Retentive Lenses

ANONYMOUS AUTHOR(S)

Based on Foster et al.'s [2007] 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, since Hippocraticness only requires that the source be unchanged if the view is not modified. Thus nothing about retaining is guaranteed when the view is changed.

To address this problem, we propose an extension of the original lenses, called *retentive lenses*, satisfying a new Retentiveness law which can guarantee 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 study the practical use of retentive lenses by implementing in our DSL a synchroniser between concrete and abstract syntax trees for a small subset of Java, and demonstrate that Retentiveness helps to enforce the retaining of comments and syntactic sugar in code refactoring.

CCS Concepts: • **Software and its engineering → Domain specific languages**;

Additional Key Words and Phrases: bidirectional transformations, asymmetric lenses

ACM Reference Format:
Anonymous Author(s). 2018. Retentive Lenses. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 29 pages.

1 INTRODUCTION

We often need to write pairs of programs to synchronise data. Typical examples include view querying and updating in relational databases [Bancilhon and Spyratos 1981] for keeping a database and its view in sync, text file format conversion [MacFarlane 2013] (e.g., between Markdown and HTML) for keeping their content and common formatting in sync, and parsers and printers as front ends of compilers [Rendel and Ostermann 2010] for keeping program text and its abstract representation in sync. Asymmetric *lenses* [Foster et al. 2007] 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 (Section 5) 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.

Let us first review the definition of well-behaved lenses, borrowing some of Stevens's [2008] terminologies. 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. (For example, the abstract representation of a piece of program text is a view of the latter, since program text contains information like comments and layouts not preserved in the abstract representation.) 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. (For example, a piece of program text is consistent with an abstract representation if the latter indeed represents the abstract structure of the former.) Since S contains more information than V, we expect that there is

2018. 2475-1421/2018/1-ART1 $15.00
https://doi.org/

a function $get : S \longrightarrow V$ 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, as reasoned by Stevens, should coincide with R extensionally — that is, $s : S$ and $v : V$ are related by R if and only if $get(s) = v$.[1] Consistency restoration in the other direction is performed by another function $put : S \times V \longrightarrow S$, which produces an updated source that is consistent with the input view and can retain some information of the input source (e.g., comments in the original program text). Well-behavedness consists of two laws regarding the restoration behaviour of *put* with respect to *get* (and thus the consistency relation R):

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

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

Correctness states that necessary changes must be made by *put* such that the updated source is consistent with the view, and 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.

An important synchronisation problem that lenses appear to be able to address is code refactoring [Fowler and Beck 1999]. Let us walk through a concrete scenario illustrated in Figure 1. 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 *for-each loop* (also known as an *enhanced for 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 abstract syntax tree (AST), perform code refactoring on the AST, and regenerate new program text. The bottom-left corner of Figure 1 shows the desired program text after refactoring, where we see that the comment associated with *fuel* is also pushed down, and the *for-each* 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, include neither comments nor the form of the original `for` loop. So if the *print* function in Figure 1 takes only the AST as input, it can only give the user an unsatisfactory result in which much information is lost (the comment and *for-each* syntactic sugar), like the one shown in Figure 2. This is a well-known problem [de Jonge and Visser 2012; Fritzson et al. 2008; Li and Thompson 2006], and one way to remedy this is to implement the transformations between program text and ASTs using lenses [Martins et al. 2014; Zhu et al. 2016], with *parse* as *get* and *print* as *put*, so that the additional information in program text can be retained by *print*/*put*.

This attempt, however, reveals a fundamental weakness about the definition of lenses: while lenses are designed to retain information, well-behavedness actually says very little about the retaining of information. In our refactoring scenario, knowing that a lens between program text and ASTs is well-behaved does not preclude the unsatisfactory outcome in Figure 2: Correctness only requires that the updated program text be parsed to the refactored AST, and this does not require the comment and syntactic sugar to be preserved; Hippocraticness has effect only when the AST is not modified, whereas the AST has been changed in this case. The root cause of the unintended *put* behaviour is that Hippocraticness only requires the *whole* source to be unchanged

[1] Therefore it is only sensible to consider functional consistency relations in the asymmetric setting.

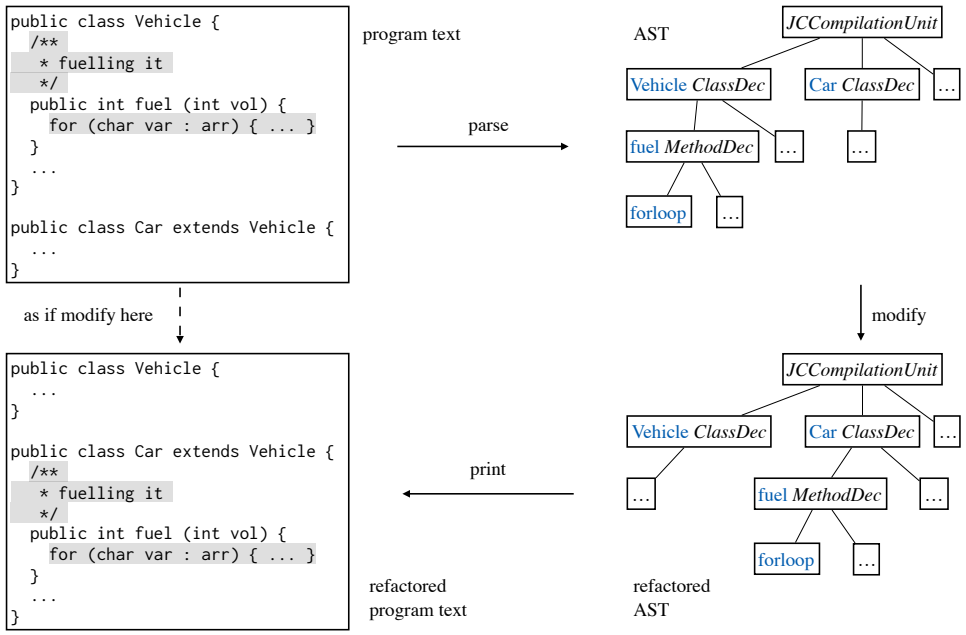


Fig. 1. An example of the *push-down* code refactoring. (For simplicity, subclasses `Bus` and `Bicycle` are omitted.)

```
public class Vehicle {
  ...
}

public class Car extends Vehicle {
  public int fuel (int vol) {
    for (int i = 0; i < arr.length;
         i++) {
      char var = arr[i]...
    }
  }
  ...
}
```

Fig. 2. An unsatisfactory result after refactoring, losing the comment and *for-each* syntactic sugar.

if the *whole* view is, but this is too 'global' as we have seen, and it is desirable to have a law that gives such a guarantee more 'locally'.

To replace Hippocraticness with a finer-grained property, we propose an extension of the original lenses, called *retentive 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. As the view is modified, this set of links is also updated to keep track of this 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 a new law, which we call *Retentiveness*, guarantees that those parts in the original source that still correspond to some parts of the modified view are retained in the right place in the updated source computed by *put*. (In the refactoring scenario, after the AST is modified, the comment and the *for-each* loop in the

original program text are still linked to the moved methods in the AST, and will be guaranteed to be retained in the updated program text by Retentiveness.) Notably, if the view is not modified and the link structure produced by *get* is kept intact, then Retentiveness will guarantee that the source is not modified by *put* — that is, Hippocraticness is subsumed by Retentiveness and becomes a special case.

In the rest of the paper, we assume that the grammar of a programming language is unambiguous so that the synchronisation between program text and its AST can be reduced to the synchronisation between the program text's concrete syntax trees (CSTs) and ASTs [Zhu et al. 2016]. This allows us to focus on a retentiveness framework for algebraic data types (i.e., trees), instead of unstructured data like program text (strings). The following are contributions of this paper:

- We formalise the idea of links, retentive lenses and the law of Retentiveness (Section 2).
- We present a domain-specific language tailored for writing retentive lenses between trees (Section 3).
- We study the practical use of retentive lenses by writing the synchronisation between syntax trees of (a small subset of) Java 8 and demonstrate that Retentiveness helps to retain comments and syntactic sugar after code refactoring. (Section 4)

In Section 5, we present related work regarding various alignment strategies for lenses, provenance and origin between two pieces of data, and operational-based bidirectional transformations (BX). In Section 6, we conclude the paper and discuss our choice of opting for triangular diagrams, composability of retentive lenses, and the feasibility of retaining code styles for refactoring tools in our framework.

2 RETENTIVE LENSES FOR TREES

In this section, we will develop a definition of retentive lenses for algebraic data types. After formalising some auxiliary concepts in Sections 2.3 and 2.4, in Section 2.5 we will make the following revisions to the definition of a classic lens [Foster et al. 2007] (given in Section 1):

- extending *get* and *put* to incorporate links — specifically, we will make *get* return a collection of consistency links, and make *put* take a collection of input links as an additional argument — and
- replacing Hippocraticness with a finer-grained law *Retentiveness*, which formalises the statement 'if parts of the view are unchanged then the corresponding parts of the source should be retained',

and show that Hippocraticness becomes a natural consequence of Retentiveness. Before giving the formal definitions, we will start by introducing a 'region model' for decomposing trees, using the concrete and abstract representations of arithmetic expressions as a concrete example (Section 2.1), and also provide a high-level sketch of how retentive lenses work on top of this region model (Section 2.2).

2.1 Regions

Let us first get familiar with the concrete and abstract representations of arithmetic expressions — which our examples are based on — as defined in Figure 3, where the definitions are 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 explicit parentheses, negations, and annotations. The consistency relations between CSTs and ASTs

```
data Expr  = Plus  Annot Expr Term
           | Minus Annot Expr Term
           | FromT Annot Term
data Term  = Lit   Annot Int
           | Neg   Annot Term
           | Paren Annot Expr
type Annot = String
data Arith = Add   Arith Arith
           | Sub   Arith Arith
           | Num   Int
```

```
getE :: Expr → Arith
getE (Plus  _ e t) = Add (getE e) (getT t)
getE (Minus _ e t) = Sub (getE e) (getT t)
getE (FromT _   t) = getT t
getT :: Term → Arith
getT (Lit   _ i) = Num i
getT (Neg   _ t) = Sub (Num 0) (getT t)
getT (Paren _ e) = getE e
```

Fig. 3. Data types for concrete and abstract syntax of the arithmetic expressions and the consistency relations between them as *getE* and *getT* functions in Haskell.

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 Section 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 fragment of a tree at a specific location and whose data are described by a *region pattern* that consists of wildcards, constructors, and constants. For example, in Figure 4, 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. This is one particular way of decomposing trees, and we call this the *region model*.

2.2 A High-Level Sketch of Retentive Lenses

With the region model, the *get* function in a retentive lens not only computes a view but also decomposes both the source and view into regions and produces a collection of links relating source and view regions. We call this collection of links produced by *get* the *consistency links* because they are a finer-grained representation of how the whole source is consistent with the whole view. In Figure 4, for example, the light dashed links between the source *cst* and the view *ast* = *getE cst* are two of the consistency links. (For clarity, we do not draw all the consistency links in the figure — the complete collection of consistency links should fully describe how all the source and view regions correspond.) 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*.

