Retentive Lenses

Zirun Zhu[2,4], Zhixuan Yang[5], Hsiang-Shang Ko[1], and Zhenjiang Hu[2,3]

[1] Institute of Information Science, Academia Sinica, Taiwan
[2] National Institute of Informatics, Japan
[3] Peking University, China
[4] SOKENDAI (The Graduate University for Advanced Studies), Japan
[5] Sourcebrella Inc., China
zhu@nii.ac.jp, yangzhixuan@sbrella.com, hsiang-shang@nii.ac.jp, huzj@pku.edu.cn

Abstract. Based on Foster et al.'s lenses, various bidirectional programming languages and systems have been developed for helping the user to write correct data synchronisers. The two well-behavedness laws of lenses, namely Correctness and Hippocraticness, are usually adopted as the guarantee of these systems. While lenses are designed to retain information in the source when the view is modified, well-behavedness says very little about the retaining of information: Hippocraticness only requires that the source be unchanged if the view is not modified, and nothing about information retention is guaranteed when the view is changed. To address the problem, we propose an extension of the original lenses, called *retentive lenses*, which satisfy a new Retentiveness law guaranteeing that if parts of the view are unchanged, then the corresponding parts of the source are retained as well. As a concrete example of retentive lenses, we present a domain-specific language for writing tree transformations; we prove that the pair of *get* and *put* functions generated from a program in our DSL forms a retentive lens. We demonstrate the practical use of retentive lenses and the DSL by presenting case studies on code refactoring, Pombrio and Krishnamurthi's resugaring, and XML synchronisation.

Keywords: lenses, bidirectional programming, domain-specific languages

1 Introduction

We often need to write pairs of transformations to synchronise data. Typical examples include view querying and updating in relational databases [**Bancilhon1981Update**] for keeping a database and its view in sync, text file format conversion [**Macfarlane2013Pandoc**] (e.g. between Markdown and HTML) for keeping their content and common formatting in sync, and parsers and printers as front ends of compilers [**Rendel2010Invertible**] for keeping program text and its abstract representation in sync. Asymmetric *lenses* [**Foster2007Combinators**] provide a framework for modelling such pairs of programs and discussing what laws they should satisfy; among such laws, two *well-behavedness* laws (explained below) play a fundamental role. Based on lenses, various bidirectional programming languages and systems (Sect. 6) have been developed for helping the user to write correct synchronisers, and the well-behavedness laws have been adopted as the minimum—and in most cases the only—laws to guarantee. In this paper, we argue that well-behavedness is not sufficient, and a more refined law, which we call *Retentiveness*, should be developed.

To see this, let us first review the definition of well-behaved lenses, borrowing some of **Stevens2008Bidirectional**'s terminologies [**Stevens2008Bidirectional**]. Lenses are used to synchronise two pieces of data respectively of types S and V, where S contains more information and is called the *source* type, and V contains less information and is called the *view* type. Here, being synchronised means that when one piece of data is changed, the other piece of data should also be changed such that consistency is *restored* among them, i.e. a *consistency relation* R defined on S and V is satisfied. Since S contains more information than V, we expect that there is a function $\mathit{get} : S \rightarrow V$ that extracts a consistent view from a source, and this *get* function serves as a consistency restorer in the source-to-view direction: if the source is changed, to restore consistency it suffices to use *get* to recompute a new view. This *get* function should coincide with R extensionally [**Stevens2008Bidirectional**]—that is, $s : S$ and $v : V$ are related by R if and only if $\mathit{get}(s) = v$. (Therefore it is only sensible to consider functional consistency relations in the asymmetric setting.) Consistency restoration in the other direction is performed by another function $\mathit{put} : S \times V \rightarrow S$, which produces an updated source that is consistent with the input view and can retain some information of the input source. Well-behavedness consists of two laws regarding the restoration behaviour of *put* with respect to *get* (i.e. the consistency relation R):

$$\mathit{get}(\mathit{put}(s, v)) = v \qquad \text{(Correctness)}$$
$$\mathit{put}(s, \mathit{get}(s)) = s \qquad \text{(Hippocraticness)}$$

Correctness states that necessary changes must be made by *put* such that the updated source is consistent with the view; Hippocraticness says that if two pieces of data are already consistent, *put* must not make any change. A pair of *get* and *put* functions, called a *lens*, is *well-behaved* if it satisfies both laws.

Despite being concise and natural, these two properties do not sufficiently characterise the result of an update performed by *put*, and well-behaved lenses may exhibit unintended behaviour regarding what information is retained in the updated source. Let us illustrate this with a very simple example, in which *get* is a projection function that extracts the first element from a tuple of an integer and a string. (Hence a source and a view are consistent if the first element of the source tuple is equal to the view.)

```
get :: (Int, String) -> Int
get (i, s) = i
```

Given this `get`, we can define $\texttt{put}_1$ and $\texttt{put}_2$, both of which are well-behaved with this `get` but have rather different behaviour: $\texttt{put}_1$ simply replaces the integer of the source tuple with the view, while $\texttt{put}_2$ also sets the string empty when the source tuple is not consistent with the view.

```
put₁ :: (Int, String) -> Int -> (Int, String)
put₁ (i, s) i' = (i', s)

put₂ :: (Int, String) -> Int -> (Int, String)
put₂ src    i'  | get src == i'  = src
put₂ (i, s) i'  | otherwise      = (i',  "")
```

From another perspective, put$_1$ retains the string from the old source when performing the update, while put$_2$ chooses to discard that string—which is not desired but 'perfectly legal', for the string does not contribute to the consistency relation. In fact, unexpected behaviour of this kind of well-behaved lenses could even lead to disaster in practice. For instance, relational databases can be thought of as tables consisting of rows of tuples, and well-behaved lenses used for maintaining a database and its view may erase important data after an update, as long as the data does not contribute to the consistency relation (in most cases this is because the data is simply not in the view). This fact seems fatal, as asymmetric lenses have been considered a satisfactory solution to the longstanding view update problem (stated at the beginning of Foster et al.'s seminal paper [**Foster2007Combinators**]).

The root cause of the information loss (after an update) is that while lenses are designed to retain information, well-behavedness actually says very little about the retaining of information: the only law guaranteeing information retention is Hippocraticness, which merely requires that the *whole* source should be unchanged if the *whole* view is. In other words, if we have a very small change on the view, we are free to create any source we like. This is too 'global' in most cases, and it is desirable to have a law that makes such a guarantee more 'locally'.

To have a finer-grained law, we propose *retentive lenses*, an extension of the original lenses, which can guarantee that if parts of the view are unchanged, then the corresponding parts of the source are retained as well. Compared with the original lenses, the *get* function of a retentive lens is enriched to compute not only the view of the input source but also a set of *links* relating corresponding parts of the source and the view. If the view is modified, we may also update the set of links to keep track of the correspondence that still exists between the original source and the modified view. The *put* function of the retentive lens is also enriched to take the links between the original source and the modified view as input, and it satisfies a new law, *Retentiveness*, which guarantees that those parts in the original source having correspondence links to some parts of the modified view are retained at the right places in the updated source.

The main contributions of the paper are as follows:

– We develop a formal definition of retentive lenses for trees defined by algebraic data types (Sect. 3).
– We present a simple domain-specific language (DSL) tailored for developing tree synchronisers and prove that any program written in our DSL gives rise to a retentive lens (Sect. 4).d
– We demonstrate the usefulness of retentive lenses in practice by presenting case studies on code refactoring, XML synchronisation, and resugaring (Sect. 5).

We will start from a high-level sketch of what retentive lenses do (Sect. 2), and after presenting the technical contents, we will discuss related work (Sect. 6)

```
type Annot  =  String                   getE :: Expr -> Arith
data Expr   =  Plus  Annot Expr Term    getE (Plus   _ e  t) =
            |  Minus Annot Expr Term      Add (getE e) (getT t)
            |  FromT Annot Term         getE (Minus  _ e  t) =
                                          Sub (getE e) (getT t)
data Term   =  Lit   Annot Int          getE (FromT  _    t) = getT t
            |  Neg   Annot Term
            |  Paren Annot Expr         getT :: Term -> Arith
                                        getT (Lit     _ i  ) = Num i
data Arith  =  Add Arith Arith          getT (Neg     _ t  ) =
            |  Sub Arith Arith            Sub (Num 0) (getT t)
            |  Num Int                  getT (Paren   _ e  ) = getE e
```

Fig. 1. Data types for concrete and abstract syntax of arithmetic expressions and the consistency relations between them as `getE` and `getT` functions in Haskell.

regarding various alignment strategies for lenses, provenance and origin between two pieces of data, and operational-based bidirectional transformations, and conclude the paper (Sect. 7).

2 A Sketch of Retentiveness

We will use the synchronisation of concrete and abstract representations of arithmetic expressions as the running example throughout the paper. The representations are defined in Fig. 1 (in Haskell). The concrete representation is either an expression of type `Expr`, containing additions and subtractions; or a term of type `Term`, including numbers, negated terms, and expressions in parentheses. Moreover, all the constructors have an annotation field of type `Annot` for holding comments. The two concrete types `Expr` and `Term` coalesce into the abstract representation type `Arith`, which does not include annotations, explicit parentheses, and negations—negations are considered *syntactic sugar* and represented in the AST by `Sub`. The consistency relations between CSTs and ASTs are defined in terms of the *get* functions—`e :: Expr` (resp. `t :: Term`) is consistent with `a :: Arith` exactly when `getE e = a` (resp. `getT t = a`).

As mentioned in Sect. 1, the core idea of Retentiveness is to use links to relate *parts* of the source and view. For trees, an intuitive notion of a 'part' is a *region*, by which we mean a top part of a subtree described by a *region pattern* that consists of wildcards, constructors, and constants. For example, in Fig. 2, the grey areas are some of the possible regions: The topmost region in `cst` is described by the region pattern `Plus "a plus" _ _`, which says that the region includes the `Plus` node and the annotation `"a plus"`, but not the other two subtrees with roots `Minus` and `Neg` matched by the wildcards.

After decomposing source and view trees into regions, we can then put in links to record the correspondences between source and view regions. In particular, we expect that when the source and view are consistent, there should be enough

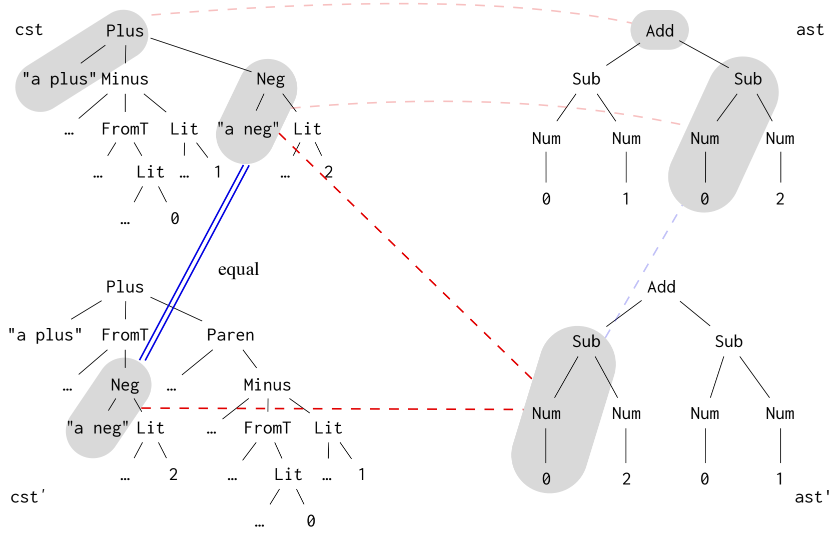


Fig. 2. Regions, links, and the triangular guarantee.

links—which we call the *consistency links*—that fully describe how all the source and view regions correspond. In Fig. 2, for example, the light dashed red links between the source `cst` and the view `ast = getE cst` are two of the consistency links. (Other consistency links are omitted for clarity.) The topmost region of pattern `Plus "a plus" _ _` in `cst` corresponds to the topmost region of pattern `Add _ _` in `ast`, and the region of pattern `Neg "a neg" _` in the right subtree of `cst` corresponds to the region of pattern `Sub (Num 0) _` in `ast`. The *get* function of a retentive lens will be responsible for producing such consistency links.

When the view is modified, the links between the source and view should also be modified to reflect the latest correspondences between regions. For example, in Fig. 2, if we change `ast` to `ast'` by swapping the two subtrees under `Add`, then there should be a 'diagonal' link recording that the `Neg "a neg" _` region and the `Sub (Num 0) _` region are still 'locally consistent' despite the fact that the source and view as a whole are no longer consistent. We will describe a particular way of computing such diagonal links from consistency links in Sect. 5.1.

When it is time to put the modified view back into the source, the diagonal links between the source and the modified view are used to guide what regions should be brought from the old source to the new one. In addition to the source and view, the *put* function of a retentive lens will also take a collection of (diagonal) links, and provide what we call the *triangular guarantee*, as illustrated in Fig. 2: When updating `cst` with `ast'`, the region `Neg "a neg" _` (i.e. syntactic sugar negation) connected by the red dashed diagonal link is guaranteed to be preserved in the result `cst'` (as opposed to changing it to a `Minus`), and `getE cst'` will link the preserved region to the same region `Sub (Num 0) _` of `ast'`. The Retentiveness law will be a formalisation of the triangular guarantee.

3 Formal Definitions

Let us now formalise what we described in Sect. 2. Apart from the definition of retentive lenses (Sect. 3.1), we will also briefly discuss how retentive lenses compose (Sect. 3.2).

3.1 Retentive Lenses

We start with some notations. Relations from set A to set B are subsets of $A \times B$, and we denote the type of these relations by $A \sim B$. Given a relation $r : A \sim B$, define its *converse* $r^\circ : B \sim A$ by $r^\circ = \{ (b,a) \mid (a,b) \in r \}$, its *left domain* by $\textsc{ldom}(r) = \{ a \in A \mid \exists b.\ (a,b) \in r \}$, and its *right domain* by $\textsc{rdom}(r) = \textsc{ldom}(r^\circ)$. The composition $r \cdot s : A \sim C$ of two relations $r : A \sim B$ and $s : B \sim C$ is defined as usual by $r \cdot s = \{ (a,c) \mid \exists b.\ (a,b) \in r \wedge (b,c) \in s \}$. The type of partial functions from A to B is denoted by $A \nrightarrow B$. The *domain* $\textsc{dom}(f)$ of a function $f : A \nrightarrow B$ is the subset of A on which f is defined; when f is total, i.e. $\textsc{dom}(f) = A$, we write $f : A \rightarrow B$. We will allow functions to be implicitly lifted to relations: a function $f : A \nrightarrow B$ also denotes a relation $f : B \sim A$ such that $(f\,x, x) \in f$ for all $x \in \textsc{dom}(f)$. This flipping of domain and codomain (from $A \nrightarrow B$ to $B \sim A$) makes function composition compatible with relation composition: a function composition $g \circ f$ lifted to a relation is the same as $g \cdot f$, i.e. the composition of g and f as relations.

