Retentive Lenses*

ZIRUN ZHU†, National Institute of Informatics, Japan
HSIANG-SHANG KO, National Institute of Informatics, Japan
ZHIXUAN YANG‡, National Institute of Informatics, Japan
ZHENJIANG HU§, National Institute of Informatics, Japan

Researchers in the field of bidirectional transformations have studied data synchronisation for a long time and proposed various properties, of which *well-behavedness* is the most fundamental. However, well-behavedness is not enough to characterise the result of an update (performed by *put*), regarding what information should be retained in the updated source. The root cause is that the property, Hippocraticness, for guaranteeing the retention of source information is too "global", only requiring that the whole source should be unchanged if the whole view is.

In this paper we propose a new property *retentiveness*, which enables us to directly reason about the retention of source information locally. Central to our formulation of retentiveness is the notion of *links*, which are used to relate fragments of sources and views. These links are passed as additional input to the extended *put* function, which produces a new source in a way that preserves all the source fragments attached to the links. We validate the feasibility of retentiveness by designing a domain-specific language (DSL) supporting mutually recursive algebraic data types. We prove that any program written in our DSL gives rise to a pair of retentive *get* and *put*. We show the usefulness of retentiveness by presenting examples in two different research areas: resugaring and code refactoring.

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

Additional Key Words and Phrases: bidirectional transformations, asymmetric lenses

ACM Reference Format:
Zirun Zhu, Hsiang-Shang Ko, Zhixuan Yang, and Zhenjiang Hu. 2018. Retentive Lenses. *Proc. ACM Program. Lang.* 0, CONF, Article 0 (October 2018), 44 pages.

1 INTRODUCTION

Every now and then, we need to write a pair of programs to synchronise data. Typical examples include: view querying and view updating in relational databases [Bancilhon and Spyratos 1981; Dayal and Bernstein 1982; Gottlob et al. 1988] for keeping a database and its view in sync; parsers and printers as front ends of compilers [Matsuda et al. 2007; Rendel and Ostermann 2010; Zhu et al. 2016] for keeping program text and its internal representation in sync; text file format conversion [MacFarlane 2013] (e.g., between Markdown and HTML). All these pairs of programs should satisfy some properties in order to make synchronisation work in harmony.

*Draft manuscript (28th October 2018)
†Also with SOKENDAI (The Graduate University for Advanced Studies), Department of Informatics.
‡Also with SOKENDAI (The Graduate University for Advanced Studies), Department of Informatics.
§Also with SOKENDAI (The Graduate University for Advanced Studies), Department of Informatics.

Authors' addresses: Zirun Zhu, Information Systems Architecture Science Research Division, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda, Tokyo, 101-8430, Japan, zhu@nii.ac.jp; Hsiang-Shang Ko, Information Systems Architecture Science Research Division, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda, Tokyo, 101-8430, Japan, hsiang-shang@nii.ac.jp; Zhixuan Yang, Information Systems Architecture Science Research Division, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda, Tokyo, 101-8430, Japan, yzx@nii.ac.jp; Zhenjiang Hu, Information Systems Architecture Science Research Division, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda, Tokyo, 101-8430, Japan, hu@nii.ac.jp.

2018. 2475-1421/2018/10-ART0 $15.00
https://doi.org/

Bidirectional transformations [Czarnecki et al. 2009] (BXs for short) provide a sound framework for writing these pairs of programs, where the property of *well-behavedness* [Foster et al. 2007; Stevens 2008] plays a fundamental role. Within the decade, various bidirectional programming languages and systems have been developed to help the user to write BXs easily [Barbosa et al. 2010; Bohannon et al. 2008; Foster et al. 2007; Hidaka et al. 2010; Hofmann et al. 2012; Ko et al. 2016; Lutterkort 2008; Pacheco et al. 2014], and well-behavedness has become the minimum, and in most cases the only, standard.

In this paper, we argue that for many applications, well-behavedness is too weak to capture the behaviour of the transformations, and thus a more refined property, which we call *retentiveness*, should be developed. We will use a small example to demonstrate how the behaviour of well-behaved programs can go beyond the user's expectation. But before that, let us first recall (asymmetric) *lenses* [Foster et al. 2007; Stevens 2008] (the most popular bidirectional transformation model) and introduce its well-behavedness in Stevens's terminologies [Stevens 2008]. A lens is used to maintain a functional consistency relation R between two 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. Once there are changes in the source and (or) view, two functions $get : S \to V$ and $put : S \to V \to S$ are invoked to restore the consistency relation. A bidirectional transformation is well-behaved if it satisfies the following two properties (regarding the restoration behaviour of *put* with respect to *get*):

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

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

Correctness states that necessary changes must be made so that inconsistent data becomes consistent again, while hippocraticness says that if two pieces of data are already consistent, the restorer (*put*) must not make any change.

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

$$\begin{aligned}
&get :: (Int, String) \to Int\\
&get\ (i, s) = i\\
&put_1 :: (Int, String) \to Int \to (Int, String)\\
&put_1\ (i, s)\ i' = (i', s)
\end{aligned}
\qquad
\begin{aligned}
&put_2 :: (Int, String) \to Int \to (Int, String)\\
&put_2\ src\ i' \mid get\ s == i' = src\\
&put_2\ (i, s)\ i' \mid otherwise = (i', \texttt{""})
\end{aligned}$$

If the view is modified by the user, it is obvious that replacing the first element of the source with the view is enough — in other words, we should leave the second element as it is. This is exactly what put_1 does. However, there are many other possibilities regarding the update behaviour. For example, put_2 additionally sets the string empty (if the source and view are not consistent) — which retains nothing from the old source, but is also well-behaved with *get* since the string data contributes nothing to the consistency relation.

The root cause of the above wide range of *put* behaviour is that while lenses are designed to enable retention of source information, the only property guaranteeing it is hippocraticness, which only requires that the *whole* source should be unchanged if the *whole* view is (the first case of put_2). In other words, if we have a very small change on the view, we are free to create any source we like (the second case of put_2). This is too "global" in most cases, where we want a more "local"

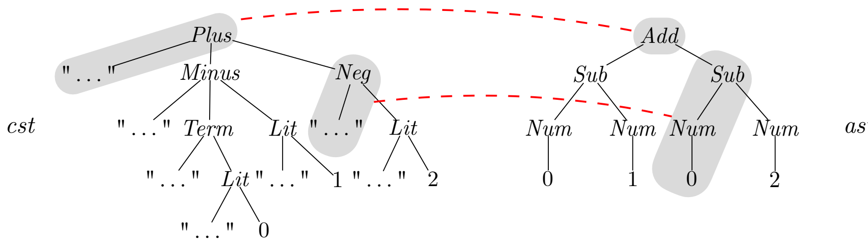


Fig. 1. Regions and consistency links.

hippocraticness property saying that if *parts* of the view are unchanged then the *corresponding parts* of the source should be retained as well.

[For this consideration, we propose a new *retentiveness* property to capture the local hippocraticness. Central to our formulation of retentiveness is to decompose sources and views into data fragments, and use *links* to relate those fragments.] A retentive *put* function accepts links as its additional input, and will produce a new source in a way that it preserves all the data fragments from the old source attached to the links. For the examples in [XX], a retentive *put* is guaranteed to preserve the string data in the tuple, by passing it a link connecting the string data. The main contributions of this paper can be summarised as follows.

- We propose a semantic framework of retentive lenses, in particular for algebraic data types. In our framework, Hippocraticness is subsumed by Retentiveness, and any (state-based) well-behaved lens can be lifted to a retentive lens (Section 2).
- To show that retentive lenses are feasible, we design a tiny but non-trivial domain-specific language, which is able to describe many interesting tree transformations. We first show a flavour of the language, by giving a solution to the problem in Figure 5. Then the syntax and semantics of the DSL are presented in order. We also prove that any program written in our DSL gives rise to a pair of retentive *get* and *put* (Section 3).
- To further show that retentiveness is useful, we present case studies in two different areas: resugaring [Pombrio and Krishnamurthi 2014, 2015] and code refactoring [Fowler and Beck 1999], showing that retentiveness simplifies the design of tools and provides additional guarantees (Section 4).

Detailed related work about various alignment strategies, provenance between two pieces of data, and operational-based BXs (Section 5). Conclusions and future work regarding composability and generalisation of the retentive lens framework come later (Section 6).

2 A SEMANTIC FRAMEWORK OF RETENTIVE LENSES

In this section we will develop a definition of retentive lenses, where we enrich the pair of functions in lenses and formalise the statement "if parts of the view are unchanged then the corresponding parts of the source should be retained" in a more abstract framework, and show that Hippocraticness becomes a consequence (Section 2.2). Before that, we will start from an intuitive description of a "region model" for tree transformations, and how a retentive lens operates on this model.

2.1 The Region Model

As mentioned in Section 1, the core idea of retentiveness is to use links to relate parts of the source and view. For trees, an intuitive notion of a "part" is a *region*, by which we mean a top portion of a subtree that can be described by a *region pattern* that consists of variables, constructors, and constants. For example, in Figure 1, the grey areas are some of the possible regions. The topmost

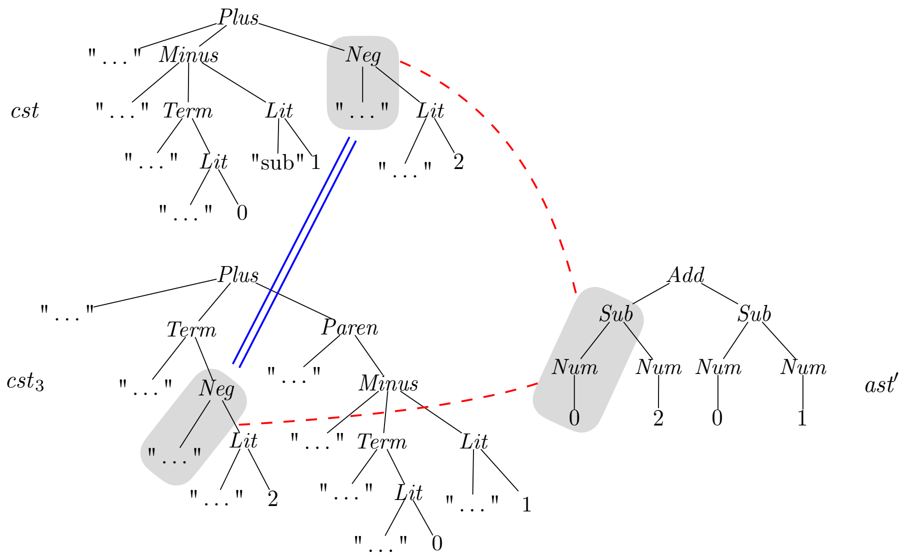


Fig. 2. The triangular guarantee.

region in *cst*, for example, is described by the region pattern '*Plus* "..." *a b*', which says that the region includes the *Plus* node and the annotation "...", i.e., the first subtree under *Plus*; the other two subtrees (with roots *Minus* and *Neg*) under *Plus* are not part of the region, but we give them names *a* and *b* (which will be used in Section 2.3). This is one particular way of decomposing trees, and we call this the *region model*.

Within the region model, the *get* function in a retentive lens not only computes a view but also decomposes both the source and the view into regions and produces a set of links relating source and view regions. We call this set of links produced by *get* the *consistency links*, because they are a finer-grained representation of how the whole source is consistent with the whole view. In Figure 1, for example, the red dashed lines are two of the consistency links between the source *cst* and the view *ast* = *getE cst*. With this particular *getE* function, the topmost region of pattern '*Plus* "..." *a b*' in *cst* corresponds to the topmost region of pattern '*Add a b*' in *ast*, and the region of pattern '*Neg* "..." *a*' in the right subtree of *cst* corresponds to the region of pattern '*Sub* (*Num* 0) *a*' in *ast*. Note that, for clarity, Figure 1 does not show all consistency links — the complete set of consistency links should fully describe how all the source and view regions correspond.

As the view is modified, the links between the source and view can also be modified to reflect the latest correspondences between regions. For example, if we change *ast* in Figure 1 to *ast′* in Figure 2 by swapping the two subtrees under *Add*, then the link relating the '*Neg* "..." *a*' region and the '*Sub* (*Num* 0) *a*' region is also maintained to record that the two regions are still "locally consistent" despite that the source and view as a whole are no longer consistent.

When it is time to put the modified view back into the source, the links that remain between the source and the view can provide valuable information regarding what should be brought from the old source to the new source. The *put* function of a retentive lens thus takes a set of links as additional input, and retains regions in a way that respects the links. More precisely, *put* should provide what we might call the *triangular guarantee*: Using Figure 2 to illustrate, if a link is provided as input to *put*, say the one relating the '*Neg* "..." *a*' region in *cst* and the '*Sub* (*Num* 0) *a*' region in ast' , then after updating cst to a new source cst_3 (which is guaranteed to be consistent with ast'), the consistency links between cst_3 and ast' should include a link that relates the 'Sub (Num 0) a' region to some region in the new source cst_3, and that region should have the same pattern as the one specified by the input link, namely 'Neg " . . . " a', preserving the negation (as opposed to changing it to a $Minus$, for example) and the associated annotation/comment.

2.2 Formalisation of Retentive Lenses

Recall that a classic lens [Foster et al. 2007] is a pair of functions $get : S \rightarrow V$ and $put : S \rightarrow V \rightarrow S$ satisfying two well-behavedness laws, Correctness $get\ (put\ s\ v) = v$ and Hippocraticness $put\ s\ (get\ s) = s$. The aim of this section is to make the following revisions to the definition:

- extending get and put to incorporate links — specifically, we will make get return a set of consistency links and put take a set of input links as an additional argument — and
- replacing Hippocraticness with a finer-grained law that captures the "triangular guarantee" described in Section 2.1 and subsumes Hippocraticness.

We will start by introducing some mathematical notations we use in this paper.

2.2.1 Preliminaries. In this section and the next we use sets, total functions, and relations as our mathematical foundation. We will write $r : A \sim B$ to denote that r is a relation between sets A and B, i.e., $r \subseteq A \times B$. Given a relation $r : A \sim B$, its "left domain" $\textsc{ldom}\ r$ is the set $\{\, x \in A \mid \exists y \in B.\ (x, y) \in r \,\}$, and its converse $r^\circ : B \sim A$ relates $b \in B$ and $a \in A$ exactly when r relates a and b. The composition $l \cdot r : A \sim C$ of two relations $l : A \sim B$ and $r : B \sim C$ is defined as usual: $(x, z) \in l \cdot r$ exactly when there exists y such that $(x, y) \in l$ and $(y, z) \in r$. We will allow functions to be implicitly lifted to relations: a function $f : A \rightarrow B$ also denotes a relation $f : B \sim A$ such that $(f\ x, x) \in f$ for all $x \in A$. This flipping of domain and codomain (from $A \rightarrow B$ to $B \sim A$) makes functional composition compatible with relational composition: a function composition $f \circ g$ lifted to a relation is the same as $f \cdot g$, i.e., the composition of f and g as relations.

To allow more precision, we will use some notations from dependent type theory. Let P be a family of sets indexed by A, and $P(x)$ the set at index $x \in A$ in the family. We write $(x \in A) \rightarrow P(x)$ to denote the set of dependent functions f that maps every $x \in A$ to $f\ x \in P(x)$. Also we write $(x \in A) \times P(x)$ to denote the set of dependent pairs (x, y) where $x \in A$ and $y \in P(x)$.

2.2.2 Abstracting Links. We will develop a series of definitions abstracting region patterns, regions, links, and sets of links, so as to arrive at a framework that does not specifically talk about the region model but still captures the essence of retentiveness. We will then be able to enrich get and put to make them handle links.

From region patterns to properties. A rough reading of the triangular guarantee described in Section 2.1 is that a fragment in the old source linked to the view should also appear in the new source. An alternative and more abstract way of understanding this is that we are preserving a kind of property that says "a tree includes a specific region pattern": if an old source has a fragment matching a region pattern, then the new source should also have a fragment matching the same region pattern. More generally, there may be other kinds of property that we want to preserve. (We will see another kind of property in Section 2.3.)

Intuitively, a property is a description that can be satisfied. Below is a more proof-relevant formulation that we will need:

Definition 2.1 (properties). A *property* on a set X is a value p equipped with a family of sets $x \models p$ where x ranges over X.

Inhabitants of $x \models p$ are considered proofs that x satisfies p. For example, given a tree t, a region pattern *pat* (Section 2.1) can be regarded as a property by equipping it with a family of sets:

$$t \models pat \quad = \quad (path \in Path) \times LeadToRegion(t, path, pat)$$

where $LeadToRegion(t, path, pat)$ is the set of proofs that *path* is a valid path from the root of t to a sub-tree which matches *pat*.

Regions as properties with proofs. We have seen that region patterns can be cast as properties: given a region pattern, a proof of the corresponding property is a path pointing to a region described by the region pattern. Now consider regions, which are fragments matching a region pattern at a specific location. This suggests that a region corresponds to a pair of a property and a proof that the property is satisfied. For brevity, we give a definition of the sets of such pairs of property and proof:

Definition 2.2. Let P be a set of properties on X and $x \in X$. Define:

$$P_x \quad = \quad (p \in P) \times (x \models p)$$

That is, an inhabitant of P_x is a property and a proof that x satisfies the property. We will sometimes refer to P_x as the set of properties provably satisfied by x.

Sets of links as relations between properties with proofs. In Section 2.1, a link connects a region in the source and another in the view; that is, it is a pair of source and view regions. We have seen how regions are abstractly represented: given a source s, the set of regions in s is abstractly represented by P_s where P is the set of all possible region patterns (regarded as properties) for sources; similarly, the set of regions in a view v is abstractly represented by Q_v where Q is the set of all possible region patterns for views. It follows that, abstractly, a link should be a pair in the set $P_s \times Q_v$, and a set of links is a relation between P_s and Q_v.

Enriching get and put with links. Now we can enrich the type of *get* to:

$$get \ : \ (s \in S) \ \rightarrow \ (v \in V) \times (P_s \sim Q_v)$$

That is, upon receiving a source $s \in S$, *get* will produce a view $v \in V$ and a relation between the regions in s and v.

It is tempting to do exactly the same for *put*, assigning it the type:

$$(s \in S) \times (v \in V) \times (P_s \sim Q_v) \rightarrow S$$

This is not enough, however, because not all triples in the above domain are valid inputs. In particular, for our syntax tree example (Section 2.1), an arbitrary input link might relate inconsistent regions, e.g., a source region of pattern '*Neg* "..." *a*' and a view region of pattern '*Add a b*'; in this case there is no hope to deliver the triangular guarantee since such invalid links cannot appear among the consistency links produced by *get*. In general, we usually do not want to consider all triples in the above domain, and only want to operate on a subset consisting of those sources and views that are "not too inconsistent" and those links that are valid for the sources and views, such that consistency restoration is possible. Inspired by Diskin et al. [2011b], we call this subset of triples a *triple space*.

