A Language for Bi-Directional Tree Transformations

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

ABSTRACT

We develop a semantic foundation and a core programming language for bi-directional transformations on tree-stuctured data. In one direction, these transformations, called *lenses*, map a complex "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 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" whose two components are constrained to fit together in a sensible way. We study definedness and continuity in this setting, show that well-behaved lenses form a complete partial order, and state a precise connection with the classical theory of "update translation under a constant complement" from databases.

We then instantiate our semantic framework as a small programming language, called Hocus Focus, whose expressions denote well-behaved lenses operating on tree-structured data. The primitives 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 Hocus Focus can be used to define a lens that translates between a native HTML representation of browser bookmarks and a generic form of abstract bookmark structures.

1. INTRODUCTION

We often want to transform a structure into a different (often smaller, simpler, or more abstract) form, in such a way that updates to the new structure can be reflected back as updates to the original structure. The transformed structure is a *view* of the original structure in which editing is more convenient.

This paper addresses an instance of "editing through a view" that arises in the context of a larger project called Harmony. The goal of the Harmony project is to build a universal synchronization framework for tree-structured data—a generic tool for propagating updates between different copies, possibly stored in different formats, of a tree data structure. For example, Harmony might be used to synchronize the bookmark files of of several different web browsers (on the same or different machines), allowing bookmarks and bookmark folders to be added, deleted, edited, and reorganized in any browser and later combining (synchronizing) the changes performed in different browsers.

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 the abstract view. For example, we can synchronize a Netscape bookmark file with an Opera bookmark file by using appropriate lenses to transform each into an *abstract bookmark structure* and synchronizing the results. However, we are not done at this point: we 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 one for pushing an updated abstract view back into an updated concrete one. (We call these the *get* and *put* components, respectively. The intuition behind the terms is that the mapping from concrete to abstract is often 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.)

Not surprisingly, the tricky parts of constructing lenses arise in the *put* direction. If the *get* component of a lens is a projection—i.e., information is suppressed when moving from concrete to abstract—then the *put* component must *restore* this information in some appropriate way. (We will see a concrete example of this shortly.) The difficulty is that there may, in general, be many ways of doing so.

Our approach to this problem is to design a language in which any expression that specifies a proper *get* function simultaneously defines the corresponding *put*. All the primitives in this language are designed to "work properly in *both* directions," and the combining forms preserve this property. We formalize the notion of proper behavior in the semantics of our language.

To define the semantics of our language, we identify a natural mathematical space of "well-behaved lenses." There is quite a bit to be said at this general level, before fixing 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 *same* a back into c, then we should get back to the original c. These laws must take partiality into account. Third, we must deal with the fact that we will later want to define lenses by recursion (because the trees that our lenses will manipulate may in general have arbitrarily deep nested structure—e.g., when they represent directory hierarchies, browser bookmark folders, etc.). This raises familiar issues of monotonicity and continuity.

Once the semantic foundations are in place, we need some syntax for constructing lenses for the specific domain of edge-labeled trees. Our surface language, Hocus Focus, comprises a collection of *primitive lenses* for tree transformations and powerful lens *combinators* (composition, repetition, conditionals, mapping, etc.) that allow complex lenses to be built up from simpler ones. From these basic constructs, we can build a rich variety of useful "derived lenses"—e.g., lenses for manipulating ordered (list-structured) data represented as trees.

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 and addresses issues of partiality and continuity. Section 4 instantiates this generic framework with primitive lenses and lens combinators for our specific application domain of lenses over trees. Section 5 illustrates the use of these constructs in actual "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 [26]) 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. For brevity, proofs are omitted; these can be found in an accompanying technical report [17].

2. A SMALL EXAMPLE

Suppose our concrete data source c is a small address book giving coordinates for two friends, represented as the following tree (we draw trees sideways to save space):

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

Each curly brace denotes a node, and each "`X` $\mapsto$ ..." on the right of the curly brace denotes a child labeled `X`. To avoid clutter, when an edge leads to an empty tree, we omit the opening brace, the $\mapsto$ symbol and the final childless node—e.g., "`333-4444`" above actually stands for "{`333-4444` $\mapsto$ {". Throughout the paper, we work with unordered, edge-labeled trees in which each node has at most one child of a given name—i.e., a tree is just a partial function from character strings to trees. We will use the word "view" instead of "tree" from now on to emphasize the fact that our "concrete" and "abstract" structures are fundamentally the same sorts of things.[1] This terminology will also facilitate sequential composition of lenses: in the composite lens "$l_1; l_2$" the abstract view generated by l_1 becomes the concrete view seen by l_2.

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

$$a \quad = \quad \begin{cases} \texttt{Pat} \mapsto \texttt{333-4444} \\ \texttt{Chris} \mapsto \texttt{888-9999} \end{cases}$$

Why would we want this? Perhaps the edits are going to be performed by synchronizing this abstract view 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.

Now we are ready to make our changes to the abstract view a, yielding a new abstract view a' of the same form but with modified content. For example, let's change `Pat`'s phone number, drop `Chris`, and add a new friend, `Jo`.

$$a' \quad = \quad \begin{cases} \texttt{Pat} \mapsto \texttt{333-4321} \\ \texttt{Jo} \mapsto \texttt{555-6666} \end{cases}$$

Note that we are only interested in the final view 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 Harmony synchronizes based on the current states of two 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 view c' reflecting the new view 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 "suppressed" (e.g., `Pat`'s `URL`) should have their values carried over from c.

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

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 in a'.

The relation between the concrete views c and c' and the abstract views a and a' can be expressed as a *lens* l consisting of a pair of functions—a *get* function that "extracts" an abstract view from a concrete view plus a *put* function

[1]Note that we use the word "view" here in a slightly different sense than some of the database papers that we cite: there, a "view" is a *function* from concrete to abstract states (i.e., it is a query that, for each concrete database state, picks out a view in our sense).

that "inserts" a new abstract view into an old concrete view to yield a new concrete view. Our goal is to design a programming language, Hocus Focus, that allows these lenses to be described in a concise, natural, and mathematically coherent manner.