We will work within a universal set *Tree* of trees, which is inductively built from all possible finitely branching constructors. (The semantics of an algebraic data type is then the subset of *Tree* that consists of those trees built with only the constructors of the data type.) We will also use the set *Pattern* of all possible region patterns and a set *Path* of all possible paths for navigating from the root of a tree to one of its subtrees. The exact representation of paths is not crucial: paths are only required to support some standard operations—for example, there should be a function $sel : Tree \times Path \nrightarrow Tree$ such that $sel(t,p)$ is the subtree of t at the end of path p (starting from the root), or undefined if p does not exist in t—and we will mention these operations in the rest of the paper as the need arises. But, to give concrete examples later, we provide one particular representation: a path can be a list of natural numbers indicating which subtree to go into at each node—for instance, starting from the root of `cst` in Fig. 2, the empty path `[]` points to the root node `Plus`, the path `[0]` points to `"a plus"` (which is the first subtree under the root), and the path `[2,0]` points to `"a neg"`.

We can now define a collection of links between two trees as a relation of type $Region \sim Region$, where $Region = Pattern \times Path$: a region is identified by a path leading to a subtree and a pattern describing the top part of the subtree included in the region, a link is a pair of regions, and a collection of links is a relation between regions of two trees. For brevity we will write *Links* for $Region \sim Region$. An arbitrary collection of links may not make sense for a given pair of trees though—a region mentioned by some link may not exist in the trees at all. We should therefore characterise when a collection of links is valid for two trees.

Definition 1 (Region Containment). *For a tree t and a set of regions $\Phi \subseteq Region$, we say that $t \models \Phi$ (read 't contains Φ') exactly when*

$$\forall (pat, path) \in \Phi. \quad sel(t, path) \text{ matches } pat.$$

Definition 2 (Valid Links). *Given $ls : Links$ and two trees t and u, we say that ls is* valid *for t and u, denoted by $t \overset{ls}{\longleftrightarrow} u$, exactly when*

$$t \models \text{LDOM}(ls) \quad \text{and} \quad u \models \text{RDOM}(ls).$$

Now we have all the ingredients for the formal definition of retentive lenses.

Definition 3 (Retentive Lenses). *For a set S of source trees and a set V of view trees, a retentive lens between S and V is a pair of functions*

$$get : S \nrightarrow V \times Links$$
$$put : S \times V \times Links \nrightarrow S$$

satisfying

$(s, v, ls) \in \text{DOM}(put) \Rightarrow s \overset{ls}{\longleftrightarrow} v$

– Hippocraticness: *if $get\ s = (v, ls)$, then $(s, v, ls) \in \text{DOM}(put)$ and*

$$s \overset{ls}{\longleftrightarrow} v \quad \wedge \quad put\ (s, v, ls) = s\ ; \qquad (1)$$

– Correctness: *if $put\ (s, v, ls) = s'$, then $s' \in \text{DOM}(get)$ and*

$$get\ s' = (v, ls') \quad \text{for some } ls'; \qquad (2)$$

– Retentiveness:

$$fst \cdot ls \subseteq fst \cdot ls' \qquad (3)$$

where $fst : A \times B \to A$ is the first projection function (lifted to a relation above).

Modulo the handling of links, Hippocraticness and Correctness stay the same as their original forms (of the well-behaved lenses) respectively. Retentiveness further states that when a region pattern (data) is preserved in the updated source, it still corresponds to exactly the same region on the view side, both the region pattern (data) itself and its location (path). Retentiveness formalises the triangular guarantee in a compact way, and we can expand it pointwise to see that it indeed specialises to the triangular guarantee.

Proposition 1 (Triangular Guarantee). *Given a retentive lens, suppose $put\ (s, v, ls) = s'$ and $get\ s' = (v, ls')$. If $((spat, spath), (vpat, vpath)) \in ls$, then for some $spath'$ we have $s' \models (spat, spath')$ and $((spat, spath'), (vpat, vpath)) \in ls'$.*

Example 1. In Fig. 2, if the `put` function takes `cst`, `ast'`, and diagonal links `ls` = { `((Neg "a neg" _ , [2]) , (Sub (Num 0) _ , [0]))`} as arguments and successfully produces an updated source `s'`, then `get s'` will succeed. Let `(v,ls')` = `get s'`; we know that we can find a consistency link with the path of its source region removed: `c = (Neg "a neg" _ , (Sub (Num 0) _ , [0]))` ∈ `fst` · `ls'`. So the view region referred to by c is indeed the same as the one referred to by the input link, and having `c` ∈ `fst` · `ls'` means that the region in s' corresponding to the view region will match the pattern `Neg "a neg" _`.

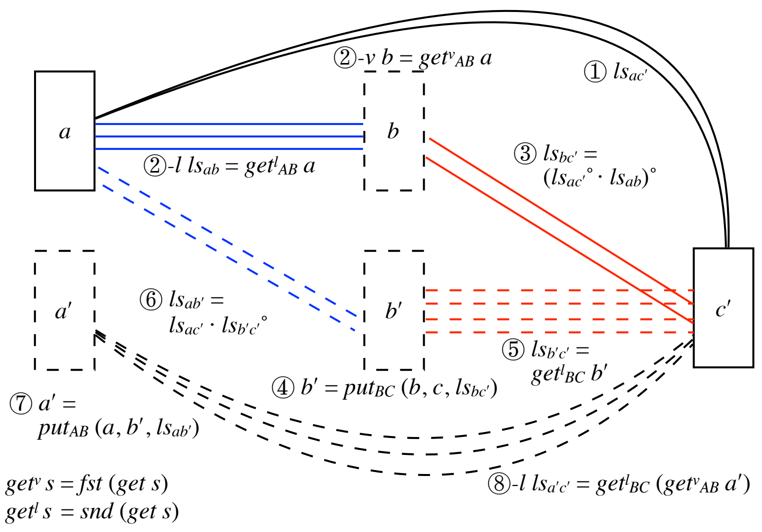

②-v $b = get^v_{AB}\ a$
① $ls_{ac'}$
a
b
②-l $ls_{ab} = get^l_{AB}\ a$
③ $ls_{bc'} = (ls_{ac'}{}^{\circ} \cdot ls_{ab})^{\circ}$
a'
b'
c'
⑥ $ls_{ab'} = ls_{ac'} \cdot ls_{b'c'}{}^{\circ}$
⑤ $ls_{b'c'} = get^l_{BC}\ b'$
④ $b' = put_{BC}\ (b, c, ls_{bc'})$
⑦ $a' = put_{AB}\ (a, b', ls_{ab'})$
$get^v\ s = fst\ (get\ s)$
$get^l\ s = snd\ (get\ s)$
⑧-l $ls_{a'c'} = get^l_{BC}\ (get^v_{AB}\ a')$

Fig. 3. The *put* behaviour of a composite retentive lens, divided into steps ① to ⑧. Step ⑧ produces consistency links for showing the triangular guarantee.

Finally, we note that retentive lenses are an extension of well-behaved lenses: every well-behaved lens between trees can be directly turned into a retentive lens (albeit in a trivial way).

Example 2 (Well-behaved Lenses are Retentive Lenses). Given a well-behaved lens defined by $g : S \to V$ and $p : S \times V \to S$, we define $get : S \nrightarrow V \times Links$ and $put : S \times V \times Links \nrightarrow S$ as follows:

$$\begin{aligned} get\ \ s \qquad &= (g\ s, trivial\ (s, g\ s)) \\ put\ (s, v, ls) &= p\ (s, v) \end{aligned}$$

where

$$trivial\ (s, v) = \{\, ((ToPat\ s, []), (ToPat\ v, []))\, \}.$$

In the definition, *ToPat* turns a tree into a pattern completely describing the tree, and $\textsc{dom}(put)$ is restricted to $\{\, (s, v, ls) \mid s \overset{ls}{\longleftrightarrow} v \text{ and } ls = trivial\ (s, v) \text{ or } \emptyset \,\}$. The explanation and proof can be found in the appendix (Sect. A).

3.2 Composition of Retentive Lenses

It is standard to provide a composition operator for composing large lenses from small ones (although we will not need it for our DSL later). Here we discuss this operator for retentive lenses, which basically follows the definition of composition for well-behaved lenses, except that we need to deal with links carefully. Below we use $lens_{AB}$ to denote a retentive lens that synchronises trees of sets A and B, get_{AB} and put_{AB} the *get* and *put* functions of the lens, l_{ab} a link between tree a (of set A) and tree b (of set B), and ls_{ab} a collection of links between a and b.

Definition 4 (Retentive Lens Composition). *Given two retentive lenses $lens_{AB}$ and $lens_{BC}$, define the get and put functions of their composition by*

$$
\begin{array}{ll}
\begin{array}{l}
get_{AC}\ a = (c, ls_{ab} \cdot ls_{bc}) \\
\quad \textit{where}\ (b, ls_{ab}) = get_{AB}\ a \\
\qquad\quad (c, ls_{bc}) = get_{BC}\ b
\end{array}
&
\begin{array}{l}
put_{AC}\ (a, c', ls_{ac'}) = a' \\
\quad \textit{where}\ (b, ls_{ab}) = get_{AB}\ a \\
\qquad\quad ls_{bc'} \quad = (ls_{ac'}^{\circ} \cdot ls_{ab})^{\circ} \\
\qquad\quad b' \quad\ \ = put_{BC}\ (b, c', ls_{bc'}) \\
\qquad\quad ls_{b'c'} \ = fst\ (get_{BC}\ b') \\
\qquad\quad ls_{ab'} \quad = ls_{ac'} \cdot ls_{b'c'}^{\circ} \\
\qquad\quad a' \quad\ \ = put_{AB}\ (a, b', ls_{ab'}).
\end{array}
\end{array}
$$

The *get* behaviour of a composite retentive lens is straightforward; the *put* behaviour, on the other hand, is a little complex and can be best understood with the help of Fig. 3. Let us first recap the composite behaviour of *put* of traditional lenses: in Fig. 3, if we need to propagate changes from data c' back to data a without links, we will first construct the *intermediate* data b (by running $get_{AB}\ a$), propagate changes from c' to b and produce b', and finally use b' to update a. The composition of retentive lenses is similar: besides the intermediate data b, we also need to construct intermediate links $ls_{bc'}$ (③ in the figure) for retaining information when updating b to b', so that we can further construct intermediate links $ls_{ab'}$ (⑥ in the figure) for retaining information when updating a to a' using b'.

Theorem 1. *The composition of two retentive lenses is still a retentive lens.*

The proof is available in the appendix (Sect. D.1).

4 A DSL for Retentive Bidirectional Tree Transformation

The definition of retentive lenses is somewhat complex, but we can ease the task of constructing retentive lenses with a domain-specific language. Our DSL is simple but suitable for handling the synchronisation between syntax trees. We give a detailed description of the DSL by presenting some programming examples (Sect. 4.1), syntax (Sect. 4.2), semantics (Sect. 4.3), and finally the theorem stating that the generated lenses satisfy Retentiveness (Theorem 2).

4.1 Description of the DSL by Examples

In this subsection, we introduce our DSL by describing the consistency relations between the concrete syntax and abstract syntax of the arithmetic expression example in Fig. 1. From each consistency relation defined in the DSL, we can obtain a pair of *get* and *put* functions forming a retentive lens. Furthermore, we show how to flexibly update the `cst` in Fig. 2 in different ways using the generated *put* function with different input links.

In our DSL, we define data types in HASKELL syntax and describe consistency relations between them as if writing *get* functions. For example, the data type

```
Expr <---> Arith                    Term <---> Arith
  Plus  _  x  y  ~  Add  x  y         Lit    _  i   ~  Num  i
  Minus _  x  y  ~  Sub  x  y         Neg    _  r   ~  Sub  (Num 0)  r
  FromT _  t     ~  t                 Paren  _  e   ~  e
```

Fig. 4. The program in our DSL for synchronising data types defined in Fig. 1.

definitions for `Expr` and `Term` written in our DSL remain the same as those in Fig. 1 and the consistency relations between them (i.e. the `getE` and `getT` functions in Fig. 1) are expressed as the ones in Fig. 4. Here we describe two consistency relations similar to `getE` and `getT`: one between `Expr` and `Arith`, and the other between `Term` and `Arith`. Each consistency relation is further defined by a set of inductive rules, stating that if the subtrees matched by the same variable appearing on the left-hand side (source side) and right-hand side (view side) are consistent, then the large pair of trees constructed from these subtrees are also consistent. (For primitive types which do not have constructors such as integers and strings, we consider them consistent if and only if they are equal.) Take

```
Plus _ x y  ~  Add x y
```

for example; it means that if x_s is consistent with x_v, and y_s is consistent with y_v, then *Plus* $a\ x_s\ y_s$ and *Add* $x_v\ y_v$ are consistent for any value a, where a corresponds to the 'don't-care' wildcard in `Plus _ x y`. So the meaning of `Plus _ x y  ~  Add x y` can be better understood by the 'derivation rule' beneath:

$$\frac{x_s \sim x_v \quad y_s \sim y_v}{\mathit{Plus}\ a\ x_s\ y_s \sim \mathit{Add}\ x_v\ y_v}$$

Generally, each consistency relation is translated to a pair of *get* and *put* functions, and each of these inductive rules forms one case of the two functions.

Now we briefly explain the semantics of `Plus _ x y  ~  Add x y`. Let us assume that the pair of *get* and *put* functions generated from `Expr <---> Arith` are `getEA` and `putEA`[1] respectively. In the *get* direction, the behaviour of `getEA` is quite similar to the first case of `getE` in Fig. 1, except that it also updates the links between subtrees marked by `x` and `y` respectively, and establishes a new link between the top nodes `Plus` and `Add` recording the correspondence. In the *put* direction, `putEA` creates a tree whose top node is a `Plus` with empty annotations and recursively builds the subtrees, provided that there is no link connected to the node `Add` (top of the view). If there are some links, then the behaviour of `putEA` is guided by those links; for example, `putEA` may additionally preserve the annotation in the old source. We leave the detailed descriptions to the subsection about semantics.