As the view is modified, the links between the source and view are also modified to reflect the latest correspondences between regions. For example, in Figure 4, if we change *ast* to *ast′* by swapping the two subtrees under *Add*, then the 'diagonal' link relating the *Neg* `"a neg"` _ region and the *Sub* (*Num* 0) _ region can be created to record that the two regions are still 'locally consistent' despite that the source and view as a whole are no longer consistent. In general, the collection of diagonal links between *cst* and *ast′* may be created in many ways, such as directly comparing *cst* and *ast′* and producing links between matching areas, or composing the light red dashed consistency links between *cst* and *ast* and the light blue dashed 'vertical correspondence' between *ast* and *ast′*. How these links are obtained is a separable concern, though, and in this

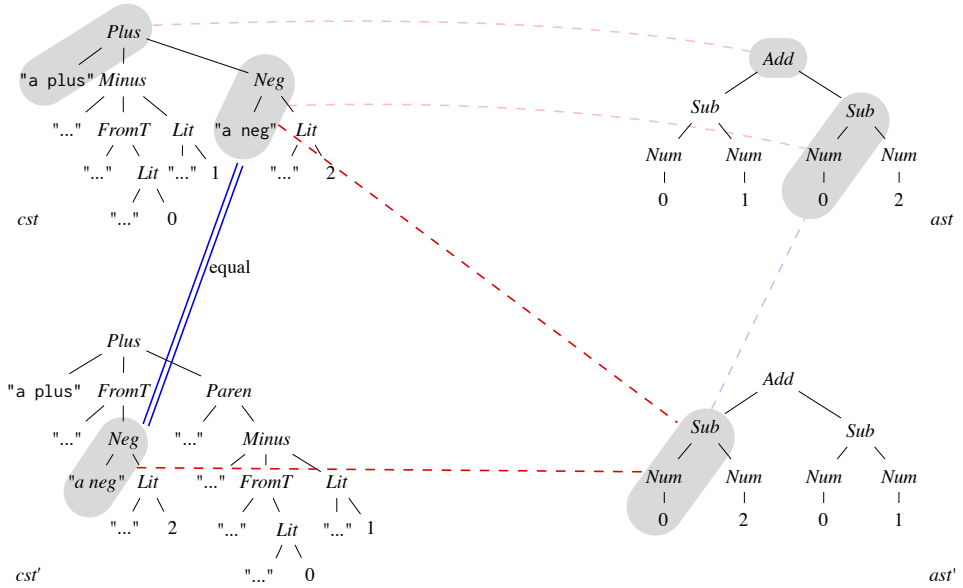


Fig. 4. The triangular guarantee. (Grey areas are some of the possible regions. Two light dashed lines between *cst* and *ast* are two of the consistency links produced by *get cst*. When updating *cst* with *ast′*, the region *Neg* `"a neg"` _ connected by the red dashed diagonal link is guaranteed to be preserved in the result *cst′*, and *get cst′* will link the preserved region to the same region *Sub* (*Num* 0) _ of *ast′*.)

paper we will focus on how to restore consistency assuming the existence of diagonal links. (We will discuss this issue again in Section 6.1.)

When it is time to put the modified view back into the source, the diagonal links between the source and the (modified) view can provide valuable information regarding what should be brought from the old source to the new source. The *put* function of a retentive lens thus takes a collection of (diagonal) links as additional input and retains regions in a way that respects these links. More precisely, *put* should provide what we might call the *triangular guarantee*: Using Figure 4 to illustrate, if the link relating the *Neg* `"a neg"` _ region in *cst* and the *Sub* (*Num* 0) _ region in *ast′* is provided as input to *put*, then after updating *cst* to a new source *cst′*, the consistency links between *cst′* and *ast′* should include a link that relates the *Sub* (*Num* 0) _ region to some region in the new source *cst′*, and that region should have the same pattern as the one specified by the input link, namely *Neg* `"a neg"` _, preserving the negation (as opposed to changing it to a *Minus*, for example) and the associated annotation.

2.3 Formalisation of Links

Now we formalise our idea of links. As described in Section 2.2, a link relates two regions, and a region consists of a location — which can be a path describing how to reach the top of the region from the root — and a pattern describing the region's content. One idea is to formalise a region as an element of the set *Region* = *Pattern* × *Path*, where *Pattern* is the set of region patterns and *Path* is the set of paths, and formalise a link as an element of *Region* × *Region*. The exact representation of paths is not important, but to give examples later, let us assume that a path is a list of natural numbers that indicate which subtree to go into: for example, starting from the root of *cst* in Figure 4, 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". The diagonal link in Figure 4 is then represented as $((\mathit{Neg}\ \texttt{"a neg"}\ \_, [2]), (\mathit{Sub}\ (\mathit{Num}\ 0)\ \_, [0]))$. This simple formalisation does not work too well, however. Recall that if *put* receives an input link between the original source and the view, it will need to guarantee that the content of the source region of the link exists in the updated source, while the location of the region within the updated source can be different from its original location. Therefore, with this definition of links, we cannot say that *put* retains the link between the updated source and the view, because the link describes the source region's location, which may not be retained.

The above discussion prompts us to use a more elaborate formalisation, in which links refer to locations only indirectly. Assuming a set *Name* of identifiers, we define:

$$\mathit{Property} = \{\ \mathsf{HasRegion}\ \mathit{pat}\ x \mid x \in \mathit{Name} \wedge \mathit{pat} \in \mathit{Pattern}\ \} \cup \cdots$$
$$\mathit{Locations} = \mathit{Name} \nrightarrow \mathit{Path}$$

where $A \nrightarrow B$ is the set of partial functions from set A to set B. To describe some regions in a tree, we first choose some locations in the tree and assign names to them by providing an element of *Locations*, i.e., a partial mapping from identifiers to paths that go from the root to the locations, and then provide a subset of *Property* describing what regions can be found at those locations — an element $\mathsf{HasRegion}\ \mathit{pat}\ x$ of *Property* is interpreted as saying that the location named x satisfies the property 'there is a region of pattern *pat* at that location'. This interpretation makes our formalisation extensible because we can introduce more varieties of properties, for example, 'a location is the root of a tree'; we will do so in Section 2.4.

Example 2.1. In *cst* in Figure 4, the two shaded regions are described by giving a (partial) location mapping $\{\ x \mapsto [], y \mapsto [2]\ \}$ and two properties $\mathsf{HasRegion}\ (\mathit{Plus}\ \texttt{"a plus"}\ \_\ \_)\ x$ and $\mathsf{HasRegion}\ (\mathit{Neg}\ \texttt{"a neg"}\ \_)\ y$.

Instead of defining the set of links and then the set of 'collections of links', we directly define the latter:

$$\mathit{Links} = \mathcal{P}\,(\mathit{Property} \times \mathit{Property}) \times \mathit{Locations} \times \mathit{Locations}$$

where $\mathcal{P}\,(A)$ is the power set of A. For $(\mathit{ps}, \mathit{ms}, \mathit{mv}) \in \mathit{Links}$, *ps* is a set consisting of pairs of properties. Each pair $(\mathsf{HasRegion}\ \mathit{pat}_x\ x, \mathsf{HasRegion}\ \mathit{pat}_y\ y)$ represents a link between a region of pattern pat_x at location x in the source tree and a region of pattern pat_y at location y in the view tree. To know where the named locations are in the source and view trees, we look them up in the mappings *ms* and *mv* respectively.

Example 2.2. The diagonal link between *cst* and *ast'* in Figure 4 is represented as the link collection $(\{\ (\mathsf{HasRegion}\ (\mathit{Neg}\ \texttt{"a neg"}\ \_)\ y, \mathsf{HasRegion}\ (\mathit{Sub}\ (\mathit{Num}\ 0)\ \_)\ z)\ \}, \{\ y \mapsto [2]\ \}, \{\ z \mapsto [0]\ \})$.

A collection of links, i.e., an element of *Links*, may not make sense for given source and view trees — for example, 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* with respect to a source tree and a view tree: all properties on the source side of the links are satisfied by the source tree and all properties on the view side are satisfied by the view tree.

Definition 2.3 (satisfaction). For a tree t, $m \in \mathit{Locations}$, and a set of properties $\Phi \subseteq \mathit{Property}$, we say that t; $m \models \Phi$ (read 't and m satisfy Φ') exactly when

$$\forall(\mathsf{HasRegion}\ \mathit{pat}\ x) \in \Phi.\quad x \in \textsc{dom}(m)\ \wedge\ \mathit{sel}(t, m(x)) \text{ matches } \mathit{pat},$$

where $\mathit{sel}(t, p)$ is a partial function that returns the subtree of t at the end of the path p (that starts from the root), or fails if path p does not exist in t.

Definition 2.4 (valid links). For a collection of links $ls = (ps, ms, mv) \in Links$, a source $s \in S$, and a view $v \in V$, we say that ls is *valid* for s and v, denoted by $s \leftrightarrow_{ls} v$, exactly when

$$s; ms \models \textsc{SrcProp}(ps) \quad \text{and} \quad v; mv \models \textsc{ViewProp}(ps),$$

where $\textsc{SrcProp}(ps) = \{\ \phi_s \mid \exists \phi_v.\ (\phi_s, \phi_v) \in ps\ \}$ and $\textsc{ViewProp}(ps) = \{\ \phi_v \mid \exists \phi_s.\ (\phi_s, \phi_v) \in ps\ \}$.

Example 2.5. The link collection given in Example 2.2 is valid for *cst* and *ast′* because $cst; \{\ y \mapsto [2]\ \} \models \{\ \mathsf{HasRegion}\ (\mathit{Neg}\ \texttt{"a neg"}\ \_)\ y\ \}$ and $ast'; \{\ z \mapsto [0]\ \} \models \{\ \mathsf{HasRegion}\ (\mathit{Sub}\ (\mathit{Num}\ 0)\ \_)\ z\ \}$.

2.4 Uniquely Identifying Property Sets

Before we give the definition of retentive lenses, there is one more ingredient that we need. One important aim of retentive lenses is to make Hippocraticness an extreme case of the triangular guarantee (Figure 4). Recall that the triangular guarantee roughly says that a set of regions in the old source will also exist in the new source. Having introduced the set *Property*, an alternative and more abstract way of understanding this is that we are preserving the HasRegion properties: if an old source has a region matching a pattern *pat*, i.e., satisfies HasRegion *pat x*, then the new source should also have a region matching *pat*, i.e., satisfy the same property HasRegion *pat x* (where *x* may refer to a new location). The more properties we require the new source to satisfy, the more restrictions we impose on the possible forms of the new source. In the extreme case, if a set of properties satisfied by the old source can only be satisfied by at most one tree, then preserving all the properties implies that the new source has to be the same as the old source. We will call such set of properties *uniquely identifying*, and require that the set of properties mentioned by the source side of the links produced by *get* be uniquely identifying.

