# 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 programming, asymmetric lenses

**ACM Reference Format:**
Anonymous Author(s). 2018. Retentive Lenses. *Proc. ACM Program. Lang.* 1, CONF, Article 1 (January 2018), 34 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.

To see this, let us first review the definition of well-behaved lenses, borrowing some of Stevens's terminologies [Stevens 2008]. 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.

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

```
get :: (Int, String) → Int
get (i, s) = i
```

Given this specific *get*, we can define $put_1$ and $put_2$, both of which are well-behaved with *get* but the behaviour is rather different: $put_1$ simply replace the integer of the source (tuple) with the view, while $put_2$ superfluously sets the string (of the tuple) empty, in silence, when the source (tuple) is not consistent with the view.

```
put1 :: (Int, String) → Int → (Int, String)      put2 :: (Int, String) → Int → (Int, String)
put1 (i, s) i′ = (i′, s)                          put2 src    i′ | get src == i′ = src
                                                  put2 (i, s) i′ | otherwise     = (i′, "")
```

From another perspective, $put_1$ retains the string value from the old source when performing the update, while $put_2$ chooses to discard that information — which is not desired but 'perfectly legal', for the string data does not contribute to the consistency relation. In fact, unexpected behaviour of this kind of well-behaved lenses could even lead to disaster in practice: For instance, relational databases are tables consisting of rows of tuples, and well-behaved lenses used for maintaining a database and its view may erase important data after an update, as long as the data does not contribute to the consistency relation. (In most cases, it is simply equal to saying that the data is not in the view.) This fact seems fatal, as asymmetric lenses have been considered a satisfactory solution [Foster et al. 2007] to the longstanding view-update problem (stated in the beginning of the paper).

The root cause of the information loss (after an update) is that while lenses are designed to retain information, well-behavedness actually says very little about it: the only law guaranteeing information retention is Hippocraticness, which merely requires that the *whole* source should be unchanged if the *whole* view is. In other words, if we have a very small change on the view, we are

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

free to create any source we like. This is too 'global' in most cases, and it is desirable to have a law that makes such a guarantee more 'locally'.

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

The main contributions of the paper are as follows:

- We formalise the idea of links and develop a formal definition of retentive lenses for algebraic data types[2]. More specifically, we extend *get* and *put* to incorporate links, and add a finer-grained law *Retentiveness*, which precisely captures the intention 'if parts of the view are unchanged, then the corresponding parts of the source should be retained' (Section 2).
- To show that retentive lenses are feasible, we present a simple domain-specific language (DSL) tailored for developing synchronisation between syntax trees. We present its syntax, semantics, and also prove that any program written in our DSL gives rise to a pair of retentive get and put (Section 3).
- We demonstrate the usefulness of Retentiveness in practice by presenting two case studies on *resugaring*and *code refactoring*. We show that retentive lenses provide a systematic way for the user to preserve information of interest in the original source after synchronisation on demand (Section 4).

After explaining technical contributions, we discuss related work regarding various alignment strategies for lenses, provenance and origin between two pieces of data, and operational-based bidirectional transformations (BX) (Section 5). Finally, we conclude the paper; we briefly discuss why we choose triangular diagrams and a simple version of Retentiveness, our thought on (retentive) lens composition, and the feasibility of retaining code styles for refactoring tools. (Section 6).

## 2 RETENTIVE LENSES FOR TREES

In this section, we will start by introducing a 'region model' for decomposing trees (Section 2.1), and providing a high-level sketch of what retentive lenses should do (Section 2.2). After that, we develop a formal definition of links (Section 2.3) and retentive lenses (Section 2.4) for algebraic data types, through revising classic lenses [Foster et al. 2007] by

- extending *get* and *put* to incorporate links — specifically, we will make *get* return a collection of consistency links, and make *put* additionally take a collection of input links — and
- adding 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'.

To be concrete, we will use the concrete and abstract representations of arithmetic expressions as the running example throughout the paper.

[2]This paper focuses on the synchronisation between algebraic data types, i.e. trees, whose structure shall be considered more general than tables in relational databases. (A table is a list of tuples, and both lists and tuples can be encoded as trees.)

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

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

## 2.1 Regions of Trees

Let us first get familiar with the concrete and abstract representations of arithmetic expressions — which our examples are based on — as defined in Figure 1, 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 annotations, explicit parentheses, and negations — negations are considered as *syntactic sugar* and represented by some other more fundamental construct, here *Sub*, in the AST. The consistency relations between CSTs and ASTs are defined in terms of the *get* functions — $e :: Expr$ (resp. $t :: Term$) is consistent with $a :: Arith$ exactly when $getE\ e = a$ (resp. $getT\ t = a$).

As mentioned in the introduction, 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 2, the grey areas are some of the possible regions: The topmost region in *cst* is described by the region pattern *Plus* `"a plus"` _ _, which says that the region includes the *Plus* node and the annotation `"a plus"`, but not the other two subtrees with roots *Minus* and *Neg* matched by the wildcards. 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 2, 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*.

If the view is modified, the links between the source and view can be modified to reflect the latest correspondences between regions. For example, in Figure 2, if we change *ast* to *ast′* by swapping

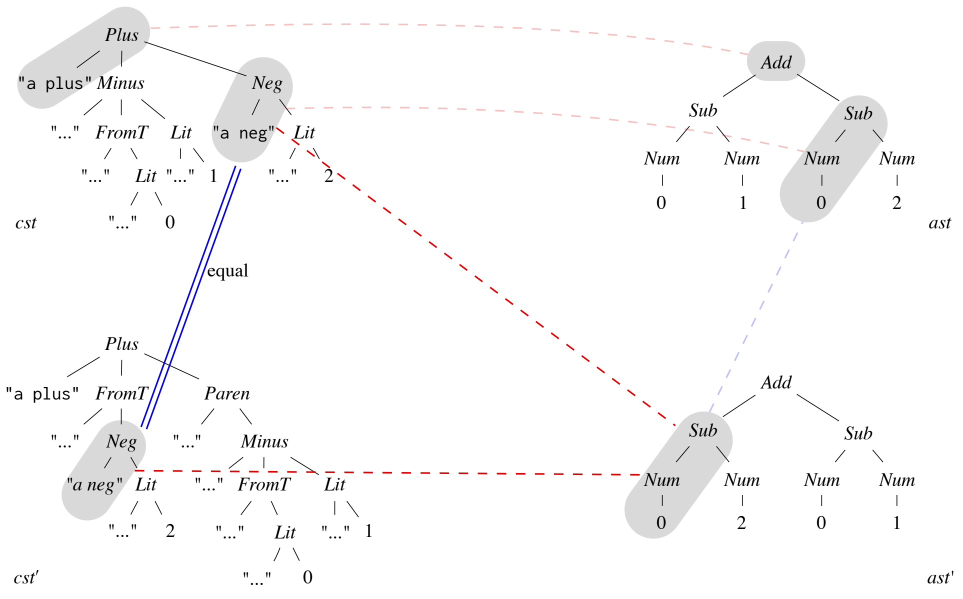


Fig. 2. The triangular guarantee. (Grey areas are some of the possible regions. Two light dashed lines between *cst* and *ast* are two of the consistency links produced by *get cst*. When updating *cst* with *ast*′, the region *Neg* `"a neg"` _ connected by the red dashed diagonal link is guaranteed to be preserved in the result *cst*′, and *get cst*′ will link the preserved region to the same region *Sub* (*Num* 0) _ of *ast*′. )

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 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 [it] respects these links. More precisely, *put* should provide the *triangular guarantee*: Using Figure 2 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 some region in the new source *cst*′ and the *Sub* (*Num* 0) _ region in the view *ast*′, and that region in *cst*′ should have the same pattern as the one specified by the input link, namely *Neg* `"a neg"` _, preserving the syntactic sugar negation (as opposed to changing it to a *Minus*, for example) and the associated annotation.

## 2.3 Formalisation of Links

*2.3.1 Basics of Links.* As described in Section 2.2, a link relates two regions, and a region consists of a pattern describing the region's content and a location, which can be a path describing how to reach the top of the region from the root. One idea is to formalise a region as an element of the set *Region*, where *Pattern* is the set of region patterns and *Path* is the set of paths; then a link is an element of the set *Link*:

$$\begin{aligned} \mathit{Region} &= \mathit{Pattern} \times \mathit{Path} \\ \mathit{Link} &= \mathit{Region} \times \mathit{Region} \ . \end{aligned}$$

The exact representation of paths is not important and may be different for each application, but to be simpler, let us assume that a path is just a list of natural numbers that indicate which subtree to go into: for instance, starting from the root of *cst* in Figure 2, the empty path $[\,]$ points to the root node *Plus*, the path $[0]$ points to `"a plus"` (which is the first subtree under the root), and the path $[2, 0]$ points to `"a neg"`.