Definition 2.3 (triple spaces). A *triple space* T between a source set S with properties P and a view set V with properties Q is a set such that:

$$T \quad \subseteq \quad (s \in S) \times (v \in V) \times (P_s \sim Q_v)$$

The definition of a retentive lens will start from specifying a triple space T, which determines the range of triples of source, view, and links that the lens operates on; the domain of *put* will be set to T, and *get* will be required to produce triples (the input sources included) only in T.

2.2.3 Uniquely Identifying Property Sets. One important aim of retentive lenses is to make Hippocraticness an extreme case of the triangular guarantee (Section 2.1). Having developed an abstract framework in Section 2.2.2, we can now give a quick sketch of how we can derive Hippocraticness. Recall that the triangular guarantee roughly says that a set of input properties satisfied by the old source will also be satisfied by the new source. The more input properties there are, the more restrictions we put on the new source. In the extreme case, if the set of input properties can only be satisfied by at most one source, then the new source has to be the same as the old source since it is required to satisfy the same properties. We will call such set of properties *uniquely identifying*, and require that the set of source properties mentioned by the links produced by *get* should be uniquely identifying.

Definition 2.4 (satisfaction of property sets). Let P be a set of properties on X.

$$x \models^* P \quad = \quad (p \in P) \to (x \models p)$$

That is, $x \models^* P$ is the set of proofs that x satisfies all properties in P, since an inhabitant of $x \models^* P$ is a function that maps every $p \in P$ to an inhabitant of $x \models p$.

Definition 2.5 (uniquely identifying property sets). A set P of properties on X is *uniquely identifying* exactly when

$$x \models^* P \text{ is inhabited} \ \wedge\ x' \models^* P \text{ is inhabited} \quad \Rightarrow \quad x = x'$$

That is, there is at most one element of X satisfying all properties in P.

2.2.4 Retentive Lenses. We now have all the ingredients for the formal definition of retentive lenses, as well as the proof that retentive lenses satisfy Hippocraticness.

Definition 2.6 (retentive lenses). Let T be a triple space between a source set S with properties P and a view set V with properties Q. A *retentive lens* on T is a pair of functions *get* and *put* of type:

$$\begin{aligned} &get : (s \in S) \to (v \in V) \times (P_s \sim Q_v) \\ &put : T \to S \end{aligned}$$

such that

$$get \subseteq T \tag{1}$$

$$get\ s = (v, l) \quad \Rightarrow \quad \textsc{ldom}(fst \cdot l) \text{ is uniquely identifying} \tag{2}$$

$$get\ (put\ (s, v, l)) = (v', l') \quad \Rightarrow \quad v = v' \ \wedge\ fst \cdot l \subseteq fst \cdot l' \tag{3}$$

where $fst : (a \in A) \times B(a) \to A$ (for any set A and family of sets B indexed by A) is the first projection function.

What is new here is (3): on the right of the implication, the left conjunct is Correctness (PutGet), and the right conjunct, which we may call *Retentiveness*, formalises the triangular guarantee in a compact way. We can expand Retentiveness pointwise to see that it indeed specialises to the triangular guarantee.

Proposition 2.7 (triangular guarantee). *Given a retentive lens, let* $(s, v, l) \in T$, $s' = put\ (s, v, l)$, *and* $((p, m), (q, n)) \in l$. *Then* $get\ s' = (v, l')$ *for some* $l' : P_{s'} \sim Q_v$, *and there exists* $m' \in (s' \models p)$ *such that* $((p, m'), (q, n)) \in l'$.

Interpreting this in the region model: Suppose that there is an input link relating two regions (p, m) in the old source and (q, n) in the view, where p and q are region patterns and m and n contain the positions of the regions. Then *get* must produce a link relating a region (p, m') in the new source and the view region (q, n), meaning that, in the new source, the region pattern p appears at some location (i.e., m') that is consistent with (q, n).

We then prove Hippocraticness with the help of two simple lemmas.

Lemma 2.8. *For a retentive lens,*

$$get\ s = (v, l) \quad \Rightarrow \quad s \models^* \textsc{ldom}(fst \cdot l) \text{ is inhabited}$$

Proof. Suppose that $get\ s = (v, l)$, where $l : P_s \sim Q_v$. For every $p \in \textsc{ldom}(fst \cdot l)$, by the definitions of left domain and relational composition, there exists $(p', m) \in P_s$ and $(q, n) \in Q_v$ such that $p' = p$ and $((p', m), (q, n)) \in l$ — note that this implies $m \in (s \models p)$. This maps every $p \in \textsc{ldom}(fst \cdot l)$ to an inhabitant of $s \models p$, and therefore constitutes an inhabitant of $s \models^* \textsc{ldom}(fst \cdot l)$. □

Lemma 2.9. *Let P and P' be sets of properties on X and $x \in X$. If $x \models^* P$ is inhabited and $P' \subseteq P$, then $x \models^* P'$ is also inhabited.*

Proof. An inhabitant of $x \models^* P'$ is a function of type $(p \in P') \to (x \models p)$, and can be obtained by restricting the domain of an inhabitant of $x \models^* P$, which is a function of type $(p \in P) \to (x \models p)$. □

Theorem 2.10 (Hippocraticness). *For a retentive lens,*

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

Proof. Supposing that $get\ s = (v, l)$ and $put\ (s, v, l) = s'$ for some s', our goal is to prove $s = s'$. By (3), $get\ s' = (v, l')$ for some l' such that $fst \cdot l \subseteq fst \cdot l'$. Applying $\textsc{ldom}$ to both sides of the inclusion yields

$$\textsc{ldom}(fst \cdot l) \subseteq \textsc{ldom}(fst \cdot l') \tag{4}$$

since $\textsc{ldom}$ is monotonic. By Lemma 2.8, both $s \models^* \textsc{ldom}(fst \cdot l)$ and $s' \models^* \textsc{ldom}(fst \cdot l')$ are inhabited. Moreover, by Lemma 2.9, $s' \models^* \textsc{ldom}(fst \cdot l)$ is also inhabited. Finally, since $\textsc{ldom}(fst \cdot l)$ is uniquely identifying by (2), we obtain $s = s'$ as required. □

Classic lenses as retentive lenses. As a simple example, we show that every well-behaved lens can be turned into a retentive lens. The idea is that a classic lens corresponds to a retentive lens that guarantees to retain the whole source or nothing at all. The retentive lens will be set up such that the only way to form links between a source and a view is to link the whole source with the whole view if they are consistent. If the view is unchanged, the link can remain intact, and the whole source is retained; otherwise, if the view is changed in any way, the link is broken, and the retentive *put* does not have to guarantee that anything in the old source is retained.

Formally: Let $g : S \to V$ and $p : S \to V \to S$ be a well-behaved lens. The property sets associated with S and V are S and V themselves respectively; given $s \in S$ (resp. $v \in V$), the set $x \models s$ (resp. $x \models v$) is inhabited by a single element '$*$' if $x = s$ (resp. $x = v$) or uninhabited otherwise. The triple space T consists of triples (s, v, l) such that $fst \cdot l \cdot fst^\circ \subseteq g^\circ$ — that is, a link can only exist between s and $g\ s$. Now we define:

$$\begin{array}{lcl} get\ s & = & (g\ s, \{((s, *), (g\ s, *))\}) \\ put\ (s, v, l) & = & p\ s\ v \end{array}$$

It is easy to verify (1), (2), and the first part of (3) (which is just PutGet). For the second part of (3): If l is empty, the conclusion holds trivially. Otherwise, l can only be $\{((s, *), (g\ s, *))\}$, implying that $v = g\ s$. We then have $put\ (s, v, l) = p\ s\ (g\ s) = s$ by GetPut, and $(v', l') = get\ (put\ (s, v, l)) = get\ s = (g\ s, \{((s, *), (g\ s, *))\})$. Therefore $l' = \{((s, *), (g\ s, *))\} = l$, which implies $fst \cdot l \subseteq fst \cdot l'$.

2.3 Retentive Lenses for the Region Model

Driven by the region model, we have arrived at a definition of retentive lenses (Definition 2.6) and seen that it entails Hippocraticness (Theorem 2.10). This definition does not immediately work for the region model, however. This is because region patterns by themselves cannot form uniquely identifying property sets (Definition 2.5) — requiring that a source must contain a set of region patterns does not uniquely determine it, however complete the set of region patterns is. This can be intuitively understood by analogy with jigsaw puzzles: a picture cannot be determined by only specifying what jigsaw pieces are used to form the picture (if the jigsaw pieces can be freely assembled); to fully solve the puzzle and determine the picture, we also need to specify how the jigsaw pieces are assembled — that is, the relative positions of all the jigsaw pieces.

Below we will introduce *second-order property sets* into the theory so that it is possible to form uniquely identifying property sets in the region model. We will also show that Definition 2.6 and Theorem 2.10 only loosely depend on the specific structure of property sets, and can be adapted for second-order property sets with almost no structural change. The general idea is that while second-order property sets have more internal constraints, they are still sets of properties, so the ideas behind the formal definitions for retentive lenses remain more or less the same.

2.3.1 Uniquely Identifying Property Sets for the Region Model. For brevity, we will use the *Arith* data type in the examples below (even though *Arith* is used elsewhere as a view type, on which we do not need property sets to be uniquely identifying). We have been using property sets as conjunctive predicates. For example, the property set $\{Add\ a\ b, Sub\ a\ b\}$ is used as a predicate $(t \models Add\ a\ b) \wedge (t \models Sub\ a\ b)$ on t, saying that t should contain two regions, one of pattern $Add\ a\ b$ and the other of pattern $Sub\ a\ b$. If we want to additionally specify the relative position of these two regions — or more abstractly, how the proofs of the two properties relate — we need to switch to a dependent conjunction like $(r \in (t \models Add\ a\ b)) \wedge (r' \in (t \models Sub\ a\ b)) \wedge Child(Add\ a\ b, a, r, r')$, where $Child(Add\ a\ b, a, r, r')$ says that the path in r' is the one in r extended with one step that navigates from the root of the region of pattern $Add\ a\ b$ to the subtree named a — that is, the region of pattern $Sub\ a\ b$ is directly below the region of pattern $Add\ a\ b$ at the position named a. (This form of *Child* is in fact slightly simplified; in our formalisation below, *Child* will need to take more arguments.) If a tree t satisfies this predicate, it means that t contains the two specified regions and, furthermore, the *Sub* region is the left child of the *Add* region. We call properties like '*Child*' *second-order properties*, which can refer to proofs of other properties.

With second-order properties capable of expressing relative position, it is now possible for a set of properties to be uniquely identifying. For example, consider the following set of properties (expressed as a conjunctive predicate):

$$\begin{aligned}
&(r_0 \in (t \models Num\ 0)) \wedge (r_1 \in (t \models Num\ 1)) \wedge (r_2 \in (t \models Num\ 2)) \wedge \\
&(r_3 \in (t \models Add\ a\ b)) \wedge (r_4 \in (t \models Add\ a\ b)) \wedge \\
&Top(r_3) \wedge Child(Add\ a\ b, a, r_3, r_4) \wedge Child(Add\ a\ b, a, r_4, r_0) \wedge \\
&Child(Add\ a\ b, b, r_4, r_1) \wedge Child(Add\ a\ b, b, r_3, r_2)
\end{aligned} \tag{5}$$

where *Top* says that a region is the topmost one in a tree (that is, there is no other region above the named region). This property set is satisfied by exactly one tree, namely:

$Add\ (Add\ (Num\ 0)\ (Num\ 1))\ (Num\ 2)$

It is also possible to specify a subset of these properties to be retained when the above tree is updated (as long as we are careful not to refer to regions that are left out of the subset), like:

$$(r_0 \in (t \models Num\ 0)) \wedge (r_3 \in (t \models Add\ a\ b)) \wedge (r_4 \in (t \models Add\ a\ b)) \wedge$$

$$Child(Add\ a\ b, a, r_3, r_4) \land Child(Add\ a\ b, b, r_3, r_2)$$

An updated tree satisfying this property will still have a region of pattern $Add\ (Add\ (Num\ 0)\ a)\ b$, but new regions can be added above it (because we no longer require $Top(r_3)$) or as its subtrees (because we no longer specify the right child of r_3 and r_4). Note that although it may be tempting to express the property set (5) equivalently and more simply as the predicate $t \models Add\ (Add\ (Num\ 0)\ (Num\ 1))\ (Num\ 2)$, this simpler predicate is monolithic and cannot be partially retained.

2.3.2 Formalisation of Second-Order Properties. The theory becomes more complicated when second-order properties are involved. We have to deal with references to proofs of other properties, and be careful about validity of references. Satisfaction of a second-order property set is no longer just satisfaction of individual properties, because whether a second-order property is satisfied depends on which proofs are supplied to show that other properties are satisfied. To avoid unnecessary complication, we will formulate our definitions just for the structure we need: a set of (named) region patterns, whose proofs (locations of matching regions) are referred to by either a unary predicate (*Top*) or a binary one (*Child*). The theory below is still somewhat obscure though, and can be safely skipped on first reading.

Definition 2.11 (second-order property sets). A *second-order property set* on a set X is the union of three disjoint sets:

- a set P_0 of properties on X,
- a set P_1 equipped with a function $(-)^* : P_1 \to P_0$ such that every $p_1 : P_1$ is a property on $(x : X) \times (x \models p_1^*)$, and
- a set P_2 equipped with two functions $(-)^* : P_2 \to P_0$ and $(-)^{**} : P_2 \to P_0$ such that every $p_2 : P_2$ is a property on $(x : X) \times (x \models p_2^*) \times (x \models p_2^{**})$.

To express the property set (5) in our example, we can take the set P_0 to be the cartesian product of a name set (consisting of names like r_0, r_1, etc) and the set of region patterns, and the set P_1 to be properties of the form $Top(r, pat)$ where $(r, pat) \in P_0$. (We omit discussion of P_2, which consists of binary predicates (like *Child* in our example) and is analogous to P_1.) The function $(-)^*$ maps a P_1-property to the P_0-property it refers to — for example, $(Top(r, pat))^* = (r, pat)$. With this function, we can then say that any $p_1 \in P_1$ is a property on the proof of the property it refers to, namely p_1^*. We need to name the properties in P_0 because the same region pattern may appear multiple times, and we want to refer to the different regions matching the same pattern — in the property set (5), we give two different names r_3 and r_4 to two regions of the same pattern $Add\ a\ b$, so that the two regions can be properly referred to by the second-order properties.

The usual way to proceed from this point is to define when a subset of a second-order property set (representing a dependent conjunction like (5)) is well-formed, making sure that a P_1- or P_2-property does not refer to a P_0-property not included in the subset, and then define its proofs. We will take a slightly quicker route, directly defining a well-formed proof that a subset of a second-order property set on X is satisfied by some $x \in X$. Such a proof is a subset of the properties provably satisfied by x such that references are made correctly within the subset.

Definition 2.12. Let $x : X$ and $P = P_0 \uplus P_1 \uplus P_2$ be a second-order property set on X. The set P_x (properties provably satisfied by x) is defined by

$$\begin{aligned} P_x = {} & (p_0 : P_0) \times (x \models p_0) \\ & \uplus (p_1 : P_1) \times (m : x \models p_1^*) \times ((x, m) \models p_1) \\ & \uplus (p_2 : P_2) \times (m : x \models p_2^*) \times (m' : x \models p_2^{**}) \times ((x, m, m') \models p_2) \end{aligned}$$

Definition 2.13. Let $x : X$ and $P = P_0 \uplus P_1 \uplus P_2$ be a second-order property set on X. A subset A of P_x is *well-formed* exactly when:

- for every $p_0 : P_0$, if $(p_0, m) \in A$ and $(p_0, m') \in A$, then $m = m'$;
- for every $p_1 : P_1$, if $(p_1, m, n) \in A$, then $(p_1^*, m) \in A$;
- for every $p_2 : P_2$, if $(p_2, m, m', n) \in A$, then $(p_2^*, m) \in A$ and $(p_2^{**}, m') \in A$.

Definition 2.14. Let P' be a subset of a second-order property set P on X and $x \in X$. The inhabitants of the set $x \models^* P'$ are exactly the well-formed subsets A of P_x such that $fst[A] = P'$, i.e., the image of A under fst is P'.

Adapting Retentive Lenses for Second-Order Properties. We have redefined the set of provably satisfied properties (Definition 2.12), and for a subset P' of a second-order property set, redefined the set $x \models^* P'$ of proofs that x satisfies all properties in P' (Definition 2.14). To make retentive lenses work with second-order properties, and also make the proof of Theorem 2.10 to go through, we need one final revision to the definition of triple spaces.

Definition 2.15 (second-order triple spaces). A *second-order triple space* T between a source set S with second-order properties P and a view set V with second-order properties Q is a set such that:

$$T \quad \subseteq \quad (s \in S) \times (v \in V) \times (P_s \sim Q_v)$$

and for every $(s, v, l) \in T$, both $\textsc{ldom}\ l$ and $\textsc{ldom}(l^\circ)$ are well-formed (Definition 2.13).

We can now substitute these new definitions about property sets for the old ones used in the definition of retentive lenses (Definition 2.6). As for the proof of Theorem 2.10, which relies on Lemmas 2.8 and 2.9: the statement of Lemma 2.8 is valid for the new definitions about second-order property sets, and Lemma 2.9 needs an additional assumption that $y \models P'$ should be inhabited for some y (to ensure that P' is "syntactically well-formed"). Theorem 2.10 still holds despite the addition of the assumption, because the assumption is met ($s \models \textsc{ldom}(fst \cdot l)$ is inhabited) when using Lemma 2.9 in the proof.

3 A DSL FOR RETENTIVE BIDIRECTIONAL TREE TRANSFORMATION

With theoretical foundations of retentive lenses in hand, we shall propose a domain specific language for easily describing them. Our DSL is designed to be simple, and yet able to handle many bidirectional tree transformations, which may be used in resugaring and bidirectional refactoring[1]. In our DSL, users basically write a *get* function in the form of inductively-defined consistency relations between sources and views, and a *put* function will be generated to pair with the *get* forming a retentive lens. [in sec XXX]

3.1 A Flavour of the DSL

To give a flavour of the language, let us consider the synchronisation between concrete and abstract syntax trees (CSTs and ASTs), which are represented by the algebraic data types defined in Figure 3. The CSTs can be either expressions (*Expr*, containing additions and subtractions) or terms (*Term*, including numbers, negated terms and expressions in parentheses), and all constructors have an annotation field (*Annot*); in comparison, the ASTs do not include explicit parentheses, negations, and annotations. Consistency between CSTs and ASTs is given directly in terms of the *get* functions in Figure 3.