Definition 2.6 (uniquely identifying). $\Phi \subseteq \mathit{Property}$ is *uniquely identifying* exactly when

$$t; m \models \Phi \ \wedge\ t'; m' \models \Phi \quad \Longrightarrow \quad t = t'$$

HasRegion properties alone cannot be uniquely identifying, though: requiring that a source must contain a set of region patterns does not uniquely determine the source, however complete the set of region patterns is. This can be intuitively understood by analogy with jigsaw puzzles: a picture cannot be determined by only specifying what jigsaw pieces are used to form the picture (if the jigsaw pieces can be freely assembled); to fully solve the puzzle and determine the picture, we also need to specify how the jigsaw pieces are assembled — that is, the relative positions of all the jigsaw pieces. To make it possible for a subset of *Property* to be uniquely identifying, we introduce another two kinds of properties:

$$\begin{aligned} \mathit{Property} = &\{\ \mathsf{HasRegion}\ \mathit{pat}\ x \mid \mathit{pat} \in \mathit{Pattern} \wedge x \in \mathit{Name}\ \} \\ &\cup \{\ \mathsf{IsTop}\ x \mid x \in \mathit{Name}\ \} \\ &\cup \{\ \mathsf{RelPos}\ p\ x\ y \mid p \in \mathit{Path} \wedge x, y \in \mathit{Name}\ \} \end{aligned}$$

The intended meaning of $\mathsf{IsTop}\ x$ is that the location named x should be the root of a tree, and $\mathsf{RelPos}\ p\ x\ y$ means that we can reach the location y from the location x by following the path p. We formalise these intentions as two additional clauses of Definition 2.3.

Definition 2.7 (satisfaction extended). For a tree t, a location mapping $m \in \mathit{Locations}$, and a set of properties $\Phi \in \mathcal{P}(\mathit{Property})$, we say that $t; m \models \Phi$ exactly when

$$\begin{aligned} \forall(\mathsf{HasRegion}\ x\ \mathit{pat}) \in \Phi.\quad & x \in \textsc{dom}(m) \quad \wedge \quad \mathit{sel}(t, m(x)) \text{ matches } \mathit{pat} \\ \forall(\mathsf{IsTop}\ x) \in \Phi.\quad & x \in \textsc{dom}(m) \quad \wedge \quad m(x) = [] \\ \forall(\mathsf{RelPos}\ p\ x\ y) \in \Phi.\quad & x, y \in \textsc{dom}(m) \quad \wedge \quad m(x) +\!\!+ p = m(y) \end{aligned}$$

where '++' is path/list concatenation.

Example 2.8. The following property set is uniquely identifying:

$$\{\ \mathsf{HasRegion}\ (\mathit{Add}\ \_\ \_)\ v, \mathsf{HasRegion}\ (\mathit{Sub}\ (\mathit{Num}\ 0)\ \_)\ w, \mathsf{HasRegion}\ (\mathit{Sub}\ (\mathit{Num}\ 0)\ \_)\ x,$$
$$\mathsf{HasRegion}\ (\mathit{Num}\ 1)\ y, \mathsf{HasRegion}\ (\mathit{Num}\ 2)\ z,$$
$$\mathsf{IsTop}\ v, \mathsf{RelPos}\ [0]\ v\ w, \mathsf{RelPos}\ [1]\ v\ x, \mathsf{RelPos}\ [1]\ w\ y, \mathsf{RelPos}\ [1]\ x\ z\ \}$$

and *ast* in Figure 4 is the only tree satisfying the above property set (with location mapping $\{\ v \mapsto [], w \mapsto [0], x \mapsto [1], y \mapsto [0,1], z \mapsto [1,1]\ \}$).

2.5 Definition of Retentive Lenses

Now we have all the ingredients for the formal definition of retentive lenses and can prove that Hippocraticness is implied by the definition.

Definition 2.9 (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 partial functions *get* and *put* of type

$$\mathit{get} : S \nrightarrow V \times \mathit{Links}$$
$$\mathit{put} : S \times V \times \mathit{Links} \nrightarrow S$$

satisfying

- *Link Completeness*: if $\mathit{get}(s) = (v, ls)$ and $ls = (ps, ms, mv)$, then $(s, v, ls) \in \textsc{dom}(\mathit{put})$ and
$$s \leftrightarrow_{ls} v \quad \wedge \quad \textsc{SrcProp}(ps) \text{ is uniquely identifying;} \tag{1}$$
- *Correctness*: if $\mathit{put}(s, v, (ps, ms, mv)) = s'$, then $s' \in \textsc{dom}(\mathit{get})$ and
$$\mathit{get}(s') = (v, (ps', ms', mv')) \quad \text{for some } ps', ms', \text{ and } mv'; \tag{2}$$
- *Retentiveness*: (continuing above) there is a permutation on names, i.e., a bijective $\sigma : \mathit{Name} \to \mathit{Name}$, such that
$$ps \subseteq ps'_\sigma \quad \wedge \quad \forall x \in NV.\ mv(x) = mv'(\sigma^{-1}(x)) \tag{3}$$
where ps'_σ is ps' in which every name n is replaced with $\sigma(n)$, and NV is the set of all names appearing in $\textsc{ViewProp}(ps)$.

Modulo the link arguments, Correctness stays the same as the Correctness law of the original lenses. Link Completeness requires that the collection of links computed by *get* be valid (Definition 2.4), and that the links *completely* record all the information of s so that it is possible to retain the whole s if the links are not modified and directly given to *put* as input.

The most important law is Retentiveness, which formalises the triangular guarantee. The first conjunct says that every pair of properties of the input must be retained by *put*, modulo a one-to-one correspondence on names. The possibility of such renaming has to be considered because ps' is introduced by a subsequent *get* after *put* with ps, which may use different name bindings from ps. The second conjunct further states that when a property (which can be HasRegion, IsTop, or RelPos) is preserved in the updated source, it still corresponds to exactly the same property on the view side, both the property itself and its evidence. In the special case of a link whose two sides are both HasRegion properties, the first conjunct says the data contained in the source region must exist somewhere in the updated source, while the second conjunct requires that the piece of data in the updated source must still correspond to the same view region (considering both the data and location). Consequently, Retentiveness can be specialised to the following triangular guarantee for regions, which was our initial motivation for developing retentive lenses.

Proposition 2.10 (Triangular guarantee for regions). *Given* $ls = (ps, ms, mv)$, $(s, v, ls) \in \mathrm{dom}(put)$, *and*

$$\phi_s = \text{HasRegion } n \; spat \quad \wedge \quad \phi_v = \text{HasRegion } m \; vpat$$

if $(\phi_s, \phi_v) \in ps$, *letting* $(v, (ps', ms', mv')) = get(put(s, v, ls))$, *then we also have:*

$$(\phi_s, \phi_v) \in ps'_\sigma \quad \wedge \quad mv(m) = mv'(\sigma^{-1}(m))$$

for some renaming σ. *Intuitively speaking, the region spat with name n still exists in the updated source and corresponds to the same view region as in the input.*

Example 2.11. In Figure 4, if we give *cst*, *ast'*, the diagonal link (HasRegion (*Neg* "a neg" _) y, HasRegion (*Sub* (*Num* 0) _) z), $ms = \{ y \mapsto [2] \}$, and $mv = \{ z \mapsto [0] \}$ to *put* and successfully obtain an updated source s', then $get(s')$ will succeed. Let $get(s') = (v, (ps', ms', mv'))$. Then we know that the diagonal link can be found in ${ps'}_\sigma$ for some renaming σ; that is, we can find a consistency link c = (HasRegion (*Neg* "a neg" _) y', HasRegion (*Sub* (*Num* 0) _) z') $\in ps'$ for some names $y' = \sigma^{-1}(y)$ and $z' = \sigma^{-1}(z)$. Also we know $mv'(z') = mv'(\sigma^{-1}(z)) = mv(z) = [0]$, so the view region referred to by the link c is indeed the same as the one referred to by the input link, and having $c \in ps'$ means that the region in s' corresponding to the view region has to be of pattern *Neg* "a neg" _ (since the link collection (ps', ms', mv') is valid for s' and v).

Now we show that Hippocraticness is a consequence of Retentiveness and Link Completeness with two simple lemmas.

Lemma 2.12 (monotonicity). *If* $\Phi \subseteq \Phi'$ *and* $t; m \models \Phi'$, *then* $t; m \models \Phi$.

Proof. Direct consequence of the definition of $t; m \models \Phi'$ (Definition 2.7). □

Lemma 2.13 (renaming). *For any bijective* $\sigma : Name \to Name$, *if* $t; m \models \Phi$, *then* $t; m \circ \sigma^{-1} \models \Phi_\sigma$ *also holds, where* Φ_σ *is* Φ *with all names in it transformed with* σ.

Proof. Direct consequence of Definition 2.7. □

Theorem 2.14 (Hippocraticness). *For a retentive lens,*

$$get(s) = (v, ls) \quad \Rightarrow \quad put(s, v, ls) = s$$

Proof. Let $s' = put(s, v, ls)$ and $get(s') = (v', ls')$. If $ls = (ps, ms, mv)$ and $ls' = (ps', ms', mv')$, then we have $ps \subseteq {ps'}_\sigma$ with a renaming function σ by Retentiveness (3). Also by the first conjunct of Link Completeness (1), we have $s \leftrightarrow_{ls} v$ and $s' \leftrightarrow_{ls'} v'$, which imply $s; ms \models \textsc{SrcProp}(ps)$ and $s'; ms' \models \textsc{SrcProp}(ps')$. By Lemma 2.13, we have $s'; ms' \circ \sigma^{-1} \models \textsc{SrcProp}({ps'}_\sigma)$. Since $ps \subseteq {ps'}_\sigma$, with Lemma 2.12, we also have $s'; ms' \circ \sigma^{-1} \models \textsc{SrcProp}(ps)$. Because Link Completeness requires $\textsc{SrcProp}(ps)$ to be uniquely identifying, we have $s' = s$. □

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

Example 2.15. 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:

$$get(s) = (g(s), trivial(s, g(s)))$$
$$put(s, v, ls) = p(s, v)$$

where

$$trivial(s, v) = (\{\ (\mathsf{HasRegion}\ ToPat(s)\ n, \mathsf{HasRegion}\ ToPat(v)\ m),\ (\mathsf{IsTop}\ n, \mathsf{IsTop}\ m)\ \}\ ,\{\ n \mapsto [\,]\ \}\ ,\{\ m \mapsto [\,]\ \})$$

ToPat turns a tree into a pattern completely describing the tree, and $\textsc{dom}(put)$ is restricted to $\{\ (s, v, ls) \mid s \leftrightarrow_{ls} v$ and $ls = trivial(s, v)$ or $(\emptyset, \emptyset, \emptyset)\ \}$. The idea is that *get* generates a link collection relating the whole source tree and view tree as two regions, and *put* accepts either an empty collection of links or the trivial link produced by *get*. The trivial link generated by *get* is apparently valid and uniquely identifying so that Link Completeness is satisfied. Correctness holds because the underlying g and p is a well-behaved lens. 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. Thus *get* and *put* indeed form a retentive lens.

3 A DSL FOR RETENTIVE BIDIRECTIONAL TREE TRANSFORMATION

With theoretical foundations of retentive lenses in hand, we shall propose a domain specific language (DSL) for easily describing them. Our DSL is designed to be simple but suitable for handling the synchronisation between syntax trees. We give a detailed description of the DSL by presenting its programming examples (Section 3.1), syntax (Section 3.2), semantics (Section 3.3), and finally the proof of its generated lenses holding Retentiveness (Theorem 3.1).