*Example 2.1.* In Figure 2, the two shaded regions of *cst* are described by ($\mathit{Plus}$ `"a plus"` $\_\ \_, [])$ and ($\mathit{Neg}$ `"a neg"` $\_, [2])$ respectively. The diagonal link is represented as (($\mathit{Neg}$ `"a neg"` $\_, [2])$, $(\mathit{Sub}\ (\mathit{Num}\ 0)\ \_, [0])$).

Let $\mathcal{P}\,(A)$ denote the power set of $A$, we define the set of collections of links to be

$$\mathit{LinkSet} = \mathcal{P}\,(\mathit{Link}) \ .$$

*Links as Relations.* For the simplicity of notation, we regard a collection of links $ls : \mathit{LinkSet}$ as a relation between $\textsc{ldom}(ls)$ and $\textsc{rdom}\ (ls)$, where $\textsc{ldom}(ls) = \{\, \phi_s \mid \exists \phi_v .\ (\phi_s, \phi_v) \in ls \,\}$ and $\textsc{rdom}\ (ls) = \{\, \phi_v \mid \exists \phi_s .\ (\phi_s, \phi_v) \in ls \,\}$. Here we introduce some standard notations of relations, which will be used later. The converse of a relation $r^{\circ} : B \sim A$ relates $b \in B$ and $a \in A$ exactly when $r : A \sim B$ relates $a$ and $b$. The composition $l \cdot r : A \sim C$ of two relations $l : A \sim B$ and $r : B \sim C$ is defined as usual: $(x, z) \in l \cdot r$ exactly when there exists $y$ such that $(x, y) \in l$ and $(y, z) \in r$. We will allow functions to be implicitly lifted to relations: a function $f : A \longrightarrow B$ also denotes a relation $f : B \sim A$ such that $(f\ x, x) \in f$ for all $x \in A$. This flipping of domain and codomain (from $A \longrightarrow B$ to $B \sim A$) makes functional composition compatible with relational composition: a function composition $g \circ f$ lifted to a relation is the same as $g \cdot f$, i.e. the composition of $g$ and $f$ as relations.

*Example 2.2.* The diagonal link between *cst* and *ast′* in Figure 2 is represented as the link collection $\{\, ((\mathit{Neg}$ `"a neg"` $\_, [2]), (\mathit{Sub}\ (\mathit{Num}\ 0)\ \_, [0])) \,\}$ (despite of being a singleton), which can also be regarded as a relation relating ($\mathit{Neg}$ `"a neg"` $\_, [2])$ to $(\mathit{Sub}\ (\mathit{Num}\ 0)\ \_, [0])$ and vice versa.

*2.3.2 Validity of Links.* A collection of links may not make sense for a given pair of 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 region patterns on the source side of the links are satisfied by the source tree and all region patterns on the view side are satisfied by the view tree.

*Definition 2.3 (Link Satisfaction).* For a tree $t$ and a set of regions $\Phi \subseteq \mathcal{P}\,(\mathit{Region})$, we say that $t \models \Phi$ (read '$t$ satisfies $\Phi$') exactly when

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

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

*Definition 2.4 (Valid Links).* For a collection of links $ls \in \mathit{LinkSet}$, a source $s \in S$, and a view $v \in V$, we say that $ls$ is *valid* for $s$ and $v$, denoted by $s \overset{ls}{\longleftrightarrow} v$, exactly when

$$s \models \textsc{ldom}(ls) \quad \text{and} \quad v \models \textsc{rdom}(ls)\ .$$

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

## 2.4 Definition of Retentive Lenses

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

*Definition 2.6 (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 (denoted by $\nrightarrow$) functions $get$ and $put$ of type

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

satisfying

- *Hippocraticness*: if $get(s) = (v, ls)$, then $(s, v, ls) \in \textsc{dom}(put)$ and
$$s \overset{ls}{\longleftrightarrow} v \quad \wedge \quad put\ (s, v, ls) = s\ ; \tag{1}$$
- *Correctness*: if $put(s, v, ls) = s'$, then $s' \in \textsc{dom}(get)$ and
$$get(s') = (v, ls') \quad \text{for some } ls'; \tag{2}$$
- *Retentiveness*: (continuing above)
$$\mathit{fst} \cdot ls \subseteq \mathit{fst} \cdot ls' \tag{3}$$

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

Proposition 2.7 (Triangular guarantee for regions). *Given $ls$, $(s, v, ls) \in \textsc{dom}(put)$, and*

$$\phi_s = (spat, spath) \quad \wedge \quad \phi_v = (vpat, vpath)$$

*if $(\phi_s, \phi_v) \in ls$ and $(v, ls') = get(put(s, v, ls))$, then we have*

$$(spat, (vpat, vpath)) \in \mathit{fst} \cdot ls'\ .$$

*Intuitively, this asserts that the region pattern spat still exists in the updated source and corresponds to the same view region $(vpat, vpath)$ as in the input.*

*Example 2.8.* In Figure 2, if a $put$ function takes $cst$, $ast'$, and diagonal links $ls = \{\,((\mathit{Neg}\ \texttt{"a neg"}\ \_, [2]), (\mathit{Sub}\ (\mathit{Num}\ 0)\ \_, [0]))\,\}$ as arguments and successfully produces an updated source $s'$, then $get(s')$ will succeed. Let $(v, ls') = get(s')$; we know that we can find a consistency link with the path of its source region pattern removed $c = (\mathit{Neg}\ \texttt{"a neg"}\ \_, (\mathit{Sub}\ (\mathit{Num}\ 0)\ \_, [0])) \in \mathit{fst} \cdot ls'$. So the view region referred to by $c$ is indeed the same as the one referred to by the input link, and having $c \in \mathit{fst} \cdot ls'$ means that the region in $s'$ corresponding to the view region will match the pattern $\mathit{Neg}\ \texttt{"a neg"}\ \_$.

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

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

where

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

In the definition, *ToPat* turns a tree into a pattern completely describing the tree, and $\textsc{dom}(put)$ is restricted to $\left\{ (s, v, ls) \mid s \overset{ls}{\longleftrightarrow} v \text{ and } ls = trivial(s, v) \text{ or } \emptyset \right\}$. The idea is that *get* generates a link collection relating the whole source tree and view tree as two regions, and *put* accepts either the trivial link produced by *get* or an empty collection of links. Hippocraticness and Correctness hold because the underlying $g$ and $p$ is well-behaved. When the input link of *put* is empty, Retentiveness is satisfied vacuously; when the input link is the trivial link, Retentiveness is guaranteed by Hippocraticness of $g$ and $p$ by the following calculation; Thus *get* and *put* indeed form a retentive lens. The proof can be found in appendix (A).

## 2.5 Lens Composition

It is standard to provide a composition operator for composing large lenses from small ones; in this subsection, we discuss this operator for retentive lenses, which basically follows the definition of composition for well-behaved lenses, except that we need to carefully deal with links additionally.

We first introduce some notations that will be used. We use $(\cdot)$ to denote *link composition* and its (overloaded) lifted version that works on two collections of links. (Although $(\cdot)$ was used for relation composition before, we have shown that a single link can be lifted to a singleton link collection and a collection of links can be viewed as a relation; so it is reasonable to reuse the same symbol.) Also, for simplicity, we use $lens_{AB}$[3] to denote a (retentive) lens doing synchronisation between trees of sets $A$ and $B$, $l_{ab}$ a link between tree $a$ (of set $A$) and tree $b$ (of set $B$), and $ls_{ab}$ a collection of links between $a$ and $b$.

Now, we define the composition of two retentive lenses.

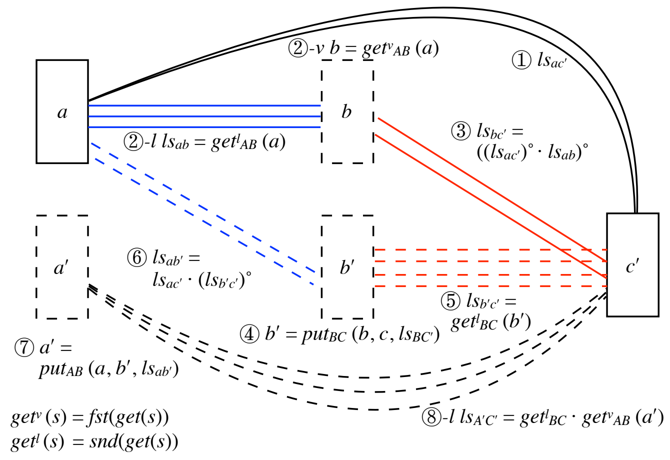
*Definition 2.10 (Retentive Lens Composition).* Given two retentive lenses $lens_{AB}$ and $lens_{BC}$, their composite lens $lens_{AC}$ is denoted by $lens_{AB}; lens_{BC}$ and the corresponding $get_{AC}$ and $put_{AC}$ functions are defined by

$$\begin{aligned} get_{AC}\ (a) &= (c, ls_{ab} \cdot ls_{bc}) \\ \text{where } (b, ls_{ab}) &= get_{AB}\ (a) \\ (c, ls_{bc}) &= get_{BC}\ (b) \end{aligned}$$

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

where $get_{AB}$ and $put_{AB}$ (resp. $get_{BC}$ and $put_{BC}$) are the corresponding *get* and *put* functions of $lens_{AB}$ (resp. $lens_{BC}$), and $ls^{\circ}$ is the collection of links produced from $ls$ by swapping each link's starting point and ending point. (Recall that a collection of links is regarded as a relation, so we use the converse symbol for 'reversed links'.)

