & \\
\iff & \exists i.F(l_i) \nearrow c = t & i = \max(i_1, i_2) \\
 & \exists i.G(l_i) \nearrow c = t & \\
\iff & \exists i.(\texttt{ccond}\ B\ F(l_i)\ G(l_i)) \nearrow c = t & \\
\iff & (\bigsqcup_i \texttt{ccond}\ B\ F(l_i)\ G(l_i)) \nearrow c = t & \text{by Corollary 3.3.4 (G\scriptsize{ET})}.
\end{array}$$

and

$$\begin{array}{lll}
 & (\texttt{ccond}\ B\ F(l)\ G(l)) \searrow (a,\ c) = t & \\
\iff & F(l) \searrow (a,\ c) = t & \text{if } c \in B \\
 & G(l) \searrow (a,\ c) = t & \text{otherwise} \\
\iff & F(\bigsqcup_i l_i) \searrow (a,\ c) = t & \\
 & G(\bigsqcup_i l_i) \searrow (a,\ c) = t & \\
\iff & (\bigsqcup_i F(l_i)) \searrow (a,\ c) = t & \text{by continuity of } F \text{ and } G \\
 & (\bigsqcup_i G(l_i)) \searrow (a,\ c) = t & \\
\iff & \exists i_1.(F(l_{i_1})) \searrow (a,\ c) = t & \text{by Corollary 3.3.4 (P\scriptsize{UT})} \\
 & \exists i_2.(G(l_{i_2})) \searrow (a,\ c) = t & \\
\iff & \exists i.F(l_i) \searrow (a,\ c) = t & i = \max(i_1, i_2) \\
 & \exists i.G(l_i) \searrow (a,\ c) = t & \\
\iff & \exists i.(\texttt{ccond}\ B\ F(l_i)\ G(l_i)) \searrow (a,\ c) = t & \\
\iff & (\bigsqcup_i \texttt{ccond}\ B\ F(l_i)\ G(l_i)) \searrow (a,\ c) = t & \text{by Corollary 3.3.4 (P\scriptsize{UT})}.
\end{array}$$ □

4.5.4 Lemma: If l_t is a total lens from $C \cap B$ to A and l_f is a total lens from $C \backslash B$ to A then `ccond` $B\ l_t\ l_f$ is total from C to A.

Proof: Suppose $c \in C$. If $c \in C \cap B$, then by the totality of l_t, we know that $l_t \nearrow c$ is defined and is in A. On the other hand, if $c \in C \backslash B$, then $l_f \nearrow c$ is defined and is in A. Thus, (`ccond` $B\ l_t\ l_f$) $\nearrow c$ is defined. Conversely, let $a \in A$ and $c \in C_\Omega$. Then, if $c \in C \cap B$, then the totality of l_t tells us that $l_t \searrow (a,\ c)$ is defined and is in $C \cap B$. Similarly, if $c \in C \backslash B$ or $c = \Omega$ then by the totality of l_f, we have $l_f \searrow (a,\ c)$ is defined and is in $C \backslash B$. Thus, we have (`ccond` $B\ l_t\ l_f$) $\searrow (a,\ c)$ is defined. □

Oblivious 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 $l_t \searrow$ or $l_f \searrow$ entirely on its abstract argument. This obliviousness to the concrete argument removes the need for any side conditions relating the behavior of l_t and l_f—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_t and l_f be disjoint guarantees this.

The oblivious conditional can be defined in terms of `map` plus two new primitive lenses, `pred` and `unpred`. The former tests a condition B in the *get* direction and plunges its argument c under an edge labeled with the special name `*t` if B holds of c (i.e., if $c \in B$) and under `*f` otherwise. In the *put* direction, `pred` ignores its concrete argument and simply throws away the edge `*t` or `*f`.