With retentive lenses `getEA` and `putEA`[2] generated from Fig. 4, we can synchronise the (old) `cst` and (modified) `ast'` in Fig. 2 in different ways by running `putEA` with different input links. For instance, supposing that the input links

[1]The function names `getEA` and `putEA` here are simplified compared with our real implementation.

[2]There are also `getTA` and `putTA` generated from the consistency relation `Term <---> Arith`, which are mutually recursive with `getEA` and `putEA` respectively.

`ls'` are obtained (updated) by swapping the two subtrees of `Add` using the edit operation *swap* introduced in Sect. 5.1, `putEA cst ast ls'` will produce exactly `cst'` in Fig. 2.

Although the DSL is tailored for describing consistency relations between syntax trees, it is also possible to handle general tree transformations. Some small but typical programming examples other than syntax tree synchronisation can be found in the appendix (Sect. B).

4.2 Syntax

Having seen the programming example, here we summarise the syntax of our language in Fig. 5. A program consists of two main parts: definitions of data types and consistency relations between these data types. We adopt the HASKELL syntax for definitions of data types, so that a new data type is defined through a set of data constructors $C_1 \ldots C_n$ followed by types; a type synonym can be defined by giving a new name to existing types. (The definition of `type` is omitted.) Now, we move to the definitions of consistency relations, where each of them starts with $type_s \leftrightarrow type_v$, representing the source type and view type for the relation. The body of each consistency relation is a list of inductively-defined rules, where each rule is defined as a relation between the source and view patterns $pat_s \sim pat_v$, and a pattern *pat* includes variables, constructors, and wildcards. Finally, two semicolons ';;' are added to finish the definition of a consistency relation. (In this paper, we always omit ';;' when there is no confusion.)

Syntactic Restrictions To guarantee that consistency relations in our DSL indeed correspond to retentive lenses, we impose several syntactic restrictions on the DSL. Our restrictions on patterns are quite natural, while those on algebraic data types are a little subtle, which the reader may skip at the first reading.

- On *patterns*, we assume (i) pattern coverage and source pattern disjointness. For any consistency relation $S \leftrightarrow V = \{\, p_i \sim q_i \mid 1 \leqslant i \leqslant n \,\}$ defined in a program, $\{\, p_i \,\}$ should cover all the possible cases of type S, and $\{\, q_i \,\}$ should cover all the cases of type V. Source pattern disjointness requires that any distinct p_i and p_j do not match any tree at the same time so that at most one pattern is matched when running *get*. Additionally, (ii) a bare variable pattern is not allowed on the source side (e.g. $x \sim D\ x$) and wildcards are not allowed on the view side (e.g. $C\ x \sim D\ \_\ x$).
- On *algebraic data types*, we require that types S_1 and S_2 be *interconvertible*[1] when (i) consistency relations $S_1 \leftrightarrow V_{12}$, $S_2 \leftrightarrow V_{12}$, and $S \leftrightarrow V$ are all defined for some types V_{12}, S, and V, and (ii) the definition of S makes use of both S_1 and S_2 (directly or indirectly), and the definition of V makes use of V_{12}. The following figure depicts some of the cases,

[1]Two types T_1 and T_2 are interconvertible if there exist injections $inj_1 :: T_1 \rightarrow T_2$ and $inj_2 :: T_2 \rightarrow T_1$ for all the inhabitants of T_1 and T_2. For example, `Lit 1 :: Term` can be converted to `FromT (Lit 1) :: Expr` and `FromT (Lit 1) :: Expr` can be converted to `Paren (FromT (Lit 1)) :: Term`.

Program

$prog ::= tDef\ \ cDef_1 \ldots cDef_n$

Type Definition

$tDef ::= tSyn_1 \ldots tSyn_m\ \ tNew_1 \ldots tNew_n$

$tSyn ::= \text{‘type’}\ \mathbf{type}\ \text{‘=’}\ \mathbf{type}$

$tNew ::= \text{‘data’}\ \mathbf{type}\ \text{‘=’}\ C_1\ \mathbf{type}_{11} \ldots \mathbf{type}_{1k_1}$
$\qquad\qquad\qquad\qquad \text{‘|’}\ \ldots$
$\qquad\qquad\qquad\qquad \text{‘|’}\ C_n\ \mathbf{type}_{n1} \ldots \mathbf{type}_{nk_n}$

$\mathbf{type} ::= tyCon\ \mathbf{type}_1 \ldots \mathbf{type}_m$ { type constructor }
$\qquad\quad |\ \ tyVar$ { type variable }

Consistency Relation Definition

$cDef ::= \mathbf{type}_s \longleftrightarrow type_v$ { relation type }
$\qquad\qquad r_1 \ldots r_n\ \text{‘;;’}$ { inductive rules }

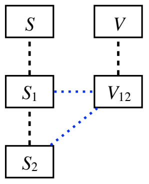
Inductive Rule

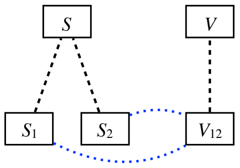
$r ::= pat_s\ \text{‘}\sim\text{’}\ pat_v$

Pattern

$pat ::= v$ { variable pattern }
$\qquad |\ \ \_$ { don’t-care wildcard }
$\qquad |\ \ C\ pat_1 \ldots pat_n$ { constructor pattern }

Fig. 5. Syntax of the DSL.





where vertical dotted links (in black) mean that there are zero or more nodes in between the connected nodes and horizontal links (in blue) show that both S_1 and S_2 can be consistent with V_{12} (i.e. there are consistency relations $S_1 \leftrightarrow V_{12}$ and $S_2 \leftrightarrow V_{12}$). In this case, in the *put* direction, there might be links asserting that values of type S_2 should be retained in a context where values of type S_1 are expected, or vice versa; thus we need a way to convert between S_1 and S_2. For instance, since the two consistency relations `Expr <---> Arith` and `Term <---> Arith` in Fig. 4 share the same view type `Arith`, there should be a way to convert (inject) `Expr` into `Term` and vice versa. Look at `cst'` in Fig. 2, the second subtree of `Plus` is of type `Expr` but created by using a link connected to the old source’s subtree `Neg ...` of type `Term`, so we need to wrap `Neg ...` into `FromT "" (Neg ...)` to make the types match.

In our DSL, the conversions (injections) for interconvertible data types are automatically generated from consistency relations. To have an injection $\mathit{inj}_{T_1 \to T_2}\ x = C \ldots x \ldots$ which directly injects data of type T_1 to data of type T_2, it is equal to saying that the consistency relation $T_2 \leftrightarrow V$ has a rule $C \ldots x \ldots \sim x$ in which the source pattern only has wildcards except C and x. For example, $\mathit{FromT}\ \_\ t \sim t$ gives rise to an injection `inj`$_{\mathit{Term} \to \mathit{Expr}}$ `x = FromT "" x`. Note that T_1 can be indirectly injected to T_2 through a *chain* of such rules, in general. (There will not be such requirement if we only handle (untyped) cases where a subtree can freely occur everywhere in a tree (like the rose tree data structure and XML).)

4.3 Semantics

We give a denotational semantics of our DSL by specifying the corresponding retentive lens—a pair of *get* and *put*—of a consistency relation defined in the DSL. In other words, our (bidirectional) semantics of the DSL is defined by two (unidirectional) semantics, a *get* semantics and a *put* semantics. Our HASKELL implementation of the DSL follows the semantics described here.

Types and Patterns To define *get* and *put*, we need to first give the semantics of type declarations and patterns. The semantics of type declarations is the same as that in HASKELL so that we will not elaborate much here. For each defined algebraic data type T, we also use T to denote the set of all values of T. In addition, we define *Tree* to be the set of all the values of all the algebraic data types defined in our DSL and *Pattern* to be the set of all possible patterns. For a pattern $p \in \mathit{Pattern}$, $\mathit{Vars}\ p$ denotes the set of variables in p, $\mathit{TypeOf}\ (p, v)$ is the set corresponding to the type of $v \in \mathit{Vars}\ p$, and $\mathit{path}\ (p, v)$ is the path of variable v in pattern p.

We use the following (partial) functions to manipulate patterns:

$$\begin{aligned} \mathit{isMatch} &: (p \in \mathit{Pattern}) \times \mathit{Tree} \to \mathit{Bool} \\ \mathit{decompose} &: (p \in \mathit{Pattern}) \times \mathit{Tree} \nrightarrow \big(\mathit{Vars}\ p \to \mathit{Tree}\big) \\ \mathit{reconstruct} &: (p \in \mathit{Pattern}) \times \big(\mathit{Vars}\ p \to \mathit{Tree}\big) \nrightarrow \mathit{Tree}\ . \end{aligned}$$

Given a pattern p and a value (i.e. tree) t, $\mathit{isMatch}\ (p, t)$ tests if t matches p. If the match succeeds, $\mathit{decompose}\ (p, t)$ returns a function f mapping every variable in p to its corresponding matched subtree of t. Conversely, $\mathit{reconstruct}\ (p, f)$ produces a tree t matching pattern p by replacing every occurrence of $v \in \mathit{Vars}\ p$ in p with $f\ v$, provided that p does not contain any wildcard. Since the semantics of patterns in our DSL is rather standard, we omit detailed definitions of these functions[1].

[1]Non-linear patterns are supported: multiple occurrences of the same variable in a pattern must have the same type and they must capture the same value.

Get Semantics For a consistency relation $S \leftrightarrow V$ defined in our DSL with a set of inductive rules $R = \{\, spat_k \sim vpat_k \mid 1 \leqslant k \leqslant n \,\}$, its corresponding get_{SV} function has the following type:

$$get_{SV} : S \to V \times LinkSet$$

The idea of $get(s)$ is to use the rule $spat_k \sim vpat_k \in R$ of which $spat_k$ matches s—our DSL requires such a rule uniquely exists for all s—to generate the top portion of the view and recursively generate subtrees for all variables in $spat_k$. The get function also creates links in the recursive procedure: when a rule $spat_k \sim vpat_k \in R$ is used, it creates a link relating those matched parts as two regions. The get function defined by R is:

$$\begin{aligned}
&get_{SV}\ s = (reconstruct\ (vpat_k, fst \circ vls),\ l_{root} \cup links) \\
&\quad \textbf{where}\ spat_k \sim vpat_k \in R \text{ and } spat_k \text{ matches } s \\
&\qquad\quad vls = (get \circ decompose\ (spat_k, s) \in Vars\ spat_k \to V \times LinkSet \\
&\qquad\quad spat' = eraseVars\ (fillWildcards\ (spat_k, s)) \\
&\qquad\quad l_{root} = \big\{\, ((spat', [\,]), (eraseVars\ vpat_k, [\,])) \,\big\} \\
&\qquad\quad links = \big\{\, ((spat, path\ (spat_k, t) +\!\!+\ spath), (vpat, path\ (vpat_k, t) +\!\!+\ vpath)) \\
&\qquad\qquad\qquad \mid t \in Vars\ (vpat_k), ((spat, spath), (vpat, vpath)) \in snd\ (vls\ t) \,\big\}\ .
\end{aligned} \tag{4}$$

For a tree t matching a pattern pat, $fillWildcards\ (pat, t)$ replaces all the wildcards in pat with the corresponding subtrees of t, and $eraseVars\ pat$ replaces all the variables in pattern pat with wildcards. While the recursive call is written as $get \circ decompose\ (spat_k, s)$ in the definition above, to be precise, get should have different subscript $TypeOf\ (spat_k, v)\ TypeOf\ (vpat_k, v)$ for different $v \in Vars\ spat_k$.

Put Semantics For a consistency relation $S \leftrightarrow V$ defined in our DSL as $R = \{\, spat_k \sim vpat_k \mid 1 \leqslant k \leqslant n \,\}$, its corresponding put_{SV} function has the following type:

$$put_{SV} : Tree \times V \times LinkSet \nrightarrow S\ .$$

Given arguments (s, v, ls), put is defined by two cases depending on whether the root of the view is within a region of the input links or not, i.e., whether there is some $(\_, (\_, [])) \in ls$.

– In the first case where the root of the view is not within any region of the input links, put selects a rule $spat_k \sim vpat_k \in R$ whose $vpat_k$ matches v—again, our DSL requires that at least one such rule exist for all v—and uses $spat_k$ to build the top portion of the new source: wildcards in $spat_k$ are filled with default values and variables in $spat_k$ are filled with trees recursively constructed from their corresponding parts of the view.

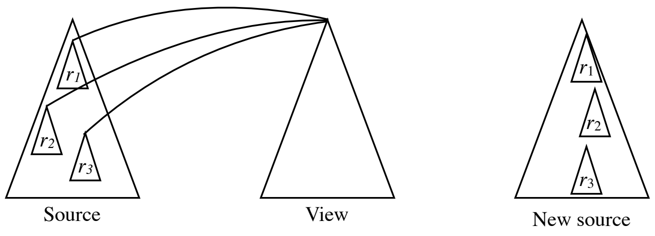
$$\begin{aligned}
&put_{SV}\ (s, v, ls) = reconstruct\ (spat'_k, ss) && (5) \\
&\quad \textbf{where}\ spat_k \sim vpat_k \in R \text{ and } vpat_k \text{ matches } v \\
&\qquad\quad vs = decompose\ (vpat_k, v) \\
&\qquad\quad ss = \lambda\,(t \in Vars\ spat_k) \to \\
&\qquad\qquad\quad put(s, vs\ t, divide\ (path\ (vpat_k, t)), ls) && (6) \\
&\qquad\quad spat'_k = fillWildcardsWithDefaults\ spat_k \\
&divide\ (prefix, ls) = \{\, (r_s, (vpat, vpath)) \mid (r_s, (vpat, prefix +\!\!+\ vpath) \in ls) \,\}
\end{aligned}$$

The omitted subscript of put in (6) is $\mathit{TypeOf}\ (\mathit{spat}_k, t)\ \mathit{TypeOf}\ (\mathit{vpat}_k, t)$. Additionally, if there is more than one rule of which vpat matches v, the rule whose vpat is *not* a bare variable pattern is preferred for avoiding infinite recursive calls: if $\mathit{vpat}_k = x$, the size of the input of the recursive call in Equation 6 does not decrease because $\mathit{vs}\ t = v$ and $\mathit{path}\ (t, \mathit{vpat}_k) = [\,]$. For example, if both $\mathit{spat}_1 \sim \mathit{Succ}\ x$ and $\mathit{spat}_2 \sim x$ can be selected, the former is preferred.