[3] We use *lens* to denote a retentive lens (instead of *rlens*) when it is clear from the context.



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

Fig. 3. The *put* behaviour of a composite retentive lens $lens_{AB}; lens_{BC}$. (The process of *put* is divided into steps ① to ⑧. Step ⑧ produces consistency links for showing triangular guarantee.

While the *get* behaviour of a composite retentive lens is straightforward from the definition, the *put* behaviour is a little complex and can be best understood with the help of Figure 3. Let us first recap the behaviour of *put* of well-behaved lenses; in Figure 3, if we need to propagate changes from data $c'$ back to data $a$ without links, we will first construct the *intermediate* data $b$ (by running $get_{AB}\ a$), propagate changes from $c'$ to $b$ and produce $b'$, and finally use $b'$ to update $a$. Now we consider retentive lenses; it is similar for them to propagate changes from $c'$ to $a$ with a collection of links $ls_{ac'}$ retaining some information: Besides the intermediate data $b$, we also need to construct intermediate links $ls_{bc'}$ (③ in the figure) for retaining information when updating $b$ to $b'$, so that we can further construct intermediate links $ls_{ab'}$ (⑥ in the figure) for retaining information when updating $a$ to $a'$ using $b'$.

Theorem 2.11 (Retentiveness Preservation). *The composite lens $lens_{AC} = lens_{AB}; lens_{BC}$ from two retentive lenses $lens_{AB}$ and $lens_{BC}$ is still a retentive lens.*

The proof is available in the appendix (Appendix B).

## 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 1. From each consistency relation defined in the DSL, we can obtain a pair of *get* and *put* functions forming a retentive lens. Furthermore, we show how to flexibly update the *cst* Figure 2 in different ways using the generated *put* function with different input links.

In our DSL, we define data types in Haskell syntax and describe consistency relations between them as if writing *get* functions. For example, the data type definitions for *Expr* and *Term* written in

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

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

our DSL remain the same as those in Figure 1 and the consistency relations between them (i.e. the *getE* and *getT* functions in Figure 1) are expressed as the ones in Figure 4. Here we describe two consistency relations similar to *getE* and *getT*: one between *Expr* and *Arith*, and the other between *Term* and *Arith*. Each consistency relation is further defined by a set of inductive rules, stating that if the subtrees matched by the same variable appearing on the left-hand side (source side) and right-hand side (view side) are consistent, then the large pair of trees constructed from these subtrees are also consistent. (For primitive types which do not have constructors such as integers and strings, we consider them consistent if and only if they are equal.) Take *Plus* _ *x y* ∼ *Add x y* for example; it means that if $x_s$ is consistent with $x_v$, and $y_s$ is consistent with $y_v$, then *Plus* $a\ x_s\ y_s$ and *Add* $x_v\ y_v$ are consistent for any value $a$, where $a$ corresponds to a 'don't-care' wildcard in *Plus* _ *x y*. So the meaning of *Plus* _ *x y* ∼ *Add x y* can be better understood by the 'derivation rule' beneath:

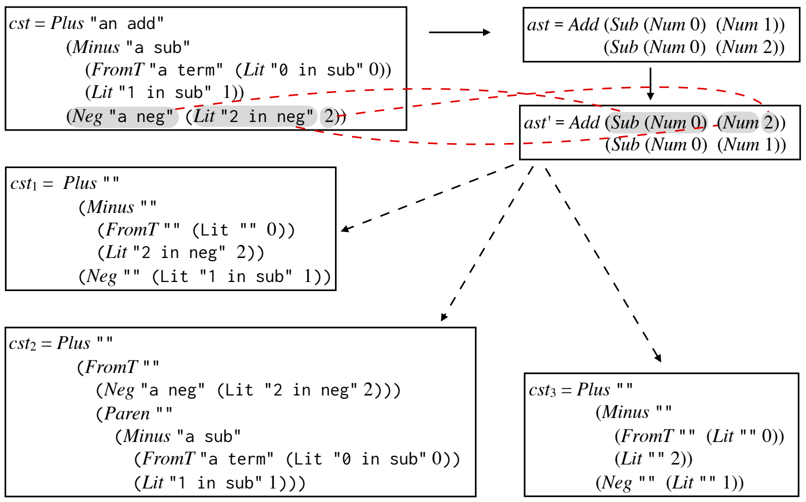
$$\frac{x_s \sim x_v \quad y_s \sim y_v}{\forall a.\ \mathit{Plus}\ a\ x_s\ y_s \sim \mathit{Add}\ x_v\ y_v}$$

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

Now we briefly explain the semantics of *Plus* _ *x y* ∼ *Add x y*. In the *get* direction, its behaviour is quite similar to the first case of *getE* in Figure 1, except that it also updates the links between subtrees marked by *x* and *y* respectively, and establishes a new link between the top nodes *Plus* and *Add* recording the correspondence. In the *put* direction, 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 retentive lenses (i.e. the above *get* and *put*) generated from Figure 4, we can synchronise the old *cst* and modified *ast′* in different ways by running *put* with different input links. Figure 5 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 5 shows the input links for *Neg* and its subtrees.



```
cst = Plus "an add"
        (Minus "a sub"
          (FromT "a term" (Lit "0 in sub" 0))
          (Lit "1 in sub" 1))
        (Neg "a neg" (Lit "2 in neg" 2))
```

```
ast = Add (Sub (Num 0) (Num 1))
          (Sub (Num 0) (Num 2))
```

```
ast' = Add (Sub (Num 0) (Num 2))
           (Sub (Num 0) (Num 1))
```

```
cst1 = Plus ""
         (Minus ""
           (FromT "" (Lit "" 0))
           (Lit "2 in neg" 2))
         (Neg "" (Lit "1 in sub" 1))
```

```
cst2 = Plus ""
         (FromT ""
           (Neg "a neg" (Lit "2 in neg" 2)))
           (Paren ""
             (Minus "a sub"
               (FromT "a term" (Lit "0 in sub" 0))
               (Lit "1 in sub" 1)))
```

```
cst3 = Plus ""
         (Minus ""
           (FromT "" (Lit "" 0))
           (Lit "" 2))
         (Neg "" (Lit "" 1))
```

Fig. 5. *cst* is updated in many ways. (Red dashed lines are some of the input links for $cst_2$.)

- If the user believes that *ast′* is created from scratch and not related to *ast*, then $cst_3$ may be the best choice, which represents the same expression as $cst_1$ except that all the annotations are removed.

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

$$\textbf{data}\ BinT\ a = Tip \mid Node\ a\ (BinT\ a)\ (BinT\ a)$$

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

$$\begin{array}{l} BinT\ Int \leftrightarrow BinT\ Int \\ \quad Tip \qquad\quad \sim Tip \\ \quad Node\ i\ x\ y \sim Node\ i\ y\ x \end{array}$$

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:

$$\begin{array}{l} Nat \leftrightarrow Bool \\ \quad Succ\ \_ \sim True \\ \quad Zero \quad \sim False \end{array}$$

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

$$\begin{array}{l} BinT\ Nat \leftrightarrow BinT\ Bool \\ \quad Tip \qquad\quad \sim Tip \\ \quad Node\ x\ ls\ rs \sim Node\ x\ ls\ rs \end{array}$$

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

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

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

Program

$$\begin{array}{lll} \mathit{prog} & ::= & \mathit{tDef}\ \mathit{cDef}_1 \ldots \mathit{cDef}_n \end{array}$$

Type Definition

$$\begin{array}{lllll} \mathit{tDef} & ::= & \mathit{tSyn}_1 \ldots \mathit{tSyn}_m\ \mathit{tNew}_1 \ldots \mathit{tNew}_n & & \\ \mathit{tSyn} & ::= & \text{`type'}\ \mathit{type}\ \text{`='}\ \mathit{type} & & \\ \mathit{tNew} & ::= & \text{`data'}\ \mathit{type} & \text{`='} & C_1\ \mathit{type}_{11} \ldots \mathit{type}_{1k_1} \\ & & & \text{`|'} & \ldots \\ & & & \text{`|'} & C_n\ \mathit{type}_{n1} \ldots \mathit{type}_{nk_n} \\ \mathit{type} & ::= & \mathit{tycon}\ \mathit{type}_1 \ldots \mathit{type}_m & & \{\text{type constructor}\} \\ & | & \mathit{tyvar} & & \{\text{type variable}\} \end{array}$$

Consistency Relation Definition

$$\begin{array}{llll} \mathit{cDef} & ::= & \mathit{type}_s \longleftrightarrow \mathit{type}_v & \{\text{relation type}\} \\ & & \quad r_1 \ldots r_n\ \text{`;;'} & \{\text{inductive rules}\} \end{array}$$

Inductive Rule

$$\begin{array}{lll} r & ::= & \mathit{pat}_s\ \text{`}\sim\text{'}\ \mathit{pat}_v \end{array}$$

Pattern

$$\begin{array}{llll} \mathit{pat} & ::= & v & \{\text{variable pattern}\} \\ & | & \_ & \{\text{don't-care wildcard}\} \\ & | & C\ \mathit{pat}_1 \ldots \mathit{pat}_n & \{\text{constructor pattern}\} \end{array}$$

Fig. 6. Syntax of the DSL.