This ("degenerated") consiscenty relation (i.e. the *get* function) can be expressed using our DSL as Figure 4 shows. (The data type definitions written in our DSL remains the same and are thus

[1]We will explain this in more detail in Section 4 as case studies.

omitted.) Similar to the *get* functions in Figure 3, here we describe two consistency relations: 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 name appearing the left-hand side and right-hand side are consistent, then the pair of trees constructed from these subtrees are also consistent. Take $Plus\ \_\ x\ y \sim Add\ x\ y$ for example. It means that if x_s is consistent with x_v, and y_s is consistent with y_v, then $Plus\ s\ x_s\ y_s$ and $Add\ x_v\ y_v$ are consistent for any value s, where s corresponds to a "don't-care" wildcard in $Plus\ \_\ x\ y$. So the meaning of $Plus\ \_\ x\ y \sim Add\ x\ y$ can be defined more formally by the following "derivation" rule:

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

Now we briefly explain its semantics. In the forward direction, its behaviour is quite similar to the first case of *getE* in Figure 3, except that it also updates the links between subtrees of the source and the view and establishes a new one between the top nodes recording the correspondence. In the backward direction, it creates a tree whose top node is $Plus\ \_\ \_\ \_$ and recursively builds the subtrees of $Plus\ \_\ \_\ \_$, provided that the view matches $Add\ \_\ \_$ and there is no link connected to the top of the view. The behaviour will be more subtle if there is some links connected to the top of the view, since *put* needs to take retentiveness into account. We leave the detailed description to the later part. In this way, each consistency relation is assigned a *get* and a *put* function, and each inductive rule of the consistency relation contributes one case to the two functions[2].

Suppose that we have a consistent pair of *cst* and *ast* shown in Figure 5, where *cst* represents "$0 - 1 + -2$" and its sub-term "-2" is desugared as "$0 - 2$" in *ast*. If *ast* is modified into *ast′* (also in Figure 5), where the two non-zero numbers are exchanged, there is not much restriction on what a well-behaved *put* should do, and we could have a wide range of *put* behaviour, producing, as listed on the right-hand side of Figure 5,

- cst_1, where the literals 1 and 2 are swapped, along with their annotations; or
- cst_2, which represents the same expression as cst_1 except that all annotations are removed; or
- cst_3, which represents "$-2 + (0 - 1)$" and retains all annotations, in effect swapping the two subtrees under *Plus* and adding two constructors, *Term* and *Paren*, to satisfy the type constraints; or
- cst_4, where the negation syntactic sugar "-2" gets lost after the update.

Among them, cst_3 might be the most expected result because the user (or the tool) indeed swapped the two subtrees of *Plus* in *ast*. However, in practice, a well-behaved *put* generated from existing bidirectional languages (tools) has a very high possibility to return cst_1 or cst_2.

3.2 Syntax

Figure 6 gives the syntax of the language. A program consists of two main parts. The first part is definitions of simple *algebraic data types*, where an algebraic data type is defined through a set of data constructors $C_1 \ldots C_k$, similar to those in functional languages such as Haskell. The second part is definitions of *consistency relations* between these data types, in which each consistency relation consists of a relation type declaration and a set of inductively-defined rules. The relation type declaration is in the form of $srcType \longleftrightarrow viewType$, representing the source and view types for the synchronisation. Each inductive rule is defined as a relation between source and view patterns $pat_s \sim pat_v$, where a pattern consists of variables, constructors, and wildcards, as we have already seen in the examples of Section 3.1.

[2]Precisely speaking, it contributes more than one cases to *put*. We will it later.

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

data Arith = Add Arith Arith
           | Sub Arith Arith
           | Num Int
```

Fig. 3. Data types for concrete and abstract syntax trees and the consistency relation between them as two *get* functions in Haskell.

```
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
  Term  _ t   ~ t       ;            Paren _ e ~ e             ;
```

Fig. 4. A program in our DSL for synchronising ADTs in Figure 3.

One might be wondering why we do not give a name to a consistency relation. This is because we assume, without loss of generality, that there is only one consistency relation between any two types. In fact, if we want to define multiple consistency relations between two types, we can indirectly achieve this by giving different names to the view types (like type synonyms), one name for writing one consistency relation.

3.2.1 Syntactic Restrictions. We shall impose several syntactic restrictions to the DSL for guaranteeing that a consistency relation $T_s \longleftrightarrow T_v$ indeed implies a view generation function $T_s \longrightarrow T_v$.

- On *variables*, we assume *linear variable usage*. The variable usage must be linear in the sense that a variable must appear exactly once in each side of an inductive rule.
- On *patterns*, we assume two restrictions: *pattern coverage* and *source pattern disjointness*. Pattern coverage guarantees totality: for a consistency relation, all the patterns on the left-hand sides should cover all the possible cases of the source type T_s, and all the patterns on the right-hand side should cover all the cases of the view type T_v. Source pattern disjointness requires that all of the source patterns in a consistency relation should be disjoint, so that at most one pattern can be matched when running *get*. This implies that the program converting a source to its view is indeed a function (regardless of the order of the inductive rules).
- On *algebraic data types*, we assume *interconvertible data types*, for allowing *put* to switch between sources of different types while retaining the core information. If we want to define two consistency relations sharing the same view type, say $T_{s1} \longleftrightarrow T_v$ and $T_{s2} \longleftrightarrow T_v$, we require that the two source types, T_{s1} and T_{s2}, should be interconvertible. For instance, since the two consistency relations *Expr* $\longleftrightarrow$ *Arith* and *Term* $\longleftrightarrow$ *Arith* in Figure 4 share the same view type *Arith*, we should be able to convert *Expr* to *Term* and vice versa. Syntactically

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

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

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

?

```
cst1 =
  Plus "an add"
    (Minus "a sub"
      (Term "a term" (Lit "0 in sub" 0))
      (Lit "1 in sub" 2))
    (Neg "a neg" (Lit "2 in neg" 1))
```

```
cst2 =
  Plus ""
    (Minus ""
      (Term "" (Lit "" 0))
      (Lit "" 2))
    (Neg "" (Lit "" 1))
```

```
cst3 =
  Plus "an add"
    (Term ""
      (Neg "a neg" (Lit "2 in neg" 2)))
    (Paren ""
      (Minus "a sub"
        (Term "a term" (Lit "0 in sub" 0)))
        (Lit "1 in sub" 1))
```

