An Algebraic Approach to Bi-directional Updating

Shin-Cheng Mu, Zhenjiang Hu, and Masato Takeichi

Department of Information Engineering
University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo 113, Japan
{scm,hu,takeichi}@ipl.t.u-tokyo.ac.jp

Abstract. In many occasions would one encounter the task of maintaining the consistency of two pieces of structured data that are related by some transform — synchronising bookmarks in different web browsers, the source and the view in an editor, or views in databases, to name a few. This paper proposes a formal model of such tasks, basing on a programming language allowing injective functions only. The programmer designs the transformation as if she is writing a functional program, while the synchronisation behaviour is automatically derived by algebraic reasoning. The main advantage is being able to deal with duplication and structural changes. The result will be integrated to our structure XML editor in the Programmable Structured Document project.

1 Introduction

In many occasions would one encounter the task of maintaining consistency of two pieces of structured data that are related by some transform. In some XML editors, for example [3, 15], a source XML document is transformed to a user-friendly, editable *view* through a transform defined by the document designer. The editing performed by the user on the view needs to be reflected back to the source document. Similar techniques can also be used to synchronise several bookmarks stored in formats of different browsers, to maintain invariance among widgets in an user interface, or to maintain the consistency of data and view in databases.

As a canonical example, consider the XML document in Figure 1(a) representing an article. When being displayed to the user, it might be converted to an HTML document as in Figure 1(b), with an additional table of contents. The conversion is defined by the document designer in some domain-specific programming language. We would then wish that when the user, for example, adds or deletes a section in (b), the original document in (a) be updated correspondingly. Further more, the changes should also trigger an update of the table of contents in (a). We may even wish that when an additional section title is added to the table of contents, a fresh, empty section will be added to the article bodies in both (a) and (b). All these are better done without too much effort, other than specifying the transform itself, from the document designer,

View-updating [5, 7, 10, 14, 1] has been intensively studied in the database community. Recently, the problem of maintaining the consistency of two pieces of structured data was brought to our attention again by [12] and [11]. Though developed separately, their results turn out to be surprisingly similar, with two important features missing. Firstly, it was assumed that the transform is total and surjective, which ruled out those transforms that duplicate data. Secondly, structural changes, such as inserting to or deleting from a list or a tree, were not sufficiently dealt with.

In this paper we will address these difficulties using a different approach inspired by previous studies of program inversion [2, 8]. We extend the injective functional

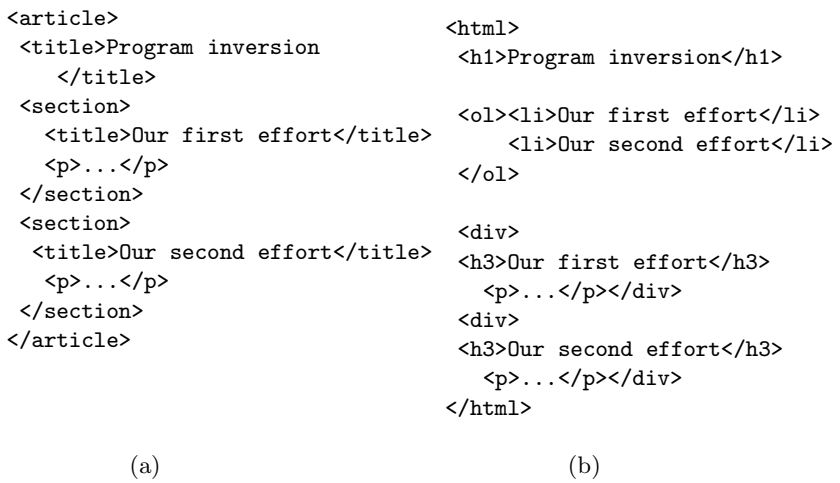

Fig. 1. An XML article and its HTML view with a table of contents.

language designed in [13], in which only injective functions are definable and therefore every program is trivially invertible. The document designer specifies the transform as if she were defining an injective function from the source to the view. A special operator for duplication specifies all element-wise dependency. To deal with inconsistencies resulting from editing, however, we define an alternative semantics, under which the behaviour of programs can be reasoned by algebraic rules. It will be a good application of program inversion [8] and algebraic reasoning, and the result will soon be integrated into our XML editor in the Programmable Structured Document project [15].

In Section 2 we give a brief introduction of the injective functional language in which the transforms are specified, and demonstrate the view-updating problem more concretely. An alternative semantics of the language is presented in Section 3, where we show, by algebraic reasoning, how to solve the view-updating problem. Section 4 shows some more useful transform, before we conclude in Section 5.

2 An Injective Language for Bi-directional Updating

Assume that a relation X, specifying the relationship between the source and the view, is given. In [11], the updating behaviour of the editor is modelled by two functions $get_X :: S \to V$ and $put_X :: (S \times V) \to S$. The function get_X transforms the source to the view. The function put_X, on the other hand, returns an updated source. It needs both the edited view and the *original* source, because some information might have been thrown away. For example, if the source is a pair and get_X simply extracts the first component, the second component is lost. The cached original source is also used for determining which value is changed by the user. A more symmetrical approach was taken in [12], where both functions take two arguments. The relation X is required to be bi-total (total and surjective), which implies that duplicating data, which would make the relation non-surjective, is not allowed.

In this paper we will explore a different approach. We make $get_X :: S \to V$ and $put_X :: V \to S$ take one argument only, and the transform has got to be injective — we shall lose no information in the source to view transform. A point-free language allowing only injective functions has been developed in [13] with this as one of the target applications. Duplication is an important primitive of the language.

Restricting ourselves to injective functions may seem like a severe limitation, but this is not true. In [13], it was shown that for all possibly non-injective functions $f :: A \to B$, we can automatically derive an injective function $f' :: A \to (B, H)$ where H records book-keeping information necessary for inversion. The extra information can be hidden from the user (for example, by setting the CSS visibility if the output is HTML). In fact, one can always make a function injective by copying the input to the output, if duplication is allowed. Therefore, the key extension here is duplication, while switching to injective functions is merely a change of presentation – rather than separating the original source and the edited view as two inputs to put_X, we move the burden of information preserving to X. This change, however, allows put_X itself to be simpler, while making it much easier to expose expose its properties, limitation, and possibly ways to overcome the limitation.

In this section we will introduce the language, Inv with some examples, and review the view-updating problem in our context. Extensions to the language and its semantics to deal with the view-updating problem will be discussed in Section 3. Some readers may consider the use of a point-free language as "not practical". We will postpone our defend to Section 5.

2.1 Views

The *View* datatype defines the basic types of data we deal with.

$$
\begin{array}{lll}
\mathit{View} & ::= & \mathit{Int} \mid \mathit{String} \mid () \\
& & \mid (\mathit{View} \times \mathit{View}) \mid \mathit{L\ View} \mid \mathit{R\ View} \\
& & \mid \mathit{List\ View} \mid \mathit{Tree\ View} \\
\mathit{List\ a} & ::= & [\,] \mid a : \mathit{List\ a} \\
\mathit{Tree\ a} & ::= & \mathit{Node\ a\ (List\ (Tree\ a))}
\end{array}
$$

The atomic types include integer, string, and unit, the type having only one value (). Composite types include pairs, sum (*L View* and *R View*), lists and rose trees. The (:) operator, forming lists, associates to the right. We also follow the common convention writing the list $1 : 2 : 3 : [\,]$ as $[1, 2, 3]$. More extensions dealing with editing will be discussed later. For XML processing we can think of XML documents as rose trees represented by the type *Tree*. This very simplified view omits features of XML which will be our future work. In fact, for the rest of this paper we will be mostly talking about lists, since the techniques can easily be generalised to trees.