– In the case where the root of the view is an endpoint of some link, put uses the source region (pattern) of the link as the top portion of the new source.

$$
\begin{aligned}
&\mathit{put}_{SV}\ (s, v, \mathit{ls}) = \mathit{inj}_{\ \mathit{TypeOf}\ \mathit{spat}_k \to S}(\mathit{reconstruct}(\mathit{spat}'_k, \mathit{ss})) && (7)\\
&\quad \textbf{where}\ l = ((\mathit{spat}^{1}, \mathit{spath}), (\mathit{vpat}, \mathit{vpath})) \in \mathit{ls}\\
&\qquad\qquad \text{such that } \mathit{vpath} = [\,],\ \mathit{spath} \text{ is the shortest}\\
&\qquad\quad \mathit{spat}_k \sim \mathit{vpat}_k \in R \ \text{ and }\ \mathit{isMatch}\ (\mathit{spat}_k, \mathit{spat})\\
&\qquad\quad \mathit{spat}'_k = \mathit{fillWildcards}\ (\mathit{spat}_k, \mathit{spat})\\
&\qquad\quad \mathit{vs} = \mathit{decompose}\ (\mathit{vpat}_k, v)\\
&\qquad\quad \mathit{ss} = \lambda\,(t \in \mathit{Vars}\ \mathit{spat}_k) \to\\
&\qquad\qquad \mathit{put}(s, \mathit{vs}\ t, \mathit{divide}\ (\mathit{path}\ (\mathit{vpat}_k, t)), \mathit{ls} \setminus \{\, l \,\})) && (8)
\end{aligned}
$$

When there is more than one source region linked to the root of the view, put chooses the source region whose path is the shortest, which ensures that the preserved region patterns in the new source will have the same relative positions as those in the old source, as the following figure shows.



Since the linked source region (pattern) does not necessarily have type S, we need to use the function $\mathit{inj}_{\ \mathit{TypeOf}\ \mathit{spat}_k \to S}$ to convert (inject) it to type S; this function is provided by the interconvertible requirement of our DSL (see Syntax Restrictions).

Domain of *put* The last piece of our definition of put is its domain, on which put will terminate and satisfy the required properties of the retentive lens formed together with its corresponding get. In the implementation, runtime checks are used to detect invalid arguments when doing put; but for clarity, here we define a separate function check and the domain of put to be $\{\, a : \mathit{Tree} \times V \times \mathit{LinkSet} \mid \mathit{check}\ a = \top \,\}$.

$$\mathit{check} : \mathit{Tree} \times V \times \mathit{LinkSet} \to \mathit{Bool}$$

[1]Here we treat region patterns (such as spat) as partial trees, i.e. trees with holes (represented by wildcards) in them. Wildcards in partial trees can be captured by either wildcard patterns or variable patterns in pattern matching.

$$
\mathit{check}\ (s, v, \mathit{ls}) = \begin{cases} \mathit{chkWithLink}\ (s, v, \mathit{ls}) & \text{if some}((\_,\_),(\_,[\,])) \in \mathit{ls} \\ \mathit{chkNoLink}\ (s, v, \mathit{ls}) & \text{otherwise} \end{cases}
$$

chkNoLink corresponds to the first case of *put* (5).

$$
\begin{aligned}
&\mathit{chkNoLink}\ (s, v, \mathit{ls}) = \mathit{cond}_1 \wedge \mathit{cond}_2 \wedge \mathit{cond}_3 \\
&\quad \textbf{where}\ \mathit{spat}_k \sim \mathit{vpat}_k \in R \text{ and } \mathit{vpat}_k \text{ matches } v \\
&\qquad \mathit{vs} = \mathit{decompose}\ (\mathit{vpat}_k, v) \\
&\qquad \mathit{vp}\ t = \mathit{path}\ (\mathit{vpat}_k, t) \\
&\qquad \mathit{cond}_1 = \mathit{ls} == \left(\bigcup_{t \in \mathit{Vars}\ \mathit{spat}_k} \mathit{addVPrefix}\ (\mathit{vp}\ t, \mathit{divide}\ (\mathit{vp}\ t, \mathit{ls})) \right) \\
&\qquad \mathit{cond}_2 = \bigwedge_{t \in \mathit{Vars}\ \mathit{spat}_k} \mathit{check}\ (s, \mathit{vs}\ t, \mathit{divide}\ (\mathit{vp}\ t, \mathit{ls})) \\
&\qquad \mathit{cond}_3 = \textbf{if}\ \mathit{vpat}_k \text{ is some bare variable pattern `}x\text{' } \textbf{then} \\
&\qquad\qquad \mathit{TypeOf}\ (\mathit{spat}_k, x) \leftrightarrow V \text{ has a rule } \mathit{spat}_j \sim \mathit{vpat}_j \text{ s.t.} \\
&\qquad\qquad\quad \mathit{isMatch}\ (\mathit{vpat}_j, v) \text{ and } \mathit{vpat}_j \text{ is not a bare variable pattern} \\
&\qquad \mathit{addVPrefix}\ (\mathit{prefix}, \mathit{rs}) = \{\, ((a, b), (c, \mathit{prefix} +\!\!+\, d)) \mid ((a, b), (c, d) \in \mathit{rs} \,\}
\end{aligned}
$$

Condition cond_1 checks that every link in *ls* is processed in one of the recursive calls. (Specifically, if $\mathit{Vars}\ \mathit{spat}_k$ is empty, *ls* in cond_1 should also be empty meaning that all the links have already been processed.) cond_2 summarises the results of *check* for recursive calls. cond_3 guarantees the termination of recursion: When vpat_k is a bare variable pattern, the recursive call in Equation 6 does not decrease the size of any of its arguments; cond_3 makes sure that such non-decreasing recursion will not happen in the next round[1] for avoiding infinite recursive calls.

For *chkWithLink*, as in the corresponding case of *put* (Equation 7), let $l = ((\mathit{spat}, \mathit{spath}), (\mathit{vpat}, \mathit{vpath})) \in \mathit{ls}$ such that $\mathit{vpath} = [\,]$ and *spath* is the shortest if there is more than one such link.

$$
\begin{aligned}
&\mathit{chkWithLink}\ (s, v, \mathit{ls}) = \mathit{cond}_1 \wedge \mathit{cond}_2 \wedge \mathit{cond}_3 \wedge \mathit{cond}_4 \\
&\quad \textbf{where} \\
&\quad \mathit{cond}_1 = \mathit{isMatch}\ (\mathit{spat}, \mathit{sel}\ (s, \mathit{spath})) \wedge \mathit{isMatch}\ (\mathit{vpat}, \mathit{sel}\ (v, \mathit{vpath})) \\
&\quad \mathit{cond}_2 = \exists!(\mathit{spat}_k, \mathit{vpat}_k) \in R.\ \mathit{vpat} = \mathit{eraseVars}\ \mathit{vpat}_k \\
&\qquad\qquad \wedge\ \mathit{isMatch}\ (\mathit{eraseVars}\ \mathit{spat}_k, \mathit{spat}) \\
&\qquad\qquad \wedge\ \mathit{fillWildcards}\ (\mathit{spat}_k, \mathit{spat}) \text{ is wildcard-free} \\
&\qquad\qquad \wedge \bigwedge_{t \in \mathit{Vars}\ \mathit{spat}_k} \mathit{decompose}\ (\mathit{spat}_k, \mathit{spat})\ t \text{ is wildcard} \\
&\quad \mathit{cond}_3 = \mathit{ls} == \left(\{\, l \,\} \cup \bigcup_{t \in \mathit{Vars}\ \mathit{spat}_k} \mathit{addVPrefix} \mid (\mathit{path}\ (\mathit{vpat}_k, t), \mathit{divide}\ (\mathit{path}\ (\mathit{vpat}_k, t)), \mathit{ls} \setminus \{\, l \,\})) \right) \\
&\quad \mathit{cond}_4 = \bigwedge_{t \in \mathit{Vars}\ \mathit{spat}_k} \mathit{check}\ (s, \mathit{vs}\ t, \mathit{divide}\ (\mathit{path}\ (\mathit{vpat}_k, t)), \mathit{ls} \setminus \{\, l \,\}))
\end{aligned}
$$

cond_1 makes sure that the link l is valid (Definition 2) and cond_2 further checks that it can be generated from some rule of the consistency relations. cond_3 and

[1]It is often too restrictive to check two rounds; however, the restriction can be relaxed to checking arbitrary finite rounds.

$cond_4$ are for recursive calls: the former guarantees that no link will be missed and the latter summarises the results.

Retentiveness of the DSL With the definitions of get and put above, we have

Theorem 2 (Main Theorem). *Let put' be put with its domain intersected with $S \times V \times LinkSet$, get and put' form a retentive lens as in Definition 3.*

The proof goes by induction on the size of the arguments to put or get and can be found in the appendix (D.2).

5 Case Studies

In this section, we first provide a way to obtain the diagonal links between a source and a modified view. Then we demonstrate the usefulness of Retentiveness in practice by presenting case studies on code refactoring [**Fowler1999Refactoring**], XML synchronisation, and resugaring [**Pombrio2014Resugaring**, **Pombrio2015Hygienic**], all of which require that we constantly make modifications to ASTs[1] and synchronise CSTs accordingly. Retentive lenses provide a systematic way for the user to preserve information of interest in the original CST after synchronisation.

5.1 Edit Operations and Link Maintenance

In Sect. 6.4, we have discussed vertical correspondence a little and stated the reason for removing it from our framework. To demonstrate the feasibility of producing vertical correspondence and composing it with horizontal links to obtain (diagonal) input links, in this subsection, we define vertical correspondence as 'vertical links'; a vertical link consists of two paths sharing the same region pattern, i.e. $(vpath, vpat, vpath')$—meaning that a data fragment $(vpat, vpath)$ in a view is not destroyed but probably moved to $vpath'$ in the updated view. When composing horizontal links with vertical links, we first transform each vertical link $(vpath, vpat, vpath')$ to an equivalent representation $((vpat, vpath), (vpat, vpath'))$ and then reuse the link composition introduced in Sect. 3.2.

We further define four edit operations, *replace*, *copy*, *move*, and *swap*, of which *move* and *swap* are (both) defined in terms of *copy* and *replace*. The edit operations accept not only an AST but also a set of horizontal links[2], and perform their corresponding update to the AST and maintain the set of links as well. The interface has been designed in a way that the last argument is the pair of AST and links, so that the user can use Haskell's ordinary function composition to

[1]Many code refactoring tools make modifications to CSTs instead. However, the design of the tools can be simplified if they choose to modify ASTs, as the paper will show.

[2]We let edit operations accept and return horizontal links and use 'vertical links' only as an intermediate representation, for the simplicity of user interface and 'opting for triangular diagrams'.

compose a sequence of edits (partially applied to other arguments). We briefly highlight the link maintenance: *replac*ing a subtree at path p will destroy all the links previously connecting to path p. *Copy*ing a subtree from path p to path p' will duplicate the set of links previously connecting to p and redirect the duplicated links to connect to p'. *Mov*ing a subtree from p to p' will destroy links connecting to p' and redirect the links (previously) connecting to p to connect to p'. *Swap*ping subtrees at p and p' will also swap the links connecting to p and p'. The implementation of the four edit operations takes less than 40 lines of code (in Haskell), as our DSL already generates useful auxiliary functions such as fetching a subtree according to a path in some tree.

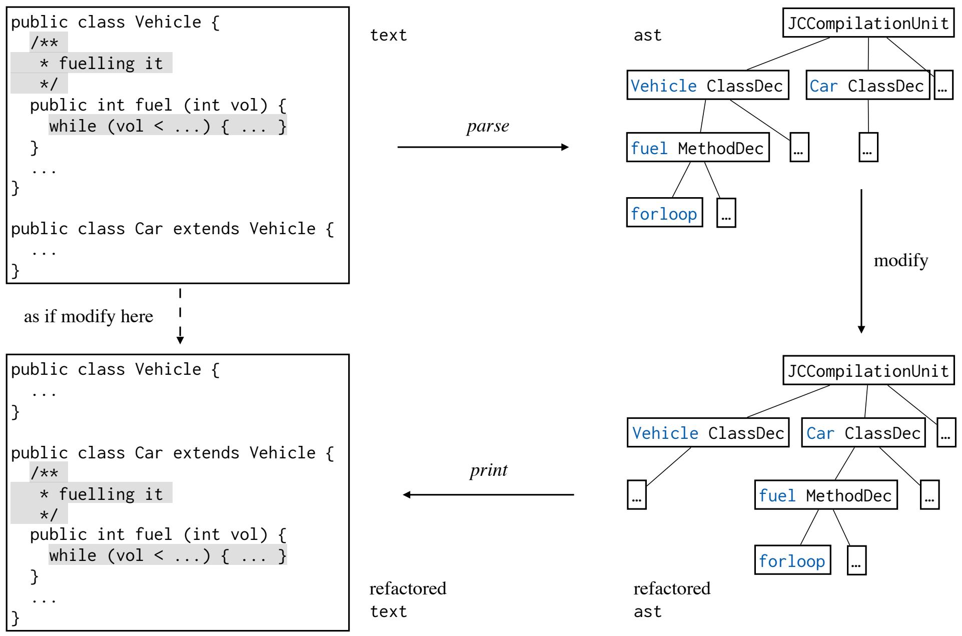
Handy operations such as *insert* and *delete* on lists can also be defined in terms of the basic ones; for instance, *insert*ing an element e at position i in a list is just to *move* each element at position j after i ($j > i$) to position $j + 1$ (in reverse order) and *replace* the element at position i with e. In the next subsection, we will explain that our edit operations are sufficient to write most of the code refactoring operations, and we demonstrate how to use them to write a particular code refactoring called *push-down*.

5.2 Refactoring

Implementation of the whole code refactoring tool for Java 8 using retentive lenses requires much engineering work, and there are further research problems not solved. For this reason, in this paper, we focus on the theoretical foundation and language design, and have only implemented the transformation system (between CSTs and ASTs) for a small subset of Java 8 to demonstrate the possibility.