3. SEMANTIC FOUNDATIONS

Although surface language, Hocus Focus, will be specialized for dealing with tree transformations, its semantic underpinnings are better presented in an abstract setting that is parameterized by the data structures manipulated by lenses. For the rest of this section, we assume we are given some set C of concrete views and some set A of abstract views; in Section 4 we will choose both of these to be the set of unordered, edge labeled trees.

3.1 Basic structure

A lens is a pair of partial functions: one that gets an abstract view from a concrete view, and one that puts a new abstract view into an old concrete view to yield a new concrete view. Since there may be cases where no old concrete view is available (as we saw with `Jo`'s `URL` in the previous section), the concrete input to the *put* function may also be a special view Ω, pronounced "missing." (There are other possible ways of dealing with missing information; the motivation for this design choice will be discussed after we define the `map` combinator in Section 4.) We write C_Ω for $C \cup \{\Omega\}$.

3.1.1 Definition [Lenses]: A *lens* l comprises two partial functions: a *get* function from C to A, written $l\nearrow$, and a *put* function from $A \times C_\Omega$ to C, written $l\searrow$.

We write $\mathsf{dom}(l\nearrow)$ for the subset of C on which $l\nearrow$ is defined and $\mathsf{dom}(l\searrow)$ for the subset of $A \times C_\Omega$ on which $l\searrow$ is defined, and similarly $\mathsf{ran}(l\nearrow)$ and $\mathsf{ran}(l\searrow)$ for the ranges of the *get* and *put* functions. Note that neither $l\nearrow$ nor $l\searrow$ may return Ω. We often say "we put view a into view c" instead of "we 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.

3.1.2 Definition [Well-behaved lenses]: A lens is *well behaved* iff its *get* and *put* functions obey the following laws:

(GetPut) $c \in \mathsf{dom}(l\nearrow) \implies l\searrow(l\nearrow c,\, c) = c$

(PutGet) $(a, c) \in \mathsf{dom}(l\searrow) \implies l\nearrow(l\searrow(a,\, c)) = a$

The GetPut law states that if some abstract view obtained from a concrete view c is unmodified, putting it back into c will yield the same concrete view. This law also requires the *put* function to be defined on $(l\nearrow c, c)$ whenever $l\nearrow c$ is defined. The PutGet law states that the *put* function captures 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. This law also requires that the *get* function be defined at least on the range of the *put* function.

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:

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

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:

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

Law PutGet fails in this case some information contained in the abstract view does not get propagated in 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 one of these two laws significantly weakens the semantic foundation.

We may also optionally consider a third law, called PutPut:

$$(a, c) \in \mathsf{dom}(l\searrow) \implies l\searrow(a',\, l\searrow(a,\, c)) = l\searrow(a',\, c)$$

This law states that the effect of a sequence of two *put*s is just the effect of the second, as long as the first *put* is defined (the reader might enjoy checking that the special case where $a = a'$ follows from the other two laws).We say that a well-behaved lens that also satisfies PutPut is *very well behaved.* Both well-behaved and very-well-behaved lenses correspond (modulo some details about partiality) 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 Hocus Focus in Section 4, we will drop PutPut because one of our most important lens combinators, `map`, fails to satisfy it. This point is discussed in more detail in Section 4.2.

3.2 Basic Properties

We now explore some simple consequences of the lens laws. To begin, we define a notion of injectivity for the *put* function. Let f be a partial function from $A \times C_\Omega$ to C. By abuse of terminology, we say that f is *injective* iff it is injective in its first argument wherever it is defined—i.e., if, for all views a, a', and c such that $f(a, c)$ and $f(a', c)$ are defined, we have $a \neq a' \implies f(a, c) \neq f(a', c)$.

The following lemma provides an easy way to show that a lens is *not* well behaved. We used it many times while designing the Hocus Focus surface language, to quickly generate and test candidate lenses.

3.2.1 Lemma: The *put* function of a well-behaved lens is injective.

Conversely, for each injective *put* function that satisfies a simple additional condition, there is exactly one *get* function that makes a well-behaved lens.

3.2.2 Lemma: Let $l\searrow$ be an injective partial function from $A \times C_\Omega$ to C such that $(a, c) \in \mathsf{dom}(l\searrow) \implies$

$l\searrow(a,\, l\searrow(a,\, c)) = l\searrow(a,\, c)$. Then there is exactly one function $l\nearrow$ such that $l = (l\nearrow,\, l\searrow)$ is a well-behaved lens.

This lemma shows that we can define a lens simply by giving a suitable *put* function. However, in most cases, we have found it more convenient to write out both *get* and *put* functions explicitly and directly check all laws.

3.3 Recursion

Since our lens framework is going to be instantiated with 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 be able to define lenses by recursion. Our final task for this foundational section, then, is to set up the necessary structure for interpreting recursive definitions in the surface language.

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' has more information than a lens l, written $l \prec l'$, if the *put* function of l' is an extension of the *put* function of l—that is, if $l'\searrow$ is defined on a larger domain than $l\searrow$ and if the two *put* functions are equal on their common domain, $\mathsf{dom}(l\searrow)$. This relation is a partial order.

A cpo is an 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 of elements, we write $\bigsqcup_{n\in\omega} l_n$ for its least upper bound. A *cpo with bottom* is a cpo that contains an element, $\perp$, that is smaller than every other element. In our setting, $\perp_l$ is the lens whose *put* and *get* function are undefined everywhere.

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

We can now apply standard domain theory (as described, for example, in [31]) to interpret a variety of constructs for defining continuous lens combinators. In particular, every continuous function on well-behaved lenses has a least fixed point that is a well behaved lens.

4. TRANSFORMING TREES

We now describe our surface language, Hocus Focus. We first formally define the set of edge-labeled trees. We then present some primitive 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 intuitions and small examples along the way; an extended example using most of the lenses together appears in Section 5.

This collection of primitive lenses is not "complete" in any strong sense. Our Harmony prototype defines a few other primitives, and we expect to add a few more as we address a broader range of applications. However, as we shall see, the lenses described here form a core language of surprising expressive power.

4.1 Views

From now on, we choose both C and A to be the set of unordered, edge-labeled trees, which we call *views*. The edge labels are drawn from some infinite set of *names*—e.g., character strings.

