Retentive Lenses

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

[1] National Institute of Informatics, Japan
[2] Peking University, China
[3] The Graduate University for Advanced Studies (SOKENDAI), Japan
zhu@nii.ac.jp, yzx@nii.ac.jp, hsiang-shang@nii.ac.jp, huzj@pku.edu.cn

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

Keywords: lenses, bidirectional programming, domain-specific languages

1 Introduction

We often need to write pairs of transformations to synchronise data. Typical examples include view querying and updating in relational databases [2] for keeping a database and its view in sync, text file format conversion [16] (e.g. between Markdown and HTML) for keeping their content and common formatting in sync, and parsers and printers as front ends of compilers [22] for keeping program text and its abstract representation in sync. Asymmetric *lenses* [11] provide a framework for modelling such pairs of programs and discussing what laws they should satisfy; among such laws, two *well-behavedness* laws (explained below) play a fundamental role. Based on lenses, various bidirectional programming languages and systems (Sect. 7) 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 [23]. Lenses are used to synchronise two pieces of data respectively of types S and V, where S contains more information and is called the *source* type, and V contains less information and is called the *view* type. Here, being synchronised means that when one piece of data is changed, the other piece of data should also be changed such that consistency is *restored* among them, i.e. a *consistency relation* R defined on S and V is satisfied. Since S contains more information than V, we expect that there is a function $get : S \to V$ that extracts a consistent view from a source, and this *get* function serves as a consistency restorer in the source-to-view direction: if the source is changed, to restore consistency it suffices to use *get* to recompute a new view. This *get* function should coincide with R extensionally [23]—that is, $s : S$ and $v : V$ are related by R if and only if $get(s) = v$. (Therefore it is only sensible to consider functional consistency relations in the asymmetric setting.) Consistency restoration in the other direction is performed by another function $put : S \times V \to S$, which produces an updated source that is consistent with the input view and can retain some information of the input source. Well-behavedness consists of two laws regarding the restoration behaviour of *put* with respect to *get* (i.e. the consistency relation R):

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

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

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

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

Given this `get`[1], we can define $\texttt{put}_1$ and $\texttt{put}_2$, both of which are well-behaved with this `get` but have rather different behaviour: $\texttt{put}_1$ simply replaces the integer of the source tuple with the view, while $\texttt{put}_2$ also sets the string empty when the source tuple is not consistent with the view.

[1] In this paper, we use Haskell notations to write functions, and concrete examples are always typeset in `typewriter` font.

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

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

From another perspective, $\texttt{put}_1$ retains the string from the old source when performing the update, while $\texttt{put}_2$ chooses to discard that string—which is not desired but 'perfectly legal', for the string does not contribute to the consistency relation. In fact, unexpected behaviour of this kind of well-behaved lenses could even lead to disaster in practice. For instance, relational databases can be thought of as tables consisting of rows of tuples, and well-behaved lenses used for maintaining a database and its view may erase important data after an update, as long as the data does not contribute to the consistency relation (in most cases this is because the data is simply not in the view). This fact seems fatal, as asymmetric lenses have been considered a satisfactory solution to the longstanding view update problem (stated at the beginning of Foster et al.'s seminal paper [11]).

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

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

The main contributions of the paper are as follows:

- We develop a formal definition of retentive lenses for tree-shaped data (Sect. 3).
- We present a domain-specific language (DSL) for writing tree synchronisers and prove that any program written in our DSL gives rise to a retentive lens (Sect. 4).
- We demonstrate the usefulness of retentive lenses in practice by presenting case studies on code refactoring, resugaring, and XML synchronisation (Sect. 6), with the help of several view editing operations that also update the links between the view and the original source (Sect. 5).

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

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

We will start from a high-level sketch of what retentive lenses do (Sect. 2), and after presenting the technical contents, we will discuss related work (Sect. 7) regarding various alignment strategies for lenses, provenance and origin between two pieces of data, and operational-based bidirectional transformations, before concluding the paper (Sect. 8).

2 A Sketch of Retentiveness

We will use the synchronisation of concrete and abstract representations of arithmetic expressions as the running example throughout the paper. The representations are defined in Fig. 1 (in Haskell). The concrete representation is either an expression of type `Expr`, containing additions and subtractions; or a term of type `Term`, including numbers, negated terms, and expressions in parentheses. Moreover, all the constructors have an annotation field of type `Annot` mocking up data that exist solely in the concrete representation like code comments and spaces. The two concrete types `Expr` and `Term` coalesce into the abstract representation type `Arith`, which does not include annotations, explicit parentheses, and negations—negations are considered *syntactic sugar* and represented in the AST by `Sub`.

As mentioned in Sect. 1, the core idea of Retentiveness is to use links to relate parts of the source and view. For data of algebraic data types (which we call 'trees' or 'terms'), a straightforward interpretation of a 'part' is a subtree of the data. But it is too restrictive in most cases, and a more useful interpretation of a 'part' is a *region* of a tree, i.e. a partial subtree. Partial trees are trees where some subtrees can be missing. We will describe the content of a partial tree with a pattern that contains wildcards at the positions of missing subtrees. In Fig. 2, all grey areas are examples of regions; the topmost region in `cst` is located at the root of the whole tree, and its content has the 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.

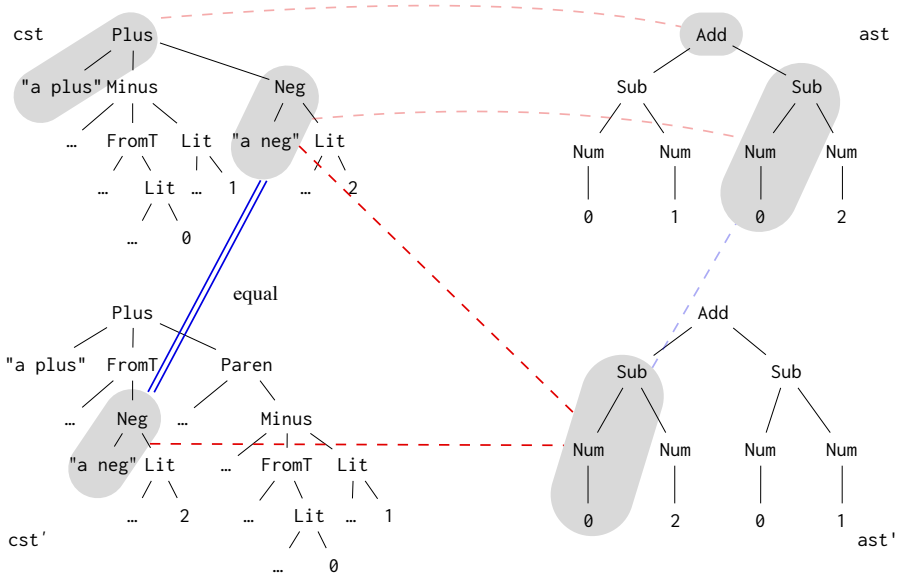


Fig. 2. Regions, links, and the triangular guarantee.

Having broken up source and view trees into regions, we can put in *links* to record the correspondences between source and view regions. In Fig. 2, for example, the light red dashed lines between the source `cst` and the view `ast = getE cst` represent two possible links. The topmost region of pattern `Plus "a plus" _ _` in `cst` corresponds to the topmost region of pattern `Add _ _` in `ast`, and the region of pattern `Neg "a neg" _` in the right subtree of `cst` corresponds to the region of pattern `Sub (Num 0) _` in `ast`. The *get* function of a retentive lens will be responsible for producing an initial set of links between a source and its view.

As the view is modified, the links between the source and view should also be modified to reflect the latest correspondences between regions. For example, in Fig. 2, if we change `ast` to `ast'` by swapping the two subtrees under `Add`, then there should be a new link (among others) recording the fact that the `Neg "a neg" _` region and the `Sub (Num 0) _` region are still related. We will describe a way of computing new links from old ones in Sect. 5.

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

3 Formal Definitions

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

3.1 Retentive Lenses

We start with some notations. Relations from set A to set B are subsets of $A \times B$, and we denote the type of these relations by $A \sim B$. Given a relation $r : A \sim B$, define its *converse* $r^\circ : B \sim A$ by $r^\circ = \{\, (b, a) \mid (a, b) \in r \,\}$, its *left domain* by $\textsc{ldom}(r) = \{\, a \in A \mid \exists b.\ (a, b) \in r \,\}$, and its *right domain* by $\textsc{rdom}(r) = \textsc{ldom}(r^\circ)$. The composition $r \cdot s : A \sim C$ of two relations $r : A \sim B$ and $s : B \sim C$ is defined as usual by $r \cdot s = \{\, (a, c) \mid \exists b.\ (a, b) \in r \wedge (b, c) \in s \,\}$. The type of partial functions from A to B is denoted by $A \nrightarrow B$. The *domain* $\textsc{dom}(f)$ of a function $f : A \nrightarrow B$ is the subset of A on which f is defined; when f is total, i.e. $\textsc{dom}(f) = A$, we write $f : A \to B$. We will allow functions to be implicitly lifted to relations: a function $f : A \nrightarrow B$ also denotes a relation $f : B \sim A$ such that $(f\,x, x) \in f$ for all $x \in \textsc{dom}(f)$[1].

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

We define a collection of links between two trees as a relation of type $\mathit{Region} \sim \mathit{Region}$, where $\mathit{Region} = \mathit{Pattern} \times \mathit{Path}$: a region is identified by a path leading to a subtree and a pattern describing the part of the subtree included in the region. Briefly, a link is a pair of regions, and a collection of links is a relation between regions of two trees. For brevity we will write *Links* for $\mathit{Region} \sim \mathit{Region}$.