3.1 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 Figure 3. From each defined consistency relation, the DSL generates us a pair of *get* and *put* functions forming a retentive lens. Furthermore, we show how to flexibly update the *cst* Figure 4 in different ways using the generated *put* function with different input links.

In our DSL, the user defines data types in Haskell syntax and describes consistency relations between them as if writing *get* functions. For example, the data type definitions for *Expr* and *Term* written in our DSL remain the same and the consistency relations between them (i.e., the *getE* and *getT* functions in Figure 3) are expressed as the ones in Figure 5. 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 $Plus\ \_\ x\ y \sim Add\ x\ y$ 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 a 'don't-care' wildcard in $Plus\ \_\ x\ y$. So the meaning of $Plus\ \_\ x\ y \sim Add\ x\ y$ can be better understood by the 'derivation rule' beneath:

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

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

Now we briefly explain the semantics of $Plus\ \_\ x\ y \sim Add\ x\ y$. In the *get* direction, its behaviour is quite similar to the first case of *getE* in Figure 3, 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*

```
Expr ↔ Arith
  Plus  _ x y ~ Add x y
  Minus _ x y ~ Sub x y
  FromT _ t   ~ t

Term ↔ Arith
  Lit   _ i ~ Num i
  Neg   _ r ~ Sub (Num 0) r
  Paren _ e ~ e
```

Fig. 5. The program in our DSL for synchronising data types defined in Figure 3.

and *Add* recording the correspondence. In the *put* direction, it 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 *put* is guided by those links; for example, *put* may additionally preserve the annotation in the old source. We leave the detailed descriptions to the subsection about semantics.

With the lenses generated from Figure 5, we can synchronise the old *cst* and modified *ast′* in different ways by running *put* with different input links. Figure 6 illustrates this:

- If the user assumes that the two non-zero numbers *Num* 1 and *Num* 2 in the *ast* are exchanged, then four links (between *cst* and *ast′*) should be established: a link connects region *Lit* `"1 in sub"` _ with region *Num* _ (of the tree *Num* 1); a link connects *Lit* `"2 in neg"` _ with *Num* _ (of the tree *Num* 2); two links connects 1 and 1, and 2 and 2. The result is cst_1, where the literals 1 and 2 are swapped, along with their annotations. Note that all other annotations disappeared because the user did not include links for preserving them.
- If the user thinks that the two subtrees of *Add* are swapped and passes *put* links that connect not only the roots of the subtrees of *Plus* and *Add* but also all their inner subtrees, the desired result should be cst_2, which represents '$-2 + (0 - 1)$' and retains all annotations, in effect swapping the two subtrees under *Plus* and adding two constructors *FromT* and *Paren* to satisfy the type constraints. Figure 6 shows the input links for *Neg* and its subtrees.
- If the user believes that *ast′* is created from scratch and is not related to *ast*, then cst_3 may be the best choice, which represents the same expression as cst_1 except that all annotations are removed.

Although the DSL is tailored for describing consistency relations between syntax trees, it is also possible to handle general tree transformations. Now we present three small but typical programming examples other than syntax tree synchronisation. For the first example, 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
```

As the second example, we demonstrate the implicit use of some other consistency relation 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
```

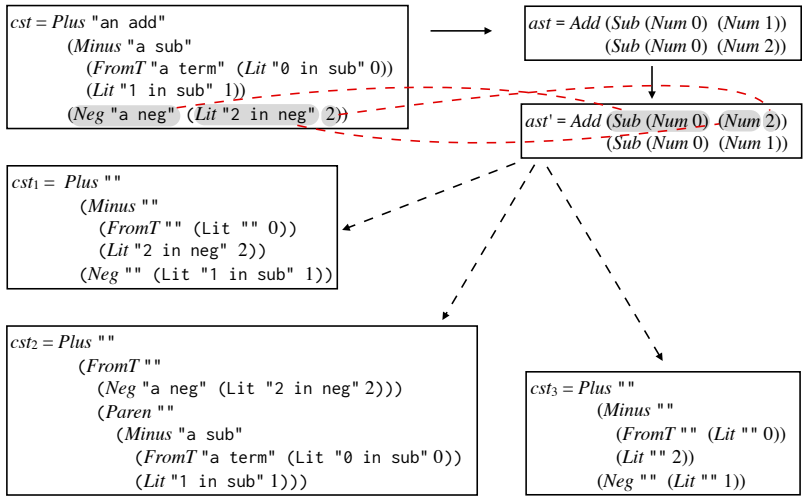


Fig. 6. *cst* is updated in many ways. (Red dashed lines are some of the input links for cst_2.)

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

$$
\begin{aligned}
&BinT\ Nat \leftrightarrow BinT\ Bool \\
&\quad Tip \sim Tip \\
&\quad Node\ x\ ls\ rs \sim Node\ x\ ls\ rs
\end{aligned}
$$

As the last example, let us consider rose trees, a data structure mutually defined with lists:

$$
\begin{aligned}
&\textbf{data}\ RTree\ a = RNode\ a\ (List\ (RTree\ a)) \\
&\textbf{data}\ List\ a = Nil \mid Cons\ a\ (List\ a)
\end{aligned}
$$

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

$$
\begin{aligned}
&RTree\ Int \leftrightarrow List\ Int \\
&\quad RNode\ i\ Nil \sim Cons\ i\ Nil \\
&\quad RNode\ i\ (Cons\ x\ \_) \sim Cons\ i\ x
\end{aligned}
$$

3.2 Syntax

Having seen some programming examples, here we summarise the syntax of our language in Figure 7. A program consists of two main parts: definitions of data types and consistency relations between these data types. We adopt the Haskell syntax for data type definitions, so that a new data type is defined through a set of data constructors $C_1 \ldots C_n$ followed by types, and a type synonym is defined by giving a new name to existing types. (Definition for `type` is omitted.) For the convenience of implementation, the part defining type synonyms should occur before the part defining new data types. At the end of type definitions, the user can describe (override) the conversions between interconvertible data types; otherwise, similar ones will be automatically generated. We will explain more about this in the syntactic restrictions below. Now we move to the syntax 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

Program

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

Type Definition

$tDef \quad ::= \quad tSyn_1 \ldots tSyn_m\ tNew_1 \ldots tNew_n\ conv_1 \ldots conv_n$

$tSyn \quad ::= \quad$ 'type' $type$ '=' $type_1 \ldots type_m$

$tNew \quad ::= \quad$ 'data' $type$ '=' $C_1\ type_{11} \ldots type_{1k_1}$
'|' ...
'|' $C_n\ type_{n1} \ldots type_{nk_n}$

$conv$ ::= 'instance' 'InterConv' $type_1\ type_2$ 'where'
'inj' var '=' $HaskellExpression$

Consistency Relation Definition

$cDef \quad ::= \quad type_s \longleftrightarrow type_v$ { relation type }
$r_1 \ldots r_n$ ';;' { inductive rules }

Inductive Rule

$r \quad ::= \quad pat_s$ '~' pat_v

Pattern

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

Fig. 7. Syntax of the DSL.

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. (Although they are not sufficient and we need further semantic checks.) Some of the restrictions are quite natural while some of them are a little subtle; the reader may skip the subtle ones at the first reading.

- On *patterns*, we assume pattern coverage and source pattern disjointness. Pattern coverage guarantees totality: for the consistency relation $Ts \leftrightarrow Tv$, all the patterns on its source side should cover all the possible cases of type Ts, and all the patterns on the view side should cover all the cases of type Tv. Source pattern disjointness requires that all of the source patterns in a consistency relation do not overlap each other so that at most one pattern is matched when running *get*. Additionally, a bare variable pattern is not allowed on the source side and wildcards are not allowed on the view side; for example, neither $x \sim C\ x$ nor $C\ x \sim D\ \_\ x$ is allowed. These requirements imply that *get* is indeed a total function.
- On *algebraic data types*, we require that types T_1 and T_2 be interconvertible; if consistency relations $T_1 \leftrightarrow V$ and $T_2 \leftrightarrow V$ are both defined for some view type V and there is some type Ts whose definition makes use of T_1 and T_2 (directly or indirectly). This allows *put* to retain regions of type T_1 when a value of type T_2 is required and vice versa. Let us consider a particular case where T_1 and T_2 are mutually recursively defined and mapped to the same view type. For instance, since the two consistency relations $Expr \leftrightarrow Arith$ and $Term \leftrightarrow Arith$

in Figure 5 share the same view type *Arith*, there should be a way to convert (inject) *Expr* into *Term* and vice versa. Look at *cst′* in Figure 4, 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.

- On *converting functions*, we require that they not discard information nor add unnecessary information. Hence they are restricted to the form $inj_{T_1 \to T_2}\ x = C\ \ldots\ x\ \ldots$ and the consistency relation $T_2 \leftrightarrow V$ should have a rule: $C\ \_\ \ldots\ x\ \ldots\ \_ \sim x$ in which the source pattern only has wildcards except C and x.

3.3 Semantics

In this subsection, we give a denotational semantics of our DSL by specifying a program's corresponding *get* and *put* as (mathematical) functions. Our language is also implemented as a compiler transforming programs in our DSL to *get* and *put* functions in Haskell. The structure of our implementation is similar to the semantics given below.

3.3.1 Types and Patterns. Before defining the semantics of *get* and *put*, we need the semantics of type declarations and patterns in our DSL. The semantics of type declarations is the same as those 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, there is a set *Tree*, which is the set of all the values of all the algebraic data types defined in our DSL. *Pattern* is the set of all possible patterns; for a pattern $p \in Pattern$, $Vars(p)$ denotes the set of variables in p, $TypeOf(p, v)$ is the set corresponding to the type of $v \in Vars(p)$, and $Path(p, v)$ is the path of v in p.