$$\boxed{\begin{array}{rcl} (\texttt{pred}\ B) \nearrow c & = & \begin{cases} \{\!|\texttt{*t} \mapsto c|\!\} & \text{if } c \in B \\ \{\!|\texttt{*f} \mapsto c|\!\} & \text{if } c \notin B \end{cases} \\ (\texttt{pred}\ B) \searrow (a,\, c) & = & \begin{cases} c_0 & \text{if } a = \{\!|\texttt{*t} \mapsto c_0|\!\} \\ c_0 & \text{if } a = \{\!|\texttt{*f} \mapsto c_0|\!\} \end{cases} \\ \hline \forall C, B {\subseteq} \mathcal{T}. & & \texttt{pred}\ B \in C \overset{\Omega}{\rightleftharpoons} (\{\!|\texttt{*t} \mapsto C{\cap}B|\!\} \cup \{\!|\texttt{*f} \mapsto C\backslash B|\!\}) \end{array}}$$

4.5.5 Lemma: $\forall C, B {\subseteq} \mathcal{T}.\ \texttt{pred}\ B \in C \overset{\Omega}{\rightleftharpoons} (\{\!|\texttt{*t} \mapsto C{\cap}B|\!\} \cup \{\!|\texttt{*f} \mapsto C\backslash B|\!\})$.

Proof:

GET: If $c \in B$ then $(\texttt{pred}\ B) \nearrow c = \{\!|\texttt{*t} \mapsto c|\!\}$. Otherwise, $(\texttt{pred}\ B) \nearrow c = \{\!|\texttt{*f} \mapsto c|\!\}$, as required.

PUT: If $a = \{\!|\texttt{*t} \mapsto c_0|\!\}$ then $c_0 \in C \cap B$ and $(\texttt{pred}\ B) \searrow (a,\, c) = c_0$. Similarly, if $a = \{\!|\texttt{*f} \mapsto c_0|\!\}$ then $c_0 \in C\backslash B$ and $(\texttt{pred}\ B) \searrow (a,\, c) = c_0$.

GETPUT: Assume that $(\texttt{pred}\ B) \nearrow c$ is defined. Then, if $c \in B$,

$$\begin{array}{rl} & (\texttt{pred}\ B) \searrow ((\texttt{pred}\ B) \nearrow c,\, c) \\ = & (\texttt{pred}\ B) \searrow (\{\!|\texttt{*t} \mapsto c|\!\},\, c) \\ = & c. \end{array}$$

The case where $c \notin B$ is symmetric.

PUTGET: Assume that $(\texttt{pred}\ B) \searrow (a,\, c)$ is defined. If $a = \{\!|\texttt{*t} \mapsto c|\!\}$, with $c \in B$, then

$$\begin{array}{rll} & (\texttt{pred}\ B) \nearrow (\texttt{pred}\ B) \searrow (a,\, c) & \\ = & (\texttt{pred}\ B) \nearrow (\texttt{pred}\ B) \searrow (\{\!|\texttt{*t} \mapsto c|\!\},\, c) & \\ = & (\texttt{pred}\ B) \nearrow c & \\ = & \{\!|\texttt{*t} \mapsto c|\!\} & \text{since } c \in B \\ = & a. & \end{array}$$

The case where $a = \{\!|\texttt{*f} \mapsto c|\!\}$ with $c \notin B$ is symmetric. □

4.5.6 Lemma: $\texttt{pred}\ B$ is a total lens from C to $\{\!|\texttt{*t} \mapsto C \cap B|\!\} \cup \{\!|\texttt{*f} \mapsto C\backslash B|\!\}$.

Proof: Straightforward. □

The `unpred` lens is dual to `pred`. In the *get* direction, `unpred` simply throws away the edge `*t` or `*f`. In the *put* direction, it tests a condition B and plunges its abstract argument a under an edge labeled with the special name `*t` if B holds of a and under `*f` otherwise; the concrete argument c is ignored.