An arbitrary collection of links may not make sense for a given pair of trees though—a region mentioned by some link may not exist in the trees at all. We should therefore characterise when a collection of links is valid for two trees.

[1] This flipping of domain and codomain (from $A \nrightarrow B$ to $B \sim A$) makes function composition compatible with relation composition: a function composition $g \circ f$ lifted to a relation is the same as $g \cdot f$, i.e. the composition of g and f as relations.

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

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

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

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

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

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

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

satisfying

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

$$put\ (s, v, ls) = s \ ; \tag{1}$$

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

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

- Retentiveness:

$$fst \cdot ls \subseteq fst \cdot ls' \tag{3}$$

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

Modulo the handling of links, Hippocraticness and Correctness remain the same as their original forms (in the definition of well-behaved lenses). Retentiveness further states that the input links ls must be preserved, except for the location of source regions (i.e. $\text{RDOM}(snd \cdot ls)$ in the compact relational notation). The region patterns (data) and the location of the view region, which are $fst \cdot ls$ in the relational notation, must be exactly the same. Retentiveness formalises the triangular guarantee in a compact way, and we can expand it pointwise to see that it indeed specialises to the triangular guarantee.

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

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

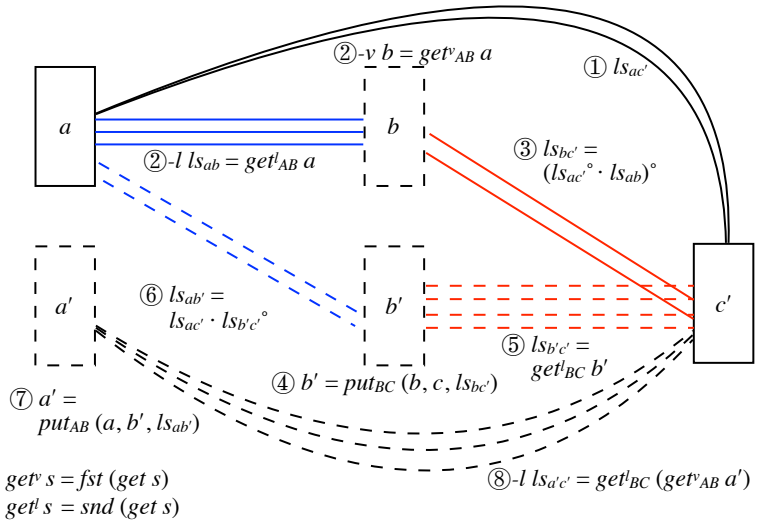


Fig. 3. The *put* behaviour of a composite retentive lens, divided into steps ① to ⑧. Step ⑧ produces consistency links for showing the triangular guarantee.

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

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

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

where

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

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

3.2 Composition of Retentive Lenses

It is standard to provide a composition operator for composing large lenses from small ones. Here we discuss this operator for retentive lenses, which basically follows the definition of composition for well-behaved lenses, except that we need to deal with links carefully. Below we use $lens_{AB}$ to denote a retentive lens that synchronises trees of sets A and B, get_{AB} and put_{AB} the *get* and *put* functions of the lens, l_{ab} a link between tree a (of set A) and tree b (of set B), and ls_{ab} a collection of links between a and b.

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

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

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

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

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

4 A DSL for Retentive Bidirectional Tree Transformations

The definition of retentive lenses is somewhat complex, but we can ease the task of constructing retentive lenses with a declarative domain-specific language. Our DSL is designed to describe consistency relations between algebraic data types, and from each consistency relation defined in the DSL, we can obtain a pair of *get* and *put* functions forming a retentive lens. Below we will give an overview of the DSL and how retentive lenses are derived from programs in the DSL using the arithmetic expression example (Sect. 4.1), the syntax (Sect. 4.2) and semantics (Sect. 4.3) of the DSL, and finally the theorem stating that the generated lenses satisfy the required laws (Theorem 2). Due to limited space, we can only provide the proof of the theorem in the appendix (Sect. D.2), but the essence is given in the last part of Sect. 4.1. Also some more programming examples other than syntax tree synchronisation can be found in the appendix (Sect. B).

4.1 Overview of the DSL

Recall the arithmetic expression example (Fig. 1). In our DSL, we define data types in Haskell syntax and describe consistency relations between them that

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

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

bear some similarity to *get* functions. For example, the data type definitions for `Expr` and `Term` written in our DSL remain the same as those in Fig. 1, and the consistency relations between them (i.e. `getE` and `getT` in Fig. 1) are expressed as the ones in Fig. 4. Here we specify 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 (i.e. source side) and right-hand side (i.e. view side) are consistent, then the larger pair of trees constructed from these subtrees are also consistent. Take

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

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

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

Each consistency relation is translated to a pair of *get* and *put* functions defined by case analysis generated from the inductive rules. Detail of the translation will be given in Sect. 4.3, but the idea behind the translation is a fairly simple one which establishes Retentiveness by construction. For *get*, the rules themselves are already close to function definitions by pattern matching, so what we need to add is only the computation of output links. For *put*, we use the rules backwards and define a function that turns the regions of an input view into the regions of the new source, reusing regions of the old source wherever required: when there is an input link connected to the current view region, *put* grabs the source region at the other end of the link in the old source; otherwise, *put* creates a new source region as described by the left-hand side of an appropriate rule.

For example, suppose that the *get* and *put* functions generated from the consistency relation `Expr <---> Arith` are named `getEA` and `putEA` respectively. The inductive rule `Plus _ x y ~ Add x y` generates the definition for `getEA s` when `s` matches `Plus _ x y`: `getEA (Plus _ x y)` computes a view recursively in the same way as `getE` in Fig. 1; furthermore, it produces a new link between the top regions `Plus` and `Add`, and keeps the links produced by the recursive calls `getEA x` and `getEA y`. In the *put* direction, the inductive rule `Plus _ x y ~ Add x y` leads to a case `putEA s (Add x y) ls`, under which there are two subcases: if there is any link in `ls` that is connected to the `Add` region at the top of the view, `putEA` grabs the region at the other end of the link in the old source and tries to use it as the top part of the new source; if such a link does not exist, `putEA` uses a `Plus`

with a default annotation as a substitute for the top part of the new source. In either case, the subtrees of the new source at the positions marked by `x` and `y` are computed recursively from the view subtrees `x` and `y`.

While the core idea is simple, there are cases in which the translated functions do not constitute valid retentive lenses, and the crux of Theorem 2 is finding suitable ways of computation or reasonable conditions to circumvent all such cases (some of which are rather subtle). The following cases should give a good idea of what is involved in the correctness of the theorem.

I. *The translated functions may not be well-defined.* For example, in the *get* direction, an arbitrary set of rules may assign zero or more than one view to a source, making *get* partial (which, though allowed by the definition, we want to avoid) or ill-defined, and we will impose (fairly standard) restrictions on patterns to preclude such rules. These restrictions are sufficient to guarantee that exactly one rule is applicable in the *get* direction but not in the *put* direction, in which we need to carefully choose a rule among the applicable ones or risk non-termination (e.g. producing an infinite number of parentheses by alternating between the `Paren` and `FromT` rules).

II. *A region grabbed by put from the old source may not have the right type.* For example, if *put* is run on `cst`, `ast'`, and the link between them in Fig. 2, it has to grab the source region `Reg "a neg" _`, which has type `Term`, and install it as the second argument of `Plus`, which has to be of type `Expr`. In this case there is a way out since we can convert a `Term` to an `Expr` by wrapping the `Term` in the `FromT` constructor. We will formulate conditions under which such conversions are needed and can be synthesised automatically.

III. *Hippocraticness may be accidentally invalidated by put.* Suppose that there is another parenthesis constructor `Brac` that has the same type as `Paren` and for which a similar rule `Brac _ e ~ e` is supplied. Given a source that starts with `Brac "" (Paren "" ...)`, *get* will produce two links (among others) relating both the `Brac` and `Paren` regions with the empty region at the top of the view. If *put* is immediately invoked on the same source, view, and links, it may choose to process the link attached to the `Paren` region first rather than the one attached to the `Brac` region, so that the new source starts with `Paren "" (Brac "" ...)`, invalidating Hippocraticness. Therefore *put* has to carefully process the links in the right order for Hippocraticness to hold.

IV. *Retentiveness may be invalidated if put does not correctly reject invalid input links.* Unlike *get*, which can easily be made total, *put* is inherently partial since input links may well be invalid and make Retentiveness impossible to hold. For example, if there is an input link relating a `Neg` region and an `Add` region, then it is impossible for *put* to produce a result that satisfies Retentiveness since *get* does not produce a link of this form. Instead, *put* must correctly reject invalid links for Retentiveness to hold. Apart from checking that input links have the right forms as specified by the rules, there are more subtle cases where the view regions referred to by a set of input links are overlapping—for example, in a view starting with `Sub (Num 0) ...` there can be links referring to both the `Sub _ _` region and the `Sub (Num 0) _`

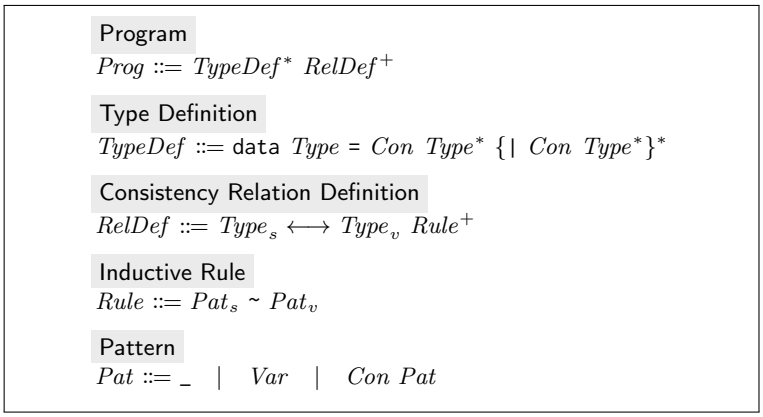


Fig. 5. Syntax of the DSL.

region at the top. Our *get* cannot produce overlapping view regions, and therefore such input links must be detected and rejected as well.