Feasibility of Retentiveness for Code Refactoring To see whether Retentiveness helps to retain comments and syntactic sugar for real-world code refactoring, we surveyed the standard set of refactoring operations for Java 8 provided by Eclipse Oxygen (with Java Development Tools). We found that the total 23 refactoring operations can be represented as the combinations of our edit operations defined in Sect. 5.1.

The *Push-Down* Code Refactoring An example of the *push-down* code refactoring is illustrated in Fig. 6. At first, the user designed a `Vehicle` class and thought that it should have a `fuel` method for all the vehicles. The `fuel` method has a JavaDoc-style comment and contains a `while` loop, which can be seen as syntactic sugar and is converted to a standard `for` loop during parsing. However, when later designing the subclasses, the user realises that bicycles cannot be fuelled, and decides to do the *push-down* code refactoring, removing the `fuel` method from `Vehicle` and pushing the method definition down to subclasses `Bus` and `Car` but not `Bicycle`. Instead of directly modifying the (program) `text`, most refactoring tools will first parse the program text into its `ast`, perform code refactoring on the `ast`, and regenerate new (program) `text`. The bottom-left corner of Fig. 6 shows the desired (program) `text` after refactoring, where we see



```
public class Vehicle {
  /**
   * fuelling it
   */
  public int fuel (int vol) {
    while (vol < ...) { ... }
  }
  ...
}

public class Car extends Vehicle {
  ...
}
```

```
public class Vehicle {
  ...
}

public class Car extends Vehicle {
  /**
   * fuelling it
   */
  public int fuel (int vol) {
    while (vol < ...) { ... }
  }
  ...
}
```

Fig. 6. An example of the *push-down* code refactoring. (For simplicity, subclasses `Bus` and `Bicycle` are omitted.)

that the comment associated with `fuel` is also pushed down, and the `while` sugar is kept. However, the preservation of the comment and syntactic sugar in fact does not come for free, as the `ast`, being a concise and compact representation of the program text, includes neither comments nor the form of the original `while` loop. So, if the user implements the *parse* and *print* functions as a well-behaved lens, it might only give the user an unsatisfactory result in which much information is lost (such as the comment and `while` syntactic sugar).

Implementation in Our DSL Following the grammar of Java 8 [**Gosling2014Java**], we define the data types for the simplified concrete syntax, which consists of only definitions of classes, methods, and variables; arithmetic expressions (including assignment and method invocation); conditional and loop statements. For convenience, we also restrict the occurrence of statements and expressions to exactly once in most cases (such as variable declarations) except for the class and method body. Then we define the corresponding simplified version of the abstract syntax that follows the one defined by the JDT parser [**OpenJDK**]. This subset of Java 8 has around 80 CST constructs (which represent production rules) and 30 AST constructs; the 70 consistency relations among them generate about 3000 lines of retentive lenses and auxiliary functions (such as the ones for handling interconvertible data types and the ones for easing the work of defining edit operations).

Since the structure of the consistency relations for the transformation system is overall similar to the ones in Fig. 4, here we only highlight two of them as examples; the reader can refer to the supplementary material to see the complete program. As for the concrete syntax, everything is a class declaration (`ClassDecl`), while for the abstract syntax everything is a tree (`JCTree`). As a `ClassDecl` should correspond to a `JCClassDec`, which by definition is yet not a `JCTree`, we use the constructors `FromJCStmt` and `FromJCClassDec` to make it a `JCTree`, emulating the inheritance in Java. This is described by the consistency relation[1]

```
ClassDecl <---> JCTree
  NormalClassDeclaration0 _ "class" n "extends" sup body ~
  FromJCStmt (FromJCClassDec (JCClassDec N n (J (JCIdent sup)) body))

  NormalClassDeclaration1 _ mdf "class" n "extends" sup body ~
  FromJCStmt (FromJCClassDec (JCClassDec (J mdf) n (J (JCIdent sup)) body))
  ...
```

Depending on whether a class has a modifier (such as `public` and `private`) or not, the concrete syntax is divided into two cases while we use a `Maybe` type in the abstract syntax representing both cases. (To save space, the constructors `Just` and `Nothing` are shortened to `J` and `N` respectively.) Similarly, there are further two cases where a class does not extend some superclass and are omitted here.

Next, we see how to represent `while` loop using the basic `for` loop, as the abstract syntax of a language should be as concise as possible[2]:

```
Statement <---> JCStatement
  While "while" "(" exp ")" stmt ~ JCForLoop Nil exp Nil stmt
```

where the four arguments of `JCForLoop` in order denote (list of) initialisation statements, the loop condition, (list of) update expressions, and the loop body. As for a `while` loop, we need to convert its loop condition `exp` and loop body `stmt` to AST types and put them in the correct places of the `for` loop. Initialisation statements and update expressions are left empty since there is none.

System Running Now we use the retentive lenses generated from our DSL to run the refactoring example in Fig. 6. We first test two basic cases: `put cst ast ls` and `put cst ast []`, where `(ast,ls) = get cst`. We obtain the same `cst` after running `put cst ast ls` and it shows that the generated lenses satisfy Hippocraticness. As a special case of Correctness, we let `cst' = put cst ast []` and check that `fst (get cst') == ast`. We observe that in `cst'` the `while` loop becomes a basic `for` loop and all the comments disappear; this demonstrates that *put* can create a new source from scratch only depending on the given view.

Then we modify `ast` to `ast'` and maintain the set of links `ls` to `ls'` using our edit operations, simulating the *push-down* code refactoring for the `fuel` method. For comparison, when building `ast'`, the `fuel` method in the `Car` class is *cop*ied from the `Vehicle` class, while the `fuel` method in the `Bus` class is built from scratch

[1]We include keywords such as `"class"` and `"extends"` in the CST patterns for improving readability, although they should be removed.

[2]Although the JDT parser does not do this.

(i.e. *replace*d with a 'new' `fuel` method). Let `cst' == put cst ast' ls'`. As for `cst'`, we observe that for the `fuel` method of its `Car` class , the `while` loop and the associated comments are preserved; but for the `fuel` method of its `Bus` class, the loop is a basic `for` loop and the comments disappear. This is where Retentiveness helps the user to retain information on demand. Finally, we also check that Correctness holds: `fst (get cst') == ast'`.

5.3 XML Synchronisation

In this subsection, we present a case study on XML synchronisation, which is pervasive in the real world and different from syntax tree manipulation. The specific example used in this subsection is from **Pacheco2014BiFluX**'s paper [**Pacheco2014BiFluX**], where they use their DSL, BiFluX, to synchronise address books.

As for their example, both the source address book and the view address book are grouped by social relationships; however, the source address book (defined by `AddrBook`) contains names, emails, and telephone numbers whereas the view (social) address book (defined by `SocialBook`) contains names only.

To synchronise `AddrBook` and `SocialBook`, we write consistency relations in our DSL and the core ones are

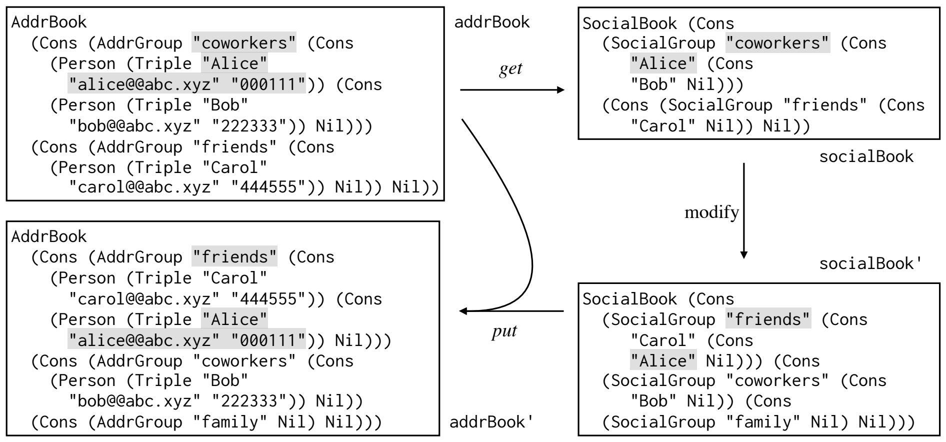
```
AddrGroup  <---> SocialGroup              Person <---> Name
  AddrGroup grp p  ~  SocialGroup grp p     Person t  ~  t

List Person <---> List Name               Triple Name Email Tel <---> Name
  Nil  ~  Nil                               Triple name  _ _  ~  name .
  Cons p xs  ~  Cons p xs
```

The consistency relations will compile to a pair of `get` and `put`.

As Fig. 7 shows, the original source is `addrBook` and its consistent view is `socialBook`, both of which have two relationship groups: `coworkers` and `friends`. The source has a record `Person (Triple "Alice" "alice@abc.xyz" "000111")` in the group `coworkers`. Then the view `socialBook` is updated in a way that we (i) reorder the two groups; (ii) change Alice's group (from `coworkers` to `friends`); (iii) create a new social relationship group `family` for family members.

In our case, to produce a new source `socialBook'`, we handle the three update steps using our basic edit operations (in this case, only *swap*, *move*, and *replace*) which also maintain the links. Feeding the original source `addrBook`, updated view `socialBook'` and links `hls'` to the (generated) `put` function, we obtain the updated `addrBook'`. In Fig. 7, it is clearly seen that carefully maintained links help us to preserve email addresses and telephone numbers associated with each person during the *put* process; note that well-behavedness does not guarantee information retention, for the updated view is not consistent with the original source in this case.

As pointed out by **Pacheco2014BiFluX** examples of this kind motivate extensions to alignment-aware languages such as Boomerang [**Bohannon2008Boomerang**] and matching lenses [**Barbosa2010Matching**]. In fact, it is hard for those languages to handle source-view alignment of this kind, where some view elements are



```
AddrBook
  (Cons (AddrGroup "coworkers" (Cons
    (Person (Triple "Alice"
      "alice@@abc.xyz" "000111")) (Cons
    (Person (Triple "Bob"
      "bob@@abc.xyz" "222333")) Nil)))
  (Cons (AddrGroup "friends" (Cons
    (Person (Triple "Carol"
      "carol@@abc.xyz" "444555")) Nil)) Nil))
```

```
SocialBook (Cons
  (SocialGroup "coworkers" (Cons
     "Alice" (Cons
     "Bob" Nil)))
  (Cons (SocialGroup "friends" (Cons
     "Carol" Nil)) Nil))
```

```
SocialBook (Cons
  (SocialGroup "friends" (Cons
     "Carol" (Cons
     "Alice" Nil))) (Cons
  (SocialGroup "coworkers" (Cons
     "Bob" Nil)) (Cons
  (SocialGroup "family" Nil) Nil)))
```

```
AddrBook
  (Cons (AddrGroup "friends" (Cons
    (Person (Triple "Carol"
      "carol@@abc.xyz" "444555")) (Cons
    (Person (Triple "Alice"
      "alice@@abc.xyz" "000111")) Nil)))
  (Cons (AddrGroup "coworkers" (Cons
    (Person (Triple "Bob"
      "bob@@abc.xyz" "222333")) Nil))
  (Cons (AddrGroup "family" Nil) Nil)))
```

Fig. 7. An example of XML synchronisation. (Grey areas highlight how the record `Alice` is updated.)

moved out of its original list-like structure (or *chunk* [**Barbosa2010Matching**]) and put into a new list-like structure, probably far away—because when using those languages, we usually lift a lens combinator k handling a single element to k^* dealing with a list of elements, so that the 'scope' of the alignment performed by k^* is always within that single list (it currently works on).

5.4 Resugaring

We have seen negation and `while` loops as syntactic sugar. The idea of *resugaring* is to print evaluation sequences in a core language using the constructs of its surface syntax (sugar) [**Pombrio2014Resugaring**, **Pombrio2015Hygienic**]. To solve the problem, **Pombrio2014Resugaring** enrich the AST to incorporate fields for holding tags that mark from which syntactic object an AST construct comes. Using retentive lenses, we can also solve the problem while leaving the AST clean—we can write consistency relations between the surface syntax and the abstract syntax and passing the generated *put* function proper links for retaining syntactic sugar, which we have already seen in the arithmetic expression example (where we retain the negation) and in the code refactoring example (where we retain the `while` loop). Both **Pombrio2014Resugaring**'s 'tag approach' and our 'link approach', in actuality, identifies where an AST construct comes from; however, the link approach has an advantage that it leaves ASTs clean and unmodified so that we do not need to patch up the existing compiler to deal with tags.

6 Related Work

6.1 Alignment

Alignment has been recognised as an important problem when we need to synchronise two lists. Our work is closely related to this.

Alignment for Lists The earliest lenses [**Foster2007Combinators**] only allow source and view elements to be matched positionally—the n-th source element is simply updated using the n-th element in the modified view. Later, lenses with more powerful matching strategies are proposed, such as dictionary lenses [**Bohannon2008Boomerang**] and their successor matching lenses [**Barbosa2010Matching**]. As for matching lenses, when a *put* is invoked, it will first find the correspondence between *chunks* (data structures that are reorderable, such as lists) of the old and new views using some predefined strategies; based on the correspondence, the chunks in the source are aligned to match the chunks in the new view. Then element-wise updates are performed on the aligned chunks. Matching lenses are designed to be practically easy to use, so they are equipped with a few fixed matching strategies (such as greedy align) from which the user can choose. However, whether the information is retained or not, still depends on the lens applied after matching. As a result, the more complex the applied lens is, the more difficult to reason about the information retained in the new source. Moreover, it suffers from a disadvantage that the alignment is only between a single source list and a single view list, as already discussed in the last paragraph of Sect. 5.3. BiFluX [**Pacheco2014BiFluX**] overcomes the disadvantage by providing the functionality that allows the user to write alignment strategies manually; in this way, when we see several lists at once, we are free to search for elements and match them in all the lists. But this alignment still has the limitation that each source element and each view element can only be matched at most once—after that they are classified as either *matched pair*, *unmatched source element*, or *unmatched view element*. Assuming that an element in the view has been copied several times, there is no way to align all the copies with the same source element. (However, it is possible to reuse an element several times for the handling of unmatched elements.)

In contrast, retentive lenses are designed to abstract out matching strategies (alignment) and are more like taking the result of matching as an additional input. This matching is not a one-layer matching but rather, a global one that produces (possibly all the) links between a source's and a view's unchanged parts. The information contained in the linked parts is preserved independently of any further applied lenses.