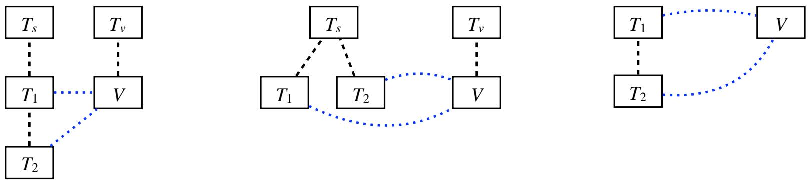
$$\begin{array}{lll} \multicolumn{3}{l}{\mathit{RTree\ Int} \leftrightarrow \mathit{List\ Int}} \\ \quad \mathit{RNode}\ i\ \mathit{Nil} & \sim & \mathit{Cons}\ i\ \mathit{Nil} \\ \quad \mathit{RNode}\ i\ (\mathit{Cons}\ x\ \_) & \sim & \mathit{Cons}\ i\ x \end{array}$$

## 3.2 Syntax

Having seen some programming examples, here we summarise the syntax of our language in Figure 6. A program consists of two main parts: definitions of data types and consistency relations between these data types. We adopt the Haskell syntax for definitions of data types, so that a new data type is defined through a set of data constructors $C_1 \ldots C_n$ followed by types, 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. Now we move to the definitions of consistency relations, where each of them starts with $\mathit{type}_s \leftrightarrow \mathit{type}_v$, 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 $\mathit{pat}_s \sim \mathit{pat}_v$, and a pattern $\mathit{pat}$ includes variables, constructors, and wildcards. Finally, two semicolons ';;' are added to finish the definition of a consistency relation. (In this paper, we always omit ';;' when there is no confusion.)

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

- On *patterns*, we assume (i) pattern coverage and source pattern disjointness. For any consistency relation $T_s \leftrightarrow T_v = \{\, p_i \sim q_i \mid 1 \leqslant i \leqslant n \,\}$ defined in a program, $\{\, p_i \,\}$ should cover all the possible cases of type $T_s$, and $\{\, q_i \,\}$ should cover all the cases of type $T_v$. Source pattern disjointness requires that any distinct $p_i$ and $p_j$ do not match any tree at the same time so that at most one pattern is matched when running *get*. Additionally, (ii) a bare variable pattern is not allowed on the source side (e.g. $x \sim D\ x$) and wildcards are not allowed on the view side (e.g. $C\ x \sim D\ \_\ x$).
- On *algebraic data types*, we require that types $T_1$ and $T_2$ be *interconvertible* when (i) consistency relations $T_1 \leftrightarrow V$, $T_2 \leftrightarrow V$, and $T_s \leftrightarrow T_v$ are all defined for some types $T_s$, $V$, and $T_v$, and (ii) the definition of $T_s$ makes use of both $T_1$ and $T_2$ (directly or indirectly), and the definition of $T_v$ makes use of $V$. The following figure depicts some of the cases, in which the vertical dotted links (in black) mean that there may be other nodes in between the connected nodes. The blue horizontal links show that both $T_1$ and $T_2$ can be consistent with $V$.



In this case, in the *put* direction, there might be links asserting values of type $T_2$ should be retained in a context where values of type $T_1$ are expected, or vice versa; thus we need a way to convert between $T_1$ and $T_2$. 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 4 share the same view type $Arith$, there should be a way to convert (inject) $Expr$ into $Term$ and vice versa. Look at $cst'$ in Figure 2, the second subtree of $Plus$ is of type $Expr$ but created by using a link connected to the old source's subtree $Neg$ ... of type $Term$, so we need to wrap $Neg$ ... into $FromT$ "" ($Neg$ ...) to make the types match.

In our DSL, the conversions (injections) for interconvertible data types are automatically generated from consistency relations. To have an injection $inj_{T_1 \to T_2}\ x = C\ \ldots\ x\ \ldots$ which directly injects data of type $T_1$ to data of type $T_2$, it is equal to saying that the consistency relation $T_2 \leftrightarrow V$ has a rule $C\ \_\ \ldots\ x \ldots\ \_ \sim x$ in which the source pattern only has wildcards except $C$ and $x$. For example, $FromT\ \_\ t \sim t$ gives rise to an injection $inj_{Term \to Expr}\ x = FromT$ "" $x$. Note that in general $T_1$ can be indirectly injected to $T_2$ through a *chain* of such rules.

## 3.3 Semantics

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

*3.3.1 Types and Patterns.* To define *get* and *put*, we need to first give the semantics of type declarations and patterns. The semantics of type declarations is the same as 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, we define $Tree$ to be the set of all the values of all the algebraic data types defined in our DSL and $Pattern$ to be the set of all possible patterns. For a pattern $p \in \mathit{Pattern}$, $\mathit{Vars}(p)$ denotes the set of variables in $p$, $\mathit{TypeOf}(p, v)$ is the set corresponding to the type of $v \in \mathit{Vars}(p)$, and $\mathit{Path}(p, v)$ is the path of variable $v$ in pattern $p$.

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

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

Given a pattern $p$ and a value (i.e. tree) $t$, $\mathit{isMatch}(p, t)$ tests if $t$ matches $p$. If the match succeeds, $\mathit{decompose}(p, t)$ returns a function $f$ mapping every variable in $p$ to its corresponding matched subtree of $t$. Conversely, $\mathit{reconstruct}(p, f)$ produces a tree $t$ matching pattern $p$ by replacing every occurrence of $v \in \mathit{Vars}(p)$ in $p$ with $f(v)$, provided that $p$ does not contain any wildcard. Since the semantics of patterns in our DSL is rather standard, we omit detailed definitions of these functions[4].

*3.3.2 Get Semantics.* For a consistency relation $S \leftrightarrow V$ defined in our DSL with a set of inductive rules $R = \{\, \mathit{spat}_k \sim \mathit{vpat}_k \mid 1 \leqslant k \leqslant n \,\}$, its corresponding $\mathit{get}_{SV}$ function has the following type:

$$\mathit{get}_{SV} : S \longrightarrow V \times \mathit{LinkSet}$$

The idea of $\mathit{get}(s)$ is to use the rule $\mathit{spat}_k \sim \mathit{vpat}_k \in R$ of which $\mathit{spat}_k$ 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 $\mathit{spat}_k$. The $\mathit{get}$ function also creates links in the recursive procedure: when a rule $\mathit{spat}_k \sim \mathit{vpat}_k \in R$ is used, it creates a link relating those matched parts as two regions. The $\mathit{get}$ function defined by $R$ is:

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

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

*3.3.3 Put Semantics.* For a consistency relation $S \leftrightarrow V$ defined in our DSL as $R = \{\, \mathit{spat}_k \sim \mathit{vpat}_k \mid 1 \leqslant k \leqslant n \,\}$, its corresponding $\mathit{put}_{SV}$ function has the following type:

$$\mathit{put}_{SV} : \mathit{Tree} \times V \times \mathit{LinkSet} \nrightarrow S\ .$$

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

- In the first case where the root of the view is not within any region of the input links, $\mathit{put}$ selects a rule $\mathit{spat}_k \sim \mathit{vpat}_k \in R$ whose $\mathit{vpat}_k$ matches $v$ — again, our DSL requires that at least one such rule exist for all $v$ — and uses $\mathit{spat}_k$ to build the top portion of the new source:

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

wildcards in $spat_k$ are filled with default values and variables in $spat_k$ are filled with trees recursively constructed from their corresponding parts of the view.

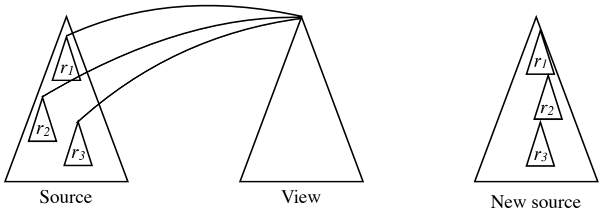
$$\begin{aligned}
& put_{SV}(s, v, ls) = reconstruct(spat'_k, ss) && (5)\\
& \quad \textbf{where } spat_k \sim vpat_k \in R \text{ and } vpat_k \text{ matches } v\\
& \qquad vs = decompose(vpat_k, v)\\
& \qquad ss = \lambda\,(t \in Vars(spat_k)) \rightarrow\\
& \qquad\quad put(s, vs(t), divide(Path(vpat_k,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

The omitted subscript of *put* in (6) is $TypeOf(spat_k, t)\ TypeOf(vpat_k, t)$. Additionally, if there is more than one rule of which *vpat* matches *v*, the rule whose *vpat* is *not* a bare variable pattern is preferred for avoiding infinite recursive calls: if $vpat_k = x$, the size of the input of the recursive call in (6) does not decrease because $vs(t) = v$ and $Path(t, vpat_k) = [\,]$. For example, if both $spat_1 \sim Succ\ x$ and $spat_2 \sim x$ can be selected, the former is preferred.