$$\boxed{\begin{array}{rcl} (\texttt{unpred}\ B) \nearrow c & = & \begin{cases} a_0 & \text{if } c = \{\!|\texttt{*t} \mapsto a_0|\!\} \\ a_0 & \text{if } c = \{\!|\texttt{*f} \mapsto a_0|\!\} \end{cases} \\ (\texttt{unpred}\ B) \searrow (a,\, c) & = & \begin{cases} \{\!|\texttt{*t} \mapsto a|\!\} & \text{if } a \in B \\ \{\!|\texttt{*f} \mapsto a|\!\} & \text{if } a \notin B \end{cases} \\ \hline \forall A, B {\subseteq} \mathcal{T}. & & \texttt{unpred}\ B \in (\{\!|\texttt{*t} \mapsto A{\cap}B|\!\} \cup \{\!|\texttt{*f} \mapsto A\backslash B|\!\}) \overset{\Omega}{\rightleftharpoons} A \end{array}}$$

4.5.7 Lemma: $\forall A, B {\subseteq} \mathcal{T}.\ \texttt{unpred}\ B \in (\{\!|\texttt{*t} \mapsto A{\cap}B|\!\} \cup \{\!|\texttt{*f} \mapsto A\backslash B|\!\}) \overset{\Omega}{\rightleftharpoons} A$.

Proof:

GET: If $c = \{\!|\texttt{*t} \mapsto a|\!\}$ then $(\texttt{unpred}\ B) \nearrow c = a$. Otherwise, $c = \{\!|\texttt{*f} \mapsto a|\!\}$ and $(\texttt{unpred}\ B) \nearrow c = a$, as required.

PUT: If $a \in B$ then $(\texttt{unpred}\ B) \searrow (a,\ c) = \{\!|\texttt{*t} \mapsto a|\!\}$. Similarly, if $a \notin B$ then $(\texttt{unpred}\ B) \searrow (a,\ c) = \{\!|\texttt{*f} \mapsto a|\!\}$, as required.

GETPUT: Assume that $(\texttt{unpred}\ B) \nearrow c$ is defined. If $c = \{\!|\texttt{*t} \mapsto a|\!\}$, with $a \in B$, then

$$\begin{array}{cll} & (\texttt{unpred}\ B) \searrow ((\texttt{unpred}\ B) \nearrow c,\ c) & \\ = & (\texttt{unpred}\ B) \searrow (a,\ c) & \\ = & \{\!|\texttt{*t} \mapsto a|\!\} & \text{as } a \in B \\ = & c. & \end{array}$$

The case where $c = \{\!|\texttt{*f} \mapsto a|\!\}$ and $a \notin B$ is symmetric.

PUTGET: Assume that $(\texttt{unpred}\ B) \searrow (a,\ c)$ is defined. Then if $a \in B$:

$$\begin{array}{cl} & (\texttt{unpred}\ B) \nearrow (\texttt{unpred}\ B) \searrow (a,\ c) \\ = & (\texttt{unpred}\ B) \nearrow \{\!|\texttt{*t} \mapsto a|\!\} \\ = & a. \end{array}$$

The case where $a \notin B$ is symmetric. □

4.5.8 Lemma: $\texttt{unpred}\ B$ is a total lens from $\{\!|\texttt{*t} \mapsto A \cap B|\!\} \cup \{\!|\texttt{*f} \mapsto A \backslash B|\!\}$ to A.

Proof: Straightforward. □

Using `pred`, `unpred`, and `map`, we can now easily build the oblivious conditional lens, `cond`. It is parameterized on two sets of trees, B_C and B_A, and two lenses, l_t and l_f. In the *get* direction, `cond` first applies $\texttt{pred}\ B_C$ to transform the concrete view c into either $\{\!|\texttt{*t} \mapsto c|\!\}$ (if $c \in B_C$) or $\{\!|\texttt{*f} \mapsto c|\!\}$ (if $c \notin B_C$). Next, we use $\texttt{map}\{\texttt{*t} \mapsto l_t,\ \texttt{*f} \mapsto l_f\}$ to apply the *get* of the l_t or l_f lens. As the *get* direction of `pred` always produces a tree with one child, exactly one of l_t or l_f is applied to the original concrete view. Finally, we apply $\texttt{unpred}\ B_A$ to hoist the original c, which has now been transformed by either l_t or l_f, up to the top level. In the *put* direction, we first apply the put of `unpred`, which ignores its second argument. This produces a tree of the form $\{\!|\texttt{*t} \mapsto a|\!\}$ if $a \in B_A$ or $\{\!|\texttt{*f} \mapsto a|\!\}$ if $a \notin B_A$. To this tree, we apply the put of `map`, which applies the *put* of l_t or l_f to the original tree a. Finally, we use the *put* of `pred` to hoist the original a back up to the top level.