6.2 Provenance and Origin

Our idea of links is inspired by research on provenance [**Cheney2009Provenance**] in database communities and origin tracking [**vanDeursen1993Origin**] in the rewriting communities.

Cheney2009Provenance classify provenance into three kinds, *why*, *how*, and *where*: *why-provenance* is the information about which data in the view is from which rows in the source; *how-provenance* additionally counts the number of times a row is used (in the source); *where-provenance* in addition records the column where a piece of data is from. In our setting, we require that two pieces of data linked by vertical correspondence be equal (under a specific pattern), and hence the vertical correspondence resembles where-provenance. However, the above-mentioned provenance is not powerful enough as they are mostly restricted to relational data, namely rows of tuples—in functional programming, the algebraic data types are more complex. For this need, *dependency provenance* [**Cheney2011Provenance**] is proposed; it tells the user on which parts of a source the computation of a part of a view depends. In this sense, our consistency links are closer to dependency provenance.

The idea of inferring consistency links can be found in the work on origin tracking for term rewriting systems [**vanDeursen1993Origin**], in which the origin relations between rewritten terms can be calculated by analysing the rewrite rules statically. However, it was developed solely for building trace between intermediate terms rather than using trace information to update a tree further. Based on origin tracking, **deJonge2012Algorithm** implemented an algorithm for code refactoring systems, which 'preserves formatting for terms that are not changed in the (AST) transformation, although they may have changes in their subterms' [**deJonge2012Algorithm**]. This description shows that the algorithm also decomposes large terms into smaller ones resembling our regions. Therefore, in terms of the formatting aspect, we think that retentiveness can be in effect the same as their theorem if we adopt the 'square diagram' (see Sect. 6.4).

The use of consistency links can also be found in **Wang2011Incremental**'s work, where the authors extend state-based lenses and use links for tracing data in a view to its origin in a source [**Wang2011Incremental**]. When a sub-term in the view is edited locally, they use links to identify a sub-term in the source that 'contains' the edited sub-term in the view. When updating the old source, it is sufficient to only perform state-based *put* on the identified sub-term (in the source) so that the update becomes an incremental one. Since lenses generated by our DSL also create consistency links (albeit for a different purpose), they can be naturally incrementalised using the same technique.

6.3 Operation-based BX

Our work is closely relevant to the operation-based approaches to BX, in particular, the delta-based BX model [**Diskin2011Asymmetric**, **Diskin2011Symmetric**] and edit lenses [**Hofmann2012Edit**]. The (asymmetric) delta-based BX model regards the differences between a view state v and v' as *deltas*, which are abstractly represented as arrows (from the old view to the new view). The main law of the framework can be described as 'given a source state s and a view delta det_v, det_v should be translated to a source delta det_s between s and s' satisfying $get\ s' = v'$'. As the law only guarantees the existence of a source delta

det_s that updates the old source to a correct state, it is yet not sufficient to derive Retentiveness in their model, for there are infinite numbers of translated delta det_s which can take the old source to a correct state, of which only a few are 'retentive'. To illustrate, **Diskin2011Asymmetric** tend to represent deltas as edit operations such as *create*, *delete*, and *change*; representing deltas in this way will only tell the user what must be changed in the new source, while it requires additional work to reason about what is retained. However, it is possible to exhibit Retentiveness if we represent deltas in some other proper form. Compared to **Diskin2011Asymmetric**'s work, **Hofmann2012Edit** give concrete definitions and implementations for propagating edit operations (in a symmetric setting).

6.4 Opting for Triangular Diagrams

temporarily placed here

Including vertical correspondence (which represents view updates) in the theory was something we thought about (for quite some time), but eventually, we opted for the current, simpler theory. The rationale is that the original state-based lens framework (which we extended) does not really have the notion of view update built in. A view given to a *put* function is not necessarily modified from the view got from the source—it can be constructed in any way, and *put* does not have to consider how the view is constructed. Coincidentally, the paper about (symmetric) delta-based lenses [**Diskin2011Symmetric**] also introduces square diagrams and later switches to triangular diagrams.

We retain this separation of concern in our framework, in particular separating the jobs of retentive lenses and third-party tools that operate in a view update setting: third-party tools are responsible for producing vertical correspondence between the consistent view and a modified view (not between sources and views), and when it is time to run *put*, the vertical correspondence are composed (as shown Fig. 2) with the consistency links produced by *get* to compute the input links between the source and the modified view. We will present a concrete example regarding vertical correspondence and edit operations in Sect. 5.1.

7 Conclusion

In this paper, we showed that well-behavedness is not sufficient for retaining information after an update and it may cause problems in many real-world applications. To address the issue, we illustrated how to use links to preserve desired data fragments of the original source, and developed a semantic framework of (asymmetric) retentive lenses for region models. Then we presented a small DSL tailored for describing consistency relations between syntax trees; we showed its syntax, semantics, and proved that the pair of *get* and *put* functions generated from any program in the DSL form a retentive lens. We further illustrated the practical use of retentive lenses by giving examples in the field of code refactoring, XML synchronisation, and resugaring. In the related work, we discussed the relations with alignment, origin tracking, and operation-based BX. Discussions of the paper can be found in the appendix (Sect. C).

A Well-behaved Lenses Are Retentive Lenses

In Sect. 3, we say that retentive lenses are an extension of well-behaved lenses: every well-behaved lens between trees can be directly turned into a retentive lens (albeit in a trivial way).

Example 3 (Well-behaved Lenses are Retentive Lenses). Given a well-behaved lens defined by $g : S \to V$ and $p : S \times V \to S$, we define $get : S \nrightarrow V \times LinkSet$ and $put : S \times V \times LinkSet \nrightarrow S$ as follows:

$$
\begin{aligned}
get\ s &= (g\ s, trivial\ (s, g\ s)) \\
put\ (s, v, ls) &= p\ (s, v)
\end{aligned}
$$

where

$$trivial\ (s, v) = \{\, ((ToPat\ s, []) , (ToPat\ v, []))\, \}.$$

The idea is that *get* generates a link collection relating the whole source tree and view tree as two regions, and *put* accepts either the trivial link produced by *get* or an empty collection of links. Hippocraticness and Correctness hold because the underlying g and p are well-behaved. When the input link of *put* is empty, Retentiveness is satisfied vacuously; when the input link is the trivial link, Retentiveness is guaranteed by Hippocraticness of g and p, which is obviously seen from the proof below. Thus *get* and *put* indeed form a retentive lens.

Proof.

$$
\begin{aligned}
& get\ (put\ (s, v, trivial\ (s, v))) \\
=\{ & \ v = g\ s\ \} \\
& get\ (put\ (s, g\ s), trivial\ (s, g\ s)) \\
=\{ & \text{ Definition of } put \text{ and } (s, g\ s, trivial\ (s, g\ s)) \in \textsc{dom}\ put\ \} \\
& get\ (p\ (s, g\ s)) \\
=\{ & \text{ Hippocraticness of } g \text{ and } p\ \} \\
& get\ s \\
=\{ & \text{ Definition of } get\ \} \\
& (g\ s, trivial\ (s, g\ s)) \\
=\{ & \ v = g\ s\ \} \\
& (v, trivial\ (s, v))\ .
\end{aligned}
$$

B Programming Examples in Our DSL

Although the DSL is tailored for describing consistency relations between syntax trees, it is also possible to handle general tree transformations and the following are small but typical programming examples other than syntax tree synchronisation.

– Let us consider the binary trees

```
data BinT a = Tip | Node a (BinT a) (BinT a) .
```

We can concisely define the *mirror* consistency relation between a tree and its mirroring as

```
BinT Int <---> BinT Int
  Tip        ~  Tip
  Node i x y  ~  Node i y x .
```

– We demonstrate the implicit use of some other consistency relations when defining a new one. Suppose that we have defined the following consistency relation between natural numbers and boolean values:

```
Nat <---> Bool
  Succ _  ~  True
  Zero    ~  False .
```

Then we can easily describe the consistency relation between a binary tree over natural numbers and a binary tree over boolean values:

```
BinT Nat  <---> BinT Bool
  Tip           ~  Tip
  Node x ls rs  ~  Node x ls rs .
```

– Let us consider rose trees, a data structure mutually defined with lists:

```
data RTree a = RNode a (List (RTree a))
data List  a = Nil | Cons a (List a) .
```

We can define the following consistency relation to associate the left spine of a tree with a list:

```
RTree Int <---> List Int
  RNode i Nil         ~  Cons i Nil
  RNode i (Cons x _)  ~  Cons i x .
```

C Discussions of the Paper

We will briefly discuss Strong Retentiveness (that subsumes Hippocraticness), our thought on (retentive) lens composition, the feasibility of retaining code styles for refactoring tools, and our choice of the word 'retentive'.

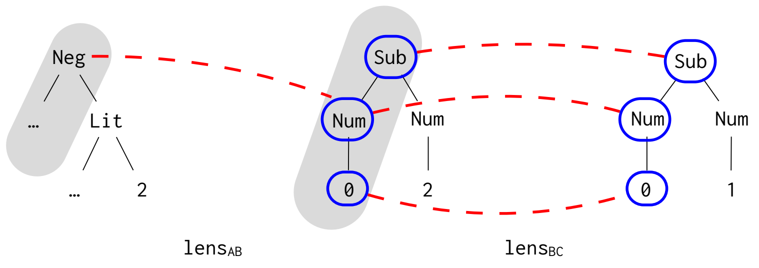
C.1 Strong Retentiveness

Through our research into Retentiveness, we also tried a different theory, which we call *Strong Retentiveness* now, that requires that the consistency links generated by *get* should additionally capture all the 'information' of the source and *uniquely identify* it. Strong Retentiveness is appealing in the sense that (we proved that) it subsumes Hippocraticness: the more information we require that the new source have, the more restrictions we impose on the possible forms of the new source; in the extreme case where the input links are consistency links—which capture all the information and can only be satisfied by at most one source—the new source has to be the same as the original one. However, using Strong Retentiveness demands extra effort in practice, for a set of region patterns can never uniquely identify a tree; as a result, much more information is required. For instance, `cst`$_1$ `= Minus "" (Lit 1) (Lit 2)` has region patterns `reg`$_1$ `= Minus "" _ _`, `reg`$_2$ `= Lit 1`, and `reg`$_3$ `= Lit 2`, which, however, are also satisfied by `cst`$_2$ `= Minus "" (Lit 2) (Lit 1)` in which regions are assembled in a different way.

This observation inspires us to generalise region patterns to *properties* in order for holding more information (that can eventually uniquely identify a tree) and generalise links connecting *Pattern* × *Path* to links connecting *Property* × *Path* accordingly. We eventually formalised three kinds of properties that are sufficient to capture all the information of a tree (e.g. `cst₁`): *region patterns* (e.g. `reg₁`, `reg₂`, and `reg₃`), *relative positions* between two regions (e.g. `reg₂` is the first child of `reg₁` and `reg₃` is the second child of `reg₁`), and *top* that marks the top of a tree (e.g. `reg₁` is the top). Worse still, observant readers might have found that properties need to be named so that they can be referred to by other properties; for instance, the region pattern `Minus "" _ _` is named `reg₁` and is referred to as the top of `cst₁`. This will additionally cause many difficulties in lens composition, as different lenses might assign the same region different names and we need to do 'alpha conversion'. Take everything into consideration, finally, we opted for the 'weaker' but simpler version of Retentiveness.

C.2 Rethinking Lens Composition

We defined retentive lens composition (Definition 4) in which we treat link composition as relation composition. In this case, however, the composition of two lenses $lens_{AB}$ and $lens_{BC}$ may not be satisfactory because the link composition might (trivially) produce an empty set as the result, if $lens_{AB}$ and $lens_{BC}$ decompose a tree b (of type B) in a different way, as the following example shows:



In the above figure, `lens`$_{AB}$ connects the region (pattern) `Neg a _` with `Sub (Num 0) _` (the grey parts); while `lens`$_{BC}$ decomposes `Sub (Num 0) _` into three small regions and establishes links for them respectively. For this case, our current link composition simply produces an empty set as the result.

Coincidentally, similar problems can also be found in quotient lenses [**Foster2008Quotient**]: A quotient lens operates on sources and views that are divided into many equivalent classes, and the well-behavedness is defined on those equivalent classes rather than a particular pair of source and view. In order to establish a sequential composition $l; k$, the authors require that the abstract (view-side) equivalence relation of lens l is identical to the concrete (source-side) equivalence of lens k. We leave other possibilities of link composition to future work.

As for our DSL, the lack of composition does not cause problems because of the philosophy of design. Take the scenario of writing a parser for example where there are two main approaches for the user to choose: to use parser combinators (such as Parsec) or to use parser generators (such as Happy). While parser combinators offer the user many small composable components, parser generators usually provide the user with a high-level syntax for describing the grammar of a language using production rules (associated with semantic actions). Then the generated parser is used as a 'standalone black box' and usually will not be composed with some others (although it is still possible to be composed 'externally'). Our DSL is designed to be a 'lens generator' and we have no difficulty in writing bidirectional transformations for the subset of Java 8 in Sect. 5.2.

C.3 Retaining Code Styles

A challenge to refactoring tools is to retain the style of program text such as indentation, vertical alignment of identifiers, and the place of line breaks. For example, an argument of a function application may be vertically aligned with a previous argument; when a refactoring tool moves the application to a different place, what should be retained is not the absolute number of spaces preceding the arguments but the *property* that these two arguments are vertically aligned.

Although not implemented in the DSL, these properties can be added to the set of *Property* as introduced in Sect. C.1.For instance, we may have $\mathsf{VertAligned}\ x\ y \in \mathit{Property}$ for $x, y \in \mathit{Name}$ (i.e. x and y are names of some regions); a CST satisfies such a property if region y is vertically aligned with region x.When *get* computes an AST from such a vertically aligned argument and produces consistency links, the links will not include (real) spaces preceding the argument as a part of the source region; instead, the links connect the property $\mathsf{VertAligned}\ x\ y$ (and the corresponding AST region).In the *put* direction, such links serve as directives to adjust the number of spaces preceding the argument to conform to the styling rule. In general, handling code styles can be very language-specific and is beyond the scope of this thesis but could be considered a direction of future work.

D Proofs About Retentive Lenses

D.1 Composability

In this section, we show the proof of Theorem 1 with the help of Fig. 3 and the definition of retentive lens composition (Definition 4).