2.2 An Injective Language Inv

The syntax of the language Inv is defined as below. We abuse the notation a bit by using X_V to denote the union of X and the set of variable names V. The $*$ operator denotes "a possibly empty sequence of".

$$
\begin{array}{lll}
X & ::= & X^{\smile} \mid \mathit{nil} \mid \mathit{zero} \mid C \\
& & \mid \delta \mid \mathit{dup\ P} \mid \mathit{cmp\ B} \mid \mathit{inl} \mid \mathit{inr} \\
& & \mid X; X \mid \mathit{id} \mid X \cup X \\
& & \mid X \times X \mid \mathit{assocr} \mid \mathit{assocl} \mid \mathit{swap} \\
& & \mid \mu(V\colon X_V) \\
C & ::= & \mathit{succ} \mid \mathit{cons} \mid \mathit{node} \\
B & ::= & < \mid \leq \mid \neq \mid \geq \mid > \\
P & ::= & \mathit{nil} \mid \mathit{zero} \mid \mathit{str\ String} \mid (S;)^* \mathit{id} \\
S & ::= & C^{\smile} \mid \mathit{fst} \mid \mathit{snd}
\end{array}
$$

The semantics of each Inv construct is given in Figure 2. A relation of type $A \to B$ is a set of pairs whose first components have type A and second components type

B, while a function[1] is one such that a value in A is mapped to at most one value in B. A function is injective if all values in B are mapped to at most one value in A as well. The semantics of every Inv program is an injective function from *View* to *View*. That is, the semantics function $[\![\,]\!]$ has type $\mathsf{Inv} \to \mathit{View} \to \mathit{View}$. For example, *nil* is interpreted as a constant function always returning the empty list, while *zero* always returns zero. Their domain is restricted to the unit type, to preserve injectivity.

The function *id* is the identity function, the unit of composition. The semicolon (;) is overloaded both as functional composition and as an Inv construct. It is defined by $(f; g)\ a = g\ (f\ a)$.

Union of functions is simply defined as set union. To avoid non-determinism, however, we require in $f \cup g$ that f and g have disjoint domains. To ensure injectivity, we require that they have disjoint ranges as well. The *domain* of a function $f :: A \to B$, written $\mathit{dom}\, f$, is the partial function (and a set) $\{(a, a) \in A \mid \exists b \in B :: (a, b) \in f\}$. The *range* of f, written $\mathit{ran}\, f$, is defined symmetrically. The product $(f \times g)$ is a function taking a pair and applying f and g to the two components respectively. We make composition bind tighter than product. Therefore $(f; g \times h)$ means $((f; g) \times h)$.

The fixed-point of F, a function from Inv expressions to Inv expressions, is denoted by μF. We will be using the notation $(X\colon \mathit{expr})$ to denote a function taking an argument X and returning *expr*.

The *converse* of a relation R is defined by

$$(b, a) \in R^\circ \equiv (a, b) \in R$$

The *reverse* operator $^\smile$ corresponds to converses on relations. Since all functions here are injective, their converses are functions too. The reverse of *cons*, for example, decomposes a non-empty list into the head and the tail. The reverse of *nil* matches only the empty list and maps it to the unit value. The reverse operator distributes into composition, products and union by the following rules, all implied by the semantics definition $[\![f^\smile]\!] = [\![f]\!]^\circ$:

$$\begin{array}{ll} [\![(f; g)^\smile]\!] = [\![g^\smile]\!]; [\![f^\smile]\!] & [\![f^{\smile\smile}]\!] = [\![f]\!] \\ [\![(f \times g)^\smile]\!] = [\![(f^\smile \times g^\smile)]\!] & [\![(\mu F)^\smile]\!] = [\![\mu(X\colon (F\ X^\smile)^\smile)]\!] \\ [\![(f \cup g)^\smile]\!] = [\![f^\smile]\!] \cup [\![g^\smile]\!] & \end{array}$$

The δ operator is worth our attention. It generates an extra copy of its argument. Written as a set comprehension, we have $\delta_A = \{(n, (n, n)) \mid n \in A\}$, where A is the type δ gets instantiated to. We restrict A to atomic types (integers, strings, and unit) only, and from now on use variable n and m to denote values of atomic types. To duplicate a list, we can always use $\mathit{map}\, \delta; \mathit{unzip}$, where *map* and *unzip* are to be introduced in the sections to come. Taking its reverse, we get:

$${\delta_A}^\smile = \{((n, n), n) \mid n \in A\}$$

That is, $\delta^\smile$ takes a pair and lets it go through only if the two components are equal. That explains the observation in [8] that to "undo" a duplication, we have to perform an equality test.

In many occasions we may want to duplicate not all but some sub-component of the input. For convenience, we include another Inv construct *dup* which takes a sequence of "labels" and duplicates the selected sub-component. The label is either *fst*, *snd*, $\mathit{cons}^\smile$, and $\mathit{node}^\smile$. Informally, think of the sequence of labels as the composition of selector functions (*fst* and *snd*) or deconstructors, and *dup* can be understood as:

$$[\![\mathit{dup}\, f]\!]\ x = (x, [\![f]\!]\ x)$$

[1] For convenience, we refer to possibly partial functions when we say "functions".

$$
\begin{array}{ll}
[\![nil]\!]\,() & = [\,] \\
[\![zero]\!]\,() & = 0 \\
[\![succ]\!]\,n & = n+1 \\
[\![cons]\!]\,(a,x) & = a{:}\,x \\
[\![node]\!]\,(a,x) & = Node\ a\ x \\
[\![inl]\!]\,a & = L\ a \\
[\![inr]\!]\,a & = R\ a \\
[\![id]\!]\,a & = a \\
\\
[\![swap]\!]\,(a,b) & = (b,a) \\
[\![assocr]\!]\,((a,b),c) & = (a,(b,c))
\end{array}
\qquad
\begin{array}{ll}
[\![assocl]\!]\,(a,(b,c)) & = ((a,b),c) \\
\\
[\![cmp\ \trianglelefteq]\!]\,(a,b) & = (a,b),\ \text{if } a \trianglelefteq b \\
[\![\delta]\!]\,a & = (a,a) \\
\\
[\![f;g]\!]\,x & = [\![g]\!]\,([\![f]\!]\,x) \\
[\![f \times g]\!]\,(a,b) & = ([\![f]\!]\,a, [\![g]\!]\,b) \\
[\![f \cup g]\!] & = [\![f]\!] \cup [\![g]\!], \\
 & \quad \text{if } dom\,f \cap dom\,g = ran\,f \cap ran\,g = \emptyset \\
[\![f^{\smile}]\!] & = [\![f]\!]^{\circ} \\
[\![\mu F]\!] & = [\![F\ \mu F]\!]
\end{array}
$$

Fig. 2. Functional semantics of Inv constructs apart from *dup*.

If we invert it, $(dup\,f)^{\smile}$ becomes a partial function taking a pair (x,n), and returns x unchanged if $f\ x$ equals n. The second component n can be safely dropped because we know its value already. We write $(dup\,f)^{\smile}$ as $eq\,f$. For example, $dup\,(fst;snd)\,((a,n),b)$ yields $(((a,n),b),n)$, while $eq\,(fst;snd)\,(((a,n),b),m)$ returns $((a,n),b)$ if $n=m$. Formally, *dup* is defined as a function taking a list of function names and returns a function:

$$
\begin{array}{ll}
dup\,id & = \delta \\
dup\,(fst;P) & = (dup\,P \times id);subl \\
dup\,(snd;P) & = (id \times dup\,P);assocl \\
dup\,(cons^{\smile};P) & = cons^{\smile};dup\,P;(cons \times id) \\
dup\,(node^{\smile};P) & = node^{\smile};dup\,P;(node \times id)
\end{array}
$$

Here, $[\![subl]\!]\,((a,b),c) = ((a,c),b)$, whose formal definition is given in Section 2.3.