In the rest of this section we will describe the DSL in more detail.

4.2 Syntax

The syntax of our DSL is summarised in Fig. 5, where nonterminals are in *italic*; terminals are typeset in `typewriter` font; {} is for grouping; $^?$, *, and $^+$ represent zero-or-one occurrence, zero-or-more occurrence, and one-or-more occurrence respectively, and *Type*, *Con*, and *Var* are syntactic categories (whose definitions are omitted) for the names of types, constructors, and variables respectively. We sometimes additionally attach a subscript s or v to a symbol to mean that the symbol is related to sources or views. A program consists of two parts: data types definitions and consistency relations between these data types. We adopt the Haskell syntax for data type definitions—a data type is defined by specifying a set of data constructors and their argument types. As for the definitions of consistency relations, each of them starts with $\mathit{Type}_s \leftrightarrow \mathit{Type}_v$, declaring the source and view types for the relation. The body of each consistency relation is a list of inductive rules, each of which defined by a pair of source and view patterns $\mathit{Pat}_s \sim \mathit{Pat}_v$, where a pattern can include wildcards, variables, and constructors.

Syntactic Restrictions We impose some syntactic restrictions to guarantee that programs in our DSL indeed give rise to retentive lenses (Theorem 2).

On *patterns*, we require (i) pattern coverage: for any consistency relation $S \leftrightarrow V = \{\, p_i \sim q_i \mid 1 \leqslant i \leqslant n \,\}$ defined in a program, $\{\, p_i \,\}$ should cover all possible cases of type S, and $\{\, q_i \,\}$ should cover all cases of type V. We also require (ii) source pattern disjointness: any distinct p_i and p_j should not be matched by the same tree. Finally, (iii) a bare variable pattern is not allowed on the source side (e.g. $x \sim D\ x$), and (iv) wildcards are not allowed on the view side

(e.g. $C\ x \sim D\ \_\ x$), and (v) the source side and the view side must use exactly the same set of variables. These conditions ensure that *get* is total and well-defined (ruling out Case I in Sect. 4.1).

To state the next requirement we need a definition: two data types S_1 and S_2 defined in a program are *interchangeable* in data type S exactly when (i) there are some data type V' and V for which consistency relations $S_1 \leftrightarrow V'$, $S_2 \leftrightarrow V'$ and $S \leftrightarrow V$ are defined in the program, and (ii) S may have subterms of type S_1 and S_2, and V may have subterms of type V'. If S_1 and S_2 are interchangeable, then Case II (Sect. 4.1) may happen: when doing *put* on S and V there might be input links dictating that values of type S_2 should be retained in a context where values of type S_1 are expected, or vice versa. When this happens, we need two-way conversions between S_1 and S_2.

We choose a simple way to ensure the existence of conversions:for any interchangeable types S_1 and S_2 with $S_1 \leftrightarrow V'$ and $S_2 \leftrightarrow V'$ defined, we require that there exists a sequence of data types in the program

$$S_1 = T_1,\ T_2,\ \cdots,\ T_{n-1},\ T_n = S_2$$

with $n >= 2$ such that for any $1 \leqslant i < n$, consistency relation $T_i \leftrightarrow V'$ is defined and has a rule $Pat_i \sim x$ whose source pattern Pat_i contains exactly one variable, and its type in Pat_i is T_{i+1} (we also require such a sequence with the roles of S_1 and S_2 switched). With rule $Pat_i \sim x$, we immediately get a function $t_i : T_{i+1} \to T_i$ contructing a T_i from a term v of T_{i+1} by substituting v for x in Pat_i (and filling wildcard positions with default values). Then we have the needed conversion function:

$$inj_{S_2 \to S_1 @ V'} = t_{n-1} \circ \cdots \circ t_2 \circ t_1 \tag{4}$$

(and similary $inj_{S_1 \to S_2 @ V'}$). For example, `FromT _ t ~ t` gives rise to a function

$$\mathtt{inj}_{\mathtt{Term} \to \mathtt{Expr@Arith}}\ x\ =\ \mathtt{FromT}\ \mathtt{""}\ x$$

and it can be used to convert `Term` to `Expr` whenever needed when doing *put* with view type `Arith`.

4.3 Semantics

We give the semantics of our DSL in terms of a translation into 'pseudo-Haskell', where we may replace chunks of Haskell code with natural language descriptions to improve readability. As in Sect. 3.1, let *Tree* be the set of values of any algebraic data type, and *Pattern* the set of all patterns. For a pattern $p \in Pattern$, $Vars\ p$ denotes the set of variables in p. For each $v \in Vars\ p$, $TypeOf\ (p, v)$ is (the set of all values of) the type of v in pattern p, and $path\ (p, v)$ is the path of variable v in pattern p. We use the following functions (two of which are dependently typed) to manipulate patterns:

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

$$\begin{aligned} \mathit{fillWildcardsWD} : &\quad \mathit{Pattern} \to \mathit{Pattern} \\ \mathit{eraseVars} : &\quad \mathit{Pattern} \to \mathit{Pattern} \ . \end{aligned}$$

Given a pattern p and a tree t, $\mathit{isMatch}\ (p, t)$ tests whether t matches p. If the match succeeds, $\mathit{decompose}\ (p, t)$ returns a function mapping every variable in p to its corresponding matched subtree of t. Conversely, $\mathit{reconstruct}\ (p, f)$ produces a tree matching p by replacing every occurrence of $v \in \mathit{Vars}\ p$ in p with $f\ v$, provided that p does not contain any wildcard. To remove wildcards, we can use $\mathit{fillWildcards}\ (p, t)$ to replace all the wildcards in p with the corresponding subtrees of t (coerced into patterns) when t matches p, or use $\mathit{fillWildcardsWD}$ to replace all the wildcards with the default values of their types. Finally, $\mathit{eraseVars}\ p$ replaces all the variables in p with wildcards. The definitions of these functions are straightforward and omitted here.

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 get_{SV} function has the following type:

$$\mathit{get}_{SV} : S \to V \times \mathit{Links}$$

The idea of computing $\mathit{get}\ s$ is to use a rule $\mathit{spat}_k \sim \mathit{vpat}_k \in R$ such that s matches spat_k—the restrictions on patterns imply that such a rule uniquely exists for all s—to generate the top portion of the view with vpat_k, and then recursively generate subtrees for all variables in spat_k. The 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 the matched parts/regions in the source and view, and extends the paths in the recursively computed links between the subtrees. In all, the get function defined by R is:

$$\begin{aligned} &\mathit{get}_{SV}\ s = (\mathit{reconstruct}\ (\mathit{vpat}_k, \mathit{fst} \circ \mathit{vls}),\ l_{root} \cup \mathit{links}) \qquad (5) \\ &\quad \textbf{where}\ \text{find } k \text{ such that } \mathit{spat}_k \sim \mathit{vpat}_k \in R \text{ and } \mathit{isMatch}(\mathit{spat}_k, s) \\ &\qquad \mathit{vls} = (\mathit{get} \circ \mathit{decompose}\ (\mathit{spat}_k, s)) \in \mathit{Vars}\ \mathit{spat}_k \to V \times \mathit{Links} \\ &\qquad \mathit{spat}' = \mathit{eraseVars}\ (\mathit{fillWildcards}\ (\mathit{spat}_k, s)) \\ &\qquad l_{root} = \{\, ((\mathit{spat}', [\,]), (\mathit{eraseVars}\ \mathit{vpat}_k, [\,])) \,\} \\ &\qquad \mathit{links} = \{\, ((\mathit{spat}, \mathit{path}\ (\mathit{spat}_k, v) +\!\!+ \mathit{spath}), (\mathit{vpat}, \mathit{path}\ (\mathit{vpat}_k, v) +\!\!+ \mathit{vpath})) \\ &\qquad\qquad \mid v \in \mathit{Vars}\ \mathit{vpat}_k, ((\mathit{spat}, \mathit{spath}), (\mathit{vpat}, \mathit{vpath})) \in \mathit{snd}\ (\mathit{vls}\ v) \,\} \ . \end{aligned}$$

The auxiliary function $\mathit{path} : (p \in \mathit{Pattern}) \times \mathit{Vars}\ p \to \mathit{Path}$ returns the path from the root of a pattern to one of its variables, and $(+\!\!+)$ is path concatenation. While the recursive call is written as $\mathit{get} \circ \mathit{decompose}\ (\mathit{spat}_k, s)$ in the definition above, to be precise, get should have different subscripts $\mathit{TypeOf}\ (\mathit{spat}_k, v)$ and $\mathit{TypeOf}\ (\mathit{vpat}_k, v)$ for different $v \in \mathit{Vars}\ \mathit{spat}_k$.

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 put_{SV} function has the following type:

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

The source argument of *put* is given the generic type *Tree* since the type of the old source may be different from the type of the result that *put* is supposed to produce. Given arguments (s, v, ls), *put* is defined by two cases depending on whether the root of the view is within a region referred to by the input links, i.e. whether there is some $(\_, (\_, [\,])) \in ls$.