```
cst4 =
  Plus ""
    (Minus "" (Term "" (Lit "" 0))
              (Lit "" 2))
    (Paren ""
      (Minus "" (Term "" (Lit "" 0))
                (Lit "" 1)))
```

Fig. 5. *CST* is updated in many ways in response to a change on AST.

Program

$prog \ ::= \ tDef_1 \ \ldots \ tDef_m \ cDef_1 \ \ldots \ cDef_n$

Type Definition

$$
\begin{array}{llllll}
tDef & ::= & \text{data } type & = & C_1 \ type_{11} \cdots type_{1m_1} & \\
 & & & | & \ldots & \\
 & & & | & C_k \ type_{11} \cdots type_{1m_k} &
\end{array}
$$

Consistency Relation Definition

$$
\begin{array}{llll}
cDef & ::= & type \longleftrightarrow type & \{\text{relation type}\} \\
 & & r_1; \ \ldots \ r_n; & \{\text{inductive rules}\}
\end{array}
$$

Inductive Rule

$r \ ::= \ pat_1 \sim pat_2$

Pattern

$$
\begin{array}{llll}
pat & ::= & v & \{\text{variable pattern}\} \\
 & | & \_ & \{\text{don't-care wildcard}\} \\
 & | & C \ pat_1 \cdots pat_n & \{\text{constructor pattern}\}
\end{array}
$$

Fig. 6. Syntax of the DSL.

speaking, a type T_1 can be converted into T_2, if the type definition of T_2 contains a case $C_i \ type_1 \ \ldots \ T_1 \ \ldots \ type_n$, and the consistency relation of $T_2 \longleftrightarrow T_v$ contains a projection rule $C_i \ \_ \ \ldots \ x \ \ldots \ \_ \sim x$, for wrapping a piece of data of type T_1 into that of type T_2.

3.2.2 Programming Examples. Although being tiny, the DSL is able to describe many interesting bidirectional tree transformations. Here are some examples (including data type definitions):

- The *mirror* transformation, which swaps all the subtrees of a given tree in both *get* and *put* directions:

$$\begin{array}{ll} \mathit{BinT}\ \mathit{Int} \longleftrightarrow \mathit{BinT}\ \mathit{Int} & \qquad \mathbf{data}\ \mathit{BinT}\ a = \\ \quad \mathit{Tip} \qquad\quad\ \sim \mathit{Tip} & \qquad\qquad \mathit{Tip} \\ \quad \mathit{Node}\ i\ x\ y \sim \mathit{Node}\ i\ y\ x & \qquad\quad\ \mid \mathit{Node}\ a\ (\mathit{BinT}\ a)\ (\mathit{BinT}\ a) \end{array}$$

- The *spine* transformation, which collects the information carried by the left-most child of a rose tree in a list:

$$\begin{array}{ll} \mathit{RTree} \longleftrightarrow [\mathit{Int}] & \qquad \mathbf{data}\ \mathit{RTree}\ a = \\ \quad \mathit{RNode}\ i\ [\,] \qquad \sim [i] & \qquad\qquad \mathit{RNode}\ a\ (\mathit{List}\ (\mathit{RTree}\ a)) \\ \quad \mathit{RNode}\ i\ (x : \_) \sim i : x & \qquad \mathbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a) \end{array}$$

- The *map* transformation, which applies a bidirectional transformation to all the nodes in a binary tree:

$$\begin{array}{ll} \mathit{BinT}\ \mathit{Int} \longleftrightarrow \mathit{BinT}\ \mathit{Bool} & \qquad \mathit{Int} \longleftrightarrow \mathit{Bool} \\ \quad \mathit{Tip} \qquad\qquad \sim \mathit{Tip} & \qquad\quad (\lambda i \rightarrow \mathbf{if}\ i > 0\ \mathbf{then}\ \mathit{True}\ \mathbf{else}\ \mathit{False}) \\ \quad \mathit{Node}\ x\ \mathit{ls}\ \mathit{rs} \sim \mathit{Node}\ x\ \mathit{ls}\ \mathit{rs} & \qquad\quad (\lambda i\ b \rightarrow \mathbf{case}\ b\ \mathbf{of} \\ & \qquad\qquad\qquad \mathit{True} \rightarrow \mathbf{if}\ i > 0\ \mathbf{then}\ i\ \mathbf{else}\ 1 \\ & \qquad\qquad\qquad \mathit{False} \rightarrow \mathbf{if}\ i \leqslant 0\ \mathbf{then}\ i\ \mathbf{else}\ 0) \end{array}$$

We leave more interesting and practical case studies on bidirectional refactoring and resugaring to Section 4.

3.3 Semantics

In this subsection, we give a denotational semantics of our DSL by specifying the corresponding *get* and *put* as mathematical functions. Our language is also implemented as a compiler transforming programs in our DSL to *get* and *put* functions in Haskell. Our Haskell implementation directly follows the semantics given here.

Because our *put* function and its implementation will respect relative positions of regions by default—no matter whether they are required by the user or not, a design choice we made to simplify *get* and *put* to only produce and accept links between regions—so that the user will not be bothered with specifying links for second-order properties. In Section 3.4, we will prove that the *get* and *put* functions given below, when lifted to a version also producing and accepting second-order properties, actually form a retentive lens.

3.3.1 Types and Patterns. To define the semantics of *get* and *put*, we first need a semantics of type declarations and patterns in our DSL. For type declarations, their semantics is the same as in Haskell and thus we will not elaborate here. For each defined algebraic data type T, we will also use T to refer to the set of values of type T. For patterns in our DSL, we will use SPat to refer to the set of all patterns on S for every type S. For a pattern $p \in \mathit{SPat}$, $\mathit{Vars}\ p$ will be the set of variables in p and $\mathit{TypeOf}\ \ p\ v$ will be the set corresponding to the type of v in pattern p. We will also use Pat to denote the tagged union of patterns for all types, that is, $\mathit{Pat} = \{\, (S,\ sp) \mid S \text{ is a type},\ sp \in \mathit{SPat}\,\}$.

To manipulate patterns, we will use the following functions:

$$\begin{aligned} \mathit{isMatch}_S &: (p \in \mathit{SPat}) \rightarrow S \rightarrow \mathit{Bool} \\ \mathit{decompose}_S &: (p \in \mathit{SPat}) \rightarrow S \rightarrow \big((v \in \mathit{Vars}\ p) \rightarrow \mathit{TypeOf}\ \ p\ v\big) \end{aligned}$$

$$reconstruct_S : (p \in SPat) \to \big((v \in Vars\ p) \to TypeOf\ \ p\ v\big) \to S$$

isMatch p s tests if the value s matches the pattern p. If they match, *decompose* p s will result in a function f from every variable v in p to the corresponding subtree of s. Conversely, *reconstruct* p f produces a value s matching the pattern p by replacing every $v \in Vars\ p$ in p with $f\ \ v$, provided that p does not contain any wildcards. Since the semantics of patterns in our DSL is rather standard, we will also omit formal definitions of these functions.

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

$$get_{SV} : S \to V \times (Region \sim Region)$$

where *Region* is $Pat \times Path$, and *Path* is the set of lists of integers, which encode paths in trees. The body of get_{SV} is:

$$\begin{aligned}
get_{SV}\ \ s = \mathbf{let}\ & spat_k = selectPat\ \ s\ R \\
& f = decompose_S\ spat_k\ \ s \\
& vs = get \circ f \\
& links = \bigcup\ \big\{\ updatePaths_t\ (snd\ (vs\ t)) \mid t \in Vars\ spat_k\ \big\} \\
& newLink = \Big\{ \Big(\ \ \big(\,(S,\ \ fillWildcards\ s\ spat_k), [\,]\,\big), \big(\,(V,\ \ vpat_k), [\,]\,\big)\ \ \Big) \Big\} \\
\mathbf{in}\ \ & \big(\ reconstruct_S\ vpat_k\ (fst \circ vs),\ \ newLink \cup\ links\ \big)
\end{aligned}$$

where *selectPat* s R returns the unique source pattern in R matching s. Such a pattern exists because our DSL syntactically requires all source patterns of R to be disjoint and total.

By matching the source with $spat_k$, we bind every variable in $spat_k$ to a subtree of s as $f : (v \in Vars\ spat_k) \to TypeOf\ \ spat_k\ v$. Then we recursively call *get* for all subtrees, expressed as $get \circ f$ in the definition above, while to be precise, we need to invoke $get_{TypeOf\ \ spat_k\ v,\ \ TypeOf\ \ vpat_k\ v}$ for every $t \in Vars\ spat_k$.

With the subtrees and links produced by recursive calls, we can construct the result of get_{SV}: (1) the returned view is created simply by reconstructing the view pattern $vpat_k$ with subtrees generated recursively; and (2) the returned links should be the union of: (2.1) a new link between two regions at the root of s and v respectively (*newLink*). (Note that subtrees of s matching wildcards of $spat_k$ are also treated as parts of the region, and this is achieved by invoking *fillWildcards* s $spat_k$ to replace all the wildcards of $spat_k$ with the corresponding subtrees of s.) (2.2) and also links produced by recursive calls, for which we need to be careful to update all paths in them. For a link $((\cdots, spath),\ (\cdots, vpath))$ generated by the recursive call for pattern variable v, we need to prepend *spath* to the path of v in $spat_k$, and *vpath* to the path of v in $vpat_k$, which is expressed as *updatePaths* in the above definition.

3.3.3 Put Semantics. To define a corresponding *put* function, we need to determine its domain T to make it total and retentive. Here we will first define the body of *put*, and later the definition of T will come out naturally.

For a consistency relation $S \longleftrightarrow V$ in our DSL defined with $R = \{\ spat_i \sim vpat_i \mid i \in \mathcal{I}\ \}$, its corresponding put_{SV} function has the signature:

$$put_{SV} : T \to S$$

where T is a subset of $S \times V \times (Region \sim Region)$. The body of put_{SV} is defined as:

$$\begin{aligned}
& put_{SV}\ (s, v, ls) = \\
& \quad \mathbf{let}\ linksToVRoot = \big\{\ \big(\ ((S', spat), spath), ((V, vpat), [\,])\ \big) \in ls\ \big\}
\end{aligned}$$

$$
\begin{array}{l}
\quad \textbf{in if } \mathit{linksToVRoot} \ \neq \ \emptyset \textbf{ then} \\
\qquad \textbf{let } l@(((S', \mathit{spat}), \mathit{spath}), ((V, \mathit{vpat}), [\,])) = \mathit{shortestSPathLink}\ \mathit{linksToVRoot} \\
\qquad\quad \mathit{vs} = \mathit{decompose}_V\ \mathit{vpat}\ v \\
\qquad\quad \mathit{ss} = \lambda(t \in \mathit{Vars}\ \mathit{spat}) \rightarrow \\
\qquad\qquad\qquad \mathit{put}\ s\ (\mathit{vs}\ t)\ (\mathit{divideLinks}_t\ (\mathit{ls}\ \setminus\ \{l\})) \\
\qquad \textbf{in } \mathit{inj}_{S'S}\ (\mathit{reconstruct}\ \mathit{spat}\ \mathit{ss}) \\
\quad\ \textbf{else let } l = \left\{\ (\mathit{spat}_k \sim \mathit{vpat}_k) \in R \mid \mathit{isMatch}\ \mathit{vpat}_k\ v\ \right\} \\
\qquad\qquad (\mathit{spat}_k \sim \mathit{vpat}_k) = \mathit{anyElement}\ l \\
\qquad\qquad \mathit{vs} = \mathit{decompose}_V\ \mathit{vpat}_k\ v \\
\qquad\qquad \mathit{ss} = \lambda(t \in \mathit{Vars}\ \mathit{spat}_k) \rightarrow \\
\qquad\qquad\qquad\quad \mathit{put}\ s\ (\mathit{vs}\ t)\ (\mathit{divideLinks}_t\ \mathit{ls}) \\
\qquad\quad \textbf{in } \mathit{reconstruct}\ (\mathit{fillWildcardsWithDefaults}\ \mathit{spat}_k)\ \mathit{ss}
\end{array}
$$

The process of put_{SV} is divided into two cases:

(1) *The root of the view is connected to input links.* In this case, we pick one link among them (*shortestSPathLink*), and use the source region of the link as the root of the new source, whose subtrees are created by recursive calls to subtrees of v. The function $\mathit{divideLinks}_t\ \mathit{lset}$ divides the set of links into appropriate parts for each recursive call, and maintains the paths of all view regions: for each link in *lset*, if the path of its view region has a prefix equal to the path of t in *vpat*, $\mathit{divideLinks}_t$ will remove that prefix; if it does not have such a prefix, $\mathit{divideLinks}_t$ will remove it from the set *lset*. Note that since the type of the newly created source S' is possibly different from the return type S, we need to wrap it into S using $\mathit{inj}_{S'S}$. Such a wrapper always exists because it is syntactically guaranteed by our DSL (Section 3.2.1).

    The choice of *shortestSPathLink* at the beginning is also important. By always choosing the link whose source region is at the shallowest position, we can guarantee that those regions in the new source will have the same relative positions as those in the original source. Thus our *put* functions will respect all of the possible second-order properties asserting relative positions of regions in the old source.

(2) *The root of the view is not attached to any input link.* In this case, we only need to perform a "traditional" *put*. We choose an arbitrary inductive rule with a matched view pattern in set $R = \{\ spat_i \sim vpat_i \mid i \in \mathcal{I}\ \}$, and the corresponding source pattern is used to create the new source. If the source pattern contains wildcards, we also need to replace those wildcards with some default values.

From the definition of put_{SV}, we can define T to be:

- All input links are valid, in the sense that: (1) the path for proving a region pattern indeed leads to a subtree matching that pattern, and (2) the source region and the view region of a link match with some inductive rule.
- For every subtree of the view whose root is not attached to any link, it always matches a view pattern in the set of rules R defining the consistency relation. This requirement guarantees that in the second case of *put*, we can always create a consistent source.
- All regions of input links are non-overlapping. Additionally, for every subtree of the view whose root is not attached to any link and matches a view pattern in R, the matched fragment is non-overlapping with any region of the input links either. This requirement guarantees that our *put* functions will eventually process all input links in the recursive procedure. (*put* will miss processing some input links if there is overlap.)

3.4 To Retentive Lenses

In this subsection, we will show that get_{SV} and put_{SV} functions specified above satisfy the following properties:

$$get_{SV}\ (put_{SV}\ (s, v, l)) = (v', l') \quad \Rightarrow \quad v = v' \qquad \text{(Correctness)}$$

$$get_{SV}\ (put_{SV}\ (s, v, l)) = (v', l') \quad \Rightarrow \quad \mathit{fst} \cdot l \subseteq \mathit{fst} \cdot l' \qquad \text{(Retentiveness)}$$

$$get_{SV}\ s = (v, l) \quad \Rightarrow \quad put_{SV}\ (s, v, l) = s \qquad \text{(Hippocraticness)}$$

However, they do not form a retentive lens immediately since they do not generate and process second-order properties and thus do not satisfy the uniquely identifying requirement. In fact, our get_{SV} and put_{SV} functions respect *all possible* second-order properties by default no matter whether they are required by the user or not, which is exactly the reason why we leave out second-order properties from the interface of get_{SV} and put_{SV}.

In the rest of this subsection, we will show that get_{SV} and put_{SV} in our semantics can be lifted to a version producing and accepting second-order properties, and that version of get_{SV} and put_{SV} will form a retentive lens. Finally, we have a corollary showing that get_{SV} and put_{SV} satisfy the properties listed above.

3.4.1 Second-Order Properties. Before defining the lifted version of *get* and *put*, we need to formalise second-order properties in our DSL, which are properties on the proofs of first-order properties (region patterns) as explained in Section 2.3. In particular, we will use them to specify: (1) some region is the root of a tree, and (2) some region is the n-th child of another region. Thus we define second-order properties as a set $Prop^2$:

$$Prop^2 = NamedPat \uplus (NamedPat \times NamedPat \times \mathbb{N})$$

$$NamedPat = Pat \times \mathbb{N}$$

where $\uplus$ is disjoint union, and *NamedPat* is the set of patterns paired with a number so that second-order properties can refer to a region pattern by its number. Similarly, we also define $NamedRegion = NamedPat \times Path$.

3.4.2 Lifted Get and Put. Given a pair of get_{SV} and put_{SV} with signatures:

$$get_{SV} : S \to V \times (Region \sim Region) \qquad put_{SV} : T \to S$$

where T is a subset of $S \times V \times (Region \sim Region)$, we can construct a pair of get' and put' with the following signatures:

$$get' : S \to V \times (SatProp \sim SatProp) \qquad put' : T' \to S$$

where $SatProp = NamedRegion \uplus Prop^2$ and T' is a subset of $S \times V \times (SatProp \sim SatProp)$.

The construction of put' is trivial. Given source s, view v, and second-order links ls, we can first obtain first-order links l by simply filtering out all links between second-order properties ($Prop^2$) in ls, and turns links between *NamedRegion* into links between *Region* by removing all names (the $\mathbb{N}$ part). Then we can invoke $put_{SV}\ s\ v\ l$ and use its result as the result of $put'\ s\ v\ ls$. (Remember that *put* in fact respects all of the possible second-order properties.)

Now we construct get', whose returned view is the same as that produced by get_{SV} and returned links should characterise all the relative positions (i.e.: uniquely identifying). More precisely, get' generates links in the following way: (1) get' transforms all links between *Region* (produced by get_{SV}) into links between *NamedRegion* by assigning unique names to region patterns of every link. (2) For the link connecting roots of the source and view, get' creates two $Prop^2$ asserting that those two regions are roots and a link between them. (3) For every pair of source regions sr_1 and sr_2 where sr_1 is the x-th child of sr_2, get' create a $p_s \in Prop^2$ asserting that the named region pattern in sr_1 should be the x-th child of the named region pattern in sr_2. Similarly, for the view regions vr_1 and vr_2 corresponding to sr_1 and sr_2 respectively, get' also creates $p_v \in Prop^2$ recording their relative position in the view. Then, a link (p_s, p_v) is added.

Finally, let us define T', the domain of put'. In Section 3.3, T is defined to restrict the input source, view, and links between regions so that put_{SV} becomes total. We shall let T' to be a subset of T with further restrictions on links between second-order properties. For every link between (second-order properties) p_s and p_v, we require that:

- The assertion of p_s and p_v should be satisfied by regions in the set of input links. For example, if $p_s = inj_1\ (pat,\ i)$ (inj_1 injects $NamedPat$ to $Prop^2$) asserts that the named region pattern $(pat,\ i)$ must be the root of the source, then its source region (referred by integer i) should indeed locate at the root.
- p_s and p_v should be correctly paired: If p_s is an assertion that source region sr_1 is the root, then p_v must be the assertion that its corresponding view region vr_1 is the root of the view. Similarly, if p_s is an assertion that source region sr_1 is the n-th child of sr_2, then p_v must be its corresponding assertion on vr_1 and vr_2.

3.4.3 *Retentiveness.*

Theorem 3.1. *get' and put' defined above form a retentive lens.*

Proof. For the sake of brevity, we only describe the overall idea here and the detailed proof can be found in the appendix. The proof is based on our Haskell implementation, which is basically a more precise version of the semantics given above. The uniquely identifying requirement is relatively easy to prove because our get' uses second-order properties that precisely specify how regions form a tree. Correctness and Retentiveness can be proved by performing inductions on the height of the view: we assume that the two properties already hold for all the views of height less than h, and then we prove that the two properties also hold for any view of height h using induction hypotheses. □

Now we have the following corolarry characterising get_{SV} and put_{SV} denoted by our semantics.

Corollary 3.2. *A pair of get_{SV} and put_{SV} satisfy:*

$$get_{SV}\ (put_{SV}\ (s, v, l)) = (v', l') \quad \Rightarrow \quad v = v'$$
$$get_{SV}\ (put_{SV}\ (s, v, l)) = (v', l') \quad \Rightarrow \quad fst \cdot l \subseteq fst \cdot l'$$
$$get_{SV}\ s = (v, l) \quad \Rightarrow \quad put_{SV}\ (s, v, l) = s$$

Proof. By Theorem 3.1 and Theorem 2.10, get' and put' satisfy these properties. From the process of constructing get' and put' using get_{SV} and put_{SV}, we know these properties are also satisfied by get_{SV} and put_{SV}. □

4 CASE STUDIES

In this section, we demonstrate the usefulness of retentiveness in practice by presenting two case studies on *resugaring* [Pombrio and Krishnamurthi 2014, 2015] and *code refactoring* [Fowler and Beck 1999]. In both cases, we need to constantly make modifications to the ASTs[3] and synchronise the CSTs accordingly. Retentiveness helps the user to know which information in the original CSTs is guaranteed to be retained after the synchronisation.