$$\boxed{\begin{array}{l} \texttt{cond}\ B_C\ B_A\ l_t\ l_f \quad = \quad \texttt{pred}\ B_C;\ \texttt{map}\ \{\texttt{*t} \mapsto l_t,\ \texttt{*f} \mapsto l_f\};\ \texttt{unpred}\ B_A \\ \hline \forall C, A, B_C, B_A \subseteq \mathcal{T}.\ \forall l_t \in C \cap B_C \overset{\Omega}{\rightleftharpoons} A \cap B_A.\ \forall l_f \in (C \backslash B_C) \overset{\Omega}{\rightleftharpoons} (A \backslash B_A). \\ \quad \texttt{cond}\ B_C\ B_A\ l_t\ l_f \in C \overset{\Omega}{\rightleftharpoons} A \end{array}}$$

The type of `cond` follows directly from the types of the primitive lenses used to define it. The following type annotations witness this fact (the notation is explained on page 32).

$$\begin{array}{cll} & \texttt{cond}\ BC\ B_A\ l_t\ l_f & \in C \\ = & \texttt{pred}\ B_C; & : \left\{\!\left|\texttt{*t} \mapsto C \cap B_C\right|\!\right\} \cup \left\{\!\left|\texttt{*f} \mapsto C \backslash B_C\right|\!\right\} \\ & \texttt{map}\ \{\texttt{*t} \mapsto l_t,\ \texttt{*f} \mapsto l_f\}; & : \left\{\!\left|\texttt{*t} \mapsto A \cap B_A\right|\!\right\} \cup \left\{\!\left|\texttt{*f} \mapsto A \backslash B_A\right|\!\right\} \\ & \texttt{unpred}\ B_A & \overset{\Omega}{\rightleftharpoons} A. \end{array}$$

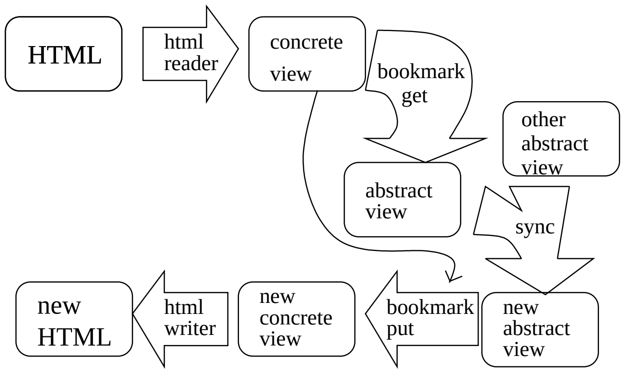
5 Extended Example: A Bookmark Lens

With these definitions in hand, we are ready to develop an extended 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.

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

Figure 2: Abstract Bookmark Types



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>` 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>
```