We use the following functions to manipulate patterns:

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

Given a pattern p and a value (i.e., tree) s, $isMatch(p, s)$ tests if s matches p. If the match succeeds, we use $decompose(p, s)$ to return a function f mapping every variable in p to its corresponding matched subtree of s. Conversely, $reconstruct(p, f)$ produces a tree s matching the pattern p by replacing every subtree denoted by $v \in Vars(p)$ 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. [2]

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

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

[2] We assume non-linear patterns are supported: multiple occurrences of the same variable in a pattern must capture the same value.

where $LinkSet = \mathcal{P}(Property \times Property) \times Locations \times Locations$ and $Locations = Name \rightharpoonup Path$ as in Section 2.3. The *get* function defined by R is:

$$
\begin{aligned}
&get_{SV} = genIsTop \circ get'_{SV} \\
&get'_{SV}(s) = (reconstruct(vpat_k, fst \circ vls),\ l_{root} \sqcup links) \qquad (4)\\
&\quad \text{where } spat_k \sim vpat_k \in R \text{ and } spat_k \text{ matches } s \\
&\qquad vls = (get' \circ decompose(spat_k,\ s)) \in Vars(spat_k) \to V \times LinkSet \\
&\qquad x,\ y \in Name \text{ and fresh} \\
&\qquad spat' = EraseVars(fillWildcards(spat_k, s)) \\
&\qquad l_{root} = \Big(\{\ (\text{HasRegion } spat'\ x,\ \text{HasRegion } EraseVars(vpat_k)\ y)\ \} \cup relPos, \\
&\qquad\qquad \{\ x \mapsto \epsilon\ \},\ \{\ y \mapsto \epsilon\ \} \Big) \\
&\qquad relPos = \big\{ (\text{RelPos } Path(spat_k, t)\ x\ st,\ \text{RelPos } Path(vpat_k, t)\ y\ vt) \qquad (5)\\
&\qquad\qquad \mid t \in Vars(vpat_k),\ (\_, (\_, m_{src}, m_{view})) = vls(t), \\
&\qquad\qquad\quad m_{src}(st) = \epsilon,\ m_{view}(vt) = \epsilon \big\} \\
&\qquad links = \bigsqcup \big\{ (ps, (Path(spat_k, t)\ +\!\!+) \circ m_{src}, (Path(vpat_k, t)\ +\!\!+) \circ m_{view}) \\
&\qquad\qquad \mid t \in Vars(vpat_k),\ (\_, (ps, m_{src}, m_{view})) = vls(t) \big\} \\
&genIsTop(v, (ps, ms, mv)) = (v,\ (ps',\ ms,\ mv)) \\
&\quad \text{where } rs, rv \in Name \text{ s.t. } ms(rs) = \epsilon \text{ and } mv(rv) = \epsilon \\
&\qquad ps' = ps \cup \{\ (\text{IsTop } rs, \text{IsTop } rv)\ \}
\end{aligned}
$$

where the operator $\sqcup \in LinkSet \times LinkSet \to LinkSet$ is the union on $LinkSet$ by taking union of corresponding components, and big $\sqcup$ is this operation extended to a set of $LinkSet$. For a tree s matching $spat_k$, $fillWildcards(spat_k, s)$ is the resulting pattern of replacing all the wildcards in $spat_k$ with the corresponding subtrees of s and $EraseVars(spat_k)$ is the resulting pattern of replacing all variables in pattern $spat_k$ with wildcards.

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 recursive call is written as $get' \circ decompose(spat_k,\ s)$ in the definition above, while to be precise, it should be $get'_{TypeOf(spat_k, t),\ TypeOf(vpat_k, t)}$ for every $t \in Vars(spat_k)$.

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. And it also creates links of RelPos recording the relative position of regions so that the produced collection of links satisfies Link Completeness. For the same purpose, a link of IsTop is added at the last step to identify the root of the source and the view.

3.3.3 Put Semantics. For a consistency relation $S \leftrightarrow V$ defined in our DSL as $R = \{\ spat_i \sim vpat_i \mid 1 \leqslant i \leqslant n\ \}$, its corresponding put_{sv} function has the following type:

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

Given arguments $(s,\ v,\ (ps,\ ms,\ mv))$, *put* is defined by two cases depending on whether the root of the view is within a region of the input links or not; more precisely, whether there is $y \in Name$ such that $mv(y) = \epsilon$ and $(\text{HasRegion } spat\ x,\ \text{HasRegion } vpat\ y) \in ps$.

If such y does not exist, i.e., the root of the view is not within any region of the input links, then *put* simply 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 use $spat_k$ as the top portion of the new source.

$$
\begin{aligned}
&put_{SV}(s,\ v,\ (ps,\ ms,\ mv)) = reconstruct(spat'_k,\ ss) && (6)\\
&\quad \text{where } spat_k \sim vpat_k \in R \text{ and } vpat_k \text{ matches } v\\
&\qquad vs = decompose(vpat_k,\ v)\\
&\qquad ss = \lambda\,(t \in Vars(spat_k)) \rightarrow\\
&\qquad\quad put(s, vs(t), trimVPath(Path(vpat_k, t),\ (ps,\ ms,\ mv))) && (7)\\
&\qquad spat'_k = fillWildcardsWithDefaults(spat_k)
\end{aligned}
$$

$$
\begin{aligned}
&trimVPath(prefix, (ps,\ ms,\ mv)) = (ps',\ ms,\ mv')\\
&\quad \text{where } mv' = \{\ x \mapsto p \mid mv(x) = prefix +\!\!+ p\ \}\\
&\qquad ps' = \{\ p \in ps \mid \text{Names appeared in } p \text{ are defined in } (ms, mv')\ \}
\end{aligned}
$$

The omitted subscription of *put* in (7) is $TypeOf(spat_k, t)\ TypeOf(vpat_k, t)$. Additionally, if there are more than one rules of which *vpat* matches v, rules whose *vpat* is *not* a bare variable pattern are preferred. For example, if both $spat_1 \sim Succ\ x$ and $spat_2 \sim x$ can be selected, the first one is selected. Such preference helps to avoid infinite recursive calls because if $vpat_k = x$, the size of the input of the recursive call in (7) does not decrease since $vs(t) = v$ and $Path(t, vpat_k) = \epsilon$.

For the other case where the root of the view is within some region, *put* 'grabs' the related source region as the top portion of the new source.

$$
\begin{aligned}
&put_{SV}(s,\ v,\ (ps,\ ms,\ mv)) = inj_{TypeOf(spat_k)\rightarrow S}(reconstruct(spat_k,\ ss)) && (8)\\
&\quad \text{where } l = (\text{HasRegion } spat\ x,\ \text{HasRegion } vpat\ y) \in ps\\
&\qquad \text{such that } mv(y) = \epsilon,\ ms(x) \text{ is the shortest}\\
&\qquad spat_k \sim vpat_k \in R \text{ and } spat_k \text{ matches } {}^{3}spat\\
&\qquad vs = decompose(vpat_k,\ v)\\
&\qquad ss = \lambda\,(t \in Vars(spat_k)) \rightarrow\\
&\qquad\quad put(s, vs(t), trimVPath(Path(vpat_k, t),\ (ps \setminus \{\ l\ \},\ ms,\ mv))) && (9)
\end{aligned}
$$

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. By this, the preserved regions in the new source will have the same relative positions as those in the old source. Thus *put* implicitly respects all properties asserting relative positions of regions in the old source without explicitly handling IsTop links and RelPos links in ps. The function $inj_{TypeOf(spat_k),\ S}$ is the transformation from the type of the linked source region to type S and is provided by the interconvertible requirement of our DSL (Syntax Restrictions). It is necessary because the linked source region does not necessarily have type S.

The last piece for our definition of *put* is its domain, on which *put* satisfies Retentiveness (3) and terminates. Some of these conditions are quite subtle and technical. Readers can safely skip them at the first reading. For a collection of links $ls = (ps,\ ms,\ mv)$, $put(s,\ v,\ ls)$ is defined when:

[3]Here we extend pattern matching to partial trees such as *spat*. Wildcards in *spat* can be captured by wildcards or variables but nothing else.

- The collection of links ls is valid for s and v (Definition 2.4) and only contains links that can be generated by our *get*. That is, there exists some $s' \in S$ such that $get(s') = (\_, (ps', \_, \_))$ and $ps \subseteq {ps'}_{\sigma}$ for some renaming σ.
- Regions are well-aligned: For every subtree t of v whose root is not within the region of any link in ps, if t matches the view-side pattern of a rule in R, the matched portion — i.e., the parts not captured by any variable in the pattern — of t does not overlap the view region of any link in ps. This restriction guarantees that when some rule $spat_k \sim vpat_k$ is used in the first case of *put*, the top portion of v matching $vpat_k$ does not overlap any view region of some link in ps; so that every region link will be handled by the second case of *put* in the recursive procedure, contributing to Retentiveness. Such well-alignment conditions also appeared in [Wang et al. 2011].
- There is no infinite recursion. In the first case of *put*, the recursive call (7) does not decrease the size of the argument of *put* when $vpat_k = t$ since $vs(t) = v$ and $Path(t, vpat_k) = \epsilon$. Thus we need to restrict the domain of *put* further to rule out the possibility of infinite recursion of such case. Given type V and an infinite sequence of types S_i such that for all $i \in \mathbb{N}$, both $S_i \leftrightarrow V$ and $S_{i+1} \leftrightarrow V$ are defined; if there is an inductive rule in $S_i \leftrightarrow V$:

  $$spat_0 \sim x \quad \text{for some variable } x$$

  where $TypeOf(spat_0, x) = S_{i+1}$, we require that for any subtree t of v, there are always an $n \in \mathbb{N}$ and a rule $spat \sim vpat$ in $S_n \leftrightarrow V$ such that $vpat$ matches t and $vpat$ is not a bare variable pattern.

With the definition of *get* and *put* above, now we have:

Theorem 3.1 (Main Theorem). *Let $put' = put$ with its domain intersected with $S \times V \times LinkSet$, get and put′ form a retentive lens as in Definition 2.9.*

The proof of Retentiveness can be found in the appendix (A). The basic idea is to prove by induction on the size of the view argument and the link argument of *put*. Correctness and Link Completeness can be proved similarly.

4 APPLICATION

We demonstrate some practical use of retentiveness by implementing the transformation system between CSTs and ASTs of a small subset of Java 8 in response to the code refactoring example in the introduction. We show that Retentiveness makes it possible to perform code refactoring on clean ASTs; and when an AST is modified by a refactoring algorithm, it is possible to retain comments and syntactic sugar on demand. In the last part of this section, we briefly discuss other potential application scenarios such as resugaring [Pombrio and Krishnamurthi 2014, 2015].

4.1 Target Language for 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 classify the total 23 refactoring operations as the combinations of four basic operations: substitution, insertion, deletion, and movement. For instance, *Extract Method*, which creates a method containing the selected statements and replace those selected statements (in the old place) with a reference to the created method [Eclipse Foundation 2018], is the combination of insertion (as for the empty method definition), movement (as for the selected statements), and substitution (as for the reference to the created method). For about half of the refactoring operations, position-wise replacement and list alignment (will be discussed in

Section 5.1) is enough, as the code (precisely, subtrees of an AST) is only moved around within a list-like structure (such as an expression list or a statement list). For the remaining half of the operations, the code is moved far from its original position so that some 'global tracking information' such as links are required. Based on the survey, we believe that a transformation system implemented by retentive lenses is able to support the set of refactoring operations.

Nevertheless, 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. To make a trade-off, in this paper we focus on the theoretical foundation and language design, and have only implemented a transformation system for a small subset of Java 8 to demonstrate the possibility of having the whole system. The full-fledged code refactoring tool is left to our future work.

4.2 Implementation in Our DSL

Following the grammar of Java 8 [Gosling et al. 2014], 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 JDT parser [Oracle Corporation and OpenJDK Community 2014]. We additionally make the AST purer by dropping unnecessary information such as a node's position information in the source file. This subset of Java 8 already has around 80 CST constructs (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).