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

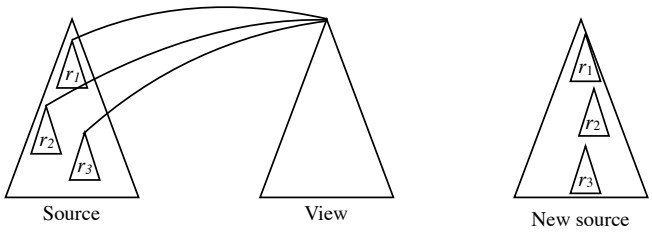
$$
\begin{aligned}
&put_{SV}\ (s, v, ls) = reconstruct\ (spat'_k, ss) && (6)\\
&\quad \textbf{where}\ \text{find } k \text{ such that } spat_k \sim vpat_k \in R \text{ and } isMatch\ (vpat_k, v)\\
&\qquad\qquad \text{and } k \text{ satisfies the extra condition below}\\
&\qquad vs = decompose\ (vpat_k, v)\\
&\qquad ss = \lambda\,(t \in Vars\ spat_k) \rightarrow\\
&\qquad\quad put(s, vs\ t, divide\ (path\ (vpat_k, t)), ls) && (7)\\
&\qquad spat'_k = fillWildcardsWD\ spat_k\\
&divide\ (prefix, ls) = \{\ (r_s, (vpat, vpath)) \mid (r_s, (vpat, prefix +\!\!+ vpath) \in ls)\ \} && (8)
\end{aligned}
$$

The omitted subscripts of *put* in (7) are $TypeOf\ (spat_k, t)$ and $TypeOf\ (vpat_k, t)$. Additionally, if there is more than one rule whose view pattern matches v, the first rule whose view pattern 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 (7) does not decrease because $vs\ t = v$ and $path\ (t, vpat_k) = [\,]$. For example, when the view patterns of both `Plus _ x y ~ Add x y` and `FromT _ t ~ t` match a view tree, the former is preferred. This helps to avoid non-termination of *put* as mentioned in Case I in Sect. 4.1.

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

$$
\begin{aligned}
&put_{SV}\ (s, v, ls) = inj_{\,TypeOf\ spat_k \rightarrow S@V}(reconstruct(spat'_k, ss)) && (9)\\
&\quad \textbf{where}\ l = ((spat, spath), (vpat, vpath)) \in ls\\
&\qquad\qquad \text{such that } vpath = [\,],\ spath \text{ is the shortest}\\
&\qquad \text{find } k \text{ such that } spat_k \sim vpat_k \in R \ \text{ and } \ spat\\
&\qquad\quad \text{is } eraseVars\ (fillWildcards\ (spat_k, t)) \text{ for some } t\\
&\qquad spat'_k = fillWildcards\ (spat_k, spat)\\
&\qquad vs = decompose\ (vpat_k, v)\\
&\qquad ss = \lambda\,(t \in Vars\ spat_k) \rightarrow\\
&\qquad\quad put(s, vs\ t, divide\ (path\ (vpat_k, t)), ls \setminus \{\ l\ \})) && (10)
\end{aligned}
$$

When there is more than one source region linked to the root of the view, to avoid Case III in Sect. 4.1, *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 $inj_{\ TypeOf\ spat_k \to S}@V$ (Equation 4) to convert it to type S; this function is available due to our requirement on interchangeable data types (see Syntax Restrictions in Sect. 4.2).

Domain of *put* To avoid Case IV in Sect. 4.1, in the actual implementation of *put* there are runtime checks for detecting invalid input links, but these checks are omitted in the above definition of *put* for clarity. We extract these checks into a separate function *check* below, which also serves as a decision procedure for the domain of *put*.

$$check : Tree \times V \times Links \to Bool$$

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

chkNoLink corresponds to the first case of *put* (6).

$$chkNoLink\ (s, v, ls) = cond_1 \wedge cond_2 \wedge cond_3$$

where find k such that $spat_k \sim vpat_k \in R$ and $isMatch\ (vpat_k, v)$
and k satisfies the same condition as in (6)

$$vs = decompose\ (vpat_k, v)$$

$$vp\ t = path\ (vpat_k, t)$$

$$cond_1 = ls == \left(\bigcup_{t \in Vars\ spat_k} addVPrefix\ (vp\ t, divide\ (vp\ t, ls)) \right)$$

$$cond_2 = \bigwedge_{t \in Vars\ spat_k} check\ (s, vs\ t, divide\ (vp\ t, ls))$$

$cond_3 =$ **if** $vpat_k$ is some bare variable pattern 'x' **then**
$TypeOf\ (spat_k, x) \leftrightarrow V$ has a rule $spat_j \sim vpat_j$ such that
$isMatch\ (vpat_j, v)$ and $vpat_j$ is not a bare variable pattern

$$addVPrefix\ (prefix, rs) = \{\, ((a, b), (c, prefix +\!\!+ d)) \mid ((a, b), (c, d)) \in rs \,\}$$

The *divide* function is defined as in Equation 8. Condition $cond_1$ checks that every link in ls is processed in one of the recursive calls, i.e. the path of every view region of ls starts with $path\ (vpat_k, t)$ for some t. (Specifically, if $Vars\ spat_k$ is empty, ls in $cond_1$ should also be empty meaning that all the links have already been processed.) $cond_2$ summarises the results of *check* for recursive calls. $cond_3$ guarantees the termination of recursion: When $vpat_k$ is a bare variable pattern,

the recursive call in Equation 7 does not decrease the size of any of its arguments; $cond_3$ makes sure that such non-decreasing recursion will not happen in the next round[1] for avoiding infinite recursive calls.

For *chkWithLink*, as in the corresponding case of *put* (Equation 9), let $l = ((spat, spath), (vpat, vpath)) \in ls$ such that $vpath = [\,]$ and $spath$ is the shortest when there is more than one such link.

$$
\begin{aligned}
&chkWithLink\ (s, v, ls) = cond_1 \wedge cond_2 \wedge cond_3 \wedge cond_4 \\
&\quad \textbf{where} \\
&\quad cond_1 = isMatch\ (spat, sel\ (s, spath)) \wedge isMatch\ (vpat, sel\ (v, vpath)) \\
&\quad cond_2 = \exists!(spat_k, vpat_k) \in R.\ vpat = eraseVars\ vpat_k \\
&\qquad\qquad \wedge\ spat \text{ is } eraseVars\ (fillWildcards\ (spat_k, t)) \text{ for some } t \\
&\quad cond_3 = ls =\!= (\{\, l \,\} \cup \bigcup_{t \in Vars\ spat_k} addVPrefix\ (path\ (vpat_k, t), \\
&\qquad\qquad\qquad\qquad divide\ (path\ (vpat_k, t), ls \setminus \{\, l \,\}))) \\
&\quad cond_4 = \bigwedge_{t \in Vars\ spat_k} check\ (s, vs\ t, divide\ (path\ (vpat_k, t)), ls \setminus \{\, l \,\}))
\end{aligned}
$$

$cond_1$ makes sure that the link l is valid (Definition 2) and $cond_2$ further checks that it can be generated from some rule of the consistency relations. $cond_3$ and $cond_4$ are for recursive calls: the latter summarises the results for the subtrees and the former guarantees that no link will be missed. It is $cond_3$ that rejects the subtle case of overlapping view regions as described at the end of Case IV in Sect. 4.1.

Main Theorem We can now state our main theorem in terms of the definitions of *get* and *put* above.

Theorem 2. *Let put′ be put with its domain intersected with $S \times V \times Links$, get and put′ form a retentive lens as in Definition 3.*

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

5 Edit Operations and Link Maintenance

Our *get* function only produces horizontal links between a source and its consistent view, while the input links to a *put* function are the ones between a source and a modified view. To bridge the gap, in this section, we demonstrate how to update the view while maintaining the links using a set of typical *edit operations* (on views). These edit operations will be used in the three case studies in the next section.

We define four edit operations, *replace*, *copy*, *move*, and *swap*, of which *move* and *swap* are defined in terms of *copy* and *replace*. The edit operations accept not

[1] For presentation purposes we only check two rounds here, but in general we should check $N + 1$ rounds where N is the number data types defined in the program.

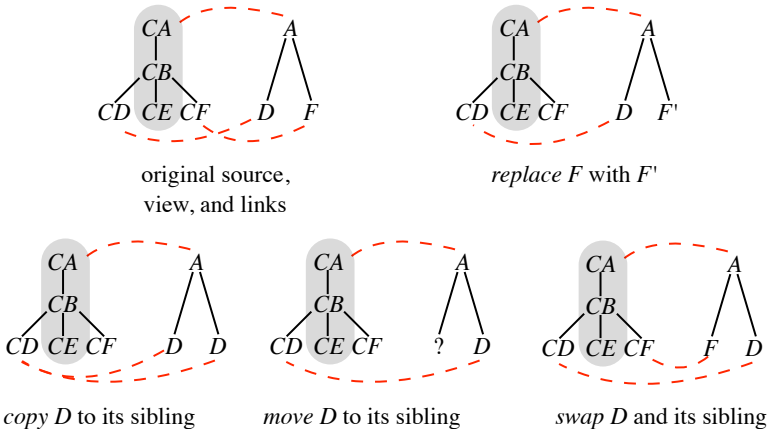


Fig. 6. How edit operations *replace*, *copy*, *move*, and *swap* main links.

only an AST but also a set of links, which is updated along with the AST. The interface has been designed in a way that the last argument of an edit operation is the pair of the AST and links, so that the user can use Haskell's ordinary function composition to compose a sequence of edits (partially applied to all the other arguments). The implementation of the four edit operations takes less than 40 lines of Haskell code, as our DSL already generates useful auxiliary functions such as fetching a subtree according to a path in some tree.