[Performing “diff” on two pieces of data can be hard [Miraldo et al. 2017]But domain-specific “diff” is not that difficult. examples.]

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

Before proceeding with these case studies, let us shortly introduce a new kind of links, dubbed *vertical link*, as they are between the old view and the modified view. The motivation for introducing vertical links is that, instead of letting third-party tools directly producing horizontal links between a source and its modified view, it is easier for them to output connections (i.e. vertical links) recording what parts in the consistent view are retained in the modified view. (For instance, recording the operation sequences.) These vertical links can be used, together with consistency links (between a source and its view), to finally obtain the connections between the source and the modified view. We assume that the third-party tools in the following case studies output vertical links.

4.1 Resugaring

For a programming language, usually the constructs of its surface syntax are richer than those of its abstract syntax (core language). The idea of *resugaring* [Pombrio and Krishnamurthi 2014, 2015] is to print evaluation sequences in a core language using the constructs of its surface syntax. We will show that our DSL is capable of reflecting AST changes resulting from evaluation back to CSTs, by writing a straightforward consistency declaration between the surface syntax and the abstract syntax and passing the generated *put* proper links.

Take the language Tiger [Appel 1998] (a general-purpose imperative language designed for educational purposes) for example: Its surface syntax offers logical conjunction ($\wedge$) and disjunction ($\vee$), which, in the core language, are desugared in terms of the conditional construct *if-then-else* during parsing. Boolean values are also eliminated: *False* is represented by integer 0 and *True* is represented by a non-zero integer. For instance, the program text $a \wedge b$ is parsed to an AST $\mathit{If}\ \ a\ b\ 0$, and $a \vee b$ is parsed to $\mathit{If}\ \ a\ 1\ b$. Suppose that evaluation is defined on the core language only. If we want to observe the evaluation sequence of $0 \wedge 10 \vee c$, then we will face a problem, as the evaluation is actually performed on $\mathit{If}\ \ (\mathit{If}\ \ 0\ \ 10\ \ 0)\ \ 1\ \ c$ and the printer will yield an evaluation sequence in terms of *if-then-else*.

To solve the problem in our DSL, let us first consider a familiar situation where we write a parser in Yacc (or similar tools such as Happy) to desugar logical expressions into conditional expressions[4]:

$$
\begin{array}{rlll}
\mathit{OrOp} & \to \mathit{Or}\quad \mathit{OrOp}\quad \mathit{AndOp} & \{\mathit{If} & \$2\quad 1\quad \$3\,\} \\
 & \mid\ \mathit{AndOp} & \{\$1 & \qquad\qquad\ \};\\
\mathit{AndOp} & \to \mathit{And}\ \mathit{AndOp}\ \mathit{CompareOp} & \{\mathit{If} & \$1\quad \$2\quad 0\ \ \} \\
 & \mid\ \mathit{CompareOp} & \{\$1 & \qquad\qquad\ \};\\
 & \ldots & &
\end{array}
$$

in which the concrete syntax is defined by production rules. Parsing can conceptually be thought of as first recovering a CST (of which the data types are automatically generated by Yacc) representing the structure of the program being parsed, and then converting the CST to an AST by the semantic actions in curly brackets. Similarly, these intentions can be expressed in our DSL as shown in Figure 7, in which the left-hand sides of the consistency declarations are the concrete syntax, and the right-hand sides are the abstract syntax. For instance, in the code snippet, we see that $\mathit{Or}\ l\ r$ is consistent with $\mathit{If}\ \ l\ (\mathit{Num}\ \ 1)\ \ r$, if the subtrees marked by the same variables are consistent (respectively). (Currently, our DSL does not handle the part of parsing where strings are converted to trees, and instead assumes that the CST has already been found. In other words, it handles the synchronisation between CSTs and ASTs, which is the more interesting part of the task of parsing. Curious readers may refer to papers [Matsuda and Wang 2013; Zhu et al. 2016] for a discussion about how string parser generators and CST–AST synchronisation can be integrated.)

[4] This example does not strictly follow Yacc's syntax and functionality.

$$
\begin{array}{lll}
\mathit{OrOp} & \longleftrightarrow & \mathit{Arith} \\
\quad \mathit{Or}\ l\ r & \sim & \mathit{If}\ \ l\ (\mathit{Num}\ 1)\ r \\
\quad \mathit{FromAnd}\ t & \sim & t \\
\mathit{AndOp} & \longleftrightarrow & \mathit{Arith} \\
\quad \mathit{And}\ l\ r & \sim & \mathit{If}\ \ l\ r\ (\mathit{Num}\ 0) \\
\quad \mathit{FromCmp}\ t & \sim & t
\end{array}
\qquad\qquad
\begin{array}{lll}
\mathit{CompOp} & \longleftrightarrow & \mathit{Arith} \\
\quad \ldots & & \\
\quad \mathit{FromAdd}\ t & \sim & t \\
\mathit{AddOp} & \longleftrightarrow & \mathit{Arith} \\
\quad \ldots & & \\
\quad \mathit{FromMul}\ t & \sim & t
\end{array}
$$

Fig. 7. Consistency declaration between logical and conditional expressions.

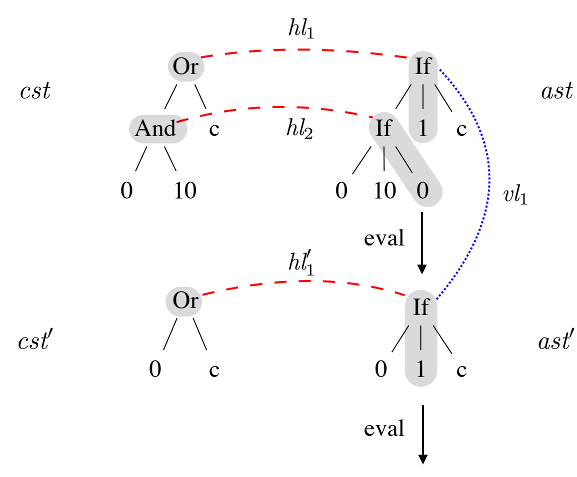


Fig. 8. Resugaring by retentiveness. (The horizontal links and vertical links are represented by dashed lines and dotted lines respectively. hl_1 and hl_2 are produced by $get_l\ cst$, and vl_1 is from a third-party tool. Let $cst' = put\ cst\ ast'\ hls$, where hls is the composition result of $[\,hl_1, hl_2\,]$ and $[\,vl_1\,]$. The constructor *Or* is retained by the composed link.)

From the consistency declarations, a pair of *get* and *put* functions are generated. As Figure 8 shows, by providing *put* proper links, we are able to achieve the same effect described by Pombrio and Krishnamurthi: When, internally, $\mathit{If}\ (\mathit{If}\ 0\ 10\ 0)\ 1\ x$ evaluates to $\mathit{If}\ 0\ 1\ x$, we can observe that (the program text represented by) the CST goes from $\mathit{Or}\ (\mathit{And}\ 0\ 10)\ x$ to $\mathit{Or}\ 0\ x$ without losing the syntactic sugar[5]. Compared to Pombrio and Krishnamurthi's method, we do not need to enrich the data types of the ASTs to incorporate fields for holding tags that mark from which syntactic object an AST construct comes. There is no need to insert tags into the ASTs during parsing or patch the compiler to be aware of the tags, either.

4.2 Code refactoring

Code refactoring is the restructuring of programs without changing its semantics [Fowler and Beck 1999]. Typical code refactoring includes, for example, renaming all the occurrences of a certain variable or moving a method to its super class. Since it is hard to perform analyses directly

[5]Here we intentionally omit many unimportant constructors such as *Num*, *Lit*, and *FromAnd* to simplify the expressions.

The original program:

$$
\begin{array}{l}
\mathit{foo}\ x\ y = \\
\quad \textbf{let}\ v_1 = a \wedge b \quad \text{-- conjunction} \\
\quad\phantom{\textbf{let}}\ v_2 = c \vee d \quad \text{-- disjunction} \\
\quad \textbf{in}\ \ e_b
\end{array}
$$

The desired program after refactoring:

$$
\begin{array}{l}
\mathit{foo}\ x\ y = e_b \\
\quad \textbf{where} \\
\qquad v_1 = a \wedge b \quad \text{-- conjunction} \\
\qquad v_2 = c \vee d \quad \text{-- disjunction}
\end{array}
$$

Fig. 9. The program for code refactoring and the desired result.

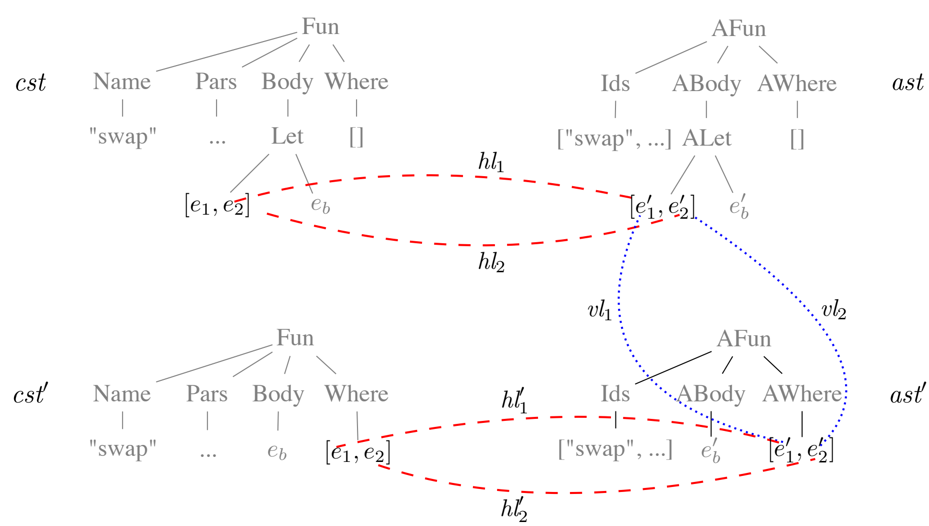


Fig. 10. Retain comments and layouts for code refactoring. (For simplicity, we use nodes e_1 and e_2 to represent the entire definitions for v_1 and v_2 respectively. In *ast* and *ast'*, we use e'_i instead of e_i to mean that e'_i does not include comments and syntactic sugar.)

on unstructured program text, refactoring usually involves (i) parsing the program text to a tree representation, (ii) modifying the tree representation (in a semantics-preserving way), and (iii) producing a new piece of program text incorporating all the modifications. Among them (i) and (iii) are often performed by a transformation system, and (ii) is done by some external tools compatible with the transformation system. Ideally, we want to choose the ASTs of a programming language to be the tree representation, as its compiler already comes with a pair of parser and printer which convert between program text and its abstract syntax representation, thereby saving the cost of defining data types and implementing the transformation system. Despite the benefit, there are inevitable difficulties with performing code refactoring on ASTs. As ASTs usually do not include information regarding comments, layouts, and syntactic sugar (as we have seen in Section 4.1) of their corresponding programs, how to elegantly retain this information when producing a new piece of program text after code refactoring on ASTs has become a research problem [de Jonge and Visser 2012].

Here we show that retentiveness makes it possible to perform code refactoring on ASTs and flexibly retain desired comments, layouts, and syntactic sugar on demand. Suppose that there is a function *foo* shown in Figure 9, of which the function body is a *let* expression binding two variables v_1 and v_2 and returning e_b. The user wants to make e_b directly the body of the function definition and move the variable definitions v_1 and v_2 to a *where* block, with the comments for v_1 and v_2 retained. This can be achieved as follows:

(1) Write consistency declarations between the language's CSTs and ASTs. This will give the user a pair of *get* and *put* functions as the transformation system between CSTs and ASTs.
(2) Use a third-party tool to perform code refactoring on the AST, by moving the variable definitions v_1' and v_2' from the *let* expression to the *where* block and making e_b' the direct child of the function body. Besides ast', the tool should in addition output vertical links $[\,vl_1, vl_2\,]$ between ast and ast' indicating that (the contents of) v_1' and v_2' are not changed through the refactoring.
(3) Pass the *put* function generated from step 1 the arguments: original cst, modified ast', and the composition result of $[\,hl_1, hl_2\,]$ and $[\,vl_1, vl_2\,]$. We get cst'. The whole process is illustrated in Figure 10.

According to retentiveness, the comments, layouts, and syntactic sugar in the variable definitions for v_1 and v_2 are all retained in cst'. We omit the consistency declarations here, since they can be defined very similarly to those in Figure 4 and Figure 7.

5 RELATED WORK AND DISCUSSIONS

[[Martins et al. 2014] also uses links to trace source information... the overview case analysis on put looks quite similar.]

5.1 Alignment

Our work on retentive lenses with links is closely related to the research on alignment in bidirectional programming.

The earliest lenses [Foster et al. 2007] only allow source and view elements to be matched positionally — the n-th source element is simply updated using the n-th element in the modified view. Later, lenses with more powerful matching strategies are proposed, such as dictionary lenses [Bohannon et al. 2008] and their successor matching lenses [Barbosa et al. 2010]. In matching lenses, the *put* transformations separate the processes of source–view element matching and element-wise updating. At the beginning, the source is divided into a "resource" and a "rigid complement": a resource consists of "chunks" of information that can be reordered, and a rigid complement stores information outside the chunk structure. This reorderable chunk structure is preserved in the view. If the view is updated, *put* first finds a correspondence between chunks of the old and new views based on some predefined strategies. Based on the correspondence, the resources are "pre-aligned" to match the new view chunks positionally, and then element-wise updates are performed. There are laws (`PutChunk` and `PutNoChunk`) dictating that correct pairs of resource and view chunks should be used during the element-wise updates, but these laws look more like an operational semantics for *put* (as remarked by the authors), and do not declaratively state what source information is retained.

To generalise list alignment, a more general notion of data structures called *containers* [Abbott et al. 2005] is used [Hofmann et al. 2012]. In the container framework, a data structure is decomposed into a shape and its content; the shape encodes a set of positions, and the content is a mapping from those positions to the elements in the data structure. The existing approaches to container alignment take advantage of this decomposition and treat shapes and contents separately. For example, if the shape of a view container changes, Hofmann et al.'s approach will update the source shape by a fixed strategy that make insertions or deletions at the rear positions of the (source) containers. By contrast, Pacheco et al.'s method permits more flexible shape changes and they call it *shape alignment*. In our setting, both the consistency on data and the consistency on shapes are specified by the same set of consistency declarations. In the *put* direction, both the data and shape of a new source is determined by (computed from) the data and shape of a view, so there is no need to have separated data and shape alignments.

It is worth noting that separation of data alignment and shape alignment will hinder the handling of some algebraic data types. First, in practice it is usually difficult for the user to define container data types and represent their data in terms of containers. We use the data types defined in Figure 3 to illustrate, where two mutually recursive data types *Expr* and *Term* are defined. If the user wants to define *Expr* and *Term* using containers, one way might be to parametrise the types of terminals (leaves in a tree, here *Integer* only):

$$
\begin{array}{lll}
\textbf{data}\ \mathit{Expr}\ i = & \mathit{Plus} & (\mathit{Expr}\ i)\ (\mathit{Term}\ i) \\
 & \mid \mathit{Minus} & (\mathit{Expr}\ i)\ (\mathit{Term}\ i) \\
 & \mid \mathit{Term} & (\mathit{Term}\ i) \\
\textbf{data}\ \mathit{Term}\ i = & \mathit{Neg} & (\mathit{Term}\ i) \\
 & \mid \mathit{Lit} & i \\
 & \mid \mathit{Paren} & (\mathit{Expr}\ i)
\end{array}
\qquad
\begin{array}{lll}
\textbf{data}\ \mathit{Arith}\ i = & \mathit{Add} & (\mathit{Arith}\ i)\ (\mathit{Arith}\ i) \\
 & \mid \mathit{Sub} & (\mathit{Arith}\ i)\ (\mathit{Arith}\ i) \\
 & \mid \mathit{Num}\ i &
\end{array}
$$

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

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

In contrast, retentiveness and "put with links as global alignment" advances the above approaches in the sense that retentiveness enables the user to know what is retained after *put*, while other approaches merely tell the user which basic lens will be applied after the alignment. As a result, the more complex the lens is, the more difficult to reason about the information retained in the new source for those approaches.

5.2 Provenance and Origin

Our work was inpired by research on provenance [Cheney et al. 2009] in the DB community and origin [van Deursen et al. 1993] in the rewriting community.

As discussed in [Cheney et al. 2009], provenance can be classified into three kinds: *why*, *how*, and *where*. *Why-provenance* is the first formalised provenance and was called *lineage* at the time [Cui et al. 2000]; it is the information about which data in the view is from which rows in the source. However, knowing only the rows a piece of data in the view is from is not informative enough for many applications. Consequently, researchers propose two refinements of why-provenance: *how-provenance*, which additionally counts the number of times a row is used (in the source), and *where-provenance*, which additionally records the column where a piece of data is from. By combining the row and column information of a piece of data, we know exactly where it is *copied* from. In our setting, we require two pieces of data linked by vertical links to be equal (under a certain pattern). Hence the vertical links resemble where-provenance.

If we leap from relational database communities to programming language communities, we will find that these kinds of provenance are not powerful enough, as they are mostly restricted to relational data, namely rows of tuples. In functional programming, we often use algebraic data types, which in general are sums of products including function types, and produce views by more general (recursive) functions rather than selection, projection, join, etc. For this need, *dependency provenance* [Cheney et al. 2011] and *expression provenance* [Acar et al. 2012] are proposed: the former tells the user on which parts of a source the computation of a part of a view depends, and the latter can even record a tree tracking how a part of a view is computed from some (predefined) primitive operations [Acar et al. 2012]. In this sense, our horizontal links are closer to dependency provenance.

The idea of inferring consistency links (horizontal links generated by *get*) can be found in the work on origin tracking for term rewriting systems [van Deursen et al. 1993], in which the origin relations between the rewritten terms can be calculated by analysing the rewrite rules statically. However, it is designed for tracing back, and nothing is mentioned about updating backwards, as we do in this paper. Using these consistency links, in the area of BX, Wang et al. propose a method to incrementalise state-based lenses by tracking links between data in the view and their origin in the source. When the view is edited locally in a sub-term, they use links to identify a sub-term in the source that contains the edited sub-term in the view. Then to update the old source, it is sufficient to only perform state-based *put* on those sub-terms. Since lenses generated by our DSL also create links (although for a different purpose), our lenses can be naturally incrementalised by their method.

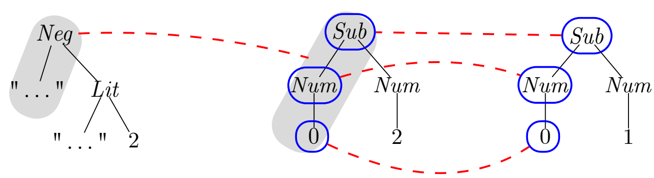
5.3 Operational-based BX

Our work is relevant to the operation-based approaches to BX, in particular, the delta-based BX model [Diskin et al. 2011a] and *edit lenses* [Hofmann et al. 2012]. The delta-based BX model regards the differences between the view state v and v' as *deltas* and the differences are abstractly represented by arrows (from the old view to the new view). The main law (property) in the framework is: Given a source state s and a view delta det_v between v and v', det_v should be translated to a source delta det_s between s and s' satisfying $get\ s' = v'$. As the law only guarantees the existence of a source delta det_s that updates the old source to a correct state, it is not sufficient for deriving retentiveness in their model because, given a translated delta det_s, there are still an infinite number of interpretations for generating a correct source s', with only few of them being retentive. To illustrate, Diskin et al. tend to represent deltas as edit operations such as *create*, *delete*, and *change*, aiming to transform edits on the view to edits on the source. However, representing deltas in this way can only tell the user in the new source what must be changed, while there is additional work to do for reasoning what must be retained. As discussed in this paper, by carefully designing and imposing proper properties on the representations of deltas, it is possible for delta-based BXs to exhibit retentiveness. Note that compared to Diskin et al.'s work, Hofmann et al. give concrete definitions and implementations for propagating edit sequences.

6 CONCLUSIONS AND FUTURE WORK

In this paper we have introduced a semantic framework of (asymmetric) retentive lenses, and designed a DSL for writing retentive transformations between simple algebraic data types. Retentive lenses enjoy a fine-grained local Hippocraticness, which subsumes the traditional global Hippocraticness. The potential use of retentiveness has been illustrated in case studies on resugaring and code refactoring. In the last part of the paper, we briefly discuss the potential future work on retentive lenses, regarding composability, generalisation, and parametricity.