Now we write the consistency relations. Since the structure of the consistency relations for the transformation system is roughly similar to the ones in Figure 5, here we only highlight some different or interesting parts; reviewers can refer to the supplementary material to see the complete program. We see that in the concrete syntax everything is a class declaration while in the abstract syntax everything is a tree. As a *ClassDecl* should correspond to a *JCClassDecl*, which by definition is yet not a *JCTree*, we use the constructors *FromJCStatement* and *FromJCClassDecl* to make it a *JCTree*, emulating the inheritance in Java. This is described by the consistency relation[4]

```
ClassDecl ↔ JCTree
  NormalClassDeclaration0 _ "class" n "extends" sup body ~
  FromJCStatement (FromJCClassDecl (JCClassDecl N n (J (JCIdent sup)) body))
  NormalClassDeclaration1 _ mdf "class" n "extends" sup body ~
  FromJCStatement (FromJCClassDecl (JCClassDecl (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. Similarly, there are further two cases where a class does not extend some superclass and are omitted here. (To save space, the constructors *Just* and *Nothing* are shortened to *J* and *N* respectively.)

Next, we see how to represent *while loop* and *for-each loop* using the basic *for loop*, as the abstract syntax of a language should be as concise as possible[5]. The conversion for *while loop* is quite simple:

```
While "while" "(" exp ")" stmt ~ JCForLoop Nil exp Nil stmt
```

[4] We include keywords such as *class* in the CST patterns for improving readability, although they should be removed.
[5] Although the JDT parser does not do this.

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 only 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.

The conversion from *for-each loop* to *for loop* is a little complex, so first consider the general conversion strategy [GeeksforGeeks 2017] through the following example.

```
for (type var : exp) {
  // statements using var;
}
```

A *for-each loop*

```
for (int i=0; i < exp.length; i++) {
  type var = exp[i];
  // statements using var;
}
```

The corresponding *for loop*

The *exp* is assumed to be an array-like object that has its length, so that in the converted *for loop* we can use a variable *i*, the length of the object `exp.length`, and a post-increment `i++` to control the times of the loop. Furthermore, since `var` ranges over each object in the array, we need to make this explicitly by using a variable declaration with initialisation in the *for loop*. These intentions can also be expressed in our DSL straightforwardly (despite being a little verbose)[6]:

EnhancedFor `"for"` `"("` *ty var* `":"` *exp* `")"` *stmt* ~
JCForLoop
  (*Cons* (*FromVarDecl* (*VarDecl N* `"i"` (*PrimTypeTree UInt*) (*J* (*FromLit* (*UInt* 0))))) *Nil*)
  (*Binary LESSTHAN* (*Ident* `"i"`) (*FieldAccess expr* `"length"`) `"<"`)
  (*Cons* (*Unary POSTINC* (*Ident* `"i"`) `"++"`) *Nil*)
  (*FromBlock* (*Cons* (*FromVarDecl* (*VarDecl N var* (*PrimTypeTree ty*)
    (*J* (*ArrayAccess expr* (*Ident* `"i"`)))))) *stmt*))

We can clearly read from the four subtrees of *JCForLoop* that, the first subtree means `int i = 0`; the second is `i < expr.length`; the third denotes `i++`; the last is the concatenation of `ty var = expr[i]` and the converted statements from the original *for-each loop* body. Another interesting point is that the variable *expr* appears twice on the view side, once in the filed access (`expr.length`) and the other in the array access (`expr[i]`). Other consistency relations can be written alike.

4.3 System Running

Now we use the retentive lenses generated from our DSL to run the refactoring example in Figure 1. We first test two basic cases: *put cst ast hls* and *put cst ast* [ ], where *ast* and *hls* are the view and consistency links generated by *get cst*. We get the same *cst* after running *put cst ast hls* and it shows that the generated lenses satisfy Hippocraticness. We get a *cst′* after running *put cst ast* [ ], in which the comments disappear and the *for-each loop* becomes a *for loop*. This demonstrates that *put* can create a new source from scratch only dependent on the given view. As a special case of Correctness, we also check that *get* (*put cst ast* [ ]) == *get cst*.

Next we modify *ast* to *ast′*, the tree after code refactoring, and create some diagonal links *hls′* between *cst* and *ast′* by hand. The diagonal links are designed only to connect the *fuel* method in the *Car* class but not the *fuel* method in the *Bus* class, so that we can observe that comments and *for-each loop* are only preserved for the *Car* class but not the *Bus* class. This is where Retentiveness helps the user to retain information on demand. Finally, we also check that Correctness holds: *get* (*put cst ast′ hls′*) == *ast′*.

[6] We removed the beginning two characters *JC* from all the constructors in the AST except for *JCForLoop*

Potential Application. Retentiveness may apply to similar scenarios such as resugaring [Pombrio and Krishnamurthi 2014, 2015], which is to print evaluation sequences in a core language using the constructs of its surface syntax. Take the language Tiger [Appel 1998] (a general-purpose imperative language designed for educational purposes) for example: Its surface syntax offers logical conjunction ($\wedge$) and disjunction ($\vee$), which, in the core language, are desugared to the conditional construct *Cond* after parsing. Boolean values are also eliminated in the AST, where integer 0 represents *False* and any non-zero integer represents *True*. For instance, the program text $a \wedge b$ is parsed to an AST *Cond* a b 0 and $a \vee b$ is parsed to *Cond* a 1 b. If the user want to inspect the result of one-step evaluation of $0 \wedge 10 \vee c$, then the user will face a problem, as the evaluation is in fact performed on the AST *Cond* (*Cond* 0 10 0) 1 c and will yield new program text in terms of a conditional like *if* 10 *then* 1 *else* c. To solve the problem, Pombrio and Krishnamurthi chooses to enrich the AST to incorporate fields for holding tags that mark from which syntactic object an AST construct comes. By comparison, we can also solve this problem by writing the transformations between the surface syntax and core language using retentive lenses, and pass the lens proper links for retaining syntactic sugar, just like what we have done for Java 8. Both their 'tags approach' and our 'links approach' need to identify where an AST construct comes from; however, the links approach has an advantage that it leaves ASTs clean so that we do not need to patch up the compiler to deal with tags.

5 RELATED WORK

5.1 Alignment

Our work on retentive lenses with links is closely related to the research on alignment in bidirectional programming. The earliest lenses [Foster et al. 2007] 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 [Bohannon et al. 2008] and their successor matching lenses [Barbosa et al. 2010]. In matching lenses, a source is divided into a 'resource' consisting of 'chunks' of information that can be reordered, and a 'rigid complement' storing information outside the chunk structure; the reorderable chunk structure is preserved in the view. When a *put* is invoked, it will first do source–view element matching, which finds the correspondence between chunks of the old and new views using some predefined strategies; based on the correspondence, the 'resource' is 'pre-aligned' to match the chunks in the new view. Then element-wise updates are performed on the aligned chunks. The design of matching lenses is 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. 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.

To generalise list alignment, a more general notion of data structures called *containers* [Abbott et al. 2005] is used [Hofmann et al. 2012]. In the container framework, a data structure is decomposed into a *shape* and its *content*; the shape encodes a set of positions, and the content is a mapping from those positions to the elements in the data structure. The existing approaches to container alignment take advantage of this decomposition and treat shapes and contents separately. For example, if the shape of a view container changes, Hofmann et al.'s approach will update the source

shape by a fixed strategy that makes insertions or deletions at the rear positions of the (source) containers. By contrast, Pacheco et al.'s method permits more flexible shape changes, and they call it *shape alignment*. In our setting, both the consistency on data and the consistency on shapes are specified by the same set of consistency declarations. In the *put* direction, both the data and shape of a new source is determined by (computed from) the data and shape of a view, so there is no need to have separated data and shape alignments.

Container-based approaches have the same situation (as list alignment) that the retention of information is dependent on the basic lens applied after alignment. Besides, as a generalisation of list alignment, it is worth noting that separation of data alignment and shape alignment will hinder the handling of some algebraic data types. First, in practice it is usually difficult for the user to define container data types and represent their data using containers. We use the data types defined in Figure 3 to illustrate, where two mutually recursive data types *Expr* and *Term* are defined. If the user wants to define *Expr* and *Term* using containers, one way might be to parametrise the types of terminals (leaves in a tree, here *Integer* only):

```
data Expr i = Plus   (Expr i) (Term i)
            | Minus  (Expr i) (Term i)
            | FromT  (Term i)
data Term i = Neg    (Term i)
            | Lit    i
            | Paren  (Expr i)
```