```
<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)

```
{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)

$$
\begin{array}{lcl}
\mathit{Val} & = & \{\!|\mathcal{N}|\!\} \\
\mathit{PCDATA} & = & \{\!|\texttt{PCDATA} \mapsto \mathit{Val}|\!\} \\[6pt]
\mathit{CLink} & = & \texttt{<dt>}\ \mathit{CLink}_1 :: \texttt{[]}\ \texttt{</dt>} \\
\mathit{CLink}_1 & = & \texttt{<a add\_date href>}\ \mathit{PCDATA} :: \texttt{[]}\ \texttt{</a>} \\[6pt]
\mathit{CFolder} & = & \texttt{<dd>}\ \mathit{CContents}\ \texttt{</dd>} \\[6pt]
\mathit{CContents} & = & \mathit{CContents}_1 :: \mathit{CContents}_2 :: \texttt{[]} \\
\mathit{CContents}_1 & = & \texttt{<h3>}\ \mathit{PCDATA} :: \texttt{[]}\ \texttt{</h3>} \\
\mathit{CContents}_2 & = & \texttt{<dl>}\ \texttt{[}\mathit{CItem}\texttt{]}\ \texttt{</dl>} \\[6pt]
\mathit{CItem} & = & \mathit{CLink} \cup \mathit{CFolder} \\[6pt]
\mathit{CBookmarks} & = & \texttt{<html>}\ \mathit{CBookmarks}_1 :: \mathit{CBookmarks}_2 :: \texttt{[]}\ \texttt{</html>} \\
\mathit{CBookmarks}_1 & = & \texttt{<head>}\ (\texttt{<title>}\ \mathit{PCDATA}\ \texttt{</title>} :: \texttt{[]})\ \texttt{</head>} \\
\mathit{CBookmarks}_2 & = & \texttt{<body>}\ \mathit{CContents}\ \texttt{</body>}
\end{array}
$$

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)

into a tree of this form:

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

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 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.

To aid in checking well-behavedness, the lenses in Figure 6 are annotated with type declarations both for each top-level lens definition and at each use of the composition operator ;. For example, the annotated composition of lenses $: C\ l_1\ ;\ :\ B\ l_2 \overset{\Omega}{\rightleftharpoons} A$ stands for $l_1;l_2$ and asserts that $l_1 \in C \overset{\Omega}{\rightleftharpoons} B$ and $l_2 \in B \overset{\Omega}{\rightleftharpoons} A$. From this, it is easy to verify that the type of the whole composite is $C \overset{\Omega}{\rightleftharpoons} A$. (The notation looks strange in-line, but works well for multi-line displays.)

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

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 $\{\!|\,\texttt{PCDATA -> Search-Engine}\,|\!\}$. In the transition from step *vi* to step *v*, the *put* direction of `prune add_date` $\{\!|\,\texttt{\$today}\,|\!\}$ 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 $\{\!|\,\texttt{\$today}\,|\!\}$ would have been used to fill in today's date. (Formally, the whole set of lenses is parameterized on the variable `$today`, which ranges over names.)

The *get* direction of the `folder` lens separates out the folder name and its contents, stripping out undesired structure where necessary. Note the use of `hoist_hd` to extract the `<h3>` and `<dl>` tags containing the folder

`link =`	$\in \{\!\{* \mapsto \mathit{Clink}_1 :: []\}\!\}$
`    hoist *;`	$: \mathit{CLink}_1 :: []$
`    hd [];`	$: \mathit{CLink}_1$
`    hoist_nonunique a {* add_date href};`	$: \{\!\{* \mapsto \mathit{PCDATA} :: [],\ \texttt{add\_date} \mapsto \mathit{Val},$
	$\texttt{href} \mapsto \mathit{Val}\}\!\}$
`    rename {*=name, href=url};`	$: \{\!\{\texttt{name} \mapsto \mathit{PCDATA} :: [],\ \texttt{add\_date} \mapsto \mathit{Val},$
	$\texttt{url} \mapsto \mathit{Val}\}\!\}$
`    prune add_date {$today};`	$: \{\!\{\texttt{name} \mapsto \mathit{PCDATA} :: [],\ \texttt{url} \mapsto \mathit{Val}\}\!\}$
`    map {name -> (hd [];`	$: \mathit{PCDATA}$
`                  hoist PCDATA)}`	$\overset{\Omega}{\rightleftharpoons} \{\!\{\texttt{name} \mapsto \mathit{Val},\ \texttt{url} \mapsto \mathit{Val}\}\!\} = \mathit{ALink}_1$
`folder =`	$\in \{\!\{* \mapsto \mathit{CContents}\}\!\}$
`    hoist *;`	$: \mathit{CContents}$
`    hoist_hd {h3};`	$: \{\!\{\texttt{h3} \mapsto \{\!\{* \mapsto \mathit{PCDATA} :: []\}\!\},\ \mathit{CContents}_2 :: []\}\!\}$
`    fork {h3} (id) (hoist_hd {dl});`	$: \{\!\{\texttt{h3} \mapsto \{\!\{* \mapsto \mathit{PCDATA} :: []\}\!\},$
	$\texttt{dl} \mapsto \{\!\{* \mapsto [\mathit{CItem}]\}\!\}\}\!\}$
`    rename {h3=name, dl=contents};`	$: \{\!\{\texttt{name} \mapsto \{\!\{* \mapsto \mathit{PCDATA} :: []\}\!\},$
	$\texttt{contents} \mapsto \{\!\{* \mapsto [\mathit{CItem}]\}\!\}\}\!\}$
`    map {name -> (hoist *;`	$: \mathit{PCDATA} :: []$
`                  hd [];`	$: \mathit{PCDATA}$
`                  hoist PCDATA)`	
`         contents -> (hoist *;`	$: [\mathit{CItem}]$
`                      map_list item)}`	
	$\overset{\Omega}{\rightleftharpoons} \{\!\{\texttt{name} \mapsto \mathit{Val},\ \texttt{contents} \mapsto [\mathit{AItem}]\}\!\} = \mathit{AFolder}_1$
`item =`	$\in \mathit{CItem}$
`    map { dd -> folder, dt -> link };`	$: \{\!\{\texttt{dd} \mapsto \mathit{AFolder}_1\}\!\} \cup \{\!\{\texttt{dt} \mapsto \mathit{ALink}_1\}\!\}$
`    rename { dd=folder, dt=link }`	$\overset{\Omega}{\rightleftharpoons} \mathit{AFolder} \cup \mathit{ALink} = \mathit{AItem}$