We briefly explain how the edit operations update links, as illustrated in Fig. 6: Replacing a subtree at path p will destroy all the links previously connecting to path p. Copying a subtree from path p to path p' will duplicate the set of links previously connecting to p and redirect the duplicated links to connect to p'. Moving a subtree from p to p' will destroy links connecting to p' and redirect the links (previously) connecting to p to connect to p'. Swapping subtrees at p and p' will also swap the links connecting to p and p'.

6 Case Studies

We demonstrate how our DSL works for the problems of code refactoring [12], resugaring [20, 21], and XML synchronisation [19], all of which require that we constantly make modifications to ASTs and synchronise them with CSTs. For all these problems, retentive lenses provide a systematic way for the user to preserve information of interest in the original CST after synchronisation. The source code for these case studies can be found on the first author's web page: http://www.prg.nii.ac.jp/members/zhu/.

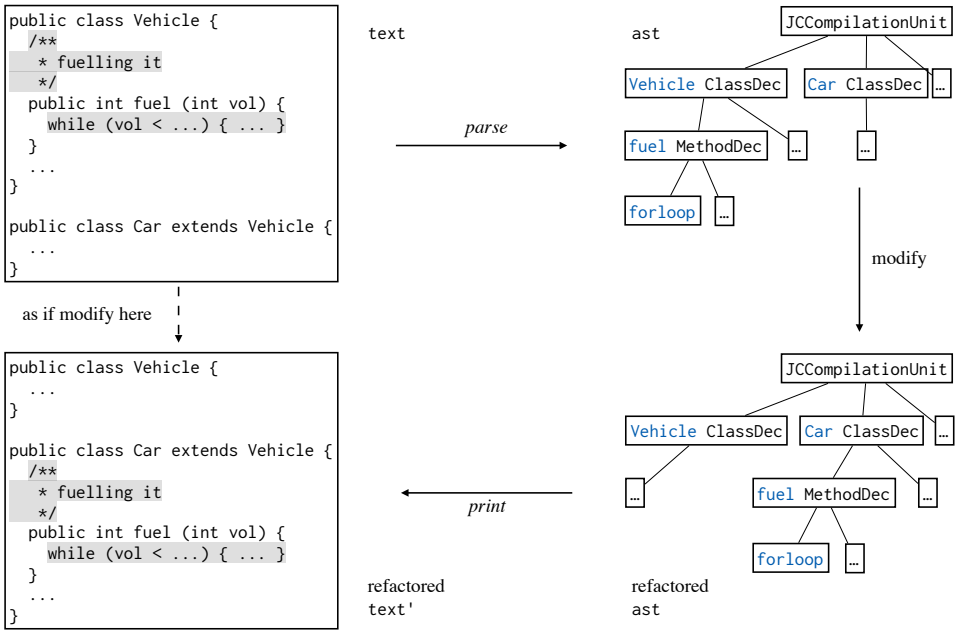


Fig. 7. An example of the *push-down* code refactoring. (Subclasses `Bus` and `Bicycle` are omitted due to space limitation.)

6.1 Refactoring

As we will report below, we have programmed the consistency relations between CSTs and ASTs for a small subset of Java 8 [13] and tested the generated retentive lens on a particular refactoring. Even though the case study is small, we believe that our framework is general enough: We have surveyed the standard set of refactoring operations for Java 8 provided by Eclipse Oxygen (with Java Development Tools) and found that all the 23 refactoring operations can be represented as the combinations of our edit operations defined in Sect. 5. A summary can be found in the appendix (Sect. E).

The *Push-Down* Code Refactoring An example of the *push-down* code refactoring is illustrated in Fig. 7. At first, the user designed a `Vehicle` class and thought that it should possess 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 `Vehicle`'s subclasses, the user realises that bicycles cannot be fuelled and decides to do the push-down code refactoring, which removes the `fuel` method from `Vehicle` and pushes the method definition down to subclasses `Bus` and `Car` but not `Bicycle`. Instead of directly modifying the (program) `text`, most refactoring tools choose to parse the program text into its `ast`, perform code

refactoring on the `ast`, and regenerate new (program) `text'`. The bottom-left corner of Fig. 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 does not come for free actually, as the `ast`—being a concise and compact representation of the program `text`—includes neither comments nor the form of the original `while` loop. So if the user implements the *parse* and *print* functions as back-and-forth conversions between CSTs ASTs (or even as a well-behaved lens), they may produce unsatisfactory results in which the comment and the `while` syntactic sugar are lost.

Implementation in Our DSL Following the grammar of Java 8, we define data types for a simplified version of its concrete syntax, which consists of definitions of classes, methods, and variables; arithmetic expressions (including assignment and method invocation); and conditional and loop statements. For convenience, we also restrict the occurrence of statements and expressions to exactly once in some cases (such as variable declarations). Then we define the corresponding simplified version of the abstract syntax that follows the one defined by the JDT parser [17]. This subset of Java 8 has around 80 CST constructs (production rules) and 30 AST constructs; the 70 consistency relations among them generate about 3000 lines of code for the retentive lenses and auxiliary functions (such as the ones for conversions between interchangeable data types and edit operations).

Demo We can now perform some experiments on Fig. 7.

- First we test `put cst ast ls`, where `(ast, ls) = get cst`. We get back the same `cst`, showing that the generated lenses do satisfy Hippocraticness.
- As a special case of Correctness, we let `cst' = put cst ast []` and check `fst (get cst') == ast`. In `cst'`, the `while` loop becomes a basic `for` loop and all the comments disappear. This shows that *put* will create a new source solely from the view if links are missing.
- Then we change `ast` to `ast'` and the set of links `ls` to `ls'` using our edit operations, simulating the *push-down* code refactoring for the `fuel` method. To show the effect of Retentiveness more clearly, when building `ast'`, the `fuel` method in the `Car` class is copied from the `Vehicle` class, while the `fuel` method in the `Bus` class is built from scratch (i.e. replaced with a 'new' `fuel` method). Let `cst' = put cst ast' ls'`. In the `fuel` method of the `Car` class, the `while` loop and its associated comments are preserved; but in the `fuel` method of the `Bus` class, there is only a `for` loop without any associated comments. This is where Retentiveness helps the user to retain information on demand. Finally, we also check that Correctness holds: `fst (get cst') == ast'`.

6.2 Resugaring

We have seen syntactic sugar such as negation and `while` loops. The idea of *resugaring* is to print evaluation sequences in a core language using the constructs

of its surface syntax (which contains sugar) [20, 21]. To solve the problem, Pombrio and Krishnamurthi [20] enrich the AST to incorporate fields for holding tags that mark from which syntactic object an AST construct comes. Using retentive lenses, we can also solve the problem while leaving the AST clean—we can write consistency relations between the surface syntax and the abstract syntax and passing the generated *put* function proper links for retaining syntactic sugar, which we have already seen in the arithmetic expression example (where we retain the negation) and in the code refactoring example (where we retain the `while` loop). Both Pombrio and Krishnamurthi's 'tag approach' and our 'link approach', in actuality, identifies where an AST construct comes from; however, the link approach has an advantage that it leaves ASTs clean and unmodified so that we do not need to patch up the existing compiler to deal with tags.

6.3 XML Synchronisation

In this subsection, we present a case study on XML synchronisation, which is pervasive in the real world. The specific example used here is adapted from Pacheco, Zan, and Hu's paper [19], where they use their DSL, BiFluX, to synchronise address books.

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

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

```
AddrGroup  <---> SocialGroup
  AddrGroup grp p  ~  SocialGroup grp p

List Person <---> List Name
  Nil  ~  Nil
  Cons p xs  ~  Cons p xs

Person <---> Name
  Person t  ~  t

Triple Name Email Tel <---> Name
  Triple name  _ _  ~  name .
```

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

As Fig. 8 shows, the original source is `addrBook` and its consistent view is `socialBook`, both of which have two relationship groups: `coworkers` and `friends`. The source has a record `Person (Triple "Alice" "alice@abc.xyz" "000111")` in the group `coworkers`, and we will see how this record changes in the new source after we update the view `socialBook` in the following way and propagate the changes back: we (i) reorder the two groups; (ii) change Alice's group from `coworkers` to `friends`; (iii) create a new social relationship group `family` for family members.

In our case, to produce a new source `socialBook'`, we handle the three updates using our basic edit operations (in this case, only *swap*, *move*, and *replace*) which also maintain the links. Feeding the original source `addrBook`, updated view `socialBook'` and links `hls'` to the (generated) `put` function, we obtain the updated `addrBook'`. In Fig. 8, it is clearly seen that carefully maintained links help us to preserve email addresses and telephone numbers associated with each

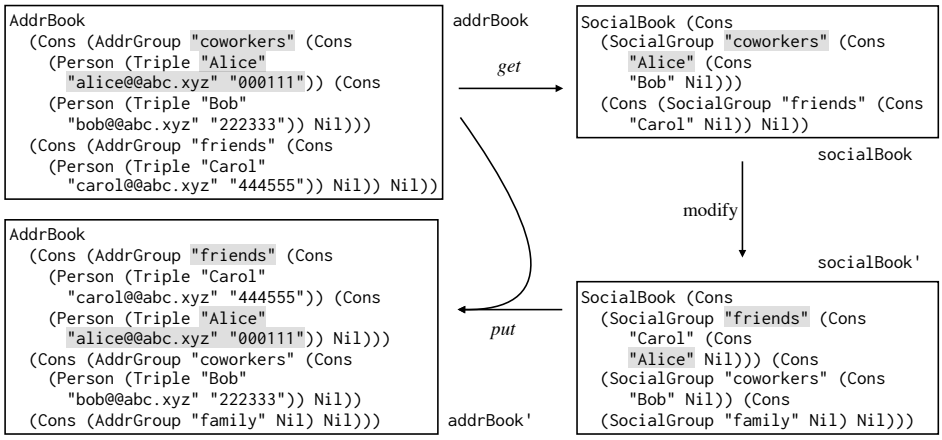


Fig. 8. An example of XML synchronisation. (Grey areas highlight how the record `Alice` is updated.)

person during the *put* process; note that well-behavedness does not guarantee the retention of this information, since the input view is not consistent with the input source in this case.