```
data Arith i = Add (Arith i) (Arith i)
             | Sub  (Arith i) (Arith i)
             | Num i
```

Here the terminals are of the same type *Integer*. However, imagine the situation where there are more than ten types of leaves, it is a boring task to parameterise all of them as type variables.

Moreover, the container-based approaches face another serious problem: they always translate a change on data in the view to another change on data in the source, without affecting the shape of a container. This is wrong in some cases, especially when the decomposition into shape and data is inadequate. For example, let the source be *Neg* (*Lit* 100) and the view *Sub* (*Num* 0) (*Num* 100). If we modify the view by changing the integer 0 to 1 (so the view becomes *Sub* (*Num* 1) (*Num* 100)), the container-based approach would not produce a correct source *Minus*..., as this data change in the view must not result in a shape change in the source. In general, the essence of container-based approaches is the decomposition into shape and data such that they can be processed independently (at least to some extent), but when it comes to scenarios where such decomposition is unnatural (like the example above), container-based approaches can hardly help.

5.2 Provenance and Origin

The idea of links is inspired by research on provenance [Cheney et al. 2009] in database communities and origin tracking [van Deursen et al. 1993] in the rewriting communities.

Cheney et al. 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. Leaping from database communities to programming language communities, we find that 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 (sums of products), and a view is produced by more general functions rather than relational operations such as selection, projection, and join. For this need,

dependency provenance [Cheney et al. 2011] 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 horizontal are closer to dependency provenance.

The idea of inferring consistency links (horizontal links generated by *get*) can be found in work on origin tracking for term rewriting systems [van Deursen et al. 1993], in which the origin relations between rewritten terms can be calculated by analysing the rewrite rules statically. However, it was developed solely for building traces between intermediate terms rather than using trace information to update a tree further. Based on origin tracking, de Jonge and Visser 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'. This description shows that the algorithm decomposes large terms into smaller ones resembling our regions. So in terms of the formatting aspect, we think that retentivenss can be in effect the same as their theorem if we adopt the 'square diagram'. (See Section 6.1.) It was a pity that they tailored the theorem for their specific printing algorithm and did not generalise the theorem to other scenarios.

Similarly, Martins et al. developed a system for attribute grammars which define transformations between tree structures (in particular CSTs and ASTs). Their bidirectional system also uses links to trace the correspondence between source nodes and view nodes, which is later used by *put* to solve the syntactic sugar problem. The differences between their system and ours are twofold: One is that in their system, links are implicitly used in the put direction. The advantage is that for the user links become transparent and are automatically maintained when a view is updated; the disadvantage is that, as a result, newly created nodes on an AST cannot have 'links back' to the old CST, even if they might be the copies of some old nodes. The second difference is the granularity of links; in their system, a link seems to connect the whole subtrees between a CST and an AST instead of between small regions. As a result, if a leaf of an AST is modified, all the nodes belonging to the spine from the leaf to the root will lose their links.

Use of consistency links can also be found in Wang et al.'s work, where the authors extend state-based lenses to use links for tracing data in a view to its origin in a source. When the view is edited locally in a sub-term, 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 so that the update becomes an incremental one. Since lenses generated by our DSL also create consistency links (although for a different purpose), they can be naturally incrementalised using the same technique.

5.3 Operational-based BX

Our work is relevant to the operation-based approaches to BX, in particular, the delta-based BX model [Diskin et al. 2011a,b] and edit lenses [Hofmann et al. 2012]. The (asymmetric) delta-based BX model regards the differences between the view state v and v' as *deltas* and the differences are abstractly represented as arrows (from the old view to the new view). The main law in the framework is: 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 not sufficient for deriving retentiveness in their model; because there are infinite numbers of translated delta det_s which can take the old source to a correct state, of which few are retentive. To illustrate, Diskin et al. tend to represent deltas as edit operations such as *create*, *delete*, and *change*; representing deltas in this way will only tell the user in the new source what must be changed, while it requires additional work for reasoning about what is retained. However, it is possible to exhibit retentiveness if we represent deltas in some other proper way. Compared to Diskin et al.'s work, Hofmann et al. give concrete definitions and implementations for propagating edit operations.

6 CONCLUSIONS AND DISCUSSIONS

In this paper, we showed that well-behavedness is not sufficient for retaining information by giving a concrete example of code refactoring, in which the syntactic sugar and comments should be preserved in the newly generated program text. To address the problem, we illustrated how to use links to preserve desired data fragments of the old source after an update, 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. Then we illustrated the practical use of retentive lenses by running the code refactoring example introduced in the introduction. In the related work, we discussed the relations with alignment, origin tracking, and operation-based BX. Now, we will briefly discuss our choice of opting for triangular diagrams, composability of retentive lenses, and the feasibility of retaining code styles for refactoring tools in our framework.

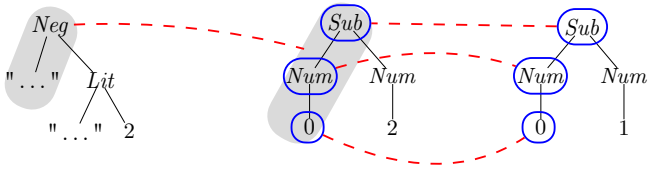
6.1 Opt for Triangular Diagrams

Including vertical correspondence (which represent 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-updating 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 [Diskin et al. 2011b] 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 Figure 4) with the consistency links produced by *get* to compute the input links between the source and the modified view. Although generic 'diff' on two pieces of algebraic data is difficult [Miraldo et al. 2017], 'domain-specific diff' is much easier. For the example of code refactoring, in which the relation between a view and its modified one is known by the refactoring algorithm: since the algorithm knows how a subtree is moved (i.e. how a subtree's position changes), it can output vertical correspondence recording this movement; other vertical correspondence between unchanged parts can be built position-wise.

6.2 Composability

Well-behaved (state-based) lenses are composable; it means that the composition of two well-behaved lenses is still a well-behaved one. For retentive lenses, we can also define a *comp* function, whose behaviour is similar to that of well-behaved lenses, apart from its producing proper intermediate links for get and put functions. Nevertheless, there are further requirements for composing retentive lenses. For the moment, we can only say that two retentive lenses $lens_1 :: RetLens\ A\ B$ and $lens_2 :: RetLens\ B\ C$ are composable, if $lens_1$ and $lens_2$ have the same property set on B; moreover, for any piece of data $b :: B$, $lens_1$ and $lens_2$ must decompose b in the same way. Otherwise the composition will cause problems, for example:



In the above figure, the first lens connects the region *Neg a* _ with *Sub* (*Num* 0) _ (the grey parts); while the second lens adopts a different decomposition strategy and decomposes *Sub* (*Num* 0) _ into three small regions and establishes links for them respectively. It is hard to determine the result of this link composition, and we leave this to the future work. Coincidentally, similar requirements can be found in quotient lenses [Foster et al. 2008]. 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. For 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.

As for our DSL, we argue that the lack of composition does not cause severe problems because of the philosophy of design. Take the scenario of writing a parser for example; 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 usually used as a 'standalone black box' and will not be composed with some others. Since our DSL is designed to be a 'lens generator', the user will have no difficulty in writing bidirectional transformations without composability.

6.3 Retaining Coding 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. Some of them can be retained by our DSL, such as the existence of a line break at a certain position, but some other stylings are more difficult to retain. 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 amount of spaces preceding the arguments. Instead, it should retain the *property* that the amount of spaces makes these two arguments vertically aligned.

Although not implemented in the DSL of this paper, these properties can be added to the set of *Property* in our framework of retentive lenses. For example, we may have VertAligned $x\ y \in$ *Property* for $x, y \in$ *Name*, and it is satisfied by a CST if the region at the location of y in the CST is vertically aligned with a region at the location of x. Then, when *get* computes AST from a vertically aligned argument, instead of including spaces preceding the argument as a part of the source region, *get* produces a link between a property VertAligned $x\ y$ and the AST region. Later when such links are supplied to *put*, they serve as directives for *put* to adjust the amount of spaces preceding the argument to conform to the styling rule. Compared with using a code formatter after refactoring, one advantage of handling styles in our framework is flexibility: the styling choices of the user — e.g., an argument opens a new line or stays at the current line — will not be overridden by predefined rules. However, handling coding styles can be very language-specific. Thus it is beyond the scope of this paper and left to the future work of a full-fledged refactoring tool.

REFERENCES

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing Strictly Positive Types. *Theoretical Computer Science* 342, 1 (2005), 3–27.

Andrew W. Appel. 1998. *Modern Compiler Implementation: In ML* (1st ed.). Cambridge University Press, New York, NY, USA.

F. Bancilhon and N. Spyratos. 1981. Update Semantics of Relational Views. *ACM Trans. Database Syst.* 6, 4 (Dec. 1981), 557–575.

Davi MJ Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C Pierce. 2010. Matching Lenses: Alignment and View Update. *ACM SIGPLAN Notices* 45, 9 (2010), 193–204.

Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. 2008. Boomerang: Resourceful Lenses for String Data. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '08)*. ACM, New York, NY, USA, 407–419.

James Cheney, Amal Ahmed, and Umut A. Acar. 2011. Provenance As Dependency Analysis. *Mathematical. Structures in Comp. Sci.* 21, 6 (Dec. 2011), 1301–1337.

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. 2009. Provenance in Databases: Why, How, and Where. *Found. Trends databases* 1, 4 (April 2009), 379–474.

Maartje de Jonge and Eelco Visser. 2012. An Algorithm for Layout Preservation in Refactoring Transformations. In *Software Language Engineering*. Springer, 40–59.

Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. 2011a. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology* 10 (2011).

Zinovy Diskin, Yingfei Xiong, Krzysztof Czarnecki, Hartmut Ehrig, Frank Hermann, and Fernando Orejas. 2011b. From State- to Delta-Based Bidirectional Model Transformations: the Symmetric Case. In *International Conference on Model Driven Engineering Languages and Systems (Lecture Notes in Computer Science)*. Springer, 304–318. https://doi.org/10.1007/978-3-642-24485-8_22