4.1.1 Definition [Views]: A *view* is a finite partial function from names to views.

We write V for the set of views, and V_Ω for $V \cup \{\Omega\}$ where $\Omega \notin V$. The special view Ω is a placeholder to indicate that a concrete view is missing. We write $\mathsf{dom}(c)$ for the domain of a view c. To make some lens definitions shorter, we assume that $\mathsf{dom}(\Omega) = \emptyset$. The metavariables a, c, d, and v range over views; by convention, we use a for views that are thought of as abstract and c or d for concrete views.

Note that our trees are different from (and simpler than) XML trees, which are node labeled and ordered. We present in Section 5 a straightforward encoding of XML/HTML trees into our views.

We now introduce some notations involving views. Let v be the view that associates v_1 to $\mathtt{n}_1$, v_2 to $\mathtt{n}_2$, ..., and v_k to $\mathtt{n}_k$. We write v as $\{\mathtt{n}_1 \mapsto v_1 \ldots \mathtt{n}_k \mapsto v_k\}$ when it appears in running text, and as an opening brace and a vertical list of name/subview pairs (dropping the closing curly brace) when it appears in a displayed figure. We write "{}" (in running text) or "{" (in displays) for the empty view, and $v(n)$ for the view associated to name n in v.

We often define views by extension. For instance, let v be a view and p be a set of names such that $p \subseteq \mathsf{dom}(v)$; we may define a view w as $w = \{n \mapsto v(n) \mid n \in p\}$. When p is a set of names (not necessarily included in $\mathsf{dom}(v)$), we write $v|_p$ for the view $\{n \mapsto v(n) \mid n \in p \cap \mathsf{dom}(v)\}$. By convention, we take $\Omega|_p = \Omega$ for any p (this shortens some definitions below), and we write $\overline{p}$ for the complement of the set p.

We now define a notion of *concatenation* for views. Let $v, v' \in V \times V$. We write $v + v'$ (in running text) or $\begin{cases} v \\ v' \end{cases}$ (in displays) for the view

$$\begin{cases} n \mapsto v(n) & n \in \mathsf{dom}(v) \\ n \mapsto v'(n) & n \in \mathsf{dom}(v') \end{cases}$$

where $\mathsf{dom}(v) \cap \mathsf{dom}(v') = \emptyset$.

A *value* is a view of the special form $\{k \mapsto \{\}\}$. For instance, a phone number $\{\mathtt{333\text{-}4444} \mapsto \{\}\}$ in the example of Section 2 is a value. We sometimes simply write `333-444` for such a value.

The only restriction we impose on the set of views is the finiteness of their width, in order to prove the continuity of the `map` lens. In practice, we might want to impose some additional restrictions, so that views only represent "well-formed" trees, corresponding for instance to schemas or consistency constraints. Our current Harmony implementation actually enforces such constraints (for reasons having to do with the way the synchronization algorithm works), but we will not discuss them further in this paper.

4.2 Primitive Lenses

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 structure of views: the identity lens, the constant lens, and sequential composition of lenses. We then introduce several lenses that inspect and manipulate tree structures—three atomic lenses (`rename`, `hoist`, 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 view in the *get* direction. Most lenses thus do not need to use the concrete view in the *put* direction.

Every atomic lens defined in this section is well-behaved, and every lens combinator is continuous. In fact, most lenses (all but `map`) are very-well-behaved.

In order to lighten the presentation, we assume that a lens defined in terms of other lenses is undefined wherever these other lenses are undefined (i.e., application of lens combinators is strict).

Generic Lenses

The simplest lens is the identity. It does nothing in the *get* direction and copies the whole abstract view in the *put* direction.

$$\boxed{\begin{array}{rcl} \texttt{id} \nearrow c & = & c \\ \texttt{id} \searrow (a,\, c) & = & a \end{array}}$$

Another simple lens is the constant lens, `const` $v\ d$, which transforms any view into the provided constant v in the *get* direction. In the *put* direction, it is defined iff the abstract view is equal to the constant one (by the PutGet law, this is the only thing it can do: if `const` $v\ d \searrow (a,\, c)$ is defined, then `const` $v\ d \nearrow (\texttt{const}\ v\ d \searrow (a,\, c)) = a = v$). In this case, the *put* function of `const` simply restores the old concrete view if it is available. If the old concrete view is missing, then there is no information as to which view should be returned; for this case, we supply the lens with a default view d.

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

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

$$\boxed{\begin{array}{rcll} (l;k) \nearrow c & = & k \nearrow l \nearrow c & \\ (l;k) \searrow (a,\, c) & = & l \searrow (k \searrow (a,\, l \nearrow c),\, c) & \text{if } c \neq \Omega \\ & & l \searrow (k \searrow (a,\, \Omega),\, \Omega) & \text{if } c = \Omega \end{array}}$$

The *get* direction applies the *get* function of l to yield a first abstract view, on which the *get* function of k is applied. In the *put* direction, if a concrete view is available, the *put* functions are applied in turn. First, the *put* function of k is used to put a into the concrete view that the *get* of k was applied to, i.e., $l \nearrow c$. The result of this *put* is then put into c using the *put* function of l. If the concrete view is missing, then k is used to put a into the missing view and then l to put the result again into the missing view.

Atomic Lenses

The `rename` lens changes the names of the immediate children of a view following some bijection b on names. 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'`.

$$\boxed{\begin{array}{rcl} (\texttt{rename}\ b) \nearrow c & = & \left\{ b(n) \mapsto c(n) \right. \\ (\texttt{rename}\ b) \searrow (a,\, c) & = & \left\{ b^{-1}(n) \mapsto a(n) \right. \end{array}}$$

The lens `hoist` n is used to remove superfluous edges. In the *get* direction, it expects a view that has only one child, which must be named n. It returns this child, removing the edge n. In the *put* direction, the value of the concrete view is ignored and a new view is created, with a single edge n pointing to the given abstract view.

$$\boxed{\begin{array}{rcll} (\texttt{hoist}\ n) \nearrow c & = & v & \text{if } c = \left\{ n \mapsto v \right. \\ & & \text{undef.} & \text{otherwise} \\ (\texttt{hoist}\ n) \searrow (a,\, c) & = & \left\{ n \mapsto a \right. & \end{array}}$$

The lens `pivot` n rearranges the structure at the top of a view.