`bookmarks =`	$\in \mathit{CBookmarks}$
`    hoist html;`	$: \{\!\{* \mapsto \mathit{CBookmarks}_1 :: \mathit{CBookmarks}_2 :: []\}\!\}$
`    hoist *;`	$: \mathit{CBookmarks}_1 :: \mathit{CBookmarks}_2 :: []$
`    tl {|head -> {|* -> [{|title -> {|* ->`	
`        [{|PCDATA -> Bookmarks|}] |} |}] |} |};`	$: \mathit{CBookmarks}_2 :: []$
`    hd [];`	$: \mathit{CBookmarks}_2$
`    hoist body;`	$: \{\!\{* \mapsto \mathit{CContents}\}\!\}$
`    folder`	$\overset{\Omega}{\rightleftharpoons} \mathit{AFolder}_1$

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 {};`	`{* -> [{PCDATA -> Google}]`
	`hoist_nonunique a {* add_date href}`	`add_date -> 1032458036`
		`href -> www.google.com}`
v:	`hoist *; hd {};`	`{name -> [{PCDATA -> Google}]`
	`hoist_nonunique a {* add_date href};`	`add_date -> 1032458036`
	`rename {*=name, href=url}`	`url -> www.google.com}`
vi:	`hoist *; hd {};`	`{name -> [{PCDATA -> Google}]`
	`hoist_nonunique a {* add_date href};`	`url -> www.google.com}`
	`rename {*=name, href=url};`	
	`prune add_date {$today}`	
vii:	`hoist *; hd {};`	`{name -> {PCDATA -> Google}`
	`hoist_nonunique a {* add_date href};`	`url -> www.google.com}`
	`rename {*=name, href=url};`	
	`prune add_date {$today};`	
	`map { name -> (hd {}) }`	
viii:	`hoist *; hd {};`	`{name -> Google`
	`hoist_nonunique a {* add_date href};`	`url -> www.google.com}`
	`rename {*=name, href=url};`	
	`prune add_date {$today};`	
	`map { name -> (hd {}; hoist PCDATA) }`	

Figure 8: Building up a link lens incrementally.

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

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.

6 Related Work

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

Our foundational structures—lenses and their laws—are not new: closely related structures have been studied for decades in the database community. However, our "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 a programming language for constructing bi-directional transformations has also been explored previously. However, we appear to be the first to have connected it with a formal semantic foundation, choosing primitives that can be combined into composite lenses whose well-behavedness is guaranteed by construction.

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 [12] 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 [6] developed an elegant semantic characterization of update translation, introducing the notion of *complement* of a view, which must include at least all information missing from the view. When a complement is fixed, there exists at most one update of the database that reflects a given update on the view while leaving the complement unmodified—i.e., that "translates updates under a constant complement." In general, a view may have many complements, each corresponding to a possible strategy for translating view updates to database updates. The problem of translating view updates then becomes a problem of finding, for a given view, a suitable complement.

Gottlob, Paolini, and Zicari [14] 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.

Recent work by Lechtenbörger [18] establishes that translations of view updates under constant complements are possible precisely if view update effects may be undone using further view updates.

In a companion report [25], we have stated 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.

The view update problem has also been studied in the context of object-oriented databases. School, Laasch, and Tresch [29] 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.

Updates 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.) However, our goals are more modest than those of most work on relational update transformation in one significant respect: we do not, at present, support any analog of relational join, a major source of update ambiguity in the relational world. (We have not yet encountered a need for join in the setting in which we use our combinators—transforming tree-structured application data to prepare it for synchronization.) We briefly review the most relevant research from the relational setting.

Masunaga [19] described an automated algorithm for translating updates on views defined by relational algebra. The core idea was to annotate where the "semantic ambiguities" arise, indicating they must be resolved either with knowledge of underlying database semantic constraints or by interactions with the user.

Keller [16] catalogued all possible strategies for handling updates to a select-project-join view and showed that these are exactly the set of translations that satisfy a small set of intuitive criteria. 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 [17] 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 [7] described a scheme for interactively constructing update translators for object-based views of relational databases.