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

$$\begin{aligned}
& put_{SV}(s, v, ls) = inj_{TypeOf(spat_k) \rightarrow S}(reconstruct(spat'_k, ss)) && (7)\\
& \quad \textbf{where } l = ((spat^{5}, spath), (vpat, vpath)) \in ls\\
& \qquad\qquad \text{such that } vpath = [\,],\ spath \text{ is the shortest}\\
& \qquad spat_k \sim vpat_k \in R \text{ and } isMatch(spat_k, spat)\\
& \qquad spat'_k = fillWildcards(spat_k, spat)\\
& \qquad vs = decompose(vpat_k, v)\\
& \qquad ss = \lambda\,(t \in Vars(spat_k)) \rightarrow\\
& \qquad\quad put(s, vs(t), divide(Path(vpat_k, t), ls \setminus \{\, l \,\})) && (8)
\end{aligned}$$

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



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

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

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

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

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

*chkNoLink* corresponds to the first case of *put* (5). Condition $\mathit{cond}_1$ checks that every link in *ls* is processed in one of the recursive calls. (Specifically, if $\mathit{Vars}\ (\mathit{spat}_k)$ is empty, *ls* in $\mathit{cond}_1$ should also be empty meaning that all the links have already been processed.) $\mathit{cond}_2$ summarises the results of *check* for recursive calls. $\mathit{cond}_3$ guarantees the termination of recursion: When $\mathit{vpat}_k$ is a bare variable pattern, the recursive call in (6) does not decrease the size of any of its arguments; $\mathit{cond}_3$ makes sure that such non-decreasing recursion will not happen in the next round[6] for avoiding infinite recursive calls.

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

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

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

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

$$cond_3 = ls == \left( \{ l \} \cup \bigcup_{t \in Vars(spat_k)} addVPrefix(Path(vpat_k, t), divide(Path(vpat_k, t), ls \setminus \{ l \})) \right)$$

$$cond_4 = \bigwedge_{t \in Vars(spat_k)} check(s, vs(t), divide(Path(vpat_k, t), ls \setminus \{ l \}))$$

*cond₁* makes sure that the link $l$ is valid (Definition 2.4) and *cond₂* further checks that it can be generated from some rule of the consistency relations. *cond₃* and *cond₄* are for recursive calls: the former guarantees that no link will be missed and the latter summarises the results.

*3.3.5 Retentiveness of the DSL.* With the definitions 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 2.6.*

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

## 4 CASE STUDIES

In this section, we demonstrate the usefulness of Retentiveness in practice by presenting two case studies on *resugaring* [Pombrio and Krishnamurthi 2014, 2015] and *code refactoring* [Fowler and Beck 1999]. In both cases, we need to constantly make modifications to the AST[7] and synchronise the CST accordingly. Retentive lenses provide a systematic way for the user to preserve information of interest in the original CST after synchronisation.

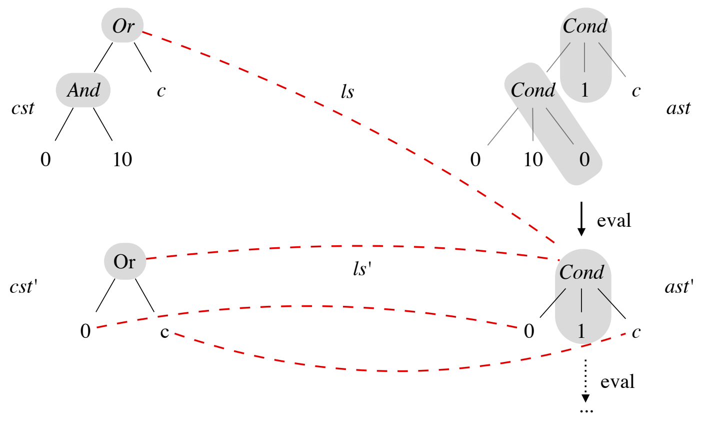
### 4.1 Resugaring

For a programming language, usually the constructs of its surface syntax are richer than those of its abstract syntax (core language). The idea of resugaring 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 non-zero integers represent *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 enrich the AST to incorporate fields for holding tags that mark from which syntactic object an AST construct comes. Conversely, we can also solve the problem while leaving the AST clean by using retentive lenses; we can write consistency relations between the surface syntax and the abstract syntax and passing the generated *put* function proper links for retaining syntactic sugar Below are the code snippet in our DSL and the illustration of resugaring by Retentiveness. Note that syntactic sugar *Or* is retained since Retentiveness requires $fst \cdot ls \subseteq fst \cdot ls'$.

```
OrOp   ↔ Arith                          AndOp ↔ Arith
  Or l r      ~ Cond l (Num 1) r          And l r      ~ Cond l r (Num 0)
  FromAnd t  ~ t                           FromCmp t ~ t
```

[7]Many code refactoring tools make modifications to the CST instead. However, the design of the tools can be simplified if they migrate to modify the AST, as the paper will show.



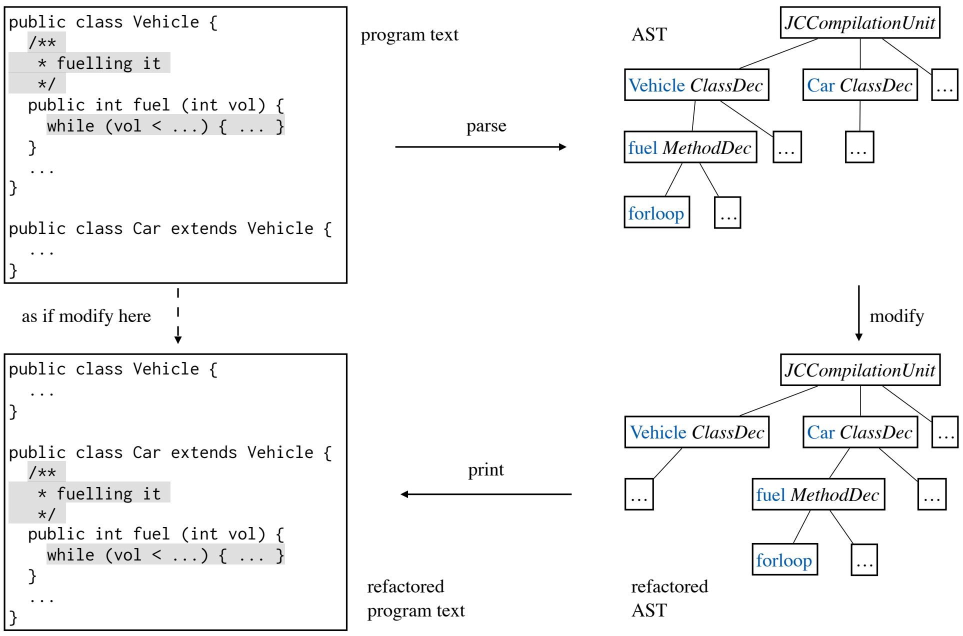
Both their 'tag approach' and our 'link approach' need to identify where an AST construct comes from; however, the links approach has an advantage that it leaves ASTs clean so that we do not need to patch up the compiler to deal with tags.

## 4.2 Refactoring

In this subsection, we first discuss the feasibility of Retentiveness for code refactoring, and in particular take code refactoring for Java 8 for example (Section 4.2.1). Then we introduce a specific refactoring scenario of Java 8 (Section 4.2.2) and use it as the running example throughout the whole subsection: We show how to use our DSL to implement the transformation system between CSTs and ASTs of a small subset of Java 8 (Section 4.2.3), and demonstrate how to use the generated (retentive) lenses to perform code refactoring on *clean* ASTs for the specific refactoring scenario while retaining comments and syntactic sugar in the updated program text on demand (Section 4.2.4).

*4.2.1 Feasibility of Retentiveness for Code Refactoring.* To see whether Retentiveness helps to retain comments and syntactic sugar for real-world code refactoring, we surveyed the standard set of refactoring operations for Java 8 provided by Eclipse Oxygen (with Java Development Tools). We classify the total 23 refactoring operations as the combinations of four basic operations: substitution, insertion, deletion, and movement. For instance, *Extract Method* that 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 sufficient, 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.