Proof (Hippocraticness Preservation). We prove that the composite lens satisfies Hippocraticness with the help of Fig. 3 and the definition of retentive lens composition (Definition 4).
Let $get_{AC}\ a = (c, ls_{ac})$. We prove $put_{AC}\ (a, c', ls_{ac'}) = a' = a$. In this case, $c' = c$ and $ls_{ac'} = ls_{ac}$.

$$\begin{aligned}
&\quad put_{AC}\ (a, c', ls_{ac'}) \\
&= \{\ put_{AC}\ (a, c', ls_{ac'}) = a' = put_{AB}\ (a, b', ls_{ab'})\ \} \\
&\quad put_{AB}\ (a, b', ls_{ab'}) \\
&= \{\ ls_{ab'} = ls_{ac'} \cdot ls^{\circ}_{b'c'}\ \} \\
&\quad put_{AB}\ (a, b', ls_{ac'} \cdot ls^{\circ}_{b'c'}) \\
&= \{\ \text{Since } c' = c,\ \text{we have } ls_{ac'} = ls_{ac} \text{ and } ls_{b'c'} = ls_{b'c}\ \}
\end{aligned}$$

$$\begin{aligned}
& put_{AB}\ (a, b', (ls_{ac} \cdot ls^{\circ}_{b'c})^{\circ}) \\
={}& \{\ b' = put_{BC}\ (b, c', ls_{bc'}) \text{ and } c' = c\ \} \\
& put_{AB}\ (a, put_{BC}\ (b, c, ls_{bc}), ls_{ac} \cdot ls^{\circ}_{b'c}) \\
={}& \{\ \text{By Hippocraticness of } lens_{BC}, b' = put_{BC}\ (b, c, ls_{bc}) = b\ \} \\
& put_{AB}\ (a, b, ls_{ac} \cdot ls^{\circ}_{bc}) \\
={}& \{\ \text{The link composition is } \textcircled{6} \text{ in Fig. 3, and } b' = b\ \} \\
& put_{AB}\ (a, b, ls_{ab}) \\
={}& \{\ \text{Hippocraticness of } lens_{AB}\ \} \\
& a\ .
\end{aligned}$$

Proof (Correctness Preservation). We prove that the composite lens satisfies Correctness with the help of Fig. 3 and the definition of retentive lens composition (Definition 4).
Let $a' = put_{AC}\ (a, c', ls_{ac'})$, we prove $fst\ (get_{AC}\ a') = c'$.

$$\begin{aligned}
& fst\ (get_{AC}\ a') \\
={}& \{\ \text{Definition of } get_{AC}\ \} \\
& fst\ (get_{BC}\ (fst\ (get_{AB}\ a'))) \\
={}& \{\ a' = put_{AC}\ (a, c', ls_{ac'})\ \} \\
& fst\ (get_{BC}\ (fst\ (get_{AB}\ (put_{AC}\ (a, c', ls_{ac'}))))) \\
={}& \{\ put_{AC}\ (a, c', ls_{ac'}) = a' = put_{AB}\ (a, b', ls_{ab'})\ \} \\
& fst\ (get_{BC}\ (fst\ (get_{AB}\ (put_{AB}\ (a, b', ls_{ab'}))))) \\
={}& \{\ \text{Correctness of } lens_{AB}\ \} \\
& fst\ (get_{BC}\ (b')) \\
={}& \{\ b' = put_{BC}\ (b, c', ls_{bc'})\ \} \\
& fst\ (get_{BC}\ (put_{BC}\ (b, c', ls_{bc'}))) \\
={}& \{\ \text{Correctness of } lens_{BC}\ \} \\
& c'\ .
\end{aligned}$$

Proof (Retentiveness Preservation). In Fig. 3, we prove $fst \cdot ls_{ac} \subseteq fst \cdot ls_{a'c'}$.
To finish the proof, we need the following lemma.

Lemma 1. *Given a relation R and a function f, we have*

$$\begin{aligned}
\textsc{rdom}(f \cdot R) &= \textsc{rdom}R \quad \textit{if } \textsc{ldom}R \subseteq \textsc{rdom}f\ , \textit{ and} \\
\textsc{ldom}(R \cdot f) &= \textsc{ldom}R \quad \textit{if } \textsc{rdom}R \subseteq \textsc{ldom}f\ .
\end{aligned}$$

Proof. We prove the first equation; the second equation is symmetric.
Suppose $f : X \to Y$ and $R : Y \sim Z$. By definition, $\textsc{rdom}R = \{z \in Z \mid \exists y \in Y,\ y\ R\ z\}$ and $\textsc{rdom}(f \cdot R) = \{z \in Z \mid \exists y \in Y,\ \exists\ x \in X,\ x\ f\ y\ R\ z\}$. Since $\textsc{ldom}R \subseteq \textsc{rdom}f$, we know that $\forall y.\ y \in \textsc{ldom}R \Rightarrow y \in \textsc{rdom}f$; on the other hand, we also have $y \in \textsc{rdom}f \Rightarrow \exists x.\ x \in X$. Therefore, $\forall y.\ y \in \textsc{ldom}R \Rightarrow \exists x.\ x \in X$ and thus $\textsc{rdom}(f \cdot R) = \{z \in Z \mid \exists y \in Y,\ \exists\ x \in X,\ x\ f\ y\ R\ z\} = \{z \in Z \mid \exists y \in Y,\ y\ R\ z\} = \textsc{rdom}R$.

Now, we present the main proof:

$$\begin{aligned}
& fst \cdot ls_{ac} \\
={}& \{\ R = R \cdot id_{\textsc{rdom}R}\ \} \\
& fst \cdot ls_{ac'} \cdot id_{\textsc{rdom}(ls_{ac'})} \\
\subseteq{}& \{\ id_{\textsc{rdom}(ls_{ac'})} \subseteq ls_{c'b'} \cdot ls^{\circ}_{c'b'} \text{ by sub-proof-1 below}\ \} \\
& fst \cdot ls_{ac'} \cdot (ls_{c'b'} \cdot ls^{\circ}_{c'b'})
\end{aligned}$$

$$
\begin{aligned}
&=\{\text{ Relation composition is associative }\}\\
&\quad \mathit{fst} \cdot (ls_{ac'} \cdot ls_{c'b'}) \cdot ls^{\circ}_{c'b'}\\
&=\{\ ls_{ab'} = ls_{ac'} \cdot ls_{c'b'}\ (⑥ \text{ in Fig. 3})\ \}\\
&\quad \mathit{fst} \cdot ls_{ab'} \cdot ls^{\circ}_{c'b'}\\
&\subseteq\{\text{ Retentiveness of } lens_{AB} \text{ and } ls^{\circ}_{c'b'} = ls_{b'c'}\ \}\\
&\quad \mathit{fst} \cdot ls_{a'b'} \cdot ls_{b'c'}\\
&=\{\ ls_{a'c'} = ls_{a'b'} \cdot ls_{b'c'}\ \}\\
&\quad \mathit{fst} \cdot ls_{a'c'}\ .
\end{aligned}
$$

sub-proof-1: $id_{\text{RDOM}(ls_{ac'})} \subseteq ls_{c'b'} \cdot ls^{\circ}_{c'b'} \Leftrightarrow \text{RDOM}(ls_{ac'}) \subseteq \text{LDOM}(ls_{c'b'})$ and we prove the latter using linear proofs. The right column of each line gives the reason how it is derived.

1.	$\text{LDOM}(ls_{c'b'}) = \text{RDOM}(ls_{b'c'})$	definition of relations
2.	$\mathit{fst} \cdot ls_{bc'} \subseteq \mathit{fst} \cdot ls_{b'c'}$	Retentiveness of $lens_{BC}$
3.	$\text{RDOM}(\mathit{fst} \cdot ls_{bc'}) \subseteq \text{RDOM}(\mathit{fst} \cdot ls_{b'c'})$	2 and definition of relation inclusion
4.	$\text{RDOM}(ls_{bc'}) \subseteq \text{RDOM}(ls_{b'c'})$	3 and Lemma 1
5.	$ls_{bc'} = (ls^{\circ}_{ac'} \cdot ls_{ab})^{\circ} = (ls_{c'a} \cdot ls_{ab})^{\circ}$	③ in Fig. 3
6.	$\text{RDOM}(ls_{bc'}) = \text{RDOM}(ls_{c'a} \cdot ls_{ab})^{\circ}$	5
7.	$\text{RDOM}(ls_{c'a} \cdot ls_{ab})^{\circ} = \text{LDOM}(ls_{c'a} \cdot ls_{ab})$	definition of converse relation
8.	$\text{LDOM}(ls_{c'a} \cdot ls_{ab}) \subseteq \text{LDOM}(ls_{c'a})$	definition of relation composition
9.	$\text{LDOM}(ls_{c'a}) = \text{RDOM}(ls_{ac'})$	definition of converse relation
10.	$\text{RDOM}(ls_{bc'}) = \text{RDOM}(ls_{ac'})$	6, 7, 8, and 9
11.	$\text{RDOM}(ls_{ac'}) \subseteq \text{LDOM}(ls_{c'b'})$	10, 4, and 1

D.2 Retentiveness of the DSL

In this section, we prove that the *get* and *put* semantics given in Sect. 4.3 does satisfy the three properties (Definition 3) of a retentive lens. Most of the proofs are proved by induction on the size of the trees.

Lemma 2. *The get function described in Sect. 4.3 is total.*

Proof. Because we require source pattern coverage, *get* is defined for all the input data. Besides, since our DSL syntactically restricts source pattern $spat_k$ to not being a bare variable pattern, for any $v \in Vars(spat_k)$, $decompose(spat_k, s)$ is a proper subtree of s. So the recursion always decreases the size of the s parameter and thus terminates.

Lemma 3. *For a pair of get and put described in Sect. 4.3 and any $s : S$, $check(s, get(s)) = \top$.*

Proof. We prove the lemma by induction on the structure of s. By the definition of *get* and *check*,

$$
\begin{aligned}
&\quad check(s, get(s))\\
&=\{\ get(s) \text{ produces consistency links }\}\\
&\quad chkWithLink(s, get(s))\\
&=\{\text{ Unfolding } get(s)\ \}
\end{aligned}
$$

$$chkWithLink(s, reconstruct(vpat_k, fst \circ vls), l_{root} \cup links)$$

where $vpat_k$, fst, vls, l_{root} and $links$ are those in the definition of get (4). In $chkWithLink$, $cond_1$ and $cond_2$ are true by the evident semantics of pattern matching functions such as $isMatch$ and $reconstruct$. $cond_3$ is true following the definition of l_{root}, $links$, and $divide$. Finally, $cond_4$ is true by the inductive hypothesis.

Lemma 4. *(Focusing) If $sel(s, p) = s'$ and for any $((\_, spath), (\_, \_)) \in ls$, p is a prefix of $spath$, then*

$$put(s, v, ls) = put(s', v, ls') \ \ \textit{and} \ \ check(s, v, ls) = check(s', v, ls')$$

where $ls' = \{\, ((a, b), (c, d)) \mid ((a, p +\!\!+ b), (c, d)) \,\}$.

Proof. From the definitions of put and $check$, we find that their first argument (of type S) is invariant during the recursive process. In fact, the first argument is only used when checking whether a link in ls is valid with respect to the source tree. Since all links in ls connect to the subtree s', the parts in s above s' can be trimmed and the identity holds.

Theorem 3. *(Hippocraticness of the DSL) For any s of type S,*

$$put(s, get(s))^{1} = s \ .$$

Proof (Proof of Hippocraticness). Also by induction on the structure of s,

$$\begin{aligned} & put(s, get(s)) \\ =& \{ \text{ Unfolding } get(s) \ \} \\ & put(s, reconstruct(vpat_k, fst \circ vls), l_{root} \cup links) \ , \end{aligned}$$

where $spat_k \sim vpat_k \in R$ is the unique rule such that $spat_k$ matches s. l_{root}, $links$, and vls are defined exactly the same as in get (4).

Now we expand put. Because l_{root} links to the root of the view, put falls to its second case.

$$put(s, get(s)) = inj(reconstruct(spat'_k, ss)) \tag{9}$$

where

$$\begin{aligned} & spat'_k \\ =& \{ \ spat \text{ in (7) is } eraseVars(fillWildcards(spat_k, s)) \ \} \\ & fillWildcards(spat_k, eraseVars(fillWildcards(spat_k, s))) \\ =& \{ \text{ See Fig. 8 } \} \\ & fillWildcards(spat_k, s) \ . \end{aligned}$$

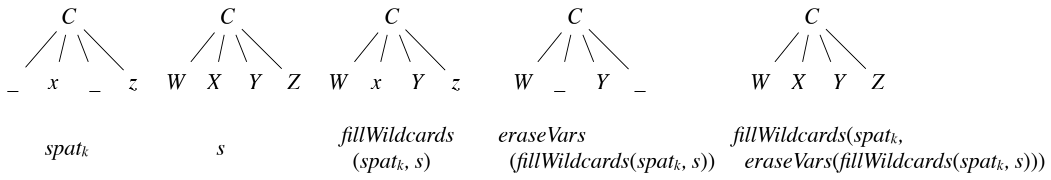
and

$$ss = \lambda(t \in Vars(spat_k)) \rightarrow put(s, vs(t), divide(Path(vpat_k, t), links))$$

where $vs = decompose(vpat_k, reconstruct(vpat_k, fst \circ vls)) = fst \circ vls$. (See the beginning of the proof.) Since $vls = get \circ decompose(spat_k, s)$, we have

$$\begin{aligned} ss = \lambda(t \in Vars(spat_k)) \rightarrow \\ & put(s, fst(get(decompose(spat_k, s)(t))), divide(Path(vpat_k, t), links)) \end{aligned}$$

[1]For simplicity, we regard $(a, (b, c))$ the same as (a, b, c).



C
_ x _ z
spat_k
C
W X Y Z
s
C
W x Y z
fillWildcards (spat_k, s)
C
W _ Y _
eraseVars (fillWildcards(spat_k, s))
C
W X Y Z
fillWildcards(spat_k, eraseVars(fillWildcards(spat_k, s)))

Fig. 8. A property regarding *fillWildcards*.