Eclipse Foundation. 2018. Eclipse documentation - Archived Release. (2018). http://help.eclipse.org/kepler/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Languages and Systems* 29, 3 (2007), 17. https://doi.org/10.1145/1232420.1232424

J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. 2008. Quotient Lenses. *SIGPLAN Not.* 43, 9 (Sept. 2008), 383–396. https://doi.org/10.1145/1411203.1411257

Martin Fowler and Kent Beck. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Peter Fritzson, Adrian Pop, Kristoffer Norling, and Mikael Blom. 2008. Comment-and Indentation Preserving Refactoring and Unparsing for Modelica. *Proceedings of the 6th International Modelica Conference*, 657–665.

GeeksforGeeks. 2017. For-each loop in Java. (2017). https://www.geeksforgeeks.org/for-each-loop-in-java/

James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. 2014. The Java Language Specification, Java SE 8 Edition (Java Series). (2014).

Martin Hofmann, Benjamin Pierce, and Daniel Wagner. 2012. Edit Lenses. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 495–508.

H. Li and S. Thompson. 2006. Comparative Study of Refactoring Haskell and Erlang Programs. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. 197–206. https://doi.org/10.1109/SCAM.2006.8

John MacFarlane. 2013. Pandoc: a Universal Document Converter. (2013).

Pedro Martins, João Saraiva, João Paulo Fernandes, and Eric Van Wyk. 2014. Generating attribute grammar-based bidirectional transformations from rewrite rules. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation*. ACM, 63–70.

Victor Cacciari Miraldo, Pierre-Évariste Dagand, and Wouter Swierstra. 2017. Type-directed Diffing of Structured Data. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Type-Driven Development (TyDe 2017)*. ACM, New York, NY, USA, 2–15. https://doi.org/10.1145/3122975.3122976

Oracle Corporation and OpenJDK Community. 2014. OpenJDK. (2014). http://openjdk.java.net/

Hugo Pacheco, Alcino Cunha, and Zhenjiang Hu. 2012. Delta Lenses Over Inductive Types. *Electronic Communications of the EASST* 49 (2012).

Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: Lifting Evaluation Sequences Through Syntactic Sugar. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, USA, 361–371. https://doi.org/10.1145/2594291.2594319

Justin Pombrio and Shriram Krishnamurthi. 2015. Hygienic Resugaring of Compositional Desugaring. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. ACM, New York, NY, USA, 75–87. https://doi.org/10.1145/2784731.2784755

Tillmann Rendel and Klaus Ostermann. 2010. Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing. *SIGPLAN Not.* 45, 11 (Sept. 2010), 1–12.

Perdita Stevens. 2008. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software & Systems Modeling* 9, 1 (2008), 7.

A. van Deursen, P. Klint, and F. Tip. 1993. Origin Tracking. *J. Symb. Comput.* 15, 5-6 (May 1993), 523–545.

Meng Wang, Jeremy Gibbons, and Nicolas Wu. 2011. Incremental Updates for Efficient Bidirectional Transformations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming (ICFP '11)*. ACM, New York, NY, USA, 392–403. https://doi.org/10.1145/2034773.2034825

Zirun Zhu, Yongzhe Zhang, Hsiang-Shang Ko, Pedro Martins, João Saraiva, and Zhenjiang Hu. 2016. Parsing and Reflective Printing, Bidirectionally. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering (SLE 2016)*. ACM, New York, NY, USA, 2–14.

A PROOF OF RETENTIVENESS

Theorem A.1. *Let $put' = put$ with its domain intersected with $S \times V \times Links$, get and put' form a retentive lens as in Definition 2.9.*

Proof. We prove they satisfy Retentiveness. Link Completeness and Correctness can be proved similarly.

Lemma A.2. *The recursion of put terminates.*

Proof. We prove by measuring the size of view argument and link argument. For the recursive call of the second case of *put* (9), $\|vs(t)\| \leqslant \|v\|$ and $\|ps \setminus \{\ l\ \}\| < \|ps\|$, thus the size of the input strictly decreased. For the recursive call of the first case of *put* (7), if $vpat_k$ is not a single variable pattern, then $\|vs(t)\| < \|v\|$, thus the size of the input strictly decreased. However, if $vpat_k = x$ for some pattern variable x, then $vs(t) = v$ and the link argument is not decreased either. But recall that we explicitly ruled out the possibility of an infinite sequence of such non-decreasing recursion in the domain of *put*. Thus *put* still terminates. □

This lemma enable us to prove Retentiveness by induction on the recursive structure of *put*. For arguments s, v and $ls = (ps,\ ms,\ mv)$, if (s, v, ls) falls into the first case of *put*, we have:

$$get(put(s, v, ls)) = get(reconstruct(spat'_k,\ ss))$$

where $spat'_k$ and ss are defined as in (6). Now we expand the definition of *get* (4). By the disjointness of source patterns, the same rule $spat_k \sim vpat_k$ will be selected as by *put*, thus:

$$\begin{aligned} get(put(s, v, ls)) &= genIsTop((reconstruct(vpat_k, fst \circ vls),\ \cdots \sqcup links)) \\ \text{where } vls &= (get' \circ decompose(spat_k,\ reconstruct(spat'_k,\ ss))) \\ &= get' \circ ss \\ &= get' \circ \lambda\,(t \in Vars(spat_k)) \longrightarrow \\ &\qquad put(s, vs(t), trimVPath(Path(vpat_k, t),\ ls)) \\ &= \lambda\,(t \in Vars(spat_k)) \longrightarrow \\ &\qquad get'(put(s, vs(t), trimVPath(Path(vpat_k, t),\ ls))) \qquad (10) \\ vs &= decompose(vpat_k, v) \end{aligned}$$

And *links* is:

$$\begin{aligned} links = \bigsqcup \big\{ &(ps_t, (Path(spat_k, t)\ ++) \circ ms_t, (Path(vpat_k, t)\ ++) \circ mv_t) \qquad (11) \\ &\mid t \in Vars(vpat_k),\ (\_, (ps_t, ms_t, mv_t)) = vls(t) \big\} \end{aligned}$$

By composing a *genIsTop* after (10), we can apply the inductive hypothesis: Let $vls(t) = (\_, (ps_t, ms_t, mv_t))$ and $trimVPath(Path(vpat_k, t),\ ls) = (rs_t, ns_t, nv_t)$, then:

$$rs_t^* \subseteq ps_{t\,\sigma} \quad \wedge \quad x \in NV_t.\ nv_t(x) = mv_t(\sigma^{-1}(x)) \tag{12}$$

where σ is some renaming function[7], NV_t is the set of all names appeared on the view side of rs_t, and $A^* = A \setminus \{\ (\mathsf{IsTop}\ x, \mathsf{IsTop}\ y) \mid x, y \in Name\ \}$ in the rest of this section.

From the definition of *trimVPath* and due to the well-aligned restriction we imposed on the domain of *put*, we have:

$$fst(ls)^* = \bigcup \{\ rs_t^* \mid t \in Vars(vpat_k)\ \}$$

Thus:

$$fst(ls)^* \subseteq \bigcup \{\ ps_t \mid t \in Vars(vpat_k)\ \} = fst(links)$$

Combined with the *genIsTop* at the last step, we have $fst(ls) \subseteq fst(links) \subseteq fst(snd(get(put(s, v, ls))))$. This is the first conjunct of Retentiveness that we want to show.

Let the argument $ls = (ps, ms, mv)$ and the result $get(put(s, v, ls)) = (ps', ms', mv')$, then the second conjunct of Retentiveness we want to show is:

$$\forall x \in NV.\ mv(x) = mv'(\sigma^{-1}(x))$$

Where NV is the names appeared on the view side of ps. Again from the well-aligned restriction, we know $\forall x \in NV$, there is a $t \in Vars(vpat_k)$ such that $x \in NV_t$. Intuitively, it means that every region with name x is within some subtree captured by t. From the definition of *trimVPath*, $mv(x) = nv_t(x) +\!\!+ Path(vpat_k, t)$. Then from the inductive hypothesis (12):

$$nv_t(x) +\!\!+ Path(vpat_k, t) = mv_t(\sigma^{-1}(x)) +\!\!+ Path(vpat_k, t)$$

From the definition of *links* (11), $mv'(\sigma^{-1}(x))$ is exactly $mv_t(\sigma^{-1}(x)) +\!\!+ Path(vpat_k, t)$. Thus $mv(x) = mv'(\sigma^{-1}(x))$, which is what we want.

What remains is the second case of *put*: showing Retentiveness when (s, v, ls) falls into (8). Similar to what we have done above:

$$get(put(s, v, ls)) = get(inj(reconstruct(spat_k,\ ss)))$$

where $spat_k$ and ss are defined as in (8). Because we restricted $inj(x)$ functions to be a *wrapping* of x, which means that if $get(x) = (v, (ps, ms, mv))$ and $get(inj(x)) = (v', (ps', ms', mv'))$ then $v' = v$ and $ps \subseteq ps' \wedge mv \subseteq mv'$. Thus it is sufficient to show Retentiveness for $get(reconstruct(spat_k,\ ss))$. As in the first case, we expand the definition of *get*:

$$\begin{aligned}
&get(put(s, v, (ps, ms, mv))) = genIsTop((reconstruct(vpat_k, fst \circ vls),\ l_{root} \sqcup links)) \\
&\quad \text{where } vls = (get' \circ decompose(spat_k,\ reconstruct(spat_k,\ ss))) \\
&\qquad = get' \circ ss \\
&\qquad = get' \circ \lambda\,(t \in Vars(spat_k)) \rightarrow \\
&\qquad\qquad put(s, vs(t), trimVPath(Path(vpat_k, t),\ (ps \setminus \{\ l\ \},\ ms,\ mv))) \\
&\qquad = \lambda\,(t \in Vars(spat_k)) \rightarrow \\
&\qquad\qquad get'(put(s, vs(t), trimVPath(Path(vpat_k, t),\ (ps \setminus \{\ l\ \},\ ms,\ mv)))) \qquad (13) \\
&\quad vs = decompose(vpat_k, v) \\
&\quad l = (\mathsf{HasRegion}\ spat_k\ x,\ \mathsf{HasRegion}\ vpat_k\ y) \in ps \\
&\qquad \text{such that } mv(y) = \epsilon,\ ms(x) \text{ is the shortest}
\end{aligned}$$

[7] Because *get* always uses fresh names, we can assume a uniform σ exists for all $t \in Vars(spat_k)$.

where *links* is the same as in (11), l_{root} is:

$$l_{root} = \Big(\{\ (\text{HasRegion } \mathit{fillWildcards}(\mathit{spat}_k, s)\ x,\ \text{HasRegion } \mathit{vpat}_k\ y)\ \} \\ \cup\ \mathit{relPos},\ \{\ x \mapsto \epsilon\ \},\ \{\ y \mapsto \epsilon\ \} \Big)$$

resPos, defined as in (5), is the set of properties asserting relative positions of the regions at the top and regions in the subtrees that are directly adjacent to the top regions. Links in (ps, ms, mv) comprises three parts: (a) a link of the top source region and view region, (b) links of regions or properties of regions that are in subtrees of the top regions, and (c) links of properties asserting the relative positions of the top region and regions below them. For (a), the link is retained by the first component of l_{root}. For (b), those links are retained by applying the inductive hypothesis to the recursive call, as what we did in the first case of the proof. What remains is to show links of (c) are also retained by *put*, though *put* does not handle them explicitly.

If a link $(\text{RelPos } p_s\ x_s\ y_s,\ \text{RelPos } p_v\ x_v\ y_v) \in ps$ with $ms(x_s) = mv(x_v) = \epsilon$, because we restricted links in *ps* can be generated by *get* and our *get* only generates pairs of RelPos between adjacent regions, we know $p_s = \mathit{Path}(\mathit{spat}_k, t_0)$ and $p_v = \mathit{Path}(\mathit{vpat}_k, t_0)$ for some $t_0 \in \mathit{Vars}(\mathit{spat}_k)$. Then by the definition of satisfaction this link, $ms(y_s) = \mathit{Path}(\mathit{spat}_k, t_0)$ and $mv(y_v) = \mathit{Path}(\mathit{vpat}_k, t_0)$. And again by our restriction that links in *ps* must be a subset of some *get*, we know that the source region with name y_s must be linked to the view region with name y_v. Thus $(\text{HasRegion } \mathit{pat}_s\ y_s,\ \text{HasRegion } \mathit{pat}_v\ y_v) \in ps$ for some patterns pat_s and pat_v. By the definition of *trimVPath*, $\mathit{trimVPath}(\mathit{Path}(\mathit{vpat}_k, t_0), (ps \setminus \{\ l\ \}, ms, mv))$ still contains the link between y_s and y_v with the path of y_v being trimmed to be ϵ. Thus the recursive call

$$\mathit{put}(s, vs(t_0), \mathit{trimVPath}(\mathit{Path}(\mathit{vpat}_k, t_0), (ps \setminus \{\ l\ \}, ms, mv)))$$

in (13) falls into the second case of *put* and the link between region y_s and y_v will be selected because region y_s is directly adjacent to the region above (with name x_s) so that it must have the shortest path among regions linked to y_v. Thus — letting $vls(t_0) = (ps_{t_0}, ms_{t_0}, mv_{t_0})$ — we have some $ms_{t_0}(x_{t_0}) = mv_{t_0}(y_{t_0}) = \epsilon$ such that $\sigma(x_{t_0}) = y_s$, $\sigma(y_{t_0}) = y_v$ and $(\text{HasRegion } \mathit{pat}_s\ x_{t_0},\ \text{HasRegion } \mathit{pat}_v\ y_{t_0})$. Recall the definition of *relPos*:

$$\begin{aligned} \mathit{relPos} = \big\{ \big(&\text{RelPos } \mathit{Path}(\mathit{spat}_k, t)\ x\ st,\ \text{RelPos } \mathit{Path}(\mathit{vpat}_k, t)\ y\ vt\big) \\ &\mid t \in \mathit{Vars}(\mathit{vpat}_k),\ (\_, (\_, m_{src}, m_{view})) = vls(t), \\ &m_{src}(st) = \epsilon,\ m_{view}(vt) = \epsilon \big\} \end{aligned}$$

Thus the link $\big(\text{RelPos } p_s\ x\ x_{t_0},\ \text{RelPos } p_v\ y\ y_{t_0}\big)$ is contained in *relPos*. Since:

$$\begin{aligned} \mathit{relPos} &\subseteq \mathit{fst}(l_{root}) \\ &\subseteq \mathit{fst}(\mathit{snd}(\mathit{put}(\mathit{get}(s, v, (ps, ms, mv)))))) \end{aligned}$$

We have:

$$(\text{RelPos } p_s\ x\ y_s,\ \text{RelPos } p_v\ y\ y_v) \in \mathit{fst}(\mathit{snd}(\mathit{put}(\mathit{get}(s, v, (ps, ms, mv))))))$$

which is what we want to show. Finally, if a link $(\text{IsTop } x_s,\ \text{IsTop } x_v) \in ps$, it is also retained by the *genIsTop* at the last step. Thus all links of properties of regions are retained as well as links of regions. This completes our proof of Retentiveness. □