Another functionality of *dup* is to introduce constants. The original input is kept unchanged but paired with a new constant:

$$
\begin{array}{ll}
[\![dup\ nil]\!]\,a & = (a,[\,]) \\
[\![dup\ zero]\!]\,a & = (a,0) \\
[\![dup\ (str\ s)]\!]\,a & = (a,s)
\end{array}
$$

Their reverses eliminates a constant whose value is known. In both directions we lose no information.

The *cmp* construct takes a pair of values, and let them go through only if they satisfy one of the five binary predicates given by non-terminal B.

2.3 Programming Examples in Inv

All functions that move around the components in a pair can be defined in terms of products, *assocr*, *assocl*, and *swap*. We find the following functions useful:

$$
\begin{array}{ll}
subr & = assocl;(swap \times id);assocr \\
subl & = assocr;(id \times swap);assocl \\
trans & = assocr;(id \times subr);assocl
\end{array}
$$

Their semantics, after expanding the definition, is given below:

$$
\begin{array}{ll}
[\![subr]\!]\,(a,(b,c)) & = (b,(a,c)) \\
[\![subl]\!]\,((a,b),c) & = ((a,c),b) \\
[\![trans]\!]\,((a,b),(c,d)) & = ((a,c),(b,d))
\end{array}
$$

Many list-processing functions can be defined recursively on the list. The function *map* applies a function to all elements of a list; the function *unzip* takes a list of pairs and splits it into a pair of lists. They can be defined by:

$$\begin{aligned} \mathit{map}\, f &= \mu(X : \mathit{nil}^\smile; \mathit{nil} \cup \\ &\qquad \mathit{cons}^\smile; (f \times X); \mathit{cons}) \\ \mathit{unzip} &= \mu(X : \mathit{nil}^\smile; \delta; (\mathit{nil} \times \mathit{nil}) \cup \\ &\qquad \mathit{cons}^\smile; (\mathit{id} \times X); \mathit{trans}; (\mathit{cons} \times \mathit{cons})) \end{aligned}$$

This is what one would expect when we write down their usual definition in a point-free style. The branches starting with $\mathit{nil}^\smile$ are the base cases, matching empty lists, while $\mathit{cons}^\smile$ matches non-empty lists. It is also provable from the semantics that $(\mathit{map}\, f)^\smile = \mathit{map}\, f^\smile$.

The function *merge* takes a pair of sorted lists and merges them into one. However, by doing so we lose information necessary to split them back to the original pair. Therefore, we tag the elements in the merged list with labels indicating where they were from. For example, $\mathit{merge}\,([1,4,7],[2,5,6]) = [L\,1, R\,2, L\,4, R\,5, R\,6, L\,7]$. It can be defined in Inv as below:

$$\begin{aligned} \mathit{merge} = \mu(X : {} & \mathit{eq}\, \mathit{nil}; \mathit{map}\, \mathit{inl} \cup \\ & \mathit{swap}; \mathit{eq}\, \mathit{nil}; \mathit{map}\, \mathit{inr} \cup \\ & (\mathit{cons}^\smile \times \mathit{cons}^\smile); \mathit{trans}; \\ & ((\mathit{leq} \times \mathit{id}); \mathit{assocr}; (\mathit{id} \times \mathit{subr}; (\mathit{id} \times \mathit{cons}); X); (\mathit{inl} \times \mathit{id}) \cup \\ & \quad ((\mathit{gt}; \mathit{swap}) \times \mathit{id}); \mathit{assocr}; (\mathit{id} \times \mathit{assocl}; (\mathit{cons} \times \mathit{id}); X); (\mathit{inr} \times \mathit{id})); \\ & \mathit{cons}) \end{aligned}$$

where $\mathit{leq} = \mathit{cmp}\,(\leq)$ and $\mathit{gt} = \mathit{cmp}\,(>)$.

As a final example, the program in Figure 3 performs the transform from Figure 1(a) to Figure 1(b). It demonstrates the use of *map*, *unzip* and *dup*. For brevity, the suffixing *id* in $\mathit{dup}\,(\mathit{fst}; \mathit{id})$ will be omitted.

$$\begin{aligned} \mathit{mktoc} &= \mathit{denode}\,\texttt{article}; \mathit{cons}^\smile; (h1 \times \mathit{cont}); \mathit{cons}; \mathit{ennode}\,\texttt{html} \\ h1 &= \mathit{denode}\,\texttt{title}; \mathit{ennode}\,\texttt{h1} \\ \mathit{cont} &= \mathit{extract}; (\mathit{enlist} \times \mathit{body}); \mathit{cons} \\ \mathit{extract} &= \mathit{map}\,(\mathit{denode}\,\texttt{section}; \mathit{cons}^\smile; (\mathit{denode}\,\texttt{title} \times \mathit{id}); \mathit{dup}\, \mathit{fst}; \mathit{swap}); \mathit{unzip} \\ \mathit{enlist} &= \mathit{map}\,(\mathit{ennode}\,\texttt{li}); \mathit{ennode}\,\texttt{ol} \\ \mathit{body} &= \mathit{map}\,((\mathit{ennode}\,\texttt{h3} \times \mathit{id}); \mathit{cons}; \mathit{ennode}\,\texttt{div}) \end{aligned}$$

$$\begin{aligned} \mathit{denode}\, s &= \mathit{node}^\smile; \mathit{swap}; \mathit{eq}\,(\mathit{str}\, s) \\ \mathit{ennode}\, s &= (\mathit{denode}\, s)^\smile \end{aligned}$$

Fig. 3. An Inv program performing the transform from Figure 1(a) to Figure 1(b). String constants are written in typewriter font.

2.4 The View-Updating Problem

Now consider the scenario of an editor, where a source document is transformed, via an Inv program, to a view editable by the user. Consider the transform $\mathit{toc} = \mathit{map}\,(\mathit{dup}\, \mathit{fst}); \mathit{unzip}$, we have:

$$\mathit{toc}\,[(1, \text{“}a\text{”}), (2, \text{“}b\text{”}), (3, \text{“}c\text{”})] = ([(1, \text{“}a\text{”}), (2, \text{“}b\text{”}), (3, \text{“}c\text{”})], [1,2,3])$$

Think of each pair as a section and the numbers as their titles, the function *toc* is a simplified version of the generation of a table of contents, thus the name.

Through a special interface, there are several things the user can do: change the value of a node, insert a new node, or delete a node. Assume that the user changes the value 3 in the "table of contents" to 4:

$$([(1, \text{“}a\text{”}), (2, \text{“}b\text{”}), (3, \text{“}c\text{”})], [1, 2, 4])$$

Now we try to perform the transformation backwards. Applying the reverse operator to *toc*, we get $(map\,(dup\,fst); unzip)^{\smile} = unzip^{\smile}; map\,(eq\,fst)$. Applying it to the modified view, $unzip^{\smile}$ maps the modified view to:

$$[((1, \text{“}a\text{”}), 1), ((2, \text{“}b\text{”}), 2), ((3, \text{“}c\text{”}), 4)]$$

pairing the sections and the titles together, to be processed by $map\,(eq\,fst)$. However, $((3, \text{“}c\text{”}), 4)$ is not in the domain of $eq\,fst$ because the equality check fails. We wish that $eq\,fst$ would return $(4, \text{“}c\text{”})$ in this case, answering the user's wish to change the section title.