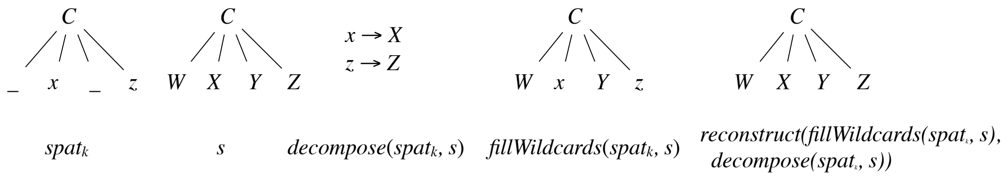


Fig. 9. A property regarding Reconstruct-Decompose.

By Lemma 4, we have

$$\begin{aligned}
&ss = \lambda(t \in \mathit{Vars}(\mathit{spat}_k)) \rightarrow \\
&\qquad\qquad \mathit{put}(\mathit{decompose}(\mathit{spat}_k, s)(t), \mathit{fst}(\mathit{get}(\mathit{decompose}(\mathit{spat}_k, s)(t))), \\
&\qquad\qquad\quad \mathit{snd}(\mathit{get}(\mathit{decompose}(\mathit{spat}_k, s)(t)))) \\
&\quad = \{ \text{ Inductive hypothesis for } \mathit{decompose}(\mathit{spat}_k, s)(t) \ \} \\
&\quad\ \ \lambda(t \in \mathit{Vars}(\mathit{spat}_k)) \rightarrow \mathit{decompose}(\mathit{spat}_k, s)(t) \\
&\quad = \mathit{decompose}(\mathit{spat}_k, s) \ .
\end{aligned}$$

Now, we substitute $\mathit{fillWildcards}(\mathit{spat}_k, s)$ for spat'_k and $\mathit{decompose}(\mathit{spat}_k, s)$ for ss in equation (9), and obtain

$$\begin{aligned}
&\ \ \mathit{put}(s, \mathit{get}(s)) \\
&=\{ \ \mathit{Equation}(9) \ \} \\
&\ \ \mathit{inj}_{SS}(\mathit{reconstruct}(\mathit{spat}'_k, ss)) \\
&=\mathit{inj}_{SS}(\mathit{reconstruct}(\mathit{fillWildcards}(\mathit{spat}_k, s), \mathit{decompose}(\mathit{spat}_k, s))) \\
&=\{ \text{ See Fig. 9 } \} \\
&\ \ \mathit{inj}_{SS}(s) \\
&=s \ .
\end{aligned}$$

This completes the proof of Hippocraticness.

Theorem 4. *(Correctness of the DSL) For any* (s, v, ls) *that makes* $\mathit{check}(s, v, \mathit{ls}) = \top$*,* $\mathit{get}(\mathit{put}(s, v, \mathit{ls})) = (v, \mathit{ls}')$*, for some* ls'*.*

Proof (Proof of Correctness). We prove Correctness by induction on the size of (v, ls). The proofs of the two cases of put are quite similar, and therefore we only present the first one, in which $\mathit{put}(s, v, \mathit{ls})$ falls into the first case of put:

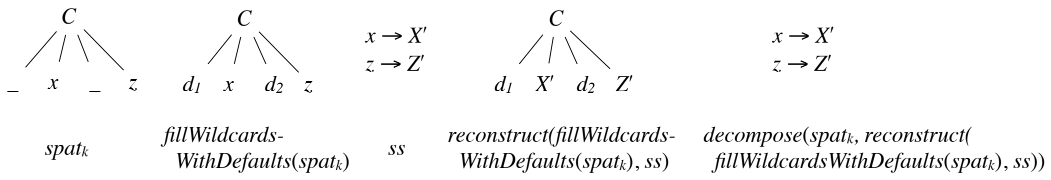

C
_
x
_
z
spat_k
C
d1
x
d2
z
fillWildcards-WithDefaults(spat_k)
x → X′
z → Z′
ss
C
d1
X′
d2
Z′
reconstruct(fillWildcards-WithDefaults(spat_k), ss)
x → X′
z → Z′
decompose(spat_k, reconstruct(fillWildcardsWithDefaults(spat_k), ss))

Fig. 10. A property regarding Decompose-Reconstruct.

i.e. $\mathit{put}(s, v, \mathit{ls}) = \mathit{reconstruct}(\mathit{fillWildcardsWithDefaults}(\mathit{spat}_k), \mathit{ss})$. Then

$$\mathit{get}(\mathit{put}(s, v, \mathit{ls})) = \mathit{get}(\mathit{reconstruct}(\mathit{fillWildcardsWithDefaults}(\mathit{spat}_k), \mathit{ss}))$$

where $\mathit{spat}_k \sim \mathit{vpat}_k \in R$, $\mathit{isMatch}(\mathit{vpat}_k, v) = \top$, and

$$\begin{aligned} \mathit{ss} = \lambda(t \in \mathit{Vars}(\mathit{spat}_k)) \to \\ \mathit{put}(s, \mathit{decompose}(\mathit{vpat}_k, v)(t), \mathit{divide}(\mathit{Path}(\mathit{vpat}_k, t), \mathit{ls})) \ . \end{aligned}$$

Now expanding the definition of get, because of the disjointness of source patterns, the same $\mathit{spat}_k \sim \mathit{vpat}_k \in R$ will be select again. Thus

$$\mathit{get}(\mathit{put}(s, v, \mathit{ls})) = (\mathit{reconstruct}(\mathit{vpat}_k, \mathit{fst} \circ \mathit{vls}), \cdots)$$

where

$$\begin{aligned} \mathit{vls} &= \mathit{get} \circ \mathit{decompose}(\mathit{spat}_k, \mathit{put}(s, v, \mathit{ls})) \\ &= \mathit{get} \circ \mathit{decompose}(\mathit{spat}_k, \mathit{reconstruct}(\mathit{fillWildcardsWithDefaults}(\mathit{spat}_k), \mathit{ss})) \\ &= \{ \text{ See Fig. 10 } \} \\ &\mathit{get} \circ \mathit{ss} \\ &= \lambda(t \in \mathit{Vars}(\mathit{spat}_k)) \to \mathit{get}(\mathit{put}(s, \mathit{decompose}(\mathit{vpat}_k, v)(t), \mathit{divide}(\mathit{Path}(\mathit{vpat}_k, t), \mathit{ls}))) \end{aligned}$$

To proceed, we want to use the inductive hypothesis to simplify $\mathit{get}(\mathit{put}(\cdots))$. When vpat_k is not a bare variable pattern, $\mathit{decompose}(\mathit{vpat}_k, v)(t)$ is a proper subtree of v and the size of the third argument (i.e. links ls) is non-increasing; thus the inductive hypothesis is applicable. On the other hand, if vpat_k is a bare variable pattern, the sizes of all the arguments stays the same; but cond_3 in $\mathit{chkNoLink}$ guarantees that in the next round of the recursion, a pattern vpat_k that is not a bare variable pattern will be selected. Therefore we can still apply the inductive hypothesis. Applying the inductive hypothesis, we get

$$\mathit{vls} = \lambda(t \in \mathit{Vars}(\mathit{spat}_k)) \to (\mathit{decompose}(\mathit{vpat}_k, v)(t), \cdots)$$

Thus $\mathit{get}(\mathit{put}(s, v, \mathit{ls})) = (\mathit{reconstruct}(\mathit{vpat}_k, \mathit{decompose}(\mathit{vpat}_k, v)), \cdots) = (v, \cdots)$, which completes the proof of Correctness.

Theorem 5. *(Retentiveness of the DSL) For any* (s, v, ls) *that* $\mathit{check}(s, v, \mathit{ls}) = \top$, $\mathit{get}(\mathit{put}(s, v, \mathit{ls})) = (v', \mathit{ls}')$, *for some* v' *and* ls' *such that*

$$\begin{aligned} &\{ (\mathit{spat}, (\mathit{vpat}, \mathit{vpath})) \mid ((\mathit{spat}, \mathit{spath}), (\mathit{vpat}, \mathit{vpath})) \in \mathit{ls} \} \\ \subseteq &\{ (\mathit{spat}, (\mathit{vpat}, \mathit{vpath})) \mid ((\mathit{spat}, \mathit{spath}), (\mathit{vpat}, \mathit{vpath})) \in \mathit{ls}' \} \end{aligned}$$

Proof (Proof of Retentiveness). Again, we prove Retentiveness by induction on the size of (v, ls). The proofs of the two cases of put are similar, and thus we only show the second one here.

If there is some $l = ((spat, spath),\ (vpat, [\,])) \in ls$, let $spat_k \sim vpat_k$ be the unique rule in $S \sim V$ that $isMatch(spat_k, spat) = \top$. We have

$$\begin{aligned}
&get(put(s, v, ls))\\
=&\{\ \text{Definition of } put\ \}\\
&get(inj_{\ TypeOf(spat_k) \to S}(s'))\\
=&\{\ get(inj(s)) = get(s) \text{ as shown in (Sect. 4.2)}\ \}\\
&get(s')
\end{aligned}$$

where $s' = reconstruct(fillWildcards(spat_k, spat), ss)$ and

$$\begin{aligned}
ss = \lambda(t \in Vars(spat_k)) \to&\\
&put(s, decompose(vpat_k, v)(t), divide(Path(vpat_k, t), ls \setminus \{\, l\, \}))
\end{aligned}$$

Now we expand the definition of get (and focus on the links)

$$get(put(s, v, ls)) = (\cdots, \{\, l_{root}\, \} \cup links)$$

where $l_{root} = \big((eraseVars(fillWildcards(spat_k, s')), [])\ ,\ (eraseVars(vpat_k), [])\big)$,

$$\begin{aligned}
links = \{\ &((a, Path(spat_k, t) +\!\!+\ b),\ (c, Path(vpat_k, t) +\!\!+\ d))\\
&\mid t \in Vars(vpat_k), ((a, b),\ (c, d)) \in snd(vls(t))\ \} \text{ ,and}
\end{aligned} \tag{10}$$

$$\begin{aligned}
&vls(t)\\
=&\{\ \text{Unfolding } vls\ \}\\
&(get \circ decompose(spat_k, s'))(t)\\
=&\{\ \text{Unfolding } s'\ \}\\
&(get \circ decompose(spat_k, reconstruct(fillWildcards(spat_k, spat), ss)))(t)\\
=&\{\ \text{Similar to the case shown in Fig. 10}\ \}\\
&(get \circ ss)(t)\\
=&\{\ \text{Definition of } ss\ \}\\
&get(put(s, decompose(vpat_k, v)(t), divide(Path(vpat_k, t), ls \setminus \{\, l\, \})))\ .
\end{aligned}$$

For l_{root}, we have

$$\begin{aligned}
&eraseVars(fillWildcards(spat_k, s'))\\
=&\{\ \text{Unfolding } s'\ \}\\
&eraseVars(fillWildcards(spat_k, reconstruct(fillWildcards(spat_k, spat), ss)))\\
=&\{\ \text{See Fig. 11}\ \}\\
&eraseVars(fillWildcards(spat_k, spat))\\
=&\{\ \text{By } cond_2 \text{ in } chkWithLink\ \}\\
&spat
\end{aligned}$$

Use the first clause of $cond_2$, we have $vpat = eraseVars(vpat_k)$. Thus

$$l_{root} = \big((eraseVars(fillWildcards(spat_k, s')), [])\ ,\ (eraseVars(vpat_k), [])\big) = ((spat, [\,]),\ (vpat, [\,]))\ ,$$

and therefore the input link $l = ((spat, spath),\ (vpat, [\,]))$ is 'preserved' by l_{root}, i.e. $fst \cdot \{l\} = fst \cdot \{l_{root}\}$.

For the links in $ls \setminus \{\, l\, \}$, we show that they are preserved in $links$ (10) above. By $cond_3$ in $chkWithLink$, for every link $m \in ls \setminus \{\, l\, \}$, there is some t_m in $Vars(spat_k)$ such that

$$m \in addVPrefix(Path(vpat_k, t_m), divide(Path(vpat_k, t_m), ls \setminus \{\, l\, \})).$$

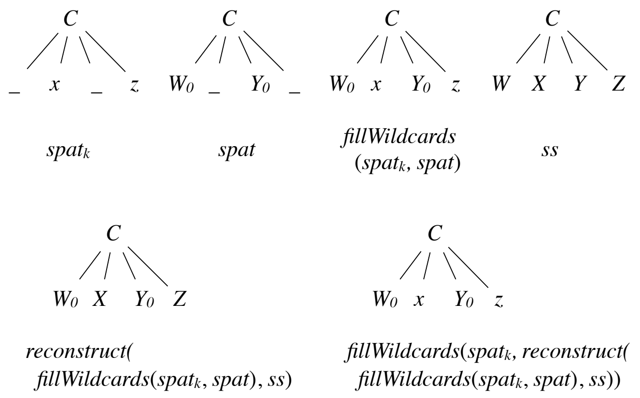

C
_ x _ z
spat_k
C
W0 _ Y0 _
spat
C
W0 x Y0 z
fillWildcards (spat_k, spat)
C
W X Y Z
ss
C
W0 X Y0 Z
reconstruct(fillWildcards(spat_k, spat), ss)
C
W0 x Y0 z
fillWildcards(spat_k, reconstruct(fillWildcards(spat_k, spat), ss))

Fig. 11. Another property regarding *fillWildcards*.

If $m = ((a, b), (c, \mathit{Path}(\mathit{vpat}_k, t_m) \mathbin{+\!\!+} d))$, then

$$m' = ((a, b),\ (c, d)) \in \mathit{divide}(\mathit{Path}(\mathit{vpat}_k, t_m), \mathit{ls} \setminus \{\, l \,\}).$$

By the inductive hypothesis for $\mathit{snd}(\mathit{vls}(t_m))$, m' is 'preserved', that is

$$\exists b'.\, ((a, b'),\ (c, d)) \in \mathit{snd}(\mathit{vls}(t_m))$$

Now by the definition of *links* (10), $((a, \mathit{Path}(\mathit{spat}_k, t_m) \mathbin{+\!\!+} b'), (c, \mathit{Path}(\mathit{vpat}_k, t_m) \mathbin{+\!\!+} d)) \in \mathit{links}$, therefore m is also preserved.

Corollary 1. *Let $\mathit{put}' = \mathit{put}$ with its domain intersected with $S \times V \times \mathit{LinkSet}$, get and put' form a retentive lens as in Definition 3 since they satisfy Hippocraticness (1), Correctness (2) and Retentiveness (3).*