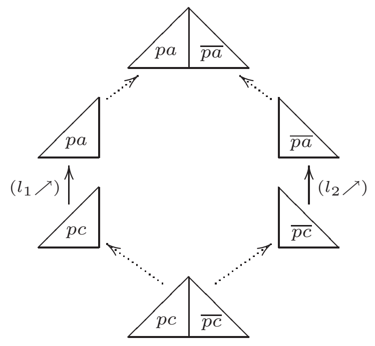
$$\left\{ \begin{array}{l} n \mapsto \left\{ k \mapsto \{ \right. \\ v \end{array} \right. \qquad \text{becomes} \qquad \left\{ k \mapsto v \right.$$

Intuitively, the value $\{k \mapsto \{\}\}$ under n represents a *key* k (a name uniquely identifying the view) for the rest of the view v. The *get* function of `pivot` returns a view where k points directly to v. The *put* function performs the reverse transformation, ignoring the old concrete view.

$$\boxed{\begin{array}{rcll} (\texttt{pivot}\ n) \nearrow c & = & \left\{ k \mapsto v \right. & \text{if } c = \left\{ \begin{array}{l} n \mapsto \left\{ k \mapsto \{ \right. \\ v \end{array} \right. \\ & & \text{undef.} & \text{otherwise} \\ (\texttt{pivot}\ n) \searrow (a,\, c) & = & \left\{ \begin{array}{l} n \mapsto \left\{ k \mapsto \{ \right. \\ v \end{array} \right. & \text{if } a = \left\{ k \mapsto v \right. \\ & & \text{undef.} & \text{otherwise} \end{array}}$$

Lens Combinators

The lens combinator `xfork` is used to apply different lenses to different parts of a view. Intuitively, it splits a view into two parts according to the names of its immediate children. It then applies one lens to the first part and a second lens to the other part and concatenates the results. Formally, `xfork` takes two predicates on names and two lenses as arguments. The *get* direction of `xfork` $pc\ pa\ l_1\ l_2$ can be visualized as follows (the concrete view is at the bottom):



The triangles labeled *pc* denote views whose immediate children have names satisfying *pc*; dotted arrows represent splitting or concatenating views. The result of applying $l_1 \nearrow$ to $c|_{pc}$ must satisfy the predicate *pa*, and similarly for l_2—that is, the lenses l_1 and l_2 are allowed to change the sets of names in the views they are given, but they must map from their own part of *pc* to their own part of *pa*. Conversely, in the *put* direction, l_1 must map from *pa* to *pc* and l_2 from $\overline{pa}$ to $\overline{pc}$.

$$\boxed{\begin{array}{lll}
(\texttt{xfork}\ pc\ pa\ l_1\ l_2) \nearrow c & = & \\
\quad (l_1 \nearrow c|_{pc}) + (l_2 \nearrow c|_{\overline{pc}}) & & \text{if (1)} \\
\quad \text{undef.} & & \text{otherwise} \\
(\texttt{xfork}\ pc\ pa\ l_1\ l_2) \searrow (a,\, c) & = & \\
\quad (l_1 \searrow (a|_{pa},\, c|_{pc})) + (l_2 \searrow (a|_{\overline{pa}},\, c|_{\overline{pc}})) & & \text{if (2)} \\
\quad \text{undef.} & & \text{otherwise} \\
 & & \\
(1) \quad \mathsf{dom}(l_1 \nearrow c|_{pc}) \subseteq pa & & \\
\quad\ \wedge\ \mathsf{dom}(l_2 \nearrow c|_{\overline{pc}}) \subseteq \overline{pa} & & \\
(2) \quad \mathsf{dom}(l_1 \searrow (a|_{pa},\, c|_{pc})) \subseteq pc & & \\
\quad\ \wedge\ \mathsf{dom}(l_2 \searrow (a|_{\overline{pa}},\, c|_{\overline{pc}})) \subseteq \overline{pc} & &
\end{array}}$$

We now define our last primitive lens and only iterator, `map`, which is a combinator that, in the *get* direction, applies a given lens l one level deeper in the view, leaving the top of the view intact.

$$\begin{cases} n_1 \mapsto v_1 \\ \dots \\ n_k \mapsto v_k \end{cases} \quad \text{becomes} \quad \begin{cases} n_1 \mapsto l \nearrow v_1 \\ \dots \\ n_k \mapsto l \nearrow v_k \end{cases}$$

We thus have $\mathsf{dom}((\texttt{map}\ l) \nearrow c) = \mathsf{dom}(c)$.

We now study the *put* direction. In the case where a and c have same domains, the definition is straighforward:

$$(\texttt{map}\ l) \searrow \left(\begin{cases} n_1 \mapsto v_1 \\ \dots \\ n_k \mapsto v_k \end{cases},\ \begin{cases} n_1 \mapsto v'_1 \\ \dots \\ n_k \mapsto v'_k \end{cases} \right) = \begin{cases} n_1 \mapsto l \searrow (v_1,\, v'_1) \\ \dots \\ n_k \mapsto l \searrow (v_k,\, v'_k) \end{cases}$$

We now consider the general case. We remark that if $(\texttt{map}\ l) \searrow (a,\, c)$ is defined, by rule PutGet, we should then have $(\texttt{map}\ l) \nearrow ((\texttt{map}\ l) \searrow (a,\, c)) = a$. Thus we necessarily have $\mathsf{dom}((\texttt{map}\ l) \searrow (a,\, c)) = \mathsf{dom}(a)$. We consider what becomes of children of a and c according to their names. Children bearing names that occur both in $\mathsf{dom}(a)$ and $\mathsf{dom}(c)$ are dealt with as before. Children bearing names that only occur in $\mathsf{dom}(c)$ are dropped. Children of a that have names that only appear in $\mathsf{dom}(a)$ need to be put into some view. Since no view is available, as there is no corresponding child in c, they are put into the missing view Ω (in the following views, $n_1, \dots, n_k \in \mathsf{dom}(c) \cap \mathsf{dom}(a)$, $r_1, \dots, r_p \in \mathsf{dom}(c) \setminus \mathsf{dom}(a)$, and $m_1, \dots, m_q \in \mathsf{dom}(a) \setminus \mathsf{dom}(c)$):

$$(\texttt{map}\ l) \searrow \left(\begin{cases} n_1 \mapsto v_1 \\ \dots \\ n_k \mapsto v_k \\ \\ \\ \\ m_1 \mapsto z_1 \\ \dots \\ m_q \mapsto z_q \end{cases},\ \begin{cases} n_1 \mapsto v'_1 \\ \dots \\ n_k \mapsto v'_k \\ r_1 \mapsto w'_1 \\ \dots \\ r_p \mapsto w'_p \\ \\ \\ \\ \end{cases} \right) = \begin{cases} n_1 \mapsto l \searrow (v_1,\, v'_1) \\ \dots \\ n_k \mapsto l \searrow (v_k,\, v'_k) \\ \\ \\ \\ m_1 \mapsto l \searrow (z_1,\, \Omega) \\ \dots \\ m_q \mapsto l \searrow (z_q,\, \Omega) \end{cases}$$