Now assume that the user inserts a new section title in the table of contents:

$$([(1, \text{“}a\text{”}), (2, \text{“}b\text{”}), (3, \text{“}c\text{”})], [1, 2, 4, 3])$$

This time the changed view cannot even pass $unzip^{\smile}$, because the two lists have different lengths. We wish that $unzip^{\smile}$ would somehow know that the two 3's should go together and the zipped list should be

$$[((1, \text{“}a\text{”}), 1), ((2, \text{“}b\text{”}), 2), (\bot, 4), ((3, \text{“}c\text{”}), 3)]$$

where $\bot$ denotes some unconstrained value, which would be further constrained by $map\,(dup\,fst)$ to $(4, \bot)$. The Inv construct $eq\,fst$ should also recognise $\bot$ and deal with it accordingly.

In short, we allow the programmer to write Inv transforms that are not surjective. Therefore it is very likely that a view modified by the user may fall out of the range of the transform. This is in contrast of the approach taken in [12] and [11]. The two problems we discussed just now are representative of the view-updating problem. There are basically two kinds of dependency we have to deal with: element-wise dependency, stating that two pieces of primary-typed data have the same value, and structural dependency, stating that two pieces of data have the same shape.

One possible solution is to provide an alternative semantics that extends the ranges of Inv constructs in a reasonable way, so that the unmodified, or barely modified programs can deal with the changes. We will discuss this in detail in the next section.

3 Alternative Semantics

We will need some labels in the view, indicating "this part has been modified by the user." We extend the *View* data type as described below:

$$\begin{aligned} View &::= \ldots \mid *Int \mid *String \\ List\ a &::= \ldots \mid a \oplus List\ a \mid a \ominus List\ a \end{aligned}$$

Here the $*$ mark applies to atomic types only, indicating that the value has been changed. The view $a \oplus x$ denotes a list $a{:}\,x$ whose head a was freshly inserted by the user, while $a \ominus x$ denotes a list x which used to have a head a but was deleted. The deleted value a is still cached for future use. The two operators associate to the right, like the cons operator (:). A similar set of operators can be introduced for *Tree* but they are out of the scope of this paper.

The original semantics of each Inv program is an injective function. When the tags are involved, however, we lost the injectivity. Multiple views may be mapped

to the same source. For example, the value 1 is mapped to $(1, 1)$ by δ. In the reverse direction, $(n, *1)$ and $(*1, n)$, for all numerical n, are all mapped to 1. Similarly, all these views are mapped back to $[1, 2, 3]$ when the transform is *map succ*: $[2, 3, 4]$, $a \ominus [2, 3, 4]$, $2 : a \ominus [3, 4]$, $2 \oplus [3, 4]$, $2 : 3 \oplus [4]$ for all a.

We define two auxiliary functions *notag*? and *ridtag*. The former is a partial function letting through the input view unchanged if it contains no tags. The latter gets rid of the tags in a view, producing a normal form. Their definitions are trivial and omitted. The behaviour of the editor, on the other hand, is specified using two functions get_X and put_X, both parameterised by an Inv program X:

$$\begin{aligned} get_X &= notag?; [\![X]\!] \\ put_X &\dot{\subseteq} [\![X^{\smile}]\!]; ridtag \end{aligned}$$

The function get_X maps the source to the view by calling X. The function put_X, on the other hand, maps the (possibly edited) view back to the document by letting it go though $X^{\smile}$ and removing the tags in the result. Here $\dot{\subseteq}$ denotes "functional refinement", defined by $f \dot{\subseteq} g$ if and only if $f \subseteq g$ and $dom\, f = dom\, g$. In general $[\![X^{\smile}]\!]; ridtag$ is not a function since $[\![X^{\smile}]\!]$ may leave some values unspecified. However, any functional refinement of $[\![X^{\smile}]\!]; ridtag$ would satisfy the properties we want. The implementation can therefore, for example, choose an "initial value" for each unspecified value according to its type. The initial view is obtained by a call to get_X. When the user performs some editing, the editor applies put_X to the view, obtaining a new source, before generating a new view by calling get_X again.

In the original semantics of Inv, the $^{\smile}$ operator is simply relational converse. In the extended semantics, the $^{\smile}$ operator deviates from relational converse for three constructs: δ, **cons** and *sync*, to be introduced later. For other cases we still have $[\![f^{\smile}]\!] = [\![f]\!]^{\circ}$. The distributivity rules of $^{\smile}$ given in Section 2.2 are still true.

In the next few sub-sections we will introduce extensions to the original semantics in the running text. A summary of the resulting semantics will be given in the end of Section 3.2.

3.1 Generalised Equality Test

The simple semantics of $\delta_A^{\smile}$, where A is an atomic type, is given by the set $\{((n, n), n) \mid n \in A\}$. To deal with editing, we generalise its semantics to:

$$\begin{aligned} [\![\delta^{\smile}]\!]\,(n, n) &= n & [\![\delta^{\smile}]\!]\,(*n, *n) &= *n \\ [\![\delta^{\smile}]\!]\,(*n, m) &= *n & [\![\delta^{\smile}]\!]\,(m, *n) &= n \end{aligned}$$

When the two values are not the same but one of them was edited by the user, the edited one gets precedence and goes through. Therefore $(*n, m)$ is mapped to $*n$. If both values are edited, however, they still have to be the same. Note that the semantics of δ does not change. Also, we are still restricted to atomic types. One will have to call $map\, \delta; unzip$ to duplicate a list, thereby separate the value and structural dependency.

The syntax of *dup* can be extended to allow, a possibly non-injective function. The results of the non-injective function, and those derive from them, are supposed to be non-editable. It is a useful functionality but we will not go into its details.

3.2 Insertion and Deletion

Recall *unzip* defined in Section 2.3. Its reverse, according to the distributivity of $^{\smile}$, is given by:

$$\begin{aligned} unzip^{\smile} = \mu(X : (nil^{\smile} \times nil^{\smile}); \delta^{\smile}; nil \cup \\ (cons^{\smile} \times cons^{\smile}); trans; (id \times X); cons) \end{aligned}$$

The puzzle is: how to make it work correctly with the presence of $\ominus$ and $\oplus$ tags? We introduce several new additional operators and types:

- two new Inv operators, *del* and *ins*, both parameterised by a view. The function $del\ a :: [A] \to [A]$ introduces an $(a \ominus)$ tag, while $ins\ a :: [A] \to [A]$ introduces an $(a \oplus)$ tag.
- two kinds of pairs in *View*: positive $(a, b)^+$ and negative $(a, b)^-$. They are merely pairs with an additional label. They can be introduced only by the reverse of $fst_b^\pm$ and $snd_a^\pm$ functions to be introduced below. The intention is to use them to denote pairs whose components are temporary left there for some reason.
- six families of functions $fst_a^\square$ and $snd_a^\square$, where $\square$ can be either $+$, $-$, or nothing, defined by

  $$fst_b^\square\ (a, b)^\square = a$$
  $$snd_a^\square\ (a, b)^\square = b$$

  That is, fst_b^+ eliminates the second component of a positive pair only if it equals b. Otherwise it fails. Similarly, snd_a eliminates the first component of an ordinary pair only of it equals a. When interacting with existing operators, they should satisfy the algebraic rules in Figure 4. In order to shorten the presentation, we use $\square$ to match $+$, $-$ and nothing, while $\pm$ matches only $+$ and $-$. The $\square$ and $\pm$ in the same rule must match the same symbol.