Medeiros and Tompa [20] presented a design tool for exploring the effects of choosing a view update policy. This tool shows the update translation for update requests supplied by the user; by considering all possible valid concrete states, the tool predicts whether the desired update would in fact be reflected back into the view after applying the translated update to the concrete database. Miller *et al.* [21] 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 [5, 4] described a tool for translating views and observed that if one can translate any concrete view to and from a *meta-model* (shared abstract view), one then gets bi-directional transformations between any pair of concrete views. They limited themselves to mappings where the concrete and abstract views are isomorphic.

Complexity bounds have also been studied for various versions of the view update inference problem. In one of the earliest, Cosmadakis and Papadimitriou [10, 11] considered the view update problem for a single relation, where the view is a projection of the underlying relation, and showed that there are polynomial time algorithms for determining whether insertions, deletions, and tuple replacements to a projection view are translatable into concrete updates. More recently, Buneman, Khanna, and Tan [9] established a variety of intractability results for the problem of inferring "minimal" view updates in the relational setting for query languages that include both join and either project or union.

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.

Programming Languages for View Update

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" [24] and in Hofmann and Pierce's work on "positive subtyping" [15]. Another related idea, proposed by Wadler [32], extended algebraic pattern matching to abstract data types using programmer-supplied *in* and *out* operators. This is related to the special case of our lenses in which the *get* and *put* functions always form an isomorphism.

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

The designers of the RIGEL language [28] 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.

A number of papers have been written in the program transformation community on *inverse* and *reversible computation*—see, for example, [**?**, **?**, **?**] and other papers cited there. These languages bear many intriguing similarities to ours, but again focus on the bijective case.

More recent work by Hu, Mu, and Takeichi [**?**] applies a bi-directional programming language much more closely related to ours to the design of "programmable" editors for structured documents.

Updates and Trees

There have been many proposals for query languages for trees (e.g., XQuery [31] 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 [13], and Braganholo, Davidson, and Heuser [8] 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 [30] 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.

7 Future Work

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 5, 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

The combinators we have described are the product of an iterative process, driven by the practical needs of the Harmony system as it has evolved. Each iteration begins with a Harmony instance that we wish to construct, for which one or more lenses is needed. We write these lenses using our current set of combinators as much as possible, but, when we get stuck, resorting to "native lenses"—pairs of ordinary functions written in a general-purpose programming language and checked for well-behavedness by ad hoc reasoning. After verifying that our combined lenses work as required, we step back and try again to see whether the native lenses we wrote along the way can be expressed using our existing combinators, or, if not, how we can introduce the smallest possible number of new primitives that can be combined to achieve the desired effect. Often, this leads to ideas for simplifying, generalizing, or combining other combinators. Then we start again. The combinators described in this paper have stood the test of time and become the core components of our lens programming toolbox.

We do not know whether this process will converge to a *complete* set of combinators in which all imaginable bi-directional transformations on trees can be expressed. Fortunately, our goals (in the context of

Harmony, at any rate) are more modest: we do not need to be able to represent all conceivable tree transformations as combinator expressions—just the majority of the ones we encounter in day to day programming of synchronizer instances. As in most high-level programming languages, we expect, occasionally, to need to drop down to a lower level language to code some special-purpose lens. However, we are still some distance from even this sort of pragmatic completeness.