We now give the concise formal definition of `map` l. We recall that if any application of l to a child is undefined, then the whole lens `map` l is undefined.

$$\boxed{\begin{array}{lcl}
(\texttt{map}\ l) \nearrow c & = & \begin{cases} n \mapsto l \nearrow c(n) \quad n \in \mathsf{dom}(c) \end{cases} \\
(\texttt{map}\ l) \searrow (a,\, c) & = & \begin{cases} n \mapsto l \searrow (a(n),\, c(n)) \\ \qquad\qquad n \in \mathsf{dom}(a) \cap \mathsf{dom}(c) \\ n \mapsto l \searrow (a(n),\, \Omega) \\ \qquad\qquad n \in \mathsf{dom}(a) \setminus \mathsf{dom}(c) \end{cases}
\end{array}}$$

The `map` combinator does not obey the PutPut law. Consider a lens l and $(a, c) \in \mathsf{dom}(l \searrow)$ such that $l \searrow (a,\, c) \neq l \searrow (a,\, \Omega)$. We have

$$\begin{array}{cl}
 & (\texttt{map}\ l) \searrow (\{\texttt{n} \mapsto a\},\, ((\texttt{map}\ l) \searrow (\{\},\, \{\texttt{n} \mapsto c\}))) \\
= & (\texttt{map}\ l) \searrow (\{\texttt{n} \mapsto a\},\, \{\}) \\
= & \{\texttt{n} \mapsto l \searrow (a,\, \Omega)\} \\
\neq & \{\texttt{n} \mapsto l \searrow (a,\, c)\} \\
= & (\texttt{map}\ l) \searrow (\{\texttt{n} \mapsto a\},\, \{\texttt{n} \mapsto c\}.)
\end{array}$$

Intuitively, there is a difference between modifying a child n and removing then adding it, as in the first case the initial value of the child is used if available, while it disappears in the second case after the child is removed.

Another interesting point is the relation between the `map` lens combinator and the missing view Ω. The *put* function of every other lens combinator only results in a *put* into the missing view 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 view may result in the application of the *put* of l to Ω if a has some children that are not in c. In order to deal with such missing children, we first tried providing a default concrete view for `map`, which would be used when no concrete view was available. However, we discovered through experimentation that in many cases it is difficult to find one default concrete view that fits all possible abstract views, especially because of `xfork` (where different lenses are applied to different parts of the view) and recursion (where the depth of a view is unknown). We thus decided to parameterize this default concrete view by the abstract view and the lens. We then discovered that most primitive lenses ignore the concrete view when defining the *put* function, as enough information is available in the abstract view. The natural choice for a concrete view parameterized by a and l was thus $l \searrow (a,\, \Omega)$, for some special view Ω. The only lens for which the *put* funtion needs to be defined on Ω is `const`, as it is the only lens that discards information. To this end, the `const` lens expects a default view d. This approach is much more local, as one only needs to provide a default view where information is discarded.

4.3 Derived Lenses

In this section, we define some useful lenses derived from the primitive ones of the previous section. Most of the lenses of this section and of section 4.4 are used in the example of section 5. We recall that these derived lenses are all well behaved by construction.

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

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

We may now define a lens that only retains the child of a view satisfying a predicate p:

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

In the *get* direction, this lens takes a concrete view, keeps the part of the view whose children have names in p (using the lens `id`), and throws away the rest of the view (using the lens `const` $\{\}\ d$). The default view d is used when putting an abstract view into a missing concrete view. It provides a default for the information that does not appear in the abstract view and is necessary to build a concrete view.

Another way to filter, or prune, a view is to explicitly specify a name that should be removed from the view:

$$\texttt{prune}\ n\ d = \texttt{fork}\ \overline{\{n\}}\ \texttt{id}\ (\texttt{const}\ \{\}\ \{n \mapsto d\})$$

This lens is very similar to `filter`, with two differences: the name given is the one to be removed, thus the predicate "all other names" $\overline{\{n\}}$ must be built, and the default view is the one to go under n if the concrete view is missing.

The following lens is useful to focus on a single child n:

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

In the *get* direction, it filters away all other children, then removes the edge n to return the corresponding view. As usual, the default view is only used in case of creation. It is used as a default for the children filtered away.

It is often useful to restrict the use of `map` to a subset of all children of a view using a predicate p.

$$\texttt{mapp}\ p\ l = \texttt{fork}\ p\ (\texttt{map}\ l)\ \texttt{id}$$

This lens splits the view in two according to the predicate p, applies `map` to the first half, and does not modify the rest.

In order to apply different lenses to different parts of the view, and concatenate the results, we define the recursive lens `dispatch`, which takes a list of tuples each containing a concrete predicate, an abstract predicate, and a lens, as:

$$\begin{array}{r}\texttt{dispatch}\ [\,] = \texttt{id}\\ \texttt{dispatch}\ (pc, pa, l) :: rest = \texttt{xfork}\ pc\ pa\ l\ (\texttt{dispatch}\ rest)\end{array}$$

In the *get* direction, `dispatch` considers the first tuple pc, pa, l. It splits the concrete view accoring to pc and applies l to it. It recurses with the rest of the tuples and the rest of the view, and concatenates the results back together. A typical use of `dispatch` is as a conditional, assuming all the lenses it uses return the empty view when given the empty view (see for instance the `item` lens in Figure 3).