As pointed out by Pacheco, Zan, and Hu, examples of this kind motivate extensions to (combinator-based) alignment-aware languages such as Boomerang [4] and matching lenses [3]. In fact, it is hard for those languages to handle source-view alignment where some view elements are moved out of its original list-like structure (or *chunk* [3]) and put into a new list-like structure, probably far away—because when using those languages, we usually lift a lens combinator k handling a single element to k^* dealing with a list of elements, so that the 'scope' of the alignment performed by k^* is always within that single list (which it currently works on).

7 Related Work

7.1 Alignment

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

Alignment for Lists The earliest lenses [11] 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 [4] and their successor matching lenses [3]. As for matching lenses, when a *put* is invoked, it will first find the correspondence between *chunks* (data structures that are reorderable, such as

lists) of the old and new views using some predefined strategies; based on the correspondence, the chunks in the source are aligned to match the chunks in the new view. Then element-wise updates are performed on the aligned chunks. Matching lenses are designed to be practically easy to use, so they are equipped with a few fixed matching strategies (such as greedy align) from which the user can choose. However, whether the information is retained or not, still depends on the lens applied after matching. As a result, the more complex the applied lens is, the more difficult to reason about the information retained in the new source. Moreover, it suffers a disadvantage that the alignment is only between a single source list and a single view list, as already discussed in the last paragraph of Sect. 6.3. BiFluX [19] overcomes the disadvantage by providing the functionality that allows the user to write alignment strategies manually; in this way, when we see several lists at once, we are free to search for elements and match them in all the lists. But this alignment still has the limitation that each source element and each view element can only be matched at most once—after that they are classified as either *matched pair*, *unmatched source element*, or *unmatched view element*. Assuming that an element in the view has been copied several times, there is no way to align all the copies with the same source element. (However, it is possible to reuse an element several times for the handling of unmatched elements.)

By 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.

Alignment for Containers To generalise list alignment, a more general notion of data structures called *containers* [1] is used [14]. 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, Pierce, and Wagner'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, Cunha, and Hu's method permits more flexible shape changes, and they call it *shape alignment* [18]. 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. 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 that 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 hardly help.

7.2 Provenance and Origin

Our idea of links is inspired by research on provenance [6] in database communities and origin tracking [7] in the rewriting communities.

Cheney, Chiticariu, and Tan classify provenance into three kinds, *why*, *how*, and *where*: *why-provenance* is the information about which data in the view is from which rows in the source; *how-provenance* additionally counts the number of times a row is used (in the source); *where-provenance* in addition records the column where a piece of data is from. In our setting, we require that two pieces of data linked by vertical correspondence be equal (under a specific pattern), and hence the vertical correspondence resembles where-provenance. However, the above-mentioned provenance is not powerful enough as they are mostly restricted to relational data, namely rows of tuples—in functional programming, the algebraic data types are more complex. For this need, *dependency provenance* [5] is proposed; it tells the user on which parts of a source the computation of a part of a view depends. In this sense, our consistency links are closer to dependency provenance.

The idea of inferring consistency links can be found in the work on origin tracking for term rewriting systems [7], in which the origin relations between rewritten terms can be calculated by analysing the rewrite rules statically. However, it was developed solely for building trace between intermediate terms rather than using trace information to update a tree further. Based on origin tracking, 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' [15]. This description shows that the algorithm also decomposes large terms into smaller ones resembling our regions. In terms of the formatting aspect, we think that retentiveness can in effect be the same as their theorem if we include vertical correspondence (representing view updates) in the theory, rather than dealing with it implicitly and externally as in Sect. 5.

The use of consistency links can also be found in Wang, Gibbons, and Wu's work, where the authors extend state-based lenses and use links for tracing data in a view to its origin in a source [24]. When a sub-term in the view is edited locally, they use links to identify a sub-term in the source that 'contains' the edited sub-term in the view. When updating the old source, it is sufficient to

only perform state-based *put* on the identified sub-term (in the source) so that the update becomes an incremental one. Since lenses generated by our DSL also create consistency links (albeit for a different purpose), they can be naturally incrementalised using the same technique.

7.3 Operation-based BX

Our work is closely relevant to the operation-based approaches to BX, in particular, the delta-based BX model [8, 9] and edit lenses [14]. The (asymmetric) delta-based BX model regards the differences between a view state v and v' as *deltas*, which are abstractly represented as arrows (from the old view to the new view). The main law of the framework can be described as 'given a source state s and a view delta det_v, det_v should be translated to a source delta det_s between s and s' satisfying $get\ s' = v'$'. As the law only guarantees the existence of a source delta det_s that updates the old source to a correct state, it is yet not sufficient to derive Retentiveness in their model, for there are infinite numbers of translated delta det_s which can take the old source to a correct state, of which only a few are 'retentive'. To illustrate, Diskin, Xiong, and Czarnecki tend to represent deltas as edit operations such as *create*, *delete*, and *change*; representing deltas in this way will only tell the user what must be changed in the new source, while it requires additional work to reason about what is retained. However, it is possible to exhibit Retentiveness if we represent deltas in some other proper form. Compared to Diskin, Xiong, and Czarnecki's work, Hofmann, Pierce, and Wagner give concrete definitions and implementations for propagating edit operations (in a symmetric setting).

8 Conclusion

In this paper, we showed that well-behavedness is not sufficient for retaining information after an update and it may cause problems in many real-world applications. To address the issue, we illustrated how to use links to preserve desired data fragments of the original source, and developed a semantic framework of (asymmetric) retentive lenses. 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 provide four edit operations which can update a view together with the links between the view and the original source, and demonstrated the practical use of retentive lenses for code refactoring, resugaring, and XML synchronisation; we discussed related work about alignment, origin tracking, and operation-based BX. Some further discussions can be found in the appendix (Sect. C).

References

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. "Containers: Constructing Strictly Positive Types". In: *Theoretical Computer Science* 342.1 (Sept. 2005), pp. 3–27. ISSN: 0304-3975. URL: https://doi.org/10.1016/j.tcs.2005.06.002.

[2] François Bancilhon and Nicolas Spyratos. "Update Semantics of Relational Views". In: *ACM Transactions on Database Systems* 6.4 (Dec. 1981), pp. 557–575. ISSN: 0362-5915. URL: https://doi.org/10.1145/319628.319634.

[3] Davi M.J. Barbosa et al. "Matching Lenses: Alignment and View Update". In: *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ICFP '10. Baltimore, Maryland, USA: ACM, 2010, pp. 193–204. ISBN: 978-1-60558-794-3. URL: https://doi.org/10.1145/1863543.1863572.

[4] Aaron Bohannon et al. "Boomerang: Resourceful Lenses for String Data". In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: ACM, 2008, pp. 407–419. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328487. URL: https://doi.org/10.1145/1328438.1328487.

[5] James Cheney, Amal Ahmed, and Umut a. Acar. "Provenance As Dependency Analysis". In: *Mathematical Structures in Computer Science* 21.6 (Dec. 2011), pp. 1301–1337. ISSN: 0960-1295. URL: https://doi.org/10.1017/S0960129511000211.

[6] James Cheney, Laura Chiticariu, and Wang-Chiew Tan. "Provenance in Databases: Why, How, and Where". In: *Found. Trends databases* 1.4 (Apr. 2009), pp. 379–474. ISSN: 1931-7883.

[7] A. van Deursen, P. Klint, and F. Tip. "Origin Tracking". In: *Journal of Symbolic Computation* 15.5-6 (May 1993), pp. 523–545. ISSN: 0747-7171. URL: https://doi.org/10.1016/S0747-7171(06)80004-0.

[8] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. "From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case". In: *Journal of Object Technology* 10 (2011), 6:1–25. ISSN: 1660-1769. URL: http://www.jot.fm/contents/issue_2011_01/article6.html.

[9] Zinovy Diskin et al. "From State- to Delta-Based Bidirectional Model Transformations: The Symmetric Case". In: *Model Driven Engineering Languages and Systems*. Ed. by Jon Whittle, Tony Clark, and Thomas Kühne. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 304–318. ISBN: 978-3-642-24485-8. URL: https://doi.org/10.1007/978-3-642-24485-8_22.

[10] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. "Quotient Lenses". In: *SIGPLAN Not.* 43.9 (Sept. 2008), pp. 383–396. ISSN: 0362-1340. URL: https://doi.org/10.1145/1411203.1411257.

[11] J. Nathan Foster et al. "Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem". In: *ACM Trans. Program. Lang. Syst.* 29.3 (May 2007). ISSN: 0164-0925. URL: https://doi.org/10.1145/1232420.1232424.

[12] Martin Fowler and Kent Beck. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley Professional, 1999.

[13] James Gosling et al. *The Java Language Specification, Java SE 8 Edition (Java Series)*. 2014. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf.

[14] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. "Edit Lenses". In: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA: ACM, 2012, pp. 495–508. ISBN: 978-1-4503-1083-3. URL: https://doi.org/10.1145/2103656.2103715.

[15] Maartje de Jonge and Eelco Visser. "An Algorithm for Layout Preservation in Refactoring Transformations". In: *Software Language Engineering*. Ed. by Anthony Sloane and Uwe Aßmann. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 40–59. ISBN: 978-3-642-28830-2. URL: https://doi.org/10.1007/978-3-642-28830-2_3.