*4.2.2 A Specific Refactoring Scenario.* The specific refactoring scenario is illustrated in Figure 7. At first, the user designed a `Vehicle` class and thought that it should have a `fuel` method for all the vehicles. The `fuel` method has a JavaDoc-style comment and contains a `while` loop, which can be seen as syntactic sugar and is converted to a standard `for` loop during parsing. However, when later designing the subclasses, the user realises that bicycles cannot be fuelled, and decides to do the *push-down* code refactoring, removing the `fuel` method from `Vehicle` and pushing the



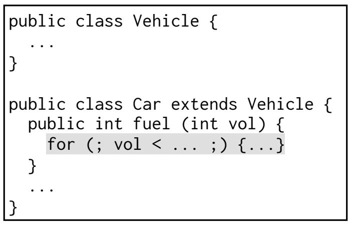
```
public class Vehicle {
  /**
   * fuelling it
   */
  public int fuel (int vol) {
    while (vol < ...) { ... }
  }
  ...
}

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

```
public class Vehicle {
  ...
}

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

Fig. 7. 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 (; vol < ... ;) {...}
  }
  ...
}
```

Fig. 8. An unsatisfactory result after refactoring, losing the comment and *while* syntactic sugar.

method definition down to subclasses `Bus` and `Car` but not `Bicycle`. Instead of directly modifying the program text, most refactoring tools will first parse the program text into its AST, perform code refactoring on the AST, and regenerate new program text. The bottom-left corner of Figure 7 shows the desired program text after refactoring, where we see that the comment associated with *fuel* is also pushed down, and the `while` sugar is kept. However, the preservation of the comment and syntactic sugar in fact does not come for free, as the AST, being a concise and compact representation of the program text, include neither comments nor the form of the original `while` loop. So if the user implements the *parse* and *print* functions as a well-behaved lens, it might only give the user an unsatisfactory result in which much information is lost (the comment and `while` syntactic sugar), like the one shown in Figure 8.

*4.2.3 Implementation in Our DSL.* 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 the 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 for future work.

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 in the source file. This subset of Java 8 has around 80 CST constructs (which represent production rules) and 30 AST constructs; the 70 consistency relations among them generate about 3000 lines of retentive lenses and auxiliary functions (such as the ones for handling interconvertible data types).

Now we define the consistency relations. Since the structure of the consistency relations for the transformation system is roughly similar to the ones in Figure 4, here we only highlight two of them as examples; reviewers can refer to the supplementary material to see the complete program. We see that for the concrete syntax everything is a class declaration (*ClassDecl*), while for the abstract syntax everything is a tree (*JCTree*). As a *ClassDecl* should correspond to a *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[8]

$$
\begin{array}{l}
\mathit{ClassDecl} \leftrightarrow \mathit{JCTree} \\
\quad \mathit{NormalClassDeclaration0}\ \_\ \texttt{"class"}\ n\ \texttt{"extends"}\ \mathit{sup}\ \mathit{body} \sim \\
\quad \mathit{FromJCStatement}\ (\mathit{FromJCClassDecl}\ (\mathit{JCClassDecl}\ N\ n\ (J\ (\mathit{JCIdent}\ \mathit{sup}))\ \mathit{body})) \\
\quad \mathit{NormalClassDeclaration1}\ \_\ \mathit{mdf}\ \texttt{"class"}\ n\ \texttt{"extends"}\ \mathit{sup}\ \mathit{body} \sim \\
\quad \mathit{FromJCStatement}\ (\mathit{FromJCClassDecl}\ (\mathit{JCClassDecl}\ (J\ \mathit{mdf})\ n\ (J\ (\mathit{JCIdent}\ \mathit{sup}))\ \mathit{body})) \\
\quad \ldots
\end{array}
$$

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

Next, we see how to represent `while` loop using the basic `for` loop, as the abstract syntax of a language should be as concise as possible[9]:

$$
\begin{array}{l}
\mathit{Statement} \leftrightarrow \mathit{JCStatement} \\
\quad \mathit{While}\ \texttt{"while"}\ \texttt{"("}\ \mathit{exp}\ \texttt{")"}\ \mathit{stmt} \sim \mathit{JCForLoop}\ \mathit{Nil}\ \mathit{exp}\ \mathit{Nil}\ \mathit{stmt}
\end{array}
$$

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.

*4.2.4 System Running.* Now we use the retentive lenses generated from our DSL to run the refactoring example in Figure 7. We first test two basic cases: *put cst ast ls* and *put cst ast* [ ], where *ast* and *ls* are respectively the view and consistency links generated by *get cst*. We obtain the same *cst* after running *put cst ast ls* and it shows that the generated lenses satisfy Hippocraticness. We obtain *cst′* after running *put cst ast* [ ], in which the comments disappear and the `while` loop

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

becomes a `for` loop. This demonstrates that *put* can create a new source from scratch only depending on the given view. As a special case of Correctness, we also check that $get\ (put\ cst\ ast\ [\,]) == get\ cst$.

Then we modify *ast* to $ast'$, the tree after code refactoring, and create some diagonal links $ls'$ between *cst* and $ast'$ manually. 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 `while` 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: $get\ (put\ cst\ ast'\ ls') == ast'$.

## 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 1 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)        data Arith i = Add (Arith i) (Arith i)
            | Minus  (Expr i) (Term i)                     | Sub (Arith i) (Arith i)
            | FromT  (Term i)                              | Num i
data Term i = Neg    (Term i)
            | Lit    i
            | Paren  (Expr 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 after an update and it may cause problems in many real world applications. To address the issue, we illustrated how to use links to preserve desired data fragments of the original source, and developed a semantic framework of (asymmetric) retentive lenses for region models. Then we presented a small DSL tailored for describing consistency relations between syntax trees; we showed its syntax, semantics, and proved that the pair of *get* and *put* functions generated from any program in the DSL form a retentive lens. We further illustrated the practical use of retentive lenses by giving examples in the field of resugaring and code refactoring. In the related work, we discussed the relations with alignment, origin tracking, and operation-based BX. At the end of the paper, we will briefly discuss our choices of opting for triangular diagrams and Strong Retentiveness (that subsumes Hippocraticness), our thought on (retentive) lens composition, and the feasibility of retaining code styles for refactoring tools.

## 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 2) with the consistency links produced by *get* to compute the input links between the source and the modified view. To be more concrete, for instance, the vertical correspondence can be regarded as 'vertical links' that share the same region pattern, i.e. $(vpath, vpat, vpath')$; when composed with (horizontal) links, $(vpath, vpat, vpath')$ is firstly transformed to $((vpat, vpath), (vpat, vpath'))$ and thus we can reuse the link composition introduced in Section 2.5. Finally, 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, 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.

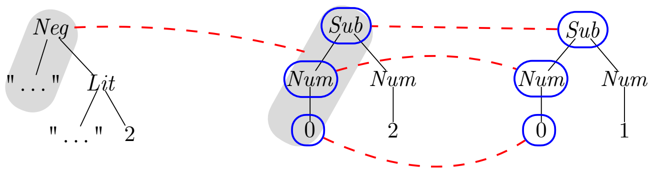