4.4 Lists

Many data formats make heavy use of ordered data, or lists. We describe in this section how we represent lists, using the usual cons cell encoding, and introduce some derived lenses to manipulate them.

4.4.1 Definition: A view v is a list iff it is the empty view or if it has exactly two children, one named $*h$ and another named $*t$, such that $v(*t)$ is a list.

In the following, we use the lighter notation $[v_1 \dots v_n]$ (writing, in displays, v_1 through v_n vertically and dropping the closing bracket) for the view

$$\left\{\begin{array}{l}\texttt{*h} \mapsto \texttt{v}_1\\ \texttt{*t} \mapsto \left\{\begin{array}{l}\texttt{*h} \mapsto \texttt{v}_2\\ \texttt{*t} \mapsto \left\{\dots \mapsto \left\{\begin{array}{l}\texttt{*h} \mapsto \texttt{v}_n\\ \texttt{*t} \mapsto \{\end{array}\right.\right.\end{array}\right.\end{array}\right.$$

We now define some lenses to work on lists. The first ones extract the head or the tail of the list.

$$\begin{array}{l}\texttt{hd}\ d = \texttt{focus}\ \{\texttt{*h}\}\ \{\texttt{*t} \mapsto d\}\\ \texttt{tl}\ d = \texttt{focus}\ \{\texttt{*t}\}\ \{\texttt{*h} \mapsto d\}\end{array}$$

The lens `hd` expects a default view which will be the tail of the created view if the concrete view is missing. In the get direction, the lens `hd` returns the view under name `*h`. Lens `tl` works similarly, with two exceptions: it expects the default view that will be put under the head in case of creation, and it returns the tail of the list, which is a list.

We define a lens that iterates over a list, applying its argument to every element of the list.

$$\texttt{map\_list}\ l = \texttt{mapp}\ \{\texttt{*h}\}\ l;\ \texttt{mapp}\ \{\texttt{*t}\}\ (\texttt{map\_list}\ l)$$

This lens simply applies l to every child named `*h`, and recurses on every child named `*t`.

We now define a lens which transforms a list into a "bush," flattening it. Such a lens may only be defined on lists of views that have pairwise-disjoint domains. Depending on whether the order matters if creation occurs, two such lenses may be defined. One lens, `flatten`, does not care about the order when putting an abstract view into a missing concrete view, and the resulting list has an arbitrary order. The other lens, `hoist_list`, expects a list of (pairwise-disjoint) predicates describing the domains of the views in the list. If an abstract view is put into a missing concrete view, the resulting list will obey the order specified by the predicate list. We now define `hoist_list`.

$$\begin{array}{rl}\texttt{hoist\_list}\ [\,] = & \texttt{id}\\ \texttt{hoist\_list}\ p :: rest = & \texttt{xfork}\ \{\texttt{*h}\}\ p\\ & (\texttt{hoist}\ \{\texttt{*h}\})\\ & (\texttt{hoist}\ \{\texttt{*t}\};\ \texttt{hoist\_list}\ rest)\end{array}$$

5. A BOOKMARK LENS

In this section, we develop an extended example of programming in Hocus Focus. The example comes from a demo application of our universal data synchronization framework, Harmony [1], in which bookmark information from diverse browsers, including Internet Explorer, Netscape, OmniWeb, Safari, and others 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 relatives.

The overall path taken by the bookmark data through the Harmony system can be described as follows. Harmony first uses a generic HTML reader to transform the HTML bookmark file into an isomorphic concrete view. This concrete view is then transformed, using the *get* direction of the `bookmark` lens, into an abstract "generic bookmark view." The abstract view is synchronized with some other abstract bookmark view (obtained from some other bookmark file by transforming its native format using an appropriate lens, not

```
{'' ->
 [{html -> {'' ->
   [{head -> {'' -> [{title ->
                     {'' ->
                      [{PCDATA -> Bookmarks}]}}]}}
    {body -> {'' ->
     [{h3 -> {'' ->
              [{PCDATA -> 'Bookmarks Folder'}]}}
      {dl -> {'' ->
       [{dt -> {'' ->
         [{a -> {'' -> [{PCDATA -> Google}]
                add_date -> 1032458036
                href -> http://www.google.com}}]}}
        {dd -> {'' ->
         [{h3 -> {'' -> [{PCDATA ->
                         'Conferences Folder'}]}}
          {dl -> {'' ->
           [{dt -> {'' ->
              [{a ->
                {'' -> [{PCDATA -> ICFP}]
                 add_date -> 1032528670
                 href -> http://www.it.uu.se/pli03/
                }}]}}]}}]}}]}}]}}]}}]}}]}
```

Figure 1: Bookmarks (concrete view)

```
{name -> 'Bookmarks Folder'
 contents ->
  [{link -> {name -> Google
             url -> http://www.google.com}}
   {folder ->
     {name -> 'Conferences Folder'
      contents ->
       [{link ->
         {name -> ICFP
          url -> http://www.it.uu.se/pli03/}}]}}]}
```

Figure 2: Bookmarks (abstract view)

shown here), yielding a new abstract view, which is transformed into a new concrete view by passing it back through the *put* direction of the `bookmark` lens (supplying the original concrete view as the second argument). Finally, the new concrete view is written back out to the filesystem as an HTML file. We now discuss these transformations in more detail.

Abstractly, bookmark data has the following recursive structure: an *item* is either a *link*, with a `name` and a `url`, or a *folder* with a `name` and a `contents`, which is a list of items.