[16] John MacFarlane. *Pandoc: a Universal Document Converter*. 2013. URL: http://pandoc.org.

[17] Oracle Corporation and OpenJDK Community. *OpenJDK*. 2014. URL: http://openjdk.java.net/.

[18] Hugo Pacheco, Alcino Cunha, and Zhenjiang Hu. "Delta lenses over inductive types". In: *Electronic Communications of the EASST* 49 (2012), pp. 1–17. URL: https://doi.org/10.14279/tuj.eceasst.49.713.

[19] Hugo Pacheco, Tao Zan, and Zhenjiang Hu. "BiFluX: A Bidirectional Functional Update Language for XML". In: *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming*. PPDP '14. Canterbury, United Kingdom: ACM, 2014, pp. 147–158. ISBN: 978-1-4503-2947-7. URL: https://doi.org/10.1145/2643135.2643141.

[20] Justin Pombrio and Shriram Krishnamurthi. "Resugaring: Lifting Evaluation Sequences Through Syntactic Sugar". In: *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '14 6. Edinburgh, United Kingdom: ACM, 2014, pp. 361–371. ISBN: 978-1-4503-2784-8. URL: https://doi.org/10.1145/2594291.2594319.

[21] Justin Pombrio and Shriram Krishnamurthi. "Hygienic Resugaring of Compositional Desugaring". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. ICFP 2015 13. Vancouver, BC, Canada: ACM, 2015, pp. 75–87. ISBN: 978-1-4503-3669-7. URL: https://doi.org/10.1145/2784731.2784755.

[22] Tillmann Rendel and Klaus Ostermann. "Invertible Syntax Descriptions: Unifying Parsing and Pretty Printing". In: *Proceedings of the Third ACM Haskell Symposium on Haskell*. Haskell '10. Baltimore, Maryland, USA: ACM, 2010, pp. 1–12. ISBN: 978-1-4503-0252-4. URL: https://doi.org/10.1145/1863523.1863525.

[23] Perdita Stevens. "Bidirectional model transformations in QVT: semantic issues and open questions". In: *Software & Systems Modeling* 9.1 (Dec. 2008), p. 7. ISSN: 1619-1374. DOI: 10.1007/s10270-008-0109-9. URL: https://doi.org/10.1007/s10270-008-0109-9.

[24] Meng Wang, Jeremy Gibbons, and Nicolas Wu. "Incremental Updates for Efficient Bidirectional Transformations". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*. ICFP '11. Tokyo, Japan: ACM, 2011, pp. 392–403. ISBN: 978-1-4503-0865-6. URL: https://doi.org/10.1145/2034773.2034825.

A Well-behaved Lenses Are Retentive Lenses

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

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

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

where

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

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

Proof.

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

B Programming Examples in Our DSL

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

– Let us consider the binary trees

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

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

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

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

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

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

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

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

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

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

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

C Discussions of the Paper

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

C.1 Strong Retentiveness

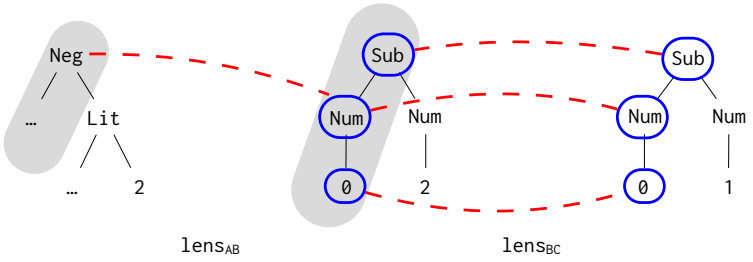
Through our research into Retentiveness, we also tried a different theory, which we call *Strong Retentiveness* now, that requires that the consistency links generated by *get* should additionally capture all the 'information' of the source and *uniquely identify* it. Strong Retentiveness is appealing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

This observation inspires us to generalise region patterns to *properties* in order for holding more information (that can eventually uniquely identify a tree) and generalise links connecting $\mathit{Pattern} \times \mathit{Path}$ to links connecting $\mathit{Property} \times \mathit{Path}$

accordingly. We eventually formalised three kinds of properties that are sufficient to capture all the information of a tree (e.g. $\texttt{cst}_1$): *region patterns* (e.g. $\texttt{reg}_1$, $\texttt{reg}_2$, and $\texttt{reg}_3$), *relative positions* between two regions (e.g. $\texttt{reg}_2$ is the first child of $\texttt{reg}_1$ and $\texttt{reg}_3$ is the second child of $\texttt{reg}_1$), and *top* that marks the top of a tree (e.g. $\texttt{reg}_1$ 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 $\texttt{reg}_1$ and is referred to as the top of $\texttt{cst}_1$. This will additionally cause many difficulties in lens composition, as different lenses might assign the same region different names and we need to do 'alpha conversion'. Take everything into consideration, finally, we opted for the 'weaker' but simpler version of Retentiveness.

C.2 Rethinking Lens Composition

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



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

Coincidentally, similar problems can also be found in quotient lenses [10]: A quotient lens operates on sources and views that are divided into many equivalent classes, and the well-behavedness is defined on those equivalent classes rather than a particular pair of source and view. In order to establish a sequential composition $l; k$, the authors require that the abstract (view-side) equivalence relation of lens l is identical to the concrete (source-side) equivalence of lens k. We leave other possibilities of link composition to future work.

As for our DSL, the lack of composition does not cause problems because of the philosophy of design. Take the scenario of writing a parser for example where there are two main approaches for the user to choose: to use parser combinators (such as Parsec) or to use parser generators (such as Happy). While parser combinators offer the user many small composable components, parser generators usually provide the user with a high-level syntax for describing the grammar

of a language using production rules (associated with semantic actions). Then the generated parser is used as a 'standalone black box' and usually will not be composed with some others (although it is still possible to be composed 'externally'). Our DSL is designed to be a 'lens generator' and we have no difficulty in writing bidirectional transformations for the subset of Java 8 in Sect. 6.1.

C.3 Retaining Code Styles

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

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

D Proofs About Retentive Lenses

D.1 Composability

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

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

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

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

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

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

Proof (Retentiveness Preservation). In Fig. 3, we prove $fst \cdot ls_{ac} \subseteq fst \cdot ls_{a'c'}$.

To finish the proof, we need the following lemma.

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

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

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

Now, we present the main proof:

$$
\begin{array}{l}
\quad fst \cdot ls_{ac} \\
=\{\ R = R \cdot id_{\textsc{rdom} R}\ \} \\
\quad fst \cdot ls_{ac'} \cdot id_{\textsc{rdom}(ls_{ac'})} \\
\subseteq\{\ id_{\textsc{rdom}(ls_{ac'})} \subseteq ls_{c'b'} \cdot ls^{\circ}_{c'b'}\ \text{by sub-proof-1 below}\ \} \\
\quad fst \cdot ls_{ac'} \cdot (ls_{c'b'} \cdot ls^{\circ}_{c'b'}) \\
=\{\ \text{Relation composition is associative}\ \}
\end{array}
$$

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

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

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

D.2 Retentiveness of the DSL

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

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

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

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

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 (5). In *chkWithLink*, $cond_1$ and $cond_2$ are true by the evident semantics of pattern matching functions such as *isMatch* and *reconstruct*. $cond_3$ is true following the definition of l_{root}, $links$, and $divide$. Finally, $cond_4$ is true by the inductive hypothesis.

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

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

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

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

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

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

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

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

where $spat_k \sim vpat_k \in R$ is the unique rule such that $spat_k$ matches s. l_{root}, $links$, and vls are defined exactly the same as in get (5).

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

$$put(s, get(s)) = inj(reconstruct(spat'_k, ss)) \tag{11}$$

where

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

and

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

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

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

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

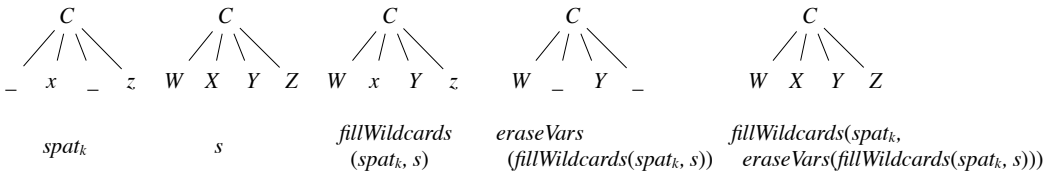


Fig. 9. A property regarding *fillWildcards*.

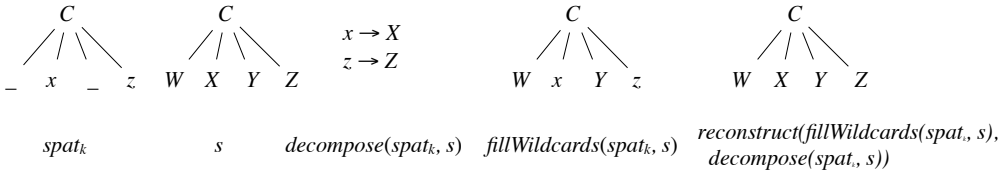


Fig. 10. A property regarding Reconstruct-Decompose.