## 6.2 Strong Retentiveness

Through our research into Retentiveness, we also tried a different theory, which we call *Strong Retentiveness* now, that requires that the consistency links generated by *get* should additionally capture all the 'information' of the source and *uniquely identify* it. Strong Retentiveness is appealing in the sense that (we proved that) it subsumes Hippocraticness: the more information we require that the new source have, the more restrictions we impose on the possible forms of the new source; in the extreme case where the input links are consistency links — which capture all the information and can only be satisfied by at most one source — the new source has to be the same as the original one. However, using Strong Retentiveness demands extra effort in practice, for a set of region patterns can never uniquely identify a tree and much more information is required; for instance, $cst_1 = Minus\ \texttt{""}\ (Lit\ 1)\ (Lit\ 2)$ has region patterns $reg_1 = Minus\ \texttt{""}\ \_\ \_$ , $reg_2 = Lit\ 1$, and $reg_3 = Lit\ 2$, which, however, are also satisfied by $cst_2 = Minus\ \texttt{""}\ (Lit\ 2)\ (Lit\ 1)$ in which regions are assembled in a different way. This observation inspires us to generalise region patterns to *properties* in order for holding more information (that can be uniquely identifying) and generalise links connecting $Pattern \times Path$ to links connecting $Property \times Path$ accordingly. We eventually formalised three kinds of properties that are sufficient to capture all the information of a tree (e.g. $cst_1$): *region patterns* (e.g. $reg_1$, $reg_2$, and $reg_3$), *relative positions* between two regions (e.g. $reg_2$ is the first child of $reg_1$ and $reg_3$ is the second child of $reg_1$), and *top* that marks the top of a tree (e.g. $reg_1$ is the top). Worse still, observant readers might have found that, properties need to be named so that they can be referred to by other properties; for instance, the region pattern

*Minus* "" _ _ is named $reg_1$ and is referred to as the top of $cst_1$. This will additionally cause much difficulty in lens composition, as different lenses might assign the same region different names and we need to do 'alpha conversion'. Take everything into consideration, finally, we opted for the simpler and 'weaker' version of Retentiveness.

## 6.3 Rethinking Lens Composition

We defined retentive lens composition (Definition 2.10) and showed that the composition of two retentive lenses is still a retentive lens (Theorem 2.11). In the definition of (retentive) lens composition, we treated link composition as relation composition and the composition of $lens_{AB}$ and $lens_{BC}$ may not be satisfactory if $lens_{AB}$ and $lens_{BC}$ decompose a tree $b$ (of type $B$) in a different way. For example:



In the above figure, $lens_{AB}$ connects the region (pattern) *Neg a* _ with *Sub* (*Num* 0) _ (the grey parts); while $lens_{BC}$ decomposes *Sub* (*Num* 0) _ into three small regions and establishes links for them respectively. Our current link composition simply produces an empty set as the result and we leave the other possibility to 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 $k$.

For our DSL, the lack of composition does not cause problems because of the philosophy of design. Take the scenario of writing a parser for example; there are two main approaches for the user to choose: to use parser combinators (such as Parsec) or to use parser generators (such as Happy). While parser combinators offer the user many small composable components, parser generators usually provide the user with a high-level syntax for describing the grammar of a language using production rules (associated with semantic actions). Then the generated parser is used as a 'standalone black box' and usually will not be composed with some others (although it is still possible to be composed 'externally'). Our DSL is designed to be a 'lens generator' and we have no difficulty in writing bidirectional transformations for the subset of Java 8 in Section 4.2.

## 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*, Anthony Sloane and Uwe Aßmann (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 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), 6:1–25.

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 *Model Driven Engineering Languages and Systems*, Jon Whittle, Tony Clark, and Thomas Kühne (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 304–318.

Eclipse Foundation. 2018. Eclipse documentation - Archived Release. 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, Boston, MA, USA.

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

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.

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

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 (PEPM '14)*. ACM, New York, NY, USA, 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. http://openjdk.java.net/

Hugo Pacheco, Alcino Cunha, and Zhenjiang Hu. 2012. Delta lenses over inductive types. *Electronic Communications of the EASST* 49 (2012), 1–17.

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

## A SUPPLEMENTARY PROOF OF WELL-BEHAVED LENSES ARE RETENTIVE LENSES

Retentiveness.

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

□

## B PROOF OF RETENTIVENESS PRESERVATION

In this section, we show the proof of Theorem 2.11 with the help of Figure 3 and the definition of retentive lens composition (Definition 2.10).

Hippocraticness Preservation. We prove that the composite lens satisfies Hippocraticness with the help of Figure 3 and the definition of retentive lens composition (Definition 2.10).
Let $get_{AC}\ (a) = (c, ls_{ac})$. We prove $put_{AC}\ (a, c', ls_{ac'}) = a' = a$. In this case, $c' = c$ and $ls_{ac'} = ls_{ac}$.

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

□

Correctness Preservation. We prove that the composite lens satisfies Correctness with the help of Figure 3 and the definition of retentive lens composition (Definition 2.10).
Let $a' = put_{AC}\ (a, c', ls_{ac'})$, we prove $fst\ (get_{AC}\ (a')) = c'$.

$$fst\ (get_{AC}\ (a'))$$

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

□

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

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

$$
\begin{aligned}
\text{RDOM}(f \cdot R) &= \text{RDOM}(R) \quad \textit{if}\ \text{LDOM}(R) \subseteq \text{RDOM}(f)\textit{, and} \\
\text{LDOM}(R \cdot f) &= \text{LDOM}(R) \quad \textit{if}\ \text{RDOM}(R) \subseteq \text{LDOM}(f)\ .
\end{aligned}
$$

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

Now, we present the main proof:

$$
\begin{aligned}
&\quad fst \cdot ls_{ac} \\
&=\{ R = R \cdot id_{\text{RDOM}(R)} \} \\
&\quad fst \cdot ls_{ac'} \cdot id_{\text{RDOM}(ls_{ac'})} \\
&\subseteq\{ id_{\text{RDOM}(ls_{ac'})} \subseteq ls_{c'b'} \cdot ls^{\circ}_{c'b'} \text{ by sub-proof-1} \} \\
&\quad fst \cdot ls_{ac'} \cdot (ls_{c'b'} \cdot ls^{\circ}_{c'b'}) \\
&=\{ \text{Relation composition is associative} \} \\
&\quad fst \cdot (ls_{ac'} \cdot ls_{c'b'}) \cdot ls^{\circ}_{c'b'} \\
&=\{ ls_{ab'} = ls_{ac'} \cdot ls_{c'b'} \text{ (⑥ in Figure 3)} \} \\
&\quad fst \cdot ls_{ab'} \cdot ls^{\circ}_{c'b'} \\
&\subseteq\{ \text{Retentiveness of } lens_{AB} \text{ and } ls^{\circ}_{c'b'} = ls_{b'c'} \} \\
&\quad fst \cdot ls_{a'b'} \cdot ls_{b'c'} \\
&=\{ ls_{a'c'} = ls_{a'b'} \cdot ls_{b'c'} \} \\
&\quad fst \cdot ls_{a'c'}\ .
\end{aligned}
$$

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

| | | |
|---|---|---|
| 1. | $\textsc{ldom}(ls_{c'b'}) = \textsc{rdom}(ls_{b'c'})$ | definition of relations |
| 2. | $fst \cdot ls_{bc'} \subseteq fst \cdot ls_{b'c'}$ | Retentiveness of $lens_{BC}$ |
| 3. | $\textsc{rdom}(fst \cdot ls_{bc'}) \subseteq \textsc{rdom}(fst \cdot ls_{b'c'})$ | 2 and definition of relation inclusion |
| 4. | $\textsc{rdom}(ls_{bc'}) \subseteq \textsc{rdom}(ls_{b'c'})$ | 3 and Lemma B.1 |
| 5. | $ls_{bc'} = ((ls_{ac'})^{\circ} \cdot ls_{ab})^{\circ} = (ls_{c'a} \cdot ls_{ab})^{\circ}$ | ③ in Figure 3 |
| 6. | $\textsc{rdom}(ls_{bc'}) = \textsc{rdom}(ls_{c'a} \cdot ls_{ab})^{\circ}$ | 5 |
| 7. | $\textsc{rdom}(ls_{c'a} \cdot ls_{ab})^{\circ} = \textsc{ldom}(ls_{c'a} \cdot ls_{ab})$ | definition of converse relation |
| 8. | $\textsc{ldom}(ls_{c'a} \cdot ls_{ab}) \subseteq \textsc{ldom}(ls_{c'a})$ | definition of relation composition |
| 9. | $\textsc{ldom}(ls_{c'a}) = \textsc{rdom}(ls_{ac'})$ | definition of converse relation |
| 10. | $\textsc{rdom}(ls_{bc'}) = \textsc{rdom}(ls_{ac'})$ | 6, 7, 8, and 9 |