Composability. Traditional state-based lenses are composable. For retentive lenses, we can also define a *comp* function, whose behaviour is similar to that in traditional state-based lenses, apart from producing proper intermediate links for the *put* functions. However, there are further requirements for composing retentive lenses. For the moment, we can only say that two retentive lenses $lens_1$ :: *RetLens A B* and $lens_2$ :: *RetLens B C* are composable, if $lens_1$ and $lens_2$ have the same property set on B. In other words, for any piece of data $b :: B$, $lens_1$ and $lens_2$ should decompose b in the same way. Here is an example showing non-composable retentive lenses:



where the first lens connects the region pattern *Neg a* _ with *Sub* (*Num* 0) _ (the grey part), while the second lens has a different decomposition strategy and further decomposes *Sub* (*Num* 0) _ into three small region patterns and establishes links for them respectively. It is hard to determine the result of this kind of link compositions, and we leave this to future work.

As for our DSL, we argue that it is not necessary to provide the user with composition functionality: it is just a different philosophy of design. Take the scenario of writing a parser for example: we can choose either to use parser combinators (such as Parsec), or to use parser generators (such as Happy). While parser combinators provide the user with many small components that are composable, 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). The generated parsers are usually used as a "black box", and no one expects to compose them. Thus, since our DSL is designed to be a "lens generator", the user will have no difficulty in writing bidirectional transformations even without composability.

Generalisation. While our theoretical framework and DSL are designed for asymmetric lenses, we believe they can be extended properly to handle symmetric settings. Many definitions in our framework are general and not limited to symmetric lenses: for example, the notion of regions, links and uniquely identifying property sets. The signature for the *get* and *put* functions might be

$$get : (s \in S, v \in V, P_s \sim Q_v) \longrightarrow (v' : V, P_s \sim Q'_v)$$
$$put : (s \in S, v \in V, P_s \sim Q_v) \longrightarrow (s' : S, P_{s'} \sim Q_v)$$

In addition, the region model in this paper is tailored for algebraic data types (trees), with which many functional programming languages are equipped. For bidirectional transformations in other languages, such as object-oriented languages, we may come up with other models that are more effective.

Parameterised Data Types. Unlike combinator-based approaches in which many combinators are generic and parameterised, our tiny DSL does not support parametric data types, with generic bidirectional transformations. To imitate functional dependent types, our DSL collects all the possible source and view types and assigns each a tag, so that *get* and *put* functions (indexed by source and view types) can pattern match against these type tags and choose a correct branch. This approach suddenly fails when facing parameterised types, for instance, *List a*, to which it is hard to assign a tag. We believe that the DSL might be drastically changed, to handle parameterised types and produce generic lenses.

ACKNOWLEDGEMENTS

We thank Jeremy Gibbons for many useful discussions and comments, and Yongzhe Zhang for helping us to create a nice figure in the introduction. This work is partially supported by the Japan Society for the Promotion of Science (JSPS) Grant-in-Aid for Scientific Research (S) No. 17H06099.

REFERENCES

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

Umut A. Acar, Amal Ahmed, James Cheney, and Roly Perera. 2012. *A Core Calculus for Provenance*. Springer Berlin Heidelberg, Berlin, Heidelberg, 410–429.

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

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

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

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

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

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

Yingwei Cui, Jennifer Widom, and Janet L. Wiener. 2000. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Trans. Database Syst.* 25, 2 (June 2000), 179–227.

Krzysztof Czarnecki, J Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective. In *Theory and Practice of Model Transformations*. Springer, 260–283.

Umeshwar Dayal and Philip A. Bernstein. 1982. On the Correct Translation of Update Operations on Relational Views. *ACM Trans. Database Syst.* 7, 3 (Sept. 1982), 381–416.

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

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

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

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

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

Georg Gottlob, Paolo Paolini, and Roberto Zicari. 1988. Properties and Update Semantics of Consistent Views. *ACM Trans. Database Syst.* 13, 4 (Oct. 1988), 486–524.

Soichiro Hidaka, Zhenjiang Hu, Kazuhiro Inaba, Hiroyuki Kato, Kazutaka Matsuda, and Keisuke Nakano. 2010. Bidirectionalizing Graph Transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, NY, USA, 205–216. https://doi.org/10.1145/1863543.1863573

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

Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: a Formally Verified Core Language for Putback-Based Bidirectional Programming. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM, 61–72.

David Lutterkort. 2008. Augeas—A configuration API. In *In: Proceedings of Linux Symposium*. 47–56.

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

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

Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. 2007. Bidirectionalization Transformation Based on Automatic Derivation of View Complement Functions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)*. ACM, New York, NY, USA, 47–58.

Kazutaka Matsuda and Meng Wang. 2013. *FliPpr: A Prettier Invertible Printing System.* Springer Berlin Heidelberg, Berlin, Heidelberg, 101–120.

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

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

Hugo Pacheco, Tao Zan, and Zhenjiang Hu. 2014. BiFluX: A Bidirectional Functional Update Language for XML. In *Proceedings of the 16th International Symposium on Principles and Practice of Declarative Programming (PPDP '14)*. ACM, New York, NY, USA, 147–158.

Justin Pombrio and Shriram Krishnamurthi. 2014. Resugaring: Lifting Evaluation Sequences Through Syntactic Sugar. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 38.

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

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

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

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

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

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

A DEFINING RETENTIVE BIDIRECTIONAL SEMANTICS VIA CODE GENERATION

In this section we explain in detail how retentive lenses are generated from a set of consistency relations defined in our DSL. We will give a source-to-source translation from the program in our DSL to Haskell code. As we go through the translation, we will sketch out why the generated lenses satisfies Equation 3 by construction, hinting at Corollary 3.2.

A.1 Overview of the Generation

The (top-level) code generation for an input program D is sketched below, in which both the generation for *get* and *put* functions, and the generating process for each inductive rule are independent. The generation for *put* is further divided into many cases depending on whether we need to handle input links. It is worth noting that, as we will see later, the *put* function (generated from *putDef*) is irrelevant to the input program D, because it will delegate all the work to *putNoLink* and *putWithLinks*, depending on whether there are links connected to the top of the input view. In addition, the function *putWithLinksDef* also delegates its work to other auxiliary functions when it need to pattern match against input data, and hence independent of D. Important auxiliary functions will be introduced later when they appear in the generated code.

$$\begin{aligned}
&\mathrm{GenLenses}[\![\mathit{dataTypes}\ \mathit{dGroups}]\!] = \mathit{dataTypes} + \mathrm{GenGets}[\![\mathit{dGroups}]\!] + \mathrm{GenPuts}[\![\mathit{dGroups}]\!]\\
&\mathrm{GenGets}[\![\mathit{dGroups}]\!] = \mathit{concatMap}\ \mathrm{GenGetG}[\![\cdot]\!]\ \mathit{dGroups}\\
&\mathrm{GenPuts}[\![\mathit{dGroups}]\!] = \mathit{putDef} : \mathit{putWithLinksDef} : \mathit{concatMap}\ \mathrm{GenPutNoLinkG}[\![\cdot]\!]\ \mathit{dGroups}
\end{aligned}$$

$$\mathsf{GenGetG}[\![st\ \texttt{'<--->'}\ vt\ ds\ \texttt{';'}]\!] = map\ (\mathsf{GenGet}\ (st, vt)\ [\![\cdot]\!])\ ds$$
$$\mathsf{GenPutNoLinkG}[\![st\ \texttt{'<--->'}\ vt\ ds\ \texttt{';'}]\!] = map\ (\mathsf{GenPutNoLink2}\ (st, vt)[\![\cdot]\!])\ ds$$

The generated *get* and *put* functions have the following signatures, in which we use some dependent function types that we can emulate to some extent in our actual Haskell code. Compared to Definition 2.6, the generated *get* and *put* functions take as additional arguments the types of the input source ($s :: STypes$) and view ($v :: VTypes$), on which the output types of *get* and *put* functions depend. For *put*, s_0 is the type of the original source, which in general can be different from the type of the generated source[6].

$$get :: (s :: STypes) \to (v :: VTypes) \to s \to (v, [HLink])$$
$$put :: (s :: STypes) \to (v :: VTypes) \to s_0 \to v \to [HLink] \to s$$

For the real Haskell implementation, we emulate dependent types by assigning each source type and view type a *type tag*. All of the source type tags and view type tags form the union types *STypes* and *VTypes* respectively. The region patterns of links also need to be encoded, to have their representations as Haskell data. Briefly, we collect all the source and view patterns appeared in the consistency relations, and assign different constructors to different patterns (in a way like hashing each pattern to a unique number). For example, the runtime representation for the region pattern $Plus\ x\ \_\ \_$ might be *S_ExprArithCase0*. Note that if the right-hand side of an inductive rule happens to be a variable, we will assign it the *Void* region pattern, meaning that there is in fact no such region (pattern) in the view. (However, since *get* need to produce a set of links that is uniquely identifying, all of the source data fragments need to connect to some parts in the view – even they does not seem to exist.) Let us call a link an *imaginary link* if it has the *Void* region pattern on the view, and otherwise a *real link*.

A.2 Generation of *get*

Generation of the *get* function is basically converting inductive rules to function definition clauses: if, for example, *Plus* is consistent with *Add*, then *get* can simply map *Plus* to *Add* and continue with the subtrees recursively. The consistency links between the source and view are thus established by construction.

The translation is more formally described by the transformation below. For every inductive rule in a group, $\mathsf{GenGet}(st, vt)[\![\cdot]\!]$ is invoked to generate a definition clause of *get*. Every inductive rule has the form $P \sim Q$ for some source pattern P and view pattern Q; we often write $P(\vec{x}) \sim Q(\vec{y})$ to name variables in P and Q as $\vec{x}$ and $\vec{y}$ respectively. (These variables should in fact be suitably renamed to avoid name clashing.) Individual variables in $\vec{x}$ and $\vec{y}$ are denoted by x_i and y_i, and the lines of code containing x_i and y_i should be repeated for every corresponding pair of source and view variables. Functions that is expanded at compile time all start with a capital character and in sans-serif. (They can be treated as macros.)

$$
\begin{array}{l}
\mathsf{GenGet}(st, vt)[\![P(\vec{x}) \sim Q(\vec{y})]\!] = \\
\quad get\ \mathsf{MkTag}[\![st]\!]\ \mathsf{MkTag}[\![vt]\!]\ P = (Q, l_0 : concat\ [ls'_0 \ldots ls'_n]) \\
\qquad where \\
\qquad\quad (y_i, ls_i) = get\ \mathsf{TypeOf}[\![x_i; P]\!]\ \mathsf{TypeOf}[\![y_i; Q]\!]\ x_i \\
\qquad\quad prefX_i\ = \mathsf{Pref}[\![x_i; P]\!] \\
\qquad\quad prefY_i\ = \mathsf{Pref}[\![y_i; Q]\!]
\end{array}
$$

[6] Many source types are interconvertible and have the same view, for instance, $0 - 3 :: Expr$ and $-3 :: Term$. Given a view, in many situations we may want to generate a new source whose type is different from the old one.

$$\begin{aligned} ls_i' &= map\ (addPrefH\ (prefX_i, prefY_i))\ ls_i \\ l_0 &= \mathsf{MkLink}[\![P; Q]\!] \end{aligned}$$

The generated *get* clause does the following:

(1) Recursively apply *get* to the subtrees x_i of the input data to get their corresponding views y_i and consistency links ls_i.

(2) Since x_i and y_i are subtrees of the source and view, we need to update the paths in their links ls_i (which previously start from the roots of the subtrees) by adding the corresponding prefixes. The prefixes are computed by Pref, and added by *addPrefH*.

(3) Finally we make a new link l_0 between the top of the current source and view using MkLink, and put l_0 into the front of the updated link list.

Example A.1. GenGet$(Expr, Arith)[\![Plus\ _\ x\ y \sim Add\ x\ y]\!]$ will produce the following Haskell code:

$$\begin{aligned}
&get\ ExprTy\ ArithTy\ (Plus\ r_0\ xS\ yS) = (Add\ xV\ yV, l_0 : concat\ [xls', yls'])\\
&\quad \textbf{where}\\
&\qquad (xV, xls) = get\ ExprTy\ ArithTy\ xS\\
&\qquad prexS \quad = [1]\\
&\qquad prexV \quad = [0]\\
&\qquad xls' \quad\ = map\ (addPrefH\ (prexS, prexV))\ xls\\
&\qquad (yV, yls) = get\ TermTy\ ArithTy\ yS\\
&\qquad preyS \quad = [2]\\
&\qquad preyV \quad = [1]\\
&\qquad yls' \quad\ = map\ (addPrefH\ (preyS, preyV))\ yls\\
&\qquad l_0 \qquad\ = ((S_ExprArithCase0, [\,]), (V_ExprArithCase0, [\,]))
\end{aligned}$$

We see that the first two arguments of *get* come from the encoded types of the group where this clause belongs. There are two variables (x and y) in the inductive rule, so they are handled by lines 1–4 and lines 5–8 in the where-block respectively. Finally the link l_0 between encoded region *S_ExprArithCase0* (representing $Plus\ x\ \_\ \_$) and *V_ExprArithCase0* (representing $Add\ \_\ \_$) is added.

A.3 Generation of *put*

Generation of the *put* function is little complex. Below we describe an algorithm that can be best understood as creating a new source following the structure of the view while satisfying retentiveness by construction. To satisfy retentiveness, we should inspect whether the top of the view is linked to some part of the original source.

- If there is no link at the top of the view, it means that retentiveness does not require that anything be retained at the top. In this case, we only need to find a inductive rule $P \sim Q$ such that the view matches Q, create a consistent source P, and call *put* on the subtrees of the view recursively.
- If there are links at the top of the view, for each link we can copy the source fragment it indicates in the original source, to the top of the new source, such that a horizontal link between the updated source and the view can be later created by *get* to satisfy retentiveness.

The *put* function thus starts with a case analysis on whether there are top-level links:

$$\begin{aligned}
&put :: (s :: STypes) \to (v :: VTypes) \to s_0 \to v \to [HLink] \to s\\
&put\ st\ vt\ os\ v\ hls \mid \neg\ (hasTopLink\ hls) = \ldots\\
&put\ st\ vt\ os\ v\ hls \mid hasTopLink\ hls \qquad = \ldots
\end{aligned}$$

Next we explain the two cases of *put* in detail.

A.3.1 No Link at the Top. In this case we need to choose a inductive rule such that the view matches the right-hand side pattern of the clause. This is done by an auxiliary function *selSPat* that tries to find such a clause and returns (the encoding of) its source pattern. This source pattern is then passed into another auxiliary function *putNoLink*:

$$put\ st\ vt\ os\ v\ hls \mid \neg\,(hasTopLink\ hls) = putNoLink\ st\ vt\ (selSPat\ st\ vt\ v)\ os\ v\ hls$$

$$putNoLink :: (s :: STypes) \to (v :: VTypes) \to SPat \to s_0 \to v \to [HLink] \to s$$
$$selSPat :: STypes \to (v :: VTypes) \to v \to SPat$$

The *putNoLink* function is defined mutually recursively with *put*. It handles the recursive calls on the subtrees, and eventually creates a new source that has the specified source pattern. For each inductive rule $P(\vec{x}) \sim Q(\vec{y})$ for synchronising data between source type *st* and view type *vt*, we generate the following *putNoLink* for it:

$$
\begin{array}{l}
\mathsf{GenPutNoLink}(st, vt)[\![P(\vec{x}) \sim Q(\vec{y})]\!] = \\
\quad putNoLink\ \mathsf{MkTag}[\![st]\!]\ \mathsf{MkTag}[\![vt]\!]\ \mathsf{MkPat}[\![P]\!]\ os\ Q\ hls = P[\vec{z}/\vec{x}] \\
\qquad where \\
\qquad\quad prefX_i = \mathsf{Pref}[\![x_i; P]\!] \\
\qquad\quad prefY_i = \mathsf{Pref}[\![y_i; Q]\!] \\
\qquad\quad hls_i \quad = map\ (delPrefH\ ([\,], prefY_i))\ (filterLink\ prefY_i\ hls) \\
\qquad\quad z_i \qquad = put\ \mathsf{TypeOf}[\![x_i; P]\!]\ \mathsf{TypeOf}[\![y_i; Q]\!]\ os\ y_i\ hls_i
\end{array}
$$

(1) We recursively invoke *put* on each subtree y_i of the view with a new environment env_i, to produce a consistent new source z_i. env_i is obtained by first dividing *env* into disjoint parts distinguished by the prefix path of y_i ($prefY_i$) and then deleting the prefix path.
(2) As specified, we create a new source using the pattern P, and use $\vec{z}$ as its subtrees.

Example A.2. Given inductive rule $Plus\ \_\ x\ y \sim Add\ x\ y$, the following Haskell code is generated for *putNoLink*:

$$
\begin{array}{l}
putNoLink\ ExprTag\ ArithTag\ S\_ExprArithCase0\ os\ (Add\ xV\ yV)\ hls = Plus\ \texttt{"X"}\ xSRes\ ySRes \\
\quad \mathbf{where} \\
\qquad prefxS\ = [1] \\
\qquad prefxV = [0] \\
\qquad hlsxV\ \ = map\ (delPrefH\ ([\,], prefxV))\ (filterLink\ prefxV\ hls) \\
\qquad xSRes\ = put\ ExprTag\ ArithTag\ os\ xV\ hlsxV \\
\qquad prefyS\ = [2] \\
\qquad prefyV = [1] \\
\qquad hlsyV\ \ = map\ (delPrefH\ ([\,], prefyV))\ (filterLink\ prefyV\ hls) \\
\qquad ySRes\ = put\ TermTag\ ArithTag\ os\ yV\ envyV
\end{array}
$$

Similar to the generated code for *get* in Example A.1, there are two code blocks for handling variables x and y in the inductive rule respectively. Since Haskell does not support dependent types, in fact we need to constantly convert the view and the new source from and to *Dynamic* values. However, to improve readability, here we intentionally removed the code for handling *Dynamic*s.

A.3.2 Links at the Top. In addition to creating a consistent source, for each link hl_0 at top of the view, we need to copy the source fragment in the original source to which hl_0 connects, so that later *get* can establish a horizontal link *hl* between the newly created source and the view, as required by retentiveness.

What makes the situation complex is that, there can be more than one links connected to the top of the view, and we need to handle all of them to avoid *putWithLinks* invoking itself. These links consist of at most one real link, and possibly many imaginary links. We sort the links according to their paths in the source, and handle them one by one in a bottom-up way. The real link is handled by the case analysis block and the imaginary links are handled by the *foldr*. Finally *mkInj* is called because the desired output source type might be different from the type of the source fragment connected by a link.