With the new operators and types, an extended *unzip* capable of dealing with deletion can be extended from the original *unzip* by (here "..." denotes the original two branches of *unzip*):

$$\begin{aligned} unzip^\smile = \mu(X : \ldots \forall a, b \cdot\ & \\ & ((ins\ a)^\smile \times (ins\ b)^\smile); X; ins\ (a, b)) \cup \\ & ((ins\ a)^\smile \times isList); X; ins\ (a, b)) \cup \\ & (isList \times (ins\ b)^\smile); X; ins\ (a, b)) \cup \\ & ((del\ a)^\smile \times (del\ b)^\smile); X; del\ (a, b)) \cup \\ & ((del\ a)^\smile \times cons^\smile; snd_b^-); X; del\ (a, b)) \cup \\ & (cons^\smile; snd_a^- \times (del\ b)^\smile); X; del\ (a, b)) \end{aligned}$$

where a and b are universally quantified, and $isList = nil^\smile; nil \cup cons^\smile; cons$, a subset of *id* letting through only lists having no tag at the head.

Look at the branch starting with $((ins\ a)^\smile \times (ins\ b)^\smile)$. It says that, given a pair of lists both starting with insertion tags $a \oplus$ and $b \oplus$, we should deconstruct them, pass the tails of the lists to the recursive call, and put back an $((a, b) \oplus)$ tag. If only the first of them is tagged (matching the branch starting with $((ins\ a)^\smile \times isList)$), we temporarily remove the $a \oplus$ tag, recursively process the lists, and put back a tag $((a, b) \oplus)$ with a freshly generated b. The choice of b is non-deterministic and might be further constrained when *unzip* is further composed with other relations. The situation is similar with deletion. In the branch starting with $(del\ a \times snd_b^{+\circ}; cons)$ where we encounter a list with an a deleted by the user, we remove an element in the other list and remember its value in b. Here universally quantified b is used to match the value — all the branches with different b's are unioned together, with only one of them resulting in a successful match.

It would be very tedious if the programmer had to explicitly write down these extra branches for all functions. Luckily, these additional branches can be derived automatically using the rules in Figure 4. In the derivations later we will omit the semantics function $[\![\ ]\!]$ and use the same notation for the language and its semantics, where no confusion would occur. This is merely for the sake of brevity.

In place of ordinary *cons*, we define two constructs addressing the dependency of structures. Firstly, the *bold* **cons** is defined by::

$$\mathbf{cons} = cons \cup \bigcup_{a::A}(snd_a^-; del\ a) \cup \bigcup_{a::A}(snd_a^+; ins\ a)$$

$$(f \times g); fst^{\square}_{(g\ b)} = fst^{\square}_b; f, \text{ if } g \text{ total} \qquad assocl; (fst^{\square}_b \times id) = (id \times snd^{\square}_b)$$
$$(f \times g); snd^{\square}_{(f\ a)} = snd^{\square}_a; g, \text{ if } f \text{ total} \qquad assocl; (snd^{\square}_a \times id) = (snd^{\square}_a \cup snd_a)$$
$$swap; snd^{\square}_a = fst^{\square}_a \qquad assocl; snd^{\square}_{(a,b)} = snd^{\square}_a; (snd^{\square}_b \cup snd_b)$$
$$snd^{\square\smile}_a; eq\ nil = (\lambda [\,] \to a)$$

Fig. 4. Algebraic rules. Here $(\lambda [\,] \to a)$ is a function mapping only empty list to a. Only rules for *assocl* are listed. The rules for *assocr* can be obtained by pre-composing *assocr* to both sides and use $asscor; assocl = id$. Free identifiers are universally quantified.

Secondly, we define the following *sync* operator:

$$\begin{aligned}
sync &= (cons \times cons)\\
sync^{\smile} &= (cons^{\smile} \times cons^{\smile})\\
&\quad \cup \bigcup_{a,b\in A}(((del\ a)^{\smile}; snd^{-\smile}_a \times (del\ b)^{\smile}; snd^{-\smile}_b)\\
&\qquad \cup ((del\ a)^{\smile}; snd^{-\smile}_a \times cons^{\smile}; snd_b; snd^{-\smile}_b)\\
&\qquad \cup (cons^{\smile}; snd_a; snd^{-\smile}_a \times (del\ b)^{\smile}; snd^{-\smile}_b))\\
&\quad \cup \bigcup_{a,b\in A}(((ins\ a)^{\smile}; snd^{+\smile}_a \times (ins\ b)^{\smile}; snd^{+\smile}_b)\\
&\qquad \cup ((ins\ a)^{\smile}; snd^{+\smile}_a \times isList; snd^{+\smile}_b)\\
&\qquad \cup (isList; snd^{+\smile}_b \times (ins\ b)^{\smile}; snd^{+\smile}_b))
\end{aligned}$$

In the definition of *unzip*, we replace every singular occurence of *cons* with **cons**, and every $(cons \times cons)$ with *sync*. The definition of $sync^{\smile}$ looks very complicated but we will shortly see its use in the derivation. Basically every product corresponds to one case we want to deal with: when both the lists are cons lists, when one or both of them has a $\ominus$ tag, or when one or both of them has a $\oplus$ tag.

After the substitution, all the branches can be derived by algebraic reasoning. The rules we need are listed in Figure 4. To derive the first branch for insertion, for example, we reason:

$$\begin{aligned}
&unzip^{\smile}\\
\supseteq\ &\{\text{fixed-point}\}\\
&sync^{\smile}; trans; (id \times unzip); \mathbf{cons}\\
\supseteq\ &\{\text{since } sync^{\smile} \supseteq ((ins\ a)^{\smile}; snd^{+\smile}_a \times (ins\ b)^{\smile}; snd^{+\smile}_b) \text{ for all } a, b\}\\
&((ins\ a)^{\smile} \times (ins\ b)^{\smile}); (snd^{+\smile}_a \times snd^{+\smile}_b); trans; (id \times unzip); \mathbf{cons}\\
\supseteq\ &\{\text{claim: } (snd^{+\smile}_a \times snd^{+\smile}_b); trans = (snd^{+}_{(a,b)})^{\smile}\}\\
&((ins\ a)^{\smile} \times (ins\ b)^{\smile}); (snd^{+}_{(a,b)})^{\smile}; (id \times unzip); \mathbf{cons}\\
=\ &\{\text{since } (f \times g); snd^{+}_{f\ a} = snd^{+}_a; g \text{ for total } f\}\\
&((ins\ a)^{\smile} \times (ins\ b)^{\smile}); unzip; (snd^{+}_{(a,b)})^{\smile}; \mathbf{cons}\\
\supseteq\ &\{\text{since } \mathbf{cons} \supseteq snd^{+}_{(a,b)}; ins\ (a, b)\}\\
&((ins\ a)^{\smile} \times (ins\ b)^{\smile}); unzip; (snd^{+}_{(a,b)})^{\smile}; snd^{+}_{(a,b)}; ins\ (a, b)\\
=\ &\{\text{since } snd^{+\smile}_x; snd^{+}_x = id\}\\
&((ins\ a)^{\smile} \times (ins\ b)^{\smile}); unzip; ins\ (a, b)
\end{aligned}$$