Besides the conditional lens combinators discussed in Section 4.5, one area where our present combinators are often awkward (and occasionally downright insufficient) is in splitting views and operating separately on both parts. The `xfork` primitive presented here works in a shallow way, splitting just the top-level children of its inputs according to some predicate on names. Sometimes we need a "deeper" split, in which the information in the given trees is partitioned by some more complex rule involving more of their structure—for example, by performing some form of pattern matching and sending matching parts to one side and non-matching parts to the other. We do not yet feel we have found the "perfect fork": in the ones we have tried, the extra flexibility is difficult to use because the side-conditions that must be checked to guarantee well-formedness become too heavy.

Expressiveness

More generally, what are the limits of "bi-directional programming"? How expressive are the combinators we have defined here? Do they cover any known or succinctly characterizable classes of computations (in the sense that the set of *get* parts of the total lenses built from these combinators coincide with this class)? We have put considerable energy into these questions, but at the moment we can only report that they are challenging!

One puzzle that we are currently wrestling with is list reverse. As we have seen, we can use our combinators, plus recursion, to define an ordinary "map" function on lists encoded as trees. Also, it is clear that list reversal is, semantically, a perfectly good lens—i.e., we can write it as a native lens and it will obey all the conditions we desire of lenses. However, we have been unable, after many attempts, to express reverse in terms of simpler primitives plus recursion (or to show that this is impossible).

We are also investigating notations for bi-directional tree transformations in the classical setting of *tree transducers* (over ranked, node-labeled trees).

Static Analysis

At present, it is the lens programmers' responsibility to check the well-behavedness of the lenses that they write. However, the types of the lens primitives are designed so that these checks are both local and essentially mechanical. The obvious next step is to reformulate them as an automatic type analysis. This task is not completely straightforward, however: there is some inherent tension between precision (how much we need to know about the arguments to a lens combinator to establish the well-behavedness of the result) and tractibility of typechecking or type inference. For example, the infinitary intersection in the current type of `map` is a probable source of difficulties.

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.)

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 would like to experiment with instantiating our semantic framework with other structures besides trees—e.g., with relations, to establish closer links with existing research 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. 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 Owen Gunden, Martin Hofmann, Zack Ives, Sanjeev Khandelwal, Sanjeev Khanna, William Lovas, Kate Moore, Cyrus Najmabadi, 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, 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] P. Atzeni and R. Torlone. Management of multiple models in an extensible database design tool. In *Proceedings of EDBT'96, LNCS 1057*, 1996.

[5] 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.

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

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

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

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

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

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

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

[13] 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.

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

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

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

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

[18] 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.

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

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

[21] 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.

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

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

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

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

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

[27] 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 CIS, University of Pennsylvania, 2004.

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

[29] 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.

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

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

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

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