```
put st vt os v env | hasTopLink env = putWithLinks st vt os v env
putWithLinks :: (s :: STypes) → (v :: VTypes) → s0 → v → Env → s
putWithLinks st vt os v env = s4
  where
    (ml, imags, env′) = getTopLinks env
    s2 = case ml of
      Just l    → let ((sPat, sPath), (vPat, [ ])) = l
                      s0 = fetch l os
                      s1 = putNoLink (typeOf s0) vt sPat os v env′
                  in  repSubtree sPat s0 s1
      Nothing → putNoLink st vt (selSPat st vt v) os v env′
    s3 = foldr (splice os) s2 imags
    s4 = mkInj st (typeOf s3) s3
```

Handling the Possible Real Link ml.

- If the real link *ml* exists, we need to fetch the source fragment s_0 indicated by *ml* from the original source and recursively invoke *put* on the subtrees (holes) of s_0. However, invoking *put* on subtrees requires us to do much redundant work very similar to what *putNoLink* does, so that we decide to reuse *putNoLink* to avoid this: We invoke *putNoLink* again on the current view without top-level links to produce a source s_1, which has complete subtrees (there are no holes in s_1) but is not retentive at the top-level node. Thus by replacing holes in s_0 with complete subtrees in s_1, we get the final result that is both retentive at the top-level node and at subtrees. This is illustrated in Figure 11 and the code generation for *repSubtree* is:

  $$repSubtree :: SPat \to (s_0 :: STypes) \to (s_0 :: STypes) \to s_0$$
  $$\begin{aligned}&\mathsf{GenRepSubtree}[\![P\ \vec{x} \sim Q\ \vec{y}]\!] = \\ &\quad repSubtree\ \mathsf{MkPat}[\![P]\!]\ \mathsf{MkRegPat}[\![P]\!]\ (P\ \vec{x}) = \mathsf{MkRegPat}[\![P]\!]\ \vec{x}\end{aligned}$$

  where $\mathsf{MkRegPat}[\![\cdot]\!]$ differs $\mathsf{MkPat}[\![\cdot]\!]$ in the sense that it convert the variable patterns into wildcard patterns and convert wildcard patterns into (fresh) variable patterns. For instance, $\mathsf{MkRegPat}[\![Plus\ \_\ x\ y]\!]$ gives *Plus fromWild0 _ _* . We have seen this function when we define regions previously.
- If there is no real link meaning nothing to be retained (at the top), we just need to invoke *putNoLink* with the new environment *env′*.

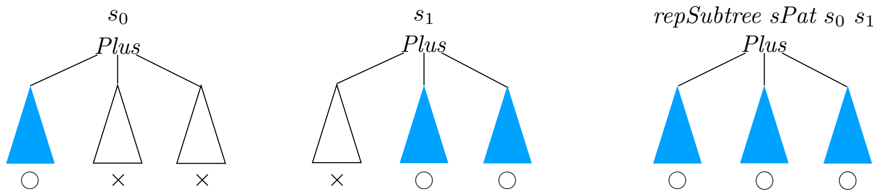


Fig. 11. Illustration of *repSubtree*. (Suppose that *sPat* is the source pattern of the inductive rule $Plus\ \_\ x\ y \sim Add\ x\ y$. Then the tree s_0 is correct (i.e., retentive) at the annotation field (marked by the wildcard in the source pattern) while has holes at subtrees marked by x and y. s_1 is like a "complement" of s_0. We can get a completely correct tree by replacing holes in s_0 with corresponding subtrees from s_1.)

Handling Imaginary Links imags. Remember that *get* in general is not an injection, and some parts of a source are discarded when producing its view. These parts can also be retained, by passing imaginary links to the *put* function. Each imaginary link is handled in a way very similar to a real link, as the *splice* function below shows. Thus the process of handling all the imaginary links can be expressed by a *foldr*, in which the (imaginary) link with longest path in the original source is dealt with first.

$$splice\ os\ imag\ s_0 = insSubtree\ sPat\ (fetch\ imag\ os)\ s_0$$

$$\begin{aligned}
&insSubtree :: SPat \longrightarrow (s_1 :: STypes) \longrightarrow s_0 :: STypes \longrightarrow s_1\\
&\mathsf{GenInsSubtree}[\![P\ x \sim Q\ y]\!] =\\
&\quad insSubtree\ \mathsf{MkPat}[\![P\ x]\!]\ s_1\ s_0 = P\ s_2\\
&\qquad where\ s_2 = mkInj\ \mathsf{TypeOf}[\![x; P\ x]\!]\ (typeOf\ s_0)\ s_0
\end{aligned}$$

Every time a source fragment is fetched, we need to insert the accumulated source s_0 into it as its subtree (by *insSubtree*). Since *insSubtree* is invoked by *splice* and *splice* only handles imaginary links, we just need to generate the code of *insSubtree* for those particular inductive rules, whose left-hand sides are possible to be source patterns of imaginary links. Those particular inductive rules, for example $Term\ \_\ t \sim t$ and $Paren\ \_\ e \sim e$, have the characteristics that there is only one hole in their left-hand sides and we precisely know where the hole is. Thus inserting a subtree s_2 into a tree P (with only one hole) is represented by $P\ s_2$ in the above code. Before the insertion, we need to convert the subtree to be inserted to the same type of the hole and meanwhile update the accumulated links. This is performed by *mkInj*. (explained below) The whole process is illustrated in Figure 12.

Making Type Correct. Importantly, the source after handling imaginary links might not be of the correct type *st* though. So functions such as *insSubtree*, *putWithLinks* need to invoke *mkInj* lastly to make the output type correct. This *mkInj* function is supposed to make changes that are discarded by *get* (for establishing Correctness) and do not disrupt retentiveness already established for the subtrees (for establishing the overall Retentiveness):

$$mkInj :: (sup :: STypes) \longrightarrow (sub :: STypes) \longrightarrow sub \longrightarrow sup$$

$$\begin{aligned}
&get_v\ s \qquad = get_v\ (mkInj_s\ _\ _\ s)\\
&fst \cdot get_l\ s\ fsubseteq\ fst \cdot get_l\ (mkInj\ _\ _\ s)
\end{aligned}$$

B PROOFS

In this section, we prove Theorem 3.1 and Corollary 3.2 based on the compiler given in Appendix A. The core of the proof is showing that the generated *get* and *put* (without second order properties

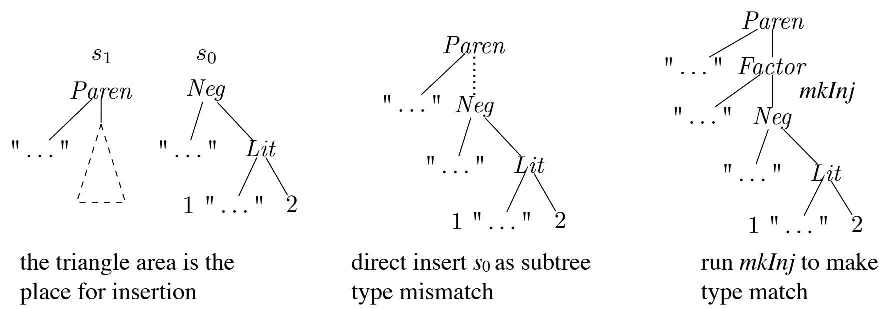


Fig. 12. Illustration of *insSubtree* (*Paren* _ *e*) s_1 s_0. We insert s_0 to the hole in s_1 according to the pattern *Paren* _ *x* of s_1. The hole for insertion is the subtree indicated by the (only) variable *e* in the pattern. However, since the type of s_0 might not match the type of the hole, we need to run *mkInj* first.

as input) retain all links of regions and satisfy *PutGet*, i.e. the first two properties of Corollary 3.2. This part is showed in detail in the rest of this section.

The key idea of our proofs is to make inductions on the height of the view: Given a source s of type st and a view v of type vt, and correspondence links hls between source s and view v, we assume that Correctness and Retentiveness already hold for all the views v' of height less than h, then we prove that the two properties also hold for the view v of height h using induction hypotheses. The structure of the proofs follow the structure of *put* functions, and are thus divided into two main parts, depending on whether there are input links connected to the top of the view.

Case 1: When There Is No Top-Level Link

We first presents some prerequisite and lemmas, which are used in the proof later.

Prerequisites 1. As mentioned in Making Type Correct, the function *mkInj* is supposed to make changes that are discarded by get (for establishing Correctness) and do not disrupt retentiveness already established for the subtrees.

$$get_v\ s = get_v\ (mkInj\ \_\ \_\ s)$$
$$fst \cdot get_l\ s\ \subseteq\ fst \cdot get_l\ (mkInj\ \_\ \_\ s)$$

Explanation. The first equation is exactly "make changes that are discarded by *get*". For the second inequation, since *mkInj* _ _ *s* contains a subtree which is exactly s, get_l on it should produce more links than on s. In addition, since *mkInj* _ _ *s* contains a subtree which is exactly s, the links of get_l (*mkInj* _ _ *s*) cover the links of get_l s, if we do not consider the proofs (paths) on the source side of the links.

Prerequisites 2. *mkInj* is an identity function if the types lifted from and converted to are the same.

$$mkInj\ toTy\ fromTy\ t\ |\ toTy == fromTy = t$$

Lemma B.1. *fstR* · *hls* does not care about the proofs (paths) on the source side:

$$fstR \cdot map\ (addPrefH\ (pref, [\,]))\ hls = fstR \cdot hls$$

Proof sketch. As for "$fstR \cdot hls$ does not care about on the source side", it means that:

$$\forall x, \exists y, \exists a, \exists b. x\ fstR\ (x, y)\ \wedge\ (x, y)\ hls\ (a, b)$$

So that the equation trivially hold.

Proofs of Case 1

Now we start to prove Case 1. Since there is no link at the top of the view, *put* will invoke *putNoLink*. Suppose the case of *putNoLink* generated from the consistency declaration $P\ xs \sim Q\ ys$ is matched, then the goal is to show (we rewrite the two properties in a one line style)

$$get_v\ (put\ st\ vt\ s\ (Q\ ys)\ hls) = v \qquad \text{(Correctness)}$$

$$fstR \cdot hls \subseteq fstR \cdot (get_l\ (put\ st\ vt\ s\ (Q\ ys)\ hls)) \qquad \text{(Retentiveness)}$$

As for Correctness:

Proof.

$$\begin{aligned}
& put\ st\ vt\ os\ (Q\ ys)\ hls \\
= \quad & \{\text{expand the body of } put\} \\
& putNoLink\ st\ vt\ (selSPat\ st\ vt\ (Q\ ys))\ os\ (Q\ ys)\ hls \\
= \quad & \{\text{expand the body of } putNoLink\} \\
& P\ zs
\end{aligned}$$

$$\begin{aligned}
& get_v\ (P\ zs) \\
= \quad & \{\text{Substitution of variables } zs \text{ (in } putNoLink)\} \\
& get_v\ (P\ [\ldots put\ \mathsf{TypeOf}[\![x_i]\!]\ \mathsf{TypeOf}[\![y_i]\!]\ os\ hls_i\ y_i \ldots]) \\
= \quad & \{\text{According to the source disjointness, the get generated from the} \\
& \quad P\ xs \sim Q\ ys \text{ must be chosen}\} \\
& Q\ [\ldots get_v\ (put\ \mathsf{TypeOf}[\![x_i]\!]\ \mathsf{TypeOf}[\![y_i]\!]\ os\ hls_i\ y_i) \ldots] \\
= \quad & \{\text{The height of } y_i \text{ is less than } h \text{ and thus Correctness for it holds}\} \\
& Q\ [\ldots y_i \ldots] \\
= \quad & \{\text{All } y_i \text{ contribute to } ys\} \\
& Q\ ys
\end{aligned}$$

□

For Retentiveness:

Proof.

$$\begin{aligned}
& put\ st\ vt\ os\ (Q\ ys)\ hls \\
= \quad & \{\text{expand the body of } put\} \\
& putNoLink\ st\ vt\ (selSPat\ st\ vt\ (Q\ ys))\ os\ (Q\ ys)\ hls \\
= \quad & \{\text{expand the body of } putNoLink\} \\
& P\ zs
\end{aligned}$$

$$
\begin{aligned}
& \mathit{fstR} \cdot \mathit{get}_l\ (P\ \mathit{zs}) \\
= \quad & \{\text{Variable substitution of } \mathit{zs} \text{ (in } \mathit{putNoLink}\text{). Assume that } \mathit{zs} = \ldots z_i \ldots\} \\
& \mathit{fstR} \cdot \mathit{get}_l\ (P \ldots \mathit{put}\ \mathsf{TypeOf}[\![x_i; P]\!]\ \mathsf{TypeOf}[\![y_i; Q]\!]\ \mathit{os}\ y_i\ \mathit{hls}_i \ldots) \\
= \quad & \{\text{Expand the body of } \mathit{get}_l\} \\
& \mathit{fstR} \cdot (\mathsf{MkLink}[\![P; Q]\!] : \mathit{concat}\ [\,..\ \mathit{ls}'_i\ ..\,]) \\
= \quad & \{\text{Distribute } \mathit{fstR} \text{ into concatenation}\} \\
& \mathit{fstR} \cdot [\,\mathsf{MkLink}[\![P; Q]\!]\,] +\!\!+ \mathit{fstR} \cdot \mathit{concat}\ [\,..\ \mathit{ls}'_i\ ..\,] \\
\supseteq \quad & \{\text{Just drop } \mathsf{MkLink}[\![P; Q]\!]\} \\
& \mathit{fstR} \cdot \mathit{concat}\ [\,..\ \mathit{ls}'_i\ ..\,] \\
= \quad & \{\text{Variable substitutions of } \mathit{ls}'_i \text{ (in } \mathit{get}\text{) }\} \\
& \mathit{fstR} \cdot \mathit{concat}\ [\ldots \mathit{map}\ (\mathit{addPrefH}\ (\mathit{prefX}_i, \mathit{prefY}_i))\ \mathit{ls}_i \ldots] \\
= \quad & \{\text{Variable substitutions of } \mathit{ls}_i \text{ (in } \mathit{get}\text{)}\} \\
& \mathit{fstR} \cdot \mathit{concat}\ [\ldots \mathit{map}\ (\mathit{addPrefH}\ (\mathit{prefX}_i, \mathit{prefY}_i)) \\
& \qquad\qquad\qquad (\mathit{get}_l\ \mathsf{TypeOf}[\![x_i; P]\!]\ \mathsf{TypeOf}[\![y_i; Q]\!]\ x_i) \ldots] \\
= \quad & \{\text{Replace } x_i \text{ with the real input } \mathit{put}\ \mathsf{TypeOf}[\![x_i; P]\!]\ \mathsf{TypeOf}[\![y_i; Q]\!]\ \mathit{os}\ y_i\ \mathit{hls}_i\ \} \\
& \mathit{fstR} \cdot \mathit{concat}\ [\ldots \mathit{map}\ (\mathit{addPrefH}\ (\mathit{prefX}_i, \mathit{prefY}_i)) \\
& \qquad\qquad (\mathit{get}_l\ \mathsf{TypeOf}[\![x_i; P]\!]\ \mathsf{TypeOf}[\![y_i; Q]\!] \\
& \qquad\qquad\qquad (\mathit{put}\ \mathsf{TypeOf}[\![x_i; P]\!]\ \mathsf{TypeOf}[\![y_i; Q]\!]\ \mathit{os}\ y_i\ \mathit{hls}_i)) \ldots] \\
\supseteq \quad & \{\text{The height of } y_i \text{ is less than } h\text{, we can use induction hypotheses}\} \\
& \mathit{fstR} \cdot \mathit{concat}\ [\ldots \mathit{map}\ (\mathit{addPrefH}\ (\mathit{prefX}_i, \mathit{prefY}_i))\ \mathit{hls}_i \ldots] \\
= \quad & \{\text{Variable substitution of } \mathit{hls}_i \text{ (in } \mathit{putNoLink}\text{)}\} \\
& \mathit{fstR} \cdot \mathit{concat}\ [\ldots \mathit{map}\ (\mathit{addPrefH}\ (\mathit{prefX}_i, \mathit{prefY}_i)) \\
& \qquad\qquad\qquad (\mathit{map}\ (\mathit{delPrefH}\ ([\,], \mathit{prefY}_i))\ (\mathit{filterLinks}\ \mathit{prefY}_i\ \mathit{hls})) \ldots] \\
= \quad & \{\text{By } \mathit{map} \text{ fusion, and } \mathit{addPrefH}\ (\mathit{PrefX}_i, [\,]) = \\
& \quad \mathit{addPrefH}\ (\mathit{prefX}_i, \mathit{prefY}_i)) \circ \mathit{delPrefH}\ ([\,], \mathit{prefY}_i)\} \\
& \mathit{fstR} \cdot \mathit{concat}\ [\ldots \mathit{map}\ (\mathit{addPrefH}\ (\mathit{prefX}_i, [\,]))\ (\mathit{filterLinks}\ \mathit{prefY}_i\ \mathit{hls}) \ldots] \\
= \quad & \{\text{Dividing a set into disjoint parts and merge them results in the same set}\} \\
& \mathit{fstR} \cdot (\mathit{map}\ (\mathit{addPrefH}\ (\mathit{prefX}_i, [\,]))\ \mathit{hls}) \\
= \quad & \{\text{By Lemma B.1}\} \\
& \mathit{fstR} \cdot \mathit{hls}
\end{aligned}
$$

□

In addition, if we do not drop $\mathsf{MkLink}[\![P; Q]\!]$ at the fourth step, we have the following conclusion:

$$\mathit{fstR} \cdot (\mathsf{MkLink}[\![P; Q]\!] : \mathit{hls}) \ \subseteq\ \mathit{fstR} \cdot (\mathit{get}_l\ (\mathit{putNoLink}\ \mathit{st}\ \mathit{vt}\ (\mathit{selSPat}\ \mathit{st}\ \mathit{vt}\ (Q\ \mathit{ys}))\ \mathit{os}\ (Q\ \mathit{ys})\ \mathit{hls})$$

Case 2: When There Are Links at Top of the View

Since there are links at the top of the view, *putWithLinks* will be executed. We divide the whole proofs into small parts by first introducing several lemmas, and then we proceed with the proofs.

Lemma B.2. Perform *get* on the source fetched by a link l will yield the same link except for their source paths.

$$fstR \cdot [\,head\ (get_l\ (fetch\ l\ os)) = fstR \cdot [\,l\,]$$

Proof.

$$
\begin{aligned}
& fstR \cdot [\,head\ (get_l\ (fetch\ l\ os))\,] \\
= \quad & \{\text{Suppose } l = ((P, sPath), (Q, [\,]))\ \} \\
& fstR \cdot [\,head\ (get_l\ (P\ xs))\,] \\
= \quad & \{\text{Expand the definition of } get\} \\
& fstR \cdot [\,head\ (l_0 : concat\ [\ldots ls'_i \ldots])\,] \\
= \quad & \{\text{Extract the head of a list}\} \\
& fstR \cdot [\,l_0\,] \\
= \quad & \{\text{Variable substitution of } l_0 \text{ (in } get)\} \\
& fstR \cdot [\,\mathsf{MkPat}[\![P; Q]\!]\,] \\
= \quad & \{\text{By the definition of } \mathsf{MkPat}[\![\cdot]\!]\} \\
& fstR \cdot [\,((P, [\,]), (Q, [\,]))\,] \\
= \quad & \{fstR \text{ does not care about the source path}\} \\
& fstR \cdot [\,l\,]
\end{aligned}
$$

□

Lemma B.3. get_v applied to the result of *repSubtree* produces the same result as get_v applied to the last argument of *repSubtree*. get_l applied to the result of the *repSubtree* is equal to $get_l\ t_2$ with a slight modification, which is to replace the top-level link of it with the first link from $get_l\ t_1$.

$$
\begin{aligned}
get_v\ (repSubtree\ sPat\ t_1\ t_2) &= get_v\ t_2 \\
get_l\ (repSubtree\ sPat\ t_1\ t_2) &= head\ (get_l\ t_1) : tail\ (get_l\ t_2)
\end{aligned}
$$

Proof. For the first equation, as Figure 11 shows, *repSubtree* takes a pattern and two trees matching the pattern, in which the second tree may only differ from the result of the *repSubtree* at some parts described by the pattern. In addition, the different parts will all be discarded by get_s, so the equation trivially hold.

For the second equation, we can consider an inductive rule $P\ xs \sim Q\ ys$. Suppose $\mathsf{MkPat}[\![P]\!] = A$ and $\mathsf{MkRegPat}[\![P]\!] = B$, according to the generated code for a inductive rule , we have

$$
\begin{aligned}
& get_l\ (repSubtree\ \mathsf{MkPat}[\![P]\!]\ \mathsf{MkRegPat}[\![P]\!]\ (P\ xs)) \\
= \quad & \{\mathsf{MkPat}[\![P]\!] = A \text{ and } \mathsf{MkRegPat}[\![P]\!] = B\} \\
& get_l\ (repSubtree\ A\ B\ (P\ xs)) \\
= \quad & \{\text{Expand the body of } repSubtree\} \\
& get_l\ (B\ xs) \\
= \quad & \{\text{Expand the body of } get_l\} \\
& l_0 : concat\ [\ldots ls'_i \ldots] \\
= \quad & \{\text{Variable substitution of } l_0 \text{ and } ls'_i \text{ (in } get)\}
\end{aligned}
$$

$$\mathsf{MkLink}[\![B; Q]\!] : concat\ [\ldots map\ (addPrefH\ (prefX_i, prefY_i))\ ls_i \ldots]$$

and

$$\begin{aligned}
& get_l\ (P\ xs) \\
=\ & \quad \{\text{Expand the body of } get_l\} \\
& l_0 : concat\ [\ldots ls'_i \ldots] \\
=\ & \quad \{\text{Variable substitution of } l_0 \text{ and } ls'_i \text{ (in } get)\} \\
& \mathsf{MkLink}[\![P; Q]\!] : concat\ [\ldots map\ (addPrefH\ (prefX_i, prefY_i))\ ls_i \ldots]
\end{aligned}$$

and

$$\begin{aligned}