We get the first branch. The claim that $trans^{\smile}; (snd^{\square}_a \times snd^{\square}_b) = snd^{\square}_{(a,b)}$ can be verified by the rules in Figure 4 and is left as an exercise. The introduction of two kinds of pairs was to avoid the suffix being reduced to $(del\ (a, b))^{\smile}$ in the last two steps. To derive one of the branches for deletion, on the other hand, one uses

the inclusion $sync^{\smile} \supseteq ((del\ a)^{\smile}; snd_a^{-\smile} \times cons^{\smile}; snd_b; snd_b^{-\smile})$ for the first step, and $\mathbf{cons} \supseteq snd_{(a,b)}^{-}; del\,(a,b)$ and $(snd_{(a,b)}^{-})^{\smile}; snd_{(a,b)}^{-} = id$ fort the last step. All the branches can be derived in a similar fashion.

$$
\begin{array}{ll}
[\![nil]\!]\,() & = [\,] \\
[\![zero]\!]\,() & = 0 \\
[\![succ]\!]\,n & = n+1 \\
[\![cons]\!]\,(a,x) & = a{:}\,x \\
[\![node]\!]\,(a,x) & = Node\ a\ x \\
[\![inl]\!]\,a & = L\,a \\
[\![inr]\!]\,a & = R\,a \\
[\![id]\!]\,a & = a
\end{array}
$$

$$
\begin{array}{ll}
[\![swap]\!]\,(a,b)^{\square} & = (b,a)^{\square} \\
[\![assocr]\!]\,((a,b)^{\pm},c)^{\pm} & = (a,(b,c)^{\pm})^{\pm} \\
[\![assocr]\!]\,((a,b)^{\pm},c) & = (a,(b,c)^{\pm}) \\
[\![assocr]\!]\,((a,b),c)^{\pm} & = (a,(b,c))^{\pm} \\
assocl & = assocr^{\smile}
\end{array}
$$

$$(f^{\smile})^{\smile} = f$$

$$
\begin{array}{ll}
[\![\delta]\!]\,n & = (n,n) \\
[\![\delta^{\smile}]\!]\,(n,n)^{\square} & = n \\
[\![\delta^{\smile}]\!]\,(*n,*n)^{\square} & = *n \\
[\![\delta^{\smile}]\!]\,(*n,m)^{\square} & = *n \\
[\![\delta^{\smile}]\!]\,(m,*n)^{\square} & = *n
\end{array}
$$

$$
\begin{array}{ll}
[\![dup\ nil]\!]\,a & = (a,[\,]) \\
[\![(dup\ nil)^{\smile}]\!]\,(a,[\,])^{\square} & = a \\
[\![dup\ zero]\!]\,a & = (a,0) \\
[\![(dup\ zer0)^{\smile}]\!]\,(a,0)^{\square} & = a \\
[\![dup\,(str\ s)]\!]\,a & = (a,s) \\
[\![(dup\,(str\ s))^{\smile}]\!]\,(a,s)^{\square} & = a
\end{array}
$$

$$
\begin{array}{l}
\mathbf{cons} = cons \\
\qquad \cup \bigcup_{a::A}(snd_a^{-}; del\ a) \\
\qquad \cup \bigcup_{a::A}(snd_a^{+}; ins\ a)
\end{array}
$$

$$
\begin{array}{ll}
[\![cmp\ \trianglelefteq]\!]\,(a,b)^{\square} & = (a,b)^{\square}, \text{ if } a \trianglelefteq b \\
[\![f;g]\!]\,x & = [\![g]\!]\,([\![f]\!]\,x) \\
[\![f \times g]\!]\,(a,b)^{\square} & = ([\![f]\!]\,a, [\![g]\!]\,b)^{\square} \\
[\![f \cup g]\!] & = [\![f]\!] \cup [\![g]\!], \\
 & \quad \text{if } dom\,f \cap dom\,g = ran\,f \cap ran\,g = \emptyset \\
[\![\mu F]\!] & = [\![F\,\mu F]\!]
\end{array}
$$

$$
\begin{array}{ll}
[\![f^{\smile}]\!] & = [\![f]\!]^{\circ} \\
[\![f;g^{\smile}]\!] & = [\![g^{\smile}]\!]; [\![f^{\smile}]\!] \\
[\![(f \times g)^{\smile}]\!] & = [\![(f^{\smile} \times g^{\smile})]\!] \\
[\![(f \cup g)^{\smile}]\!] & = [\![f^{\smile}]\!] \cup [\![g^{\smile}]\!] \\
[\![\mu F^{\smile}]\!] & = [\![\mu(X \to (F\,X^{\smile})^{\smile}]\!]
\end{array}
$$

$$
\begin{array}{ll}
[\![fst_a^{\square}]\!]\,(a,b)^{\square} & = b \\
[\![snd_b^{\square}]\!]\,(a,b)^{\square} & = a \\
[\![del\ a]\!]\,(a \ominus x) & = (a,x)^{-} \\
[\![ins\ a]\!]\,(a \oplus x) & = (a,x)^{+}
\end{array}
$$

$$
\begin{array}{ll}
dup\ id & = \delta \\
dup\,(fst;P) & = (dup\ P \times id); subl \\
dup\,(snd;P) & = (id \times dup\ P); assocl \\
dup\,(cons^{\smile};P) & = cons^{\smile}; dup\ P; (cons \times id) \\
dup\,(node^{\smile};P) & = node^{\smile}; dup\ P; (node \times id)
\end{array}
$$

$$
\begin{array}{l}
sync\ = (cons \times cons) \\
sync^{\smile} = (cons^{\smile} \times cons^{\smile}) \\
\cup \bigcup_{a,b \in A}(((del\ a)^{\smile}; snd_a^{-\smile} \times (del\ b)^{\smile}; snd_b^{-\smile}) \\
\qquad \cup ((del\ a)^{\smile}; snd_a^{-\smile} \times cons^{\smile}; snd_b; snd_b^{-\smile}) \\
\qquad \cup (cons^{\smile}; snd_a; snd_a^{-\smile} \times (del\ b)^{\smile}; snd_b^{-\smile})) \\
\cup \bigcup_{a,b \in A}(((ins\ a)^{\smile}; snd_a^{+\smile} \times (ins\ b)^{\smile}; snd_b^{+\smile}) \\
\qquad \cup ((ins\ a)^{\smile}; snd_a^{+\smile} \times isList; snd_b^{+\smile}) \\
\qquad \cup (isList; snd_b^{+\smile} \times (ins\ b)^{\smile}; snd_b^{+\smile}))
\end{array}
$$

Fig. 5. Summary of the alternative semantics. The patterns should be matched from the top-left to bottom-left, then top-right to bottom-right.

3.3 The Put-Get-Put Property and Galois Connection

A *valid* Inv program is one that does not use $fst_a^{\square}$ and $snd_b^{\square}$ apart from in **cons** and *sync*. The domain of get_X, for a valid X, is restricted to tag-free views, so is its range. In fact, $notag?; [\![X]\!]$ reduces to the injective function defined by the original semantics. Therefore, $get_X; get_X{}^{\circ} = dom\ get_X$. Furthermore, $notag?; ridtag = notag?$. As a result, for all valid Inv programs X we have the following *get-put* property:

$$get_X; put_X = dom\ get_X \tag{1}$$

This is a desired property for our editor: mapping an unedited view back to the source always gives us the same source document.

On the other hand, $put_X; get_X \subseteq id$ is not true. For example, $(put_\delta; get_\delta)\,(*a, b) = (a, a) \neq (*a, b)$. This is one of the main differences between our work and that of [12] and [11]. They both assume the relation X to be bi-total, and that the *put-get* property $put_X; get_X = id$ holds. It also implies that duplication cannot be allowed in the language.