In HTML, 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.

```
link = rename {'dt' = 'link'};
       map (hoist '';
            hd {};
            hoist 'a';
            rename {'href' = 'url'  '' = 'name'};
            prune 'add_date' {today};
            mapp {'name'} (hd {}; hoist 'PCDATA'))

folder = rename {'dd' = 'folder'};
         map (hoist ''; folder_contents)

folder_contents =
   hoist_list [{'h3'} {'dl'}];
   rename {'h3' = 'name'  'dl' = 'contents'};
   mapp {'name'} (hoist ''; hd {}; hoist 'PCDATA');
   mapp {'contents'} (hoist ''; map_list item)

item =
  dispatch [({'dd'},{'folder'},folder)
            ({'dt'},{'link'},link)]

bookmarks =
  hoist ''; hd {}; hoist 'html'; hoist '';
  tl {'head' -> {'' -> [{'title' -> {'' ->
                 [{'PCDATA' -> 'Bookmarks'}]}}]}};
  hd {}; hoist 'body'; hoist '';
  folder_contents
```

Figure 3: Bookmark lenses

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

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

into a view of the following form:

$$\left\{ \texttt{tag} \mapsto \left\{ \begin{array}{l} \texttt{attr1} \mapsto \texttt{val1} \\ \quad\vdots \\ \texttt{attrm} \mapsto \texttt{valm} \\ \texttt{''} \mapsto \left[\begin{array}{l} \langle\texttt{subelt1}\rangle \\ \quad\vdots \\ \langle\texttt{subeltn}\rangle \end{array} \right. \end{array} \right. \right.$$

Note that the sub-elements are placed in a *list* under a child named `''` (empty string). 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 view, leaving it up to whatever lens is applied to the view to throw away ordering information where

Lens expression	Resulting abstract view (from ’get’)
`id`	`{dt -> {’’ -> [{a -> {’’ -> [{PCDATA -> Google}] add_date -> 1032458036 href -> http://www.google.com}}]}}`
`rename {’dt’ = ’link’}`	`{link -> {’’ -> [{a -> {’’ -> [{PCDATA -> Google}] add_date -> 1032458036 href -> http://www.google.com}}]}}`
`rename {’dt’ = ’link’}; map (hoist ’’)`	`{link -> [{a -> {’’ -> [{PCDATA -> Google}] add_date -> 1032458036 href -> http://www.google.com}}]}`
`rename {’dt’ = ’link’}; map (hoist ’’; hd {})`	`{link -> {a -> {’’ -> [{PCDATA -> Google}] add_date -> 1032458036 href -> http://www.google.com}}}`
`rename {’dt’ = ’link’}; map (...; hd {}; hoist ’a’)`	`{link -> {’’ -> [{PCDATA -> Google}] add_date -> 1032458036 href -> http://www.google.com}}`
`rename {’dt’ = ’link’}; map (... ; hoist ’a’; rename {’’ = ’name’, ’href’ = ’url’})`	`{link -> {name -> [{PCDATA -> Google}] add_date -> 1032458036 url -> http://www.google.com}}`
`rename {’dt’ = ’link’}; map (...; rename {’’ = ’name’, ’href’ = ’url’}; prune ’add_date’ {today})`	`{link -> {name -> [{PCDATA -> Google}] url -> http://www.google.com}}`
`rename {’dt’ = ’link’}; map (...; prune ’add_date’ {today}; mapp {’name’} (hd {}))`	`{link -> {name -> {PCDATA -> Google} url -> http://www.google.com}}`
`rename {’dt’ = ’link’}; map (...; mapp {’name’} (hd {}; hoist ’PCDATA’))`	`{link -> {name -> Google url -> http://www.google.com}}`

Figure 4: Building up a link lens incrementally.

it is not needed; the `flatten` lens described in Section 4.4 provides one convenient way to do this.) A “leaf” of the HTML document—i.e., a “parsed character data” element containing a text string `str`—is converted to a view of the form `{PCDATA -> str}`. Figure 1 shows a view representing a small bookmark file.

The transformation from this concrete view to the abstract bookmark view shown in Figure 2 is implemented by means of the collection of lenses shown in Figure 3. 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.’ The *put* direction, then, restores the original names and rebuilds the necessary structure.

Notation: In the definitions of the lenses, names are enclosed in quotes to distinguish them from variables. When displaying views (here, as in the rest of the paper), we often drop these quotes for readability.

Normally we develop these lenses incrementally, slowly massaging the views into the correct shape. Figure 4 shows this process in developing the `link` lens above by transforming the representation of the HTML `<dt>` element containing a link into the desired abstract form. At each level of the structure, tree branches are relabeled with `rename`, undesired structure is removed with `prune`, `hoist`, and/or `hd`, and then work is continued at a lower level via `map` or `mapp`.

Similarly, the `folder` lens renames the `<dd>` tag to `folder`, then proceeds to separate out the folder name and its contents, stripping out undesired structure where necessary. (We have separated out the content processing into the `folder_contents` lens for code reuse, since the top-level bookmark file itself looks *almost* like a folder.) Note the use of `hoist_list` instead of `flatten` to access the `<h3>` and `<dl>` tags containing the folder name and contents respectively; although the order of these two tags does not matter to us, it matters to Mozilla, so we want to ensure that the *put* direction of the lens restores them to their proper place. Finally, we use `map_list` 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, and we use `dispatch` to call the correct sublens.

Finally, the main lens is `bookmarks`, which (in the *get* direction) takes a whole concrete bookmark view, strips off the boilerplate header information using a combination of `hoist`, `hd`, and `tl`, and then invokes `folder_contents` to deal with the rest.

6. RELATED WORK

Hocus Focus is the product of a long odyssey through a large design space, driven by the practical needs of the Harmony system as it evolved. 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 led to a formulation that is arguably simpler (transforming states rather than “update functions”) and more refined in its treatment of partiality. Our formulation is also novel in considering the issue of continuity (which was not addressed in earlier work), 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.

6.1 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 [7] developed an elegant semantic characterization of the update translation problem, introducing the notion of *complement* of a view, which must include at least all information from the database missing from the view. When a complement is fixed, there exists at most one update of the database that reflects the update on the view while leaving the complement unmodified—i.e., that *translates updates under a constant complement*. In general, a given view may have many complements, each corresponding to a possible strategy for translating view updates to database updates. The problem of translating view updates then becomes a problem of finding, for a given view, a "suitable" complement.

Gottlob, Paolini, and Zicari [16] offered a more refined theory based on a syntactic translation of view updates. They identified a hierarchy of restricted cases of their framework, the most permissive form being their "dynamic views" and the most restrictive, called "cyclic views with constant complement," being formally equivalent to Bancilhon and Spyratos's update translators.

In [26] we establish a precise correspondence between our definition of 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 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, the sets of well-behaved and very-well-behaved lenses need to be restricted to subsets that are "total" in a suitable sense. More details can be found in [26].