& get_l\ (\mathsf{MkRegPat}[\![P]\!]) \\
=\ & \quad \{\mathsf{MkRegPat}[\![P]\!] = B\} \\
& get_l\ B \\
=\ & \quad \{\text{Expand the body of } get_l\} \\
& l_0 : _ \\
=\ & \quad \{\text{Variable substitution of } l_0 \text{ (in } get)\} \\
& \mathsf{MkLink}[\![B; Q]\!] : _
\end{aligned}$$

It is obvious that $get_l\ (repSubtree\ sPat\ t_1\ t_2) = head\ (get_l\ t_1) : tail\ (get_l\ t_2)$.

□

Lemma B.4. Let $P\ s_2 = insSubtree\ sPat\ s_1\ s_0$, we have the following (in)equations, where variables should refer to their bindings in the definition of *insSbutree*.

$$get_v\ (P\ s_2) = get_v\ s_0$$
$$fstR \cdot (get_l\ s_0 +\!\!+ [\mathsf{MkLink}[\![P; Void]\!]]) \subseteq fstR \cdot (get_l\ (P\ s_2))$$

Proof. For the first equation, since *sPat* is of pattern $P\ x \sim x$ (there is only one variable), let us assume $insSubtree\ (P\ x)\ s_1\ s_0 = P\ s_2$.

$$\begin{aligned}
& get_v\ (P\ s_2) \\
=\ & \quad \{\text{There is no constructor in the right-hand side of } P\ x \sim x\} \\
& get_v\ s_2 \\
=\ & \quad \{\text{Variable substitution}\} \\
& get_v\ (mkInj\ _\ _\ s_0) \\
=\ & \quad \{\text{By Prerequisite 1}\} \\
& get_v\ s_0
\end{aligned}$$

For the second inequation,

$$\begin{aligned}
& fstR \cdot get_l\ (P\ s_2) \\
=\ & \quad \{\text{By the definition of } get_l\text{, and there is no constructor in the view.}\} \\
& fstR \cdot (\mathsf{MkLink}[\![P; Void]\!] : concat\ [\ldots ls'_i \ldots])
\end{aligned}$$

$$
\begin{aligned}
= \quad & \{\text{The pattern } P\ s_2 \text{ has only one subtree } s_2\} \\
& \mathit{fstR} \cdot (\mathsf{MkLink}[\![P; \mathit{Void}]\!] : \mathit{concat}\ [\, ls_0' \,]) \\
= \quad & \{\mathit{concat}\ [\, ls_0' \,] = ls_0'\} \\
& \mathit{fstR} \cdot (\mathsf{MkLink}[\![P; \mathit{Void}]\!] +\!\!\!+\ ls_0') \\
= \quad & \{\text{Variable substitution of } ls_0' \text{ (in } \mathit{get})\} \\
& \mathit{fstR} \cdot (\mathsf{MkLink}[\![P; \mathit{Void}]\!] +\!\!\!+\ \mathit{map}\ (\mathit{addPrefH}\ (\mathit{PrefX_0}, \mathit{PrefY_0}))\ ls_0) \\
= \quad & \{\text{Distribute } \mathit{fstR} \text{ into concatenation}\} \\
& \mathit{fstR} \cdot [\, \mathsf{MkLink}[\![P; \mathit{Void}]\!] \,] +\!\!\!+\ \mathit{fstR} \cdot (\mathit{map}\ (\mathit{addPrefH}\ (\mathit{PrefX_0}, \mathit{PrefY_0}))\ ls_0) \\
= \quad & \{\text{Variable substitution of } \mathit{PrefX_0}\ , \mathit{PrefY_0}, \text{ and } ls_0 \text{ (in } \mathit{get})\} \\
& \mathit{fstR} \cdot [\, \mathsf{MkLink}[\![P; \mathit{Void}]\!] \,] +\!\!\!+ \\
& \mathit{fstR} \cdot (\mathit{map}\ (\mathit{addPrefH}\ (\mathsf{Pref}[\![x; P\ x]\!], \mathsf{Pref}[\![y; Q\ y]\!]))\ (\mathit{get}_l\ s_2)) \\
= \quad & \{\mathsf{Pref}[\![y; Q\ y]\!] \text{ is empty because the inductive rule is } P\ x \sim x\} \\
& \mathit{fstR} \cdot [\, \mathsf{MkLink}[\![P; \mathit{Void}]\!] \,] +\!\!\!+\ \mathit{fstR} \cdot (\mathit{map}\ (\mathit{addPrefH}\ (\mathsf{Pref}[\![x; P\ x]\!], [\,]))\ (\mathit{get}_l\ s_2)) \\
= \quad & \{\text{By Lemma B.1}\} \\
& \mathit{fstR} \cdot [\, \mathsf{MkLink}[\![P; \mathit{Void}]\!] \,] +\!\!\!+\ \mathit{fstR} \cdot (\mathit{get}_l\ s_2) \\
= \quad & \{\text{Variable substitution of } s_2 \text{ (in } \mathit{insSbutree})\} \\
& \mathit{fstR} \cdot [\, \mathsf{MkLink}[\![P; \mathit{Void}]\!] \,] +\!\!\!+\ \mathit{fstR} \cdot (\mathit{get}_l\ (\mathit{mkInj}\ \_\ \_\ s_0)) \\
\supseteq \quad & \{\text{By Prerequisite 1}\} \\
& \mathit{fstR} \cdot [\, \mathsf{MkLink}[\![P; \mathit{Void}]\!] \,] +\!\!\!+\ \mathit{fstR} \cdot (\mathit{get}_l\ s_0)) \\
\supseteq \quad & \{\text{Drop } \mathit{fstR} \cdot [\, \mathsf{MkLink}[\![P; \mathit{Void}]\!] \,]\} \\
& \mathit{fstR} \cdot (\mathit{get}_l\ s_0)
\end{aligned}
$$

□

Lemma B.5. Let $s_2 = \mathit{splice}\ \mathit{os}\ \mathit{imag}\ s_0$, we have the following (in)equations, where variables should refer to their bindings in the definition of *splice*.

$$\mathit{get}_v\ s_2 = \mathit{get}_v\ s_0$$

$$\mathit{fstR} \cdot (\mathit{get}_l\ s_0 +\!\!\!+ [\, \mathit{imag} \,]) \subseteq \mathit{fstR} \cdot \mathit{get}_l\ s_2$$

Proof. For the first equation:

$$
\begin{aligned}
& \mathit{get}_v\ s_2 \\
= \quad & \{\text{Expand the body of } \mathit{splice}\ \} \\
& \mathit{get}_v\ (\mathit{insSubtree}\ \mathit{sPat}\ (\mathit{fetch}\ \mathit{imag}\ \mathit{on})\ s_0) \\
= \quad & \{\text{by Lemma B.4 }\} \\
& \mathit{get}_v\ s_0
\end{aligned}
$$

For the second inequation:

$$
\begin{aligned}
& \mathit{fstR} \cdot \mathit{get}_l\ s_2 \\
= \quad & \{\text{Expand the body of } \mathit{splice}\} \\
& \mathit{fstR} \cdot \mathit{get}_l\ (\mathit{insSubtree}\ \mathit{sPat}\ (\mathit{fetch}\ \mathit{imag}\ \mathit{os})\ s_0)
\end{aligned}
$$

$$
\begin{aligned}
\supseteq\quad & \{\text{By Lemma B.4}\} \\
& fstR \cdot (get_l\ s_0 \mathbin{+\!\!+} [\,\mathsf{MkLink}[\![P; Void]\!]\,]) \\
=\quad & \{\text{ The variable } imag \text{ in } splice \text{ is } ((sPat, sPath), (Void, [\,]))\ \} \\
& fstR \cdot (get_l\ s_0 \mathbin{+\!\!+} [\,imag\,])
\end{aligned}
$$

□

Lemma B.6. The following (in)equations hold for the code block handling imaginary links. Suppose

$$s' = foldr\ (splice\ os)\ s\ imags$$

Then

$$get_v\ s' = get_v\ s$$
$$fstR \cdot (get_l\ s \mathbin{+\!\!+} imags) \subseteq fstR \cdot get_l\ s'$$

Proof. The two (in)equations can be proved by making inductions on the length *imags*. Suppose the (in)equations hold for $foldr\ (splice\ os)\ s\ ims$, now we consider the input list which has one more imaginary link:

$$
\begin{aligned}
& foldr\ (splice\ os)\ s\ (im : ims) \\
=\quad & \{\text{Expand the body of } foldr \text{ once}\} \\
& splice\ os\ im\ (foldr\ (splice\ os)\ s\ ims) \\
=\quad & \{\text{Let } s' = foldr\ (splice\ os)\ s\ ims\} \\
& splice\ os\ im\ s' \\
=\quad & \{\text{Let } s'' = splice\ os\ im\ s'\ \} \\
& s''
\end{aligned}
$$

so we need to prove:

$$get_v\ s'' = get_v\ s$$
$$fstR \cdot (get_l\ s \mathbin{+\!\!+} (im : ims)) \subseteq fstR \cdot get_l\ s''$$

which can be easily derived from the properties of *splice* and induction hypotheses:

For the first equation:

$$
\begin{aligned}
& get_v\ s'' \\
=\quad & \{\text{By Lemma B.5, the property of } splice\} \\
& get_v\ s' \\
=\quad & \{\text{By induction hypotheses}\} \\
& get_v\ s
\end{aligned}
$$

For the second inequation:

$$
\begin{aligned}
& fstR \cdot get_l\ s'' \\
\supseteq\quad & \{\text{By Lemma B.5, the property of } splice\} \\
& fstR \cdot (get_l\ s' \mathbin{+\!\!+} [\,im\,]) \\
=\quad & \{\text{Distribute } fstR \text{ into concatenation}\} \\
& fstR \cdot (get_l\ s') \mathbin{+\!\!+} fstR \cdot [\,im\,])
\end{aligned}
$$

$$
\begin{aligned}
& \supseteq \quad \{\text{By induction hypotheses}\} \\
& \quad fstR \cdot (get_l\ s \mathbin{+\!\!+} ims) \mathbin{+\!\!+} fstR \cdot [\,im\,] \\
& = \quad \{\text{Distribute } fstR \text{ into concatenation}\} \\
& \quad fstR \cdot (get_l\ s) \mathbin{+\!\!+} fstR \cdot ims \mathbin{+\!\!+} fstR \cdot [\,im\,] \\
& = \quad \{\, get_l\ s,\ ims, \text{ and } im \text{ are ``disjoint'' in } putWithLinks.\} \\
& \quad fstR \cdot (get_l\ s \mathbin{+\!\!+} (im : ims))
\end{aligned}
$$

□

Lemma B.7. The code block for handling the real link produces a source s_2 consistent with the input view v, and the source s_2 is created in a way that takes input link l into account, if there is. The variables in the following (in)equations should refer to their bindings in *putWithLinks*.

$$
\begin{array}{lll}
get_v\ s_2 & = v & \\
fstR \cdot (l : hls') & \subseteq fstR \cdot get_l\ s_2, & \text{if there is an input real link} \\
fstR \cdot hls' & \subseteq fstR \cdot get_l\ s_2, & \text{if there is no such input real link}
\end{array}
$$

Proof. For the first equation $get_v\ s_2 = v$.

- If there is a real link,

$$
\begin{aligned}
& \quad get_v\ s_2 \\
& = \quad \{\text{Variable substitution of } s_2 \text{ (in } putWithLinks)\} \\
& \quad get_v\ (repSubtree\ sPat\ s_0\ s_1) \\
& = \quad \{\text{By Lemma B.3}\} \\
& \quad get_v\ s_1 \\
& = \quad \{\text{Variable substitution of } s_1 \text{ (in } putWithLinks)\} \\
& \quad get_v\ (putNoLink\ (typeOf\ s_0)\ vt\ sPat\ os\ hls'\ v) \\
& = \quad \{\text{No link at the top of the view. Use the result of Case 1}\} \\
& \quad v
\end{aligned}
$$

- If there is no real link,

$$
\begin{aligned}
& \quad get_v\ s_2 \\
& = \quad \{\text{Variable substitution of } s_2 \text{ (in } putWithLinks)\} \\
& \quad get_v\ (putNoLink\ st\ vt\ (selSPat\ st\ vt\ v)\ os\ hls'\ v) \\
& = \quad \{\text{No link at the top of the view. Use the result of Case 1}\} \\
& \quad v
\end{aligned}
$$

For the inequations $fstR \cdot (l : hls') \subseteq fstR \cdot get_l\ s_2$ and $fstR \cdot hls' \subseteq fstR \cdot get_l\ s_2$:

- If there is a real link,

$$
\begin{aligned}
& \quad fstR \cdot get_l\ s_2 \\
& = \quad \{\text{Variable substitution of } s_2 \text{ (in } putWithLinks)\} \\
& \quad fstR \cdot get_l\ (repSubtree\ sPat\ s_0\ s_1) \\
& = \quad \{\text{By Lemma B.3 }\} \\
& \quad fstR \cdot (head\ (get_l\ s_0) : tail\ (get_l\ s_1))
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\text{Variable substitution of } s_0 \text{ and } s_1 \text{ (in } \mathit{putWithLinks})\} \\
&\quad \mathit{fstR} \cdot (\mathit{head}\ (\mathit{get}_l\ (\mathit{fetch}\ l\ \mathit{os})) : \mathit{tail}\ (\mathit{get}_l\ (\mathit{putNoLink}\ (\mathit{typeOf}\ s_0)\ \mathit{vt}\ \mathit{sPat}\ \mathit{os}\ v\ \mathit{hls}'))) \\
&= \quad \{\text{Distribute } \mathit{fstR} \text{ over concatenation}\} \\
&\quad \mathit{fstR} \cdot [\,\mathit{head}\ (\mathit{get}_l\ (\mathit{fetch}\ l\ \mathit{os}))] \mathbin{+\!\!+} \\
&\quad \mathit{fstR} \cdot (\mathit{tail}\ (\mathit{get}_l\ (\mathit{putNoLink}\ (\mathit{typeOf}\ s_0)\ \mathit{vt}\ \mathit{sPat}\ \mathit{os}\ v\ \mathit{hls}'))) \\
&= \quad \{\text{By Lemma B.2}\} \\
&\quad \mathit{fstR} \cdot [\,l\,] \mathbin{+\!\!+} \mathit{fstR} \cdot (\mathit{tail}\ (\mathit{get}_l\ (\mathit{putNoLink}\ (\mathit{typeOf}\ s_0)\ \mathit{vt}\ \mathit{sPat}\ \mathit{os}\ v\ \mathit{hls}'))) \\
&= \quad \{\mathit{fstR} \cdot (\mathit{tail}\ R) = \mathit{tail}\ (\mathit{fstR} \cdot R) \text{ since } R \text{ is an ordered list.}\ \} \\
&\quad \mathit{fstR} \cdot [\,l\,] \mathbin{+\!\!+} \mathit{tail}\ (\mathit{fstR} \cdot (\mathit{get}_l\ (\mathit{putNoLink}\ (\mathit{typeOf}\ s_0)\ \mathit{vt}\ \mathit{sPat}\ \mathit{os}\ v\ \mathit{hls}'))) \\
&\supseteq \quad \{\text{No link at the top of the view. Use the result (last inequation) of Case 1}\} \\
&\quad \mathit{fstR} \cdot [\,l\,] \mathbin{+\!\!+} (\mathit{tail}\ (\mathit{fstR} \cdot (\mathsf{MkLink}[\![\_;\_]\!] : \mathit{hls}'))) \\
&= \quad \{\mathit{tail} \text{ drops the first link in the list}\} \\
&\quad \mathit{fstR} \cdot [\,l\,] \mathbin{+\!\!+} \mathit{fstR} \cdot \mathit{hls}' \\
&= \quad \{\text{By the property of list concatenation}\} \\
&\quad \mathit{fstR} \cdot (l : \mathit{hls}')
\end{aligned}
$$

- If there is no real link.

$$
\begin{aligned}
&\quad \mathit{fstR} \cdot \mathit{get}_l\ s_2 \\
&= \quad \{\text{Variable substitution of } s_2 \text{ (in } \mathit{putWithLink})\ \} \\
&\quad \mathit{fstR} \cdot \mathit{get}_l\ (\mathit{putNoLink}\ \mathit{st}\ \mathit{vt}\ (\mathit{selSPat}\ \mathit{st}\ \mathit{vt}\ v)\ \mathit{os}\ v\ \mathit{hls}')) \\
&\supseteq \quad \{\text{No link at the top of the view. Use the result of Case 1}\} \\
&\quad \mathit{fstR} \cdot \mathit{hls}'
\end{aligned}
$$

□

Proofs of Case 2

Now we start to prove Case 2. Since there are links at the top of the view, *put* will invoke *putWithLinks*. The goal is to show

$$\mathit{get}_v\ (\mathit{putWithLinks}\ \mathit{st}\ \mathit{vt}\ \mathit{os}\ v\ \mathit{hls}) = v \qquad \text{(Correctness)}$$

$$\mathit{fstR} \cdot \mathit{hls}\ \subseteq\ \mathit{fstR} \cdot \mathit{get}_l\ (\mathit{putWithLinks}\ \mathit{st}\ \mathit{vt}\ \mathit{os}\ v\ \mathit{hls}) \qquad \text{(Retentiveness)}$$

As for Correctness:

Proof.

$$
\begin{aligned}
&\quad \mathit{get}_v\ (\mathit{putWithLinks}\ \mathit{st}\ \mathit{vt}\ \mathit{os}\ v\ \mathit{hls}) \\
&= \quad \{\text{Expand the body of } \mathit{putWithLinks}\} \\
&\quad \mathit{get}_v\ s_4 \\
&= \quad \{\text{Variable substitution of } s_4 \text{ (in } \mathit{putWithLinks})\} \\
&\quad \mathit{get}_v\ (\mathit{mkInj}\ \_\ \_\ s_3) \\
&= \quad \{\text{By Prerequisite 1}\ \} \\
&\quad \mathit{get}_v\ s_3
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\text{Variable substitution of } s_3 \text{ (in } \mathit{putWithLinks}\text{)}\}\\
&\quad \mathit{get}_v\ (\mathit{foldr}\ (\mathit{splice}\ \mathit{os})\ s_2\ \mathit{imags})\\
&= \quad \{\text{By Lemma B.6}\}\\
&\quad \mathit{get}_v\ s_2\\
&= \quad \{\text{Variable substitution of } s_2 \text{ (in } \mathit{putWithLinks}\text{)}\}\\
&\quad \mathit{get}_v\ (\mathit{repSubtree}\ \mathit{sPat}\ s_0\ s_1)\\
&= \quad \{\text{By Lemma B.3}\}\\
&\quad \mathit{get}_v\ s_1\\
&= \quad \{\text{Variable substitution of } s_1 \text{ (in } \mathit{putWithLinks}\text{)}\}\\
&\quad \mathit{get}_v\ (\mathit{putNoLink}\ (\mathit{typeOf}\ s_0)\ \mathit{vt}\ \mathit{sPat}\ \mathit{os}\ v\ \mathit{hls}')\\
&= \quad \{\text{No link at the top. Use the result of Case 1}\}\\
&\quad v
\end{aligned}
$$

□

As for Retentiveness:

Proof.

$$
\begin{aligned}
&\quad \mathit{fstR} \cdot \mathit{get}_l\ (\mathit{putWithLinks}\ \mathit{st}\ \mathit{vt}\ \mathit{os}\ v\ \mathit{hls})\\
&= \quad \{\text{Expand the body of } \mathit{putWithLinks}\ \}\\
&\quad \mathit{fstR} \cdot \mathit{get}_l\ s_4\\
&= \quad \{\text{Variable substitutions of } s_4 \text{ (in } \mathit{putWithLinks}\text{)}\}\\
&\quad \mathit{fstR} \cdot \mathit{get}_l\ (\mathit{mkInj}\ \mathit{st}\ (\mathit{typeOf}\ s_3)\ s_3)\\
&\supseteq \quad \{\text{By Prerequisite 1}\}\\
&\quad \mathit{fstR} \cdot \mathit{get}_l\ s_3\\
&= \quad \{\text{Variable substitutions of } s_3 \text{ (in } \mathit{putWithLinks}\text{)}\}\\
&\quad \mathit{fstR} \cdot \mathit{get}_l\ (\mathit{foldr}\ (\mathit{splice}\ \mathit{os})\ s_2\ \mathit{imags})\\
&\supseteq \quad \{\text{By Lemma B.6}\}\\
&\quad \mathit{fstR} \cdot (\mathit{get}_l\ s_2 +\!\!+\ \mathit{imags})\\
&= \quad \{\text{Distribute } \mathit{fstR} \text{ into concatenation}\}\\
&\quad \mathit{fstR} \cdot (\mathit{get}_l\ s_2) +\!\!+\ \mathit{fstR} \cdot \mathit{imags}\\
&\supseteq \quad \{\text{By Lemma B.7, suppose the real link } l \text{ exists. }\}\\
&\quad \mathit{fstR} \cdot (l : \mathit{hls}') +\!\!+\ \mathit{fstR} \cdot \mathit{imags}\\
&\supseteq \quad \{\text{By the property of list concatenation}\}\\
&\quad \mathit{fstR} \cdot ([\,l\,] +\!\!+\ \mathit{hls}' +\!\!+\ \mathit{imags})\\
&= \quad \{\text{Previously } \mathit{hls} \text{ is divided into } (l, \mathit{imags}, \mathit{hls}')\}\\
&\quad \mathit{hls}
\end{aligned}
$$

If the real link l does not exists, the last three steps are:

$$\mathit{fstR} \cdot (\mathit{get}_l\ s_2) +\!\!+\ \mathit{fstR} \cdot \mathit{imags}$$

$$
\begin{aligned}
&\supseteq \quad \{\text{By Lemma B.7, suppose the real link } l \text{ does not exist. }\}\\
&\quad \mathit{fstR} \cdot \mathit{hls}' \mathbin{+\!\!+} \mathit{fstR} \cdot \mathit{imags}\\
&\supseteq \quad \{\text{By the property of list concatenation}\}\\
&\quad \mathit{fstR} \cdot (\mathit{hls}' \mathbin{+\!\!+} \mathit{imags})\\
&= \quad \{\text{Previously } \mathit{hls} \text{ is divided into } (\mathit{imags}, \mathit{hls}')\}\\
&\quad \mathit{hls}
\end{aligned}
$$

□

As A Retentive Lens

Theorems proved above will be the core part of proving Theorem 3.1, but we still need to show two more things:

- Links produced by get' is uniquely identifying.
- Our put retains all links of second-order properties.

For the uniquely identifying property, we should verify that our construction of get' generates a set of second-order properties completely specifying how regions are assembled into a tree. Given a concrete implementation of constructing get', in particular, a concrete implementation of generating second-order properties described in Section 3.4, the verification should be straightforward, thus omitted here.

For the retentiveness of links of second-order properties, the proof will be quite similar to the proof above showing retentiveness of regions, thus we also omit it here.

Now we can conclude Theorem 3.1, that is, our get and put functions can be lifted into a retentive lens. Following this, our get and put functions satisfy Corollary 3.2, that is, they satisfy:

$$get\ (put\ (s, v, l)) = (v', l') \quad \Rightarrow \quad v = v' \qquad \text{(Correctness)}$$

$$get\ (put\ (s, v, l)) = (v', l') \quad \Rightarrow \quad fst \cdot l \subseteq fst \cdot l' \qquad \text{(Retentiveness)}$$

$$get\ s = (v, l) \quad \Rightarrow \quad put_{SV}\ (s, v, l) = s \qquad \text{(Hippocraticness)}$$