Instead, we have a weaker property. First of all, for all valid X we have $dom\ get_X \subseteq ran\ put_X$. That is, every valid source input to get_X must be a result of put_X for at least one view, namely, the view the source get mapped to under the original semantics. Pre-composing $put\ X$ to (1) and use $put_X; dom\ get_X \subseteq put_X; ran\ put_X = put_X$, we get the following *put-get-put* property:

$$put_X; get_X; put_X \subseteq put_X \tag{2}$$

When the user edits the view, the editor calls the function $put\ X$ to calculate an updated source, and then calls get_X to update the view as well. For example, $(*a, b)$ is changed to (a, a) after $put_\delta; get_\delta$. With the *put-get-put* property we know that another put_X is not necessary, because it is not going to change the view — the result of $put_X; get_X; put_X$, if anything, is the same as that of put_X.

It is desirable to have $put_X; get_X; put_X = put_X$. However, this is not true, and $dom\ get_X \neq ran\ put_X$. For a counter-example, take $X = (\delta \times id); assocr; (id \times \delta)$. The function get_X takes only pairs with equal components and returns it unchanged. Applying put_X to $(*b, a)$ results in (b, a), which is not in the domain of get_X. Such a result is theoretically not satisfactory, but does not cause a problem for our application. The editor can signal an error to the user, saying that such a modification is not allowed, when the new source is not in the domain of get_X. The domain check is not an extra burden since we have to call get_X anyway.

A Galois connection is a pair of functions $f :: A \to B$ and $g :: B \to A$ satisfying

$$f\ x \trianglelefteq y \equiv x \preceq g\ y \tag{3}$$

Galois connected functions satisfy a number of properties, including $f; g; f = f$. For those X that $dom\ get_X = ran\ put_X$ do hold, get_X and put_X satisfy (3), if we take $\preceq$ to be equality on tag-free *View*s and $\trianglelefteq$ to be $(put_X; get_X)^\circ$. That is, $s \preceq s'$ if and only if the two sources s and s' are exactly the same, while a view v is no bigger than v' under $\trianglelefteq$ if there exists a source s such that $v = get_X\ s$ and $s = put\ v'$. For example, (n, n) is no bigger than $(*n, m)$, $(m, *n)$, $(*n, *n)$, and (n, n) itself under $\trianglelefteq$, when the transform is δ. The only glitch here is that $\trianglelefteq$ is not reflexive! In fact it is reflexive only in the range of get_X — the set of tag-free views. However, this is enough for get_X and put_X to satisfy most properties of a Galois connection.

3.4 Implementation Issues

In our experimental implementation, we have a simple interpreter for Inv. One way to incorporate the algebraic rules in the previous section in the implementation is to call a pre-processor before the program is interpreted. Another possibility is to build the rules implicitly in the interpreter. In this section we will talk about how.

The abstract syntax tree of Inv is extended with new constructs **cons** and *sync*. The "intermediate" functions introduced in the last section, namely *ins*,*del*, $fst^\pm$s and $snd^\pm$s, are not actually represented in the abstract syntax. Instead, we extend the value domain *View* with additional constructs:

$$\begin{aligned} View ::= \ & \ldots \mid (View, {}_+View) \mid ({}_+View, View) \\ & \mid (View, {}_-View) \mid ({}_-View, View) \\ & \mid \bot \mid NilTo\ View \end{aligned}$$

Conceptually, after we apply $snd_a^{+\smile}$ to a value b, we get $({}_+a, b)$, while $({}_-a, b)$ is the result of applying $snd_a^{-\smile}$ to b. The reader can think of them as a note saying "the

value should have been b only, but we temporarily pair it with an a, just to allow the computation to carry on." Or one can think of it as a pending application of snd_a^+ or snd_a^-. The $\perp$ symbol denotes an unconstrained value. Finally, *NilTo a* denotes a function taking only $[\,]$ and returns a.

To implement the *sync* operator, we add the following definitions (some cases are omitted):

$$\begin{array}{ll} [\![sync^\smile]\!]\,(a\colon x, b\colon y) = ((a,x),(b,y)) & [\![sync^\smile]\!]\,(a \oplus x, y) = ((_+a,x),(_+\perp,y)) \\ [\![sync^\smile]\!]\,(a \ominus x, b\colon y) = ((\text{-}a,x),(\text{-}b,y)) & [\![sync^\smile]\!]\,(x, b \oplus y) = ((_+\perp,x),(_+b,y)) \\ [\![sync^\smile]\!]\,(a\colon x, b \ominus y) = ((\text{-}a,x),(\text{-}b,y)) & \end{array}$$

The first clause is simply what $(cons \times cons)^\circ$ would do. The second clause shows that when there is a deletion in the first list, we throw away an element in the second list as well, while keeping note of the fact by the $(\text{-}_-, \_)$ tag. It corresponds to the $(del\ a^\circ; snd_a^{-\circ} \times cons^\circ; snd_b^-; snd_b^{-\circ})$ branch of $(cons \mathbin{\hat{\times}} cons)^\circ$. The fourth branch, on the other hand, corresponds to $((ins\ a)^\circ; snd_a^{+\circ} \times isList; snd_b^{+\circ})$. The newly introduced, unconstrained value b is represented by $\perp$.

Now we build in some extra rules for *cons* and $cons^\smile$:

$$\begin{array}{ll} [\![cons]\!]\,(\text{-}a, x) = a \ominus x & [\![cons^\smile]\!]\,(a \ominus x) = (\text{-}a, x) \\ [\![cons]\!]\,(_+a, x) = a \oplus x & [\![cons^\smile]\!]\,(a \oplus x) = (_+a, x) \end{array}$$

They correspond to the fact that $snd^\smile; \mathbf{cons} = del$ and $snd_a{}^\smile; \mathbf{cons} = ins\ a$. Also, some additional rules for *assocr*:

$$\begin{array}{ll} [\![assocr]\!]\,((a,{}_+b),c) = (a,(_+b,c)) & [\![assocr]\!]\,(_+(a,b),c) = (_+a,(_+b,c)) \\ [\![assocr]\!]\,((_+a,b),c) = (_+a,(b,c)) & \end{array}$$

The three clauses correspond to the rules for *assocl* in the left column of Figure 4. Finally we need some rules for *dup nil* and its inverse *eq nil*:

$$[\![(eq\ nil)]\!]\,(\text{-}a,[\,]) = NilTo\ a \qquad [\![(dup\ nil)]\!]\,(NilTo\ a) = (\text{-}a,[\,])$$

which corresponds to the rule $snd_a^{\square\smile}; eq\ nil = (\lambda\,[\,] \to a)$ in Figure 4.

4 More Examples

In this section we will show more transforms defined in Inv that do satisfy $dom\ get_X = ran\ put_X$ and how they react to user editing.