6.2 Updates for Relational Views

Research on view update translation in the database literature has tended to focus on taking an existing language (e.g., relational algebra) for defining *get* functions and then considering how to infer (either automatically or with some programmer assistance) corresponding *put* functions. By contrast, we have designed a completely new language in which the definitions of *get* and *put* go hand-in-hand. Our approach can be described both as more demanding (because we deal with trees) and as more straightforward (because we do not attempt to deal with joins, a major source of update ambiguity in the relational world), compared to most of the classical database work. We briefly review the most relevant of this work.

Masunaga [21] described an automated algorithm for translating updates on views defined by relational algebra. The core idea was to annotate where the "semantic ambiguities" arise, indicating they must be resolved either with knowledge of underlying database semantic constraints or by interactions with the user.

Keller [19] outlined 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. Keller [20] later 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 [8] described a scheme for interactively constructing update translators for object-based views of relational databases.

Medeiros and Tompa [22] presented a design tool for exploring the effects of choosing a view update policy. This tool shows the update translation for update requests supplied by the user; by considering all possible valid concrete states, the tool predicts whether the desired update would in fact be reflected back into the view after applying the translated update to the concrete database.

Atzeni and Torlone [6, 5] describe a tool for translating views, and observe 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 limit themselves to mappings where the concrete and abstract views are isomorphic.

A variety of complexity results have been shown for different 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 body of work that is sometimes mentioned in connection with view update translation is the problem of *incremental view maintainance* (e.g., [4])—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 our problem of translating view updates to updates on an underlying concrete structure.

6.3 Languages for View Update

In the programming languages literature, laws similar to our lens laws (but somewhat simpler, since they deal only with total *get* and *put* functions) appear in Oles' category of "state shapes" [25] and in Hofmann and Pierce's work on "positive subtyping" [18]. Another related idea, proposed by Wadler [30], extended algebraic pattern matching to abstract data types using programmer-supplied *in* and *out* operators. This is essentially the special case of our lenses in which the *get* and *put* functions must always form an isomorphism.

Abiteboul, Cluet, and Milo [2] defined a declarative language for describing *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; however, this process requires an isomorphism between the two data formats (again, a special case of our lenses).

The same authors [3] later defined a system for bi-directional transformations based around the concept of *structuring schemas* (parse grammars annotated with semantic information). Thus their *get* involved parsing, whereas their *put* consisted of "unparsing." Again, to resolve ambiguous abstract updates, they restrict themselves to *lossless* grammars that define an isomorphism between concrete and abstract views.

Ohori and Tajima [24] develop a statically-typed polymorphic record calculus for defining views on object-oriented databases. They specifically restrict which fields of a view are updatable, allowing only those with a ground (simple) type to be updated, whereas our lenses can accomodate structural updates as well.

6.4 Updates and Trees

There have been many proposals for query languages for trees (e.g., XQuery [15]), but most of these do not consider the view update problem, and those that do tend not to consider ambiguous updates.

For example, Braganholo, Heuser, and Vittori [13] study 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 view updates unambiguous.

Tatarinov, Ives, Halevy, and Weld [29] describe 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

Our interest in bi-directional tree transformations arose in the context of our data synchronization framework, Harmony. We plan to develop many more synchronizers (e.g., ones for calendars/appointments, email, structured text, filesystems, and more) that will further exercise Hocus Focus's set of basic operators and perhaps suggest new ones.

This process should shed light on some intriguing follow-on questions to our work here. For example, is the set of tree lenses expressible in (some variant or extension of) Hocus Focus "complete" in some natural sense? Can we characterize the complexity of Hocus Focus programs? Is there an algebraic theory of lens combinators that would underpin optimization of Hocus Focus programs in the same way that the relational calculus and its algebraic theory are used to optimize relational database queries?

From a programming point of view, a static type system for views and Hocus Focus programs would be very useful, particularly when building very complicated lenses. Such a type system would allow the programmer to codify the well-formedness of certain views (*e.g.*, "this tree should have exactly one child, named *foo* (because I want to `hoist` it)").

It would be useful to generate lens programs 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 could perhaps do most of the work for a programmer wanting to add synchronization support for a new application (where the abstract form was already defined, for example), leaving just a few spots to fill in.

A growing body of work deals with the problem of translating between heterogeneous representations of similar data to enable different applications to cooperate. Such representations (e.g. directed graphs and XML) are a superset of the concrete views (namely trees) that we handle. Although much of this work (one way transformations that do not address the update problem) is not directly relevant to Hocus Focus, it may be useful as a set of examples against which to compare the expressiveness of Hocus Focus. Further, one class of proposed solutions uses schema matching (as well as representation mapping and model mapping) to perform all or part of the translation automatically ([23, 14, 28] include useful introductions). We may be able to employ the similar methods to automatically construct lenses to translate between two given views.

Finally, it would be intriguing to experiment with instantiating our semantic framework with relations instead of trees, thereby establishing a closer link with existing research in the database community.

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 Unison (a file synchronization tool on which many of Harmony's intuitions are based and of which Trevor was a co-designer) should be synchronizing XML. Insightful conversations with Steve

Zdancewic, Cyrus Najmabadi, Kate Moore, William Lovas, Sanjeev Khanna, Zack Ives, and Owen Gunden 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, 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.

8. REFERENCES

[1] Harmony project. `http://www.cis.upenn.edu/~bcpierce/harmony/`.

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

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

[4] 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.

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

[6] 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.

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

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

[9] 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] A. Doan. *Learning to map between structured representations of Data*. PhD thesis, 2002.

[15] P. Fankhauser, M. Fernández, A. Malhotra, M. Rys, J. Siméon, and P. Wadler. XQuery 1.0 Formal Semantics. `http://www.w3.org/TR/query-semantics/`, 2001.

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

[17] M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. A language for bi-directional tree transformations. Technical Report MS-CIS-03-08, University of Pennsylvania, 2003.

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

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

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

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

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

[23] T. Milo and S. Zohar. Using schema matching to simplify heterogeneous data translation. In *VLDB'98*, 1998.

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

[25] 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.

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

[27] B. C. Pierce and J. Vouillon. Unison: A file synchronizer and its specification. Technical report; available through `http://www.cis.upenn.edu/~bcpierce`, 2001.

[28] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10(4):334–350, 2001.

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

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

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