| 11. | $\textsc{rdom}(ls_{ac'}) \subseteq \textsc{ldom}(ls_{c'b'})$ | 10, 4, and 1 |

□

## C PROOF OF RETENTIVENESS OF THE DSL

In this section, we prove that the *get* and *put* semantics given in Section 3.3 does satisfy the three properties (Definition 2.6) of a retentive lens. Most of the proofs are proved by induction on the size of the trees.

Lemma C.1. *The get function described in Section 3.3.2 is total.*

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

Lemma C.2. *For a pair of get and put described in Section 3.3.3 and any $s : S$, $check(s, get(s)) = \top$.*

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

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

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

Lemma C.3. *(Focusing) If $sel(s, p) = s'$ and for any $((\_, spath), (\_, \_)) \in ls$, $p$ is a prefix of spath, then*

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

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

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

Theorem C.4. *(Hippocraticness of the DSL) For any $s$ of type $S$,*

$$put(s, get(s))^{10} = s\ .$$

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

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

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

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

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

where

$$\begin{aligned} & spat_k{}' \\ & = \{ \ spat \text{ in } (7) \text{ is } eraseVars(fillWildcards(spat_k, s)) \ \} \\ & fillWildcards(spat_k, eraseVars(fillWildcards(spat_k, s))) \\ & = \{ \text{ See Figure 9 } \} \\ & fillWildcards(spat_k, s)\ . \end{aligned}$$

and

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

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

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

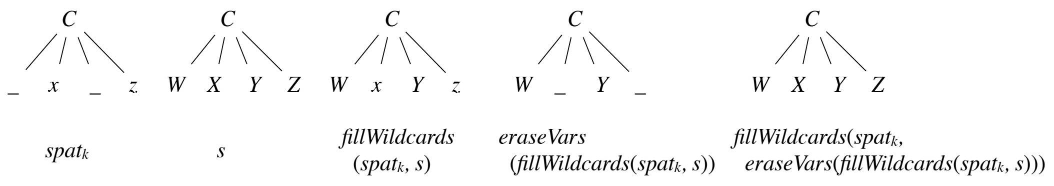
By Lemma C.3, we have

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

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

$$\begin{aligned} & put(s, get(s)) \\ = & \{ \ Equation(9) \ \} \end{aligned}$$

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



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

Fig. 9. A property regarding *fillWildcards*.

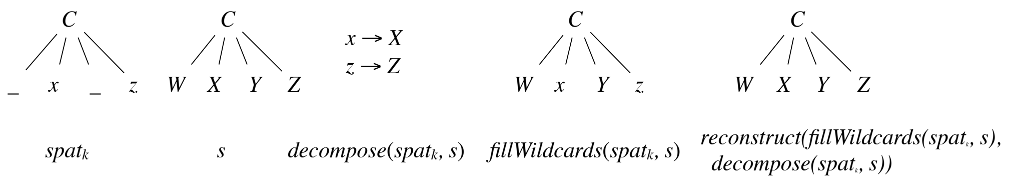


Fig. 10. A property regarding Reconstruct-Decompose.

$$
\begin{aligned}
& inj_{SS}(reconstruct(spat_k{}', ss)) \\
= & inj_{SS}(reconstruct(fillWildcards(spat_k, s), decompose(spat_k, s))) \\
= & \{ \text{ See Figure 10 } \} \\
& inj_{SS}(s) \\
= & s\ .
\end{aligned}
$$

This completes the proof of Hippocraticness. □

Theorem C.5. *(Correctness of the DSL) For any* $(s, v, ls)$ *that makes* $check(s, v, ls) = \top$, $get(put(s, v, ls)) = (v, ls')$, *for some* $ls'$.

Proof of Correctness. We prove Correctness by induction on the size of $(v, ls)$. The proofs of the two cases of *put* are quite similar, and therefore we only present the first one, in which $put(s, v, ls)$ falls into the first case of *put*: i.e. $put(s, v, ls) = reconstruct(fillWildcardsWithDefaults(spat_k), ss)$. Then

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

where $spat_k \sim vpat_k \in R$, $isMatch(vpat_k, v) = \top$, and

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

Now expanding the definition of *get*, because of the disjointness of source patterns, the same $spat_k \sim vpat_k \in R$ will be select again. Thus

$$get(put(s, v, ls)) = (reconstruct(vpat_k, fst \circ vls), \cdots)$$

where

$$
\begin{aligned}
vls &= get \circ decompose(spat_k, put(s, v, ls)) \\
&= get \circ decompose(spat_k, reconstruct(fillWildcardsWithDefaults(spat_k), ss)) \\
&= \{ \text{ See Figure 11 } \}
\end{aligned}
$$

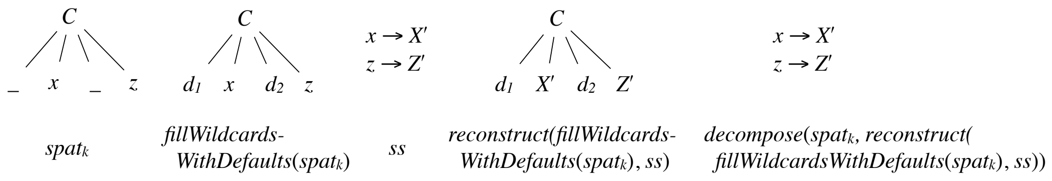

C
_ x _ z
spat_k
C
d1 x d2 z
fillWildcards-WithDefaults(spat_k)
x → X'
z → Z'
ss
C
d1 X' d2 Z'
reconstruct(fillWildcards-WithDefaults(spat_k), ss)
x → X'
z → Z'
decompose(spat_k, reconstruct(fillWildcardsWithDefaults(spat_k), ss))

Fig. 11. A property regarding Decompose-Reconstruct.

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

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

$$vls = \lambda(t \in Vars(spat_k)) \longrightarrow (decompose(vpat_k, v)(t), \cdots)$$

Thus $get(put(s, v, ls)) = (reconstruct(vpat_k, decompose(vpat_k, v)), \cdots) = (v, \cdots)$, which completes the proof of Correctness. □

Theorem C.6. *(Retentiveness of the DSL) For any* $(s, v, ls)$ *that* $check(s, v, ls) = \top$, $get(put(s, v, ls)) = (v', ls')$, *for some* $v'$ *and* $ls'$ *such that*

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

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

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

$$\begin{aligned}&get(put(s, v, ls))\\ =&\{\ \text{Definition of } put\ \}\\ &get(inj_{TypeOf(spat_k)\to S}(s'))\\ =&\{\ get(inj(s)) = get(s) \text{ as shown in (Section 3.2)}\ \}\\ &get(s')\end{aligned}$$

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

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

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

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

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

$$links = \{\, ((a, Path(spat_k, t) +\!\!+ b), (c, Path(vpat_k, t) +\!\!+ d)) \qquad (10)$$

$$\mid t \in \mathit{Vars}(\mathit{vpat}_k), ((a,b),\ (c,d)) \in \mathit{snd}(\mathit{vls}(t))\ \}\ \text{,and}$$

$$\begin{aligned}
& \mathit{vls}(t) \\
={}& \{\ \text{Unfolding } \mathit{vls}\ \} \\
& (\mathit{get} \circ \mathit{decompose}(\mathit{spat}_k, s'))(t) \\
={}& \{\ \text{Unfolding } s'\ \} \\
& (\mathit{get} \circ \mathit{decompose}(\mathit{spat}_k, \mathit{reconstruct}(\mathit{fillWildcards}(\mathit{spat}_k, \mathit{spat}), \mathit{ss})))(t) \\
={}& \{\ \text{Similar to the case shown in Figure 11}\ \} \\
& (\mathit{get} \circ \mathit{ss})(t) \\
={}& \{\ \text{Definition of } \mathit{ss}\ \} \\
& \mathit{get}(\mathit{put}(s, \mathit{decompose}(\mathit{vpat}_k, v)(t), \mathit{divide}(\mathit{Path}(\mathit{vpat}_k, t), \mathit{ls} \setminus \{\ l\ \}))) \ .
\end{aligned}$$

For $l_{root}$, we have

$$\begin{aligned}
& \mathit{eraseVars}(\mathit{fillWildcards}(\mathit{spat}_k, s')) \\
={}& \{\ \text{Unfolding } s'\ \} \\
& \mathit{eraseVars}(\mathit{fillWildcards}(\mathit{spat}_k, \mathit{reconstruct}(\mathit{fillWildcards}(\mathit{spat}_k, \mathit{spat}), \mathit{ss}))) \\
={}& \{\ \text{See Figure 12}\ \} \\
& \mathit{eraseVars}(\mathit{fillWildcards}(\mathit{spat}_k, \mathit{spat})) \\
={}& \{\ \text{By } \mathit{cond}_2 \text{ in } \mathit{chkWithLink}\ \} \\
& \mathit{spat}
\end{aligned}$$

Use the first clause of $\mathit{cond}_2$, we have $\mathit{vpat} = \mathit{eraseVars}(\mathit{vpat}_k)$. Thus

$$l_{root} = ((\mathit{eraseVars}(\mathit{fillWildcards}(\mathit{spat}_k, s')), []) \ , \ (\mathit{eraseVars}(\mathit{vpat}_k), [])) = ((\mathit{spat}, [\,]),\ (\mathit{vpat}, [\,])) \ ,$$

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

For the links in $\mathit{ls} \setminus \{\ l\ \}$, we show that they are preserved in $\mathit{links}$ (10) above. By $\mathit{cond}_3$ in *chkWithLink*, for every link $m \in \mathit{ls} \setminus \{\ l\ \}$, there is some $t_m$ in $\mathit{Vars}(\mathit{spat}_k)$ such that

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

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

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

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

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

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

Corollary C.7. *Let put′ = put with its domain intersected with S × V × LinkSet, get and put′ form a retentive lens as in Definition 2.6 since they satisfy Hippocraticness (1), Correctness (2) and Retentiveness (3).*

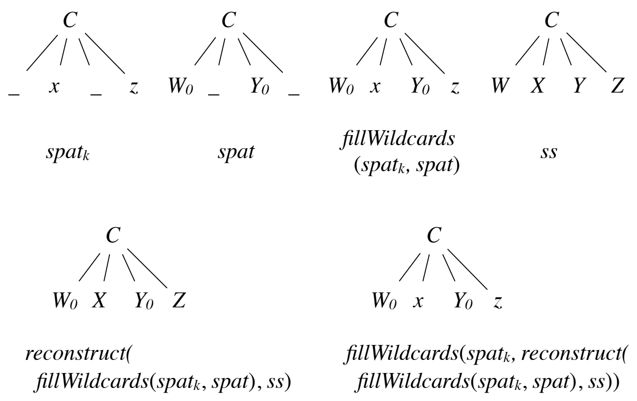

C
_ x _ z
C
W0 _ Y0 _
C
W0 x Y0 z
C
W X Y Z
spatk
spat
fillWildcards (spatk, spat)
ss
C
W0 X Y0 Z
C
W0 x Y0 z
reconstruct( fillWildcards(spatk, spat), ss)
fillWildcards(spatk, reconstruct( fillWildcards(spatk, spat), ss))

Fig. 12. Another property regarding *fillWildcards*.