4.1 Snoc and List Reversal

The function $snoc :: (a, List\ a) \to List\ a$, appending an element to the end of a list, can be defined recursively as:

$$\begin{aligned} snoc = \mu(X :\ & eq\ nil; dup\ nil; \mathbf{cons} \cup \\ & (id \times \mathbf{cons}^\circ); subr; (id \times X); \mathbf{cons}) \end{aligned}$$

For example $[\![snoc]\!]\,(4,[1,2,3]) = [1,2,3,4]$. Conversely, $snoc^\smile$ extracts the last element of a list. But what is the result of extracting the last element of a list whose last element was just removed? We expand the base case:

$$\begin{array}{ll} & snoc^\smile \\ \supseteq & \{\text{fixed-point}\} \\ & \mathbf{cons}^\smile; eq\ nil; dup\ nil \\ \supseteq & \{\text{specialising } \mathbf{cons} \supseteq snd_a^-; del\ a\} \\ & (del\ a)^\smile; snd_a^{-\smile}; ea\ nil; dup\ nil \end{array}$$

$$
\begin{array}{ll}
= & \{\text{since } \mathit{snd}_a^{-\smile}; \mathit{eq}\ \mathit{nil} = (\lambda[\,] \to a)\} \\
& (\mathit{del}\ a)^{\smile}; (\lambda[\,] \to a); \mathit{dup}\ \mathit{nil} \\
= & \{\text{since } \mathit{snd}_a^{-\smile}; \mathit{eq}\ \mathit{nil} = (\lambda[\,] \to a) \Rightarrow \mathit{snd}_a^{-\smile} = (\lambda[\,] \to a); \mathit{dup}\ \mathit{nil}\} \\
& (\mathit{del}\ a)^{\smile}; \mathit{snd}_a^{-\smile}
\end{array}
$$

That is, for example, $\mathit{eval}\ \mathit{snoc}^{\smile}\,(4 \ominus [\,]) = (\text{-}4, [\,])$. Inductively, we have $\mathit{eval}\ \mathit{snoc}^{\smile}\,(1 : 2 : 3 : 4 \ominus [\,]) = (\text{-}4, 1 : 2 : 3 : [\,])$, which is reasonable enough: by extracting the last element of a list whose last element, 4, is missing, we get a pair whose first element should not have been there.

The ubiquitous fold function on lists can be defined by

$$\mathit{fold}\ f\ g = \mu(X : \mathit{nil}^{\smile}; g\ \cup\ \mathbf{cons}^{\smile}; (\mathit{id} \times X); f)$$

The function *reverse*, reverting a list, can be defined in terms of fold as $\mathit{reverse} = \mathit{fold}\ \mathit{snoc}\ \mathit{nil}$. Unfolding its definition, we can perform the following refinement:

$$
\begin{array}{ll}
& \mathit{reverse}^{\smile} \\
\supseteq & \{\text{unfolding the definitions}\} \\
& \mathit{snoc}^{\smile}; (\mathit{id} \times \mathit{reverse}^{\smile}); \mathbf{cons} \\
\supseteq & \{\text{by the reasoning above, } \mathit{snoc}^{\smile} \supseteq (\mathit{del}\ a)^{\smile}; \mathit{snd}_a^{-\smile}\} \\
& (\mathit{del}\ a)^{\smile}; \mathit{snd}_a^{-\smile}; (\mathit{id} \times \mathit{reverse}^{\smile}); \mathbf{cons} \\
= & \{\text{since } (f \times g); \mathit{snd}_{f\ a}^{-} = \mathit{snd}_a^{-}; g \text{ for total } f\} \\
& (\mathit{del}\ a)^{\smile}; \mathit{reverse}^{\smile}; \mathit{snd}_a^{-\smile}; \mathbf{cons} \\
\supseteq & \{\text{since } \mathbf{cons} \supseteq \mathit{snd}_a^{-}; \mathit{del}\ a \text{ and } \mathit{snd}_a^{-\smile}; \mathit{snd}_a^{-} = \mathit{id}\} \\
& (\mathit{del}\ a)^{\smile}; \mathit{reverse}^{\smile}; \mathit{del}\ a
\end{array}
$$

which shows that $\mathit{reverse}^{\smile}$ regenerates the $\ominus$ tags (and, similiarly, $\oplus$ tags) upon receipt of the "partial" pairs returned by *snoc*. For example, we have $\mathit{eval}\ \mathit{reverse}\,(1 : 2 : 3 \ominus 4 : [\,]) = 4 : 3 \ominus 2 : 1 : [\,]$ which is exactly what we want. A lesson is that to deal with lists, we have to first learn to deal with pairs.

4.2 Merging and Filtering

Recall the function *merge* defined in Section 2.3, merging two sorted lists into one, while marking the elements with labels remembering where they were from:

$$\mathit{merge}\,([1, 4, 7], [2, 5, 6]) = [L\,1, R\,2, L\,4, R\,5, R\,6, L\,7]$$

Filtering is an often needed feature. For example, in a list of $(\mathit{author}, \mathit{article})$ pairs we may want to extract the articles by a chosen author. The Haskell Prelude function $\mathit{filter} :: (a \to \mathit{Bool}) \to \mathit{List}\ a \to \mathit{List}\ a$, returning only the elements in the list satisfying a given predicate, however, is not injective because it throws away some items. A common scenario of filtering is when we have a list of *sorted* items to filter. For example, the articles in the database may be sorted by the date of creation, and splitting the list retains the order. If we simplify the situation a bit further, it is exactly the converse of what *merge* does, if we think of L and R as true and false!

To make *merge* work with editing tags, we simply replace every occurrence of *cons* with **cons**, including the *cons* in $(\mathit{cons} \times \mathit{cons})$. This time the latter shall not be replaced by *sync* because we certainly do not want to delete or invent elements in one list when the user edits the other! This *merge* does behave as what we would expect. For example, when an element is added to the split list:

$$\mathit{merge}\,(1 : 3 \oplus 4 : 7 : [\,], [2, 5, 6]) = L\,1 : R\,2 : L\,3 \oplus [L\,4, R\,5, R\,6, L\,7]$$

the new element is inserted back to the original list as well.

5 Conclusion

Bi-directional updating, though an old problem[5, 7, 10, 14, 1], has recently attracted much interests, each took a slightly different approach according to their target application. We have developed a formalisation of bi-directional updating which is able to deal with duplication and structural changes like insertion and deletion. From a specification X, written as an injective function, we induce two functions get_X and put_X that satisfy the important *get-put* and *put-get-put* properties. To find out how put_X reacts to user editing, one can make use of algebraic reasoning, which also provides a hint how the formalisation can be implemented in an interpreter.

Our formalisation deals with duplication and structural changes at the cost of introducing editing tags, which is okay for our application — to integrate it to our structural editor in [15]. The complementary approach taken by [11], on the other hand, chooses not to use any information how the new view was constructed. Upon encountering inconsistency, the system generates several possible ways to resolve the inconsistency for the user to choose from. It would be interesting to see whether there is a general framework covering both approaches.

Another feature of our work is the use of an injective language, and various program derivation and inversion techniques. The injective language Inv has been introduced in [13], where it is also described how to automatically derive an injective variant for every non-injective program. So far we have a primitive implementation. For an efficient implementation, however, the techniques described in [9] based on parsing may be of help.

The history of point-free functional languages many be traced to Backus's FP [4], although our style in this paper is more influenced by [6]. Readers who tend to equate the use of point-free languages with being "not practical" should probably be apprised of the existence of a number of well-established libraries that adopt a point-free style, such as the XML processing library HaXml [16]. Furthermore, there is a tedious, uninteresting way of converting a certain class of pointwise functional programs into Inv. The class of programs is essentially the same as the source language in [9], that is, first-order functional programs with disjoint, linear patterns in case expressions.

Acknowledgements

The idea of using algebraic rules and program transformation to guide the processing of editing tags was proposed by Lambert Meertens during the first author's visit to Kestrel Institute, CA. The authors would like to thank Johan Jeuring for useful improvements to an earlier draft of this paper, and the members of the Programmable Structured Document in Information Processing Lab, Tokyo University for valuable discussions. This research is partly supported by the e-Society Infrastructure Project of the Ministry of Education, Culture, Sports, Science and Technology, Japan.