By Lemma 4, we have

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

Now, we substitute $\mathit{fillWildcards}(\mathit{spat}_k, s)$ for spat'_k and $\mathit{decompose}(\mathit{spat}_k, s)$ for ss in equation (11), and obtain

$$
\begin{aligned}
& \mathit{put}(s, \mathit{get}(s)) \\
= {} & \{ \ \mathit{Equation}(11) \ \} \\
& \mathit{inj}_{S \rightarrow S@V}(\mathit{reconstruct}(\mathit{spat}'_k, ss)) \\
= {} & \mathit{inj}_{S \rightarrow S@V}(\mathit{reconstruct}(\mathit{fillWildcards}(\mathit{spat}_k, s), \mathit{decompose}(\mathit{spat}_k, s))) \\
= {} & \{ \text{ See Fig. 10 } \} \\
& \mathit{inj}_{S \rightarrow S@V}(s) \\
= {} & s \ .
\end{aligned}
$$

This completes the proof of Hippocraticness.

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

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

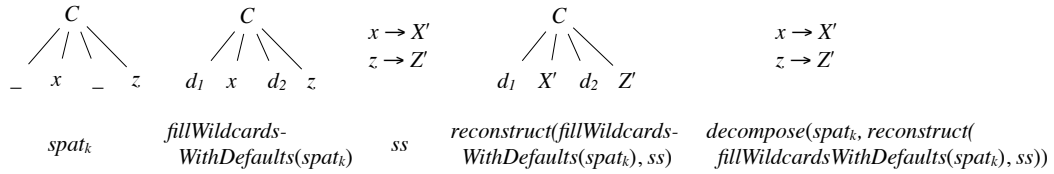


Fig. 11. A property regarding Decompose-Reconstruct.

i.e. $put(s, v, ls) = reconstruct(fillWildcardsWithDefaults(spat_k), ss)$. Then

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

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

$$ss = \lambda(t \in Vars(spat_k)) \rightarrow put(s, decompose(vpat_k, v)(t), divide(Path(vpat_k, t), ls)) \ .$$

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

$$\begin{aligned} vls &= get \circ decompose(spat_k, put(s, v, ls)) \\ &= get \circ decompose(spat_k, reconstruct(fillWildcardsWithDefaults(spat_k), ss)) \\ &= \{ \text{ See Fig. 11 } \} \\ &\ get \circ ss \\ &= \lambda(t \in Vars(spat_k)) \rightarrow get(put(s, decompose(vpat_k, v)(t), divide(Path(vpat_k, t), ls))) \end{aligned}$$

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

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

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

Theorem 5. *(Retentiveness of the DSL) For any* (s, v, ls) *that* $check(s, v, ls) = True$, $get(put(s, v, ls)) = (v', ls')$, *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 (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 $isMatch(spat_k, spat) = True$. We have

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

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

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

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

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

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

$$\begin{aligned}
links = \{\ &((a, Path(spat_k, t) +\!\!+ b),\ (c, Path(vpat_k, t) +\!\!+ d))\\
&\mid t \in Vars(vpat_k), ((a, b),\ (c, d)) \in snd(vls(t))\ \}\ \text{,and}
\end{aligned} \tag{12}$$

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

For l_{root}, we have

$$\begin{aligned}
&eraseVars(fillWildcards(spat_k, s'))\\
=&\{\ \text{Unfolding } s'\ \}\\
&eraseVars(fillWildcards(spat_k, reconstruct(fillWildcards(spat_k, spat), ss)))\\
=&\{\ \text{See Fig. 12}\ \}\\
&eraseVars(fillWildcards(spat_k, spat))\\
=&\{\ \text{By } cond_2 \text{ in } chkWithLink\ \}\\
&spat
\end{aligned}$$

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

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

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

For the links in $ls \setminus \{\, l \,\}$, we show that they are preserved in *links* (12) above. By $cond_3$ in *chkWithLink*, for every link $m \in ls \setminus \{\, l \,\}$, there is some t_m in $Vars(spat_k)$ such that

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

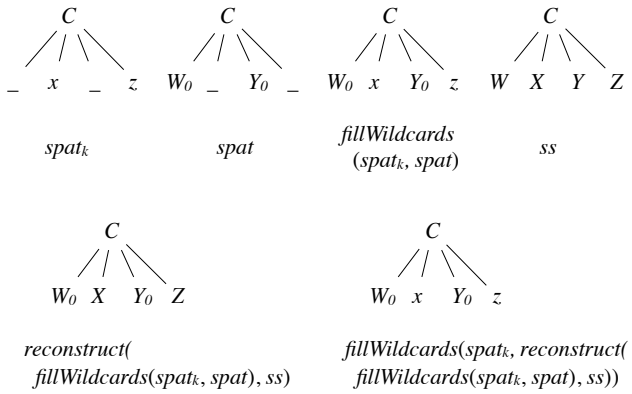


Fig. 12. Another property regarding *fillWildcards*.

If $m = ((a, b), (c, Path(vpat_k, t_m) +\!\!+ d))$, then

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

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

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

Now by the definition of *links* (12), $((a, Path(spat_k, t_m) +\!\!+ b'), (c, Path(vpat_k, t_m) +\!\!+ d)) \in links$, therefore m is also preserved.

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

E Refactoring Operations as Edit Operation Sequences

We summarise how the 23 refactoring operations for Java 8 in Eclipse Oxygen could be described by *replace*, *copy*, *move*, *swap*, *insert*, and *delete*, where the *insert* and *delete* operations on lists can be implemented in terms of the first four. For instance, to *insert* an element e at position i in a list of length n (where $1 \leqslant i \leqslant n$), we can follow these steps: (i) Change the list to length $n + 1$. (ii) Starting from the tail of the list, *move* each element at position j such that $j > i$ to

position $j+1$. (iii) *replace* the element at position i with e. Deleting the element at position i is almost as simple as moving each element after i one position ahead and decrease the length of the list by one.

Table 1: Refactoring Operations as Edit Operation Sequences.

Refactor Operation	Description	Edit Operations
Rename	Renames the selected element and (if enabled) corrects all references to the elements	*replace* the selected element and all references with the new name.
Use Supertype Where Possible	Replaces occurrences of a type with one of its supertypes after identifying all places where this replacement is possible.	*replace* all occurrences.
Generalize Declared Type	Allows the user to choose a supertype of the reference's current type. If the reference can be safely changed to the new type, it is.	*replace* all occurrences.
Infer Generic Type Arguments	Replaces raw type occurrences of generic types by parameterized types after identifying all places where this replacement is possible.	*replace* all occurrences.
Encapsulate Field	Replaces all references to a field with getter and setter methods.	*insert* getters and setters; *replace* all occurrences (with getters or setters respectively).
Change Method Signature	Changes parameter names, parameter types, parameter order and updates all references to the corresponding method.	*replace* all occurrences. Use *swap* if we need to change the parameter order.
Extract Method	Creates a new method containing the statements or expression currently selected and replaces the selection with a reference to the new method.	*insert* a new method; *move* selected code; *replace* the selection.
Extract Local Variable	Creates a new variable assigned to the expression currently selected and replaces the selection with a reference to the new variable.	*insert* a new variable; *copy* the selected expression to the variable assignment; *replace* the selected expression.
Extract Constant	Creates a static final field from the selected expression and substitutes a field reference, and optionally rewrites other places where the same expression occurs.	*insert* a field; *copy* the selected expression; *replace* the selected expression.
Introduce Parameter	Replaces an expression with a reference to a new method parameter, and updates all callers of the method to pass the expression as the value of that parameter.	*insert* a method parameter; *insert* the selected expression to all the callers (use *copy* if we want to preserve the information attached to the expression); *replace* the expression with the new method parameter.

Introduce Factory	Creates a new factory method, which will call a selected constructor and return the created object. All references to the constructor will be replaced by calls to the new factory method.	*insert* a factory method; *replace* all the references to the constructor.
Introduce Indirection	Creates a static indirection method delegating to the selected method.	*insert* a method.
Convert to Nested	Converts an anonymous inner class to a member class.	*insert* a member class; *move* the code within the anonymous class to the member class; *delete* the anonymous class.
Move Type to New File	Creates a new Java compilation unit for the selected member type or the selected secondary type, updating all references as needed.	*Move* the selected code to the new file; *replace* all references.
Convert Local Variable to Field	Turn a local variable into a field. If the variable is initialized on creation, then the operation moves the initialization to the new field's declaration or to the class's constructors.	*insert* a field; *copy* the initialization; *delete* the variable declaration.
Extract Superclass	Extracts a common superclass from a set of sibling types. The selected sibling types become direct subclasses of the extracted superclass after applying the refactoring.	*insert* a superclass; *move* fields to the superclass; *replace* declarations of sibling types (classes) so that they extend the superclass; *insert* lacking fields into sibling classes.
Extract Interface	Creates a new interface with a set of methods and makes the selected class implement the interface.	generally the same as above.
Move	Moves the selected elements and (if enabled) corrects all references to the elements (also in other files).	*move* the selected elements; *replace* all references.
Push Down	Moves a set of methods and fields from a class to its subclasses.	*move* the methods and fields.
Pull Up	Moves a field or method to a superclass of its declaring class or (in the case of methods) declares the method as abstract in the superclass.	*move* the field or *insert* an abstract method declaration.

Introduce Parameter Object	Replaces a set of parameters with a new class, and updates all callers of the method to pass an instance of the new class as the value to the introduce parameter.	*insert* a class definition; *move* the parameters to the class; in callers' definitions, *delete* the set of parameters and *insert* the class type as a new parameter; for callers' arguments, *delete* the arguments corresponding to the set of parameters and *insert* a class instance.
Extract Class	Replaces a set of fields with new container object. All references to the fields are updated to access the new container object.	*insert* a new class; *move* the fields; *replace* references to the fields with references to the container object and field names.
Inline	Inline local variables, methods or constants.	For a variable or a constant, *replace* the occurrences with the value; *delete* the definition. For a method, *replace* all occurrences of parameters within the method body with real arguments; *replace* the method call with the (new) method body; *delete* the method definition.