Incremental Updates for Efficient Bidirectional Transformations

Meng Wang
Computer Science and Engineering
Chalmers University of Technology
412 96 Göteborg, Sweden
wmeng@chalmers.se

Jeremy Gibbons
Department of Computer Science
University of Oxford
Wolfson Building, Parks Road
Oxford OX1 3QD, UK
jeremy.gibbons@cs.ox.ac.uk

Nicolas Wu
Well-Typed LLP
Oxford, UK
nick@well-typed.com

Abstract

A bidirectional transformation is a pair of mappings between *source* and *view* data objects, one in each direction. When the view is modified, the source is updated accordingly. The key to handling large data objects that are subject to relatively small modifications is to process the updates incrementally. Incrementality has been explored in the semi-structured settings of relational databases and graph transformations; this flexibility in structure makes it relatively easy to divide the data into separate parts that can be transformed and updated independently. The same is not true if the data is to be encoded with more general-purpose algebraic datatypes, with transformations defined as functions: dividing data into well-typed separate parts is tricky, and recursions typically create interdependencies. In this paper, we study transformations that support incremental updates, and devise a constructive process to achieve this incrementality.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages, Specialized application languages; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures, Polymorphism

General Terms Languages, Design, Performance, Theory

Keywords Functional Programming, Bidirectional Programming, Incremental Computing, Program Transformation, View-update Problem

1. Introduction

From a programming perspective, bidirectional programming is an exercise in writing programs that execute both forwards and backwards. This goal is not always achievable: non-injective functions obscure the route back, while non-surjective functions leave the 'backward' execution partial. Nevertheless, the need to invert a computation does arise in many contexts in which pairs of programs naturally interact with one another; parser/printer, embedding/projection, marshalling/unmarshalling, and compression/decompression are typical examples. For decades, the needs of bidirectionality have been fulfilled by individually crafted pairs of programs. This is a significant duplication of work, because the two programs in a pair are closely related. Furthermore, maintaining the relationship between these progams becomes a source of errors, where changes to only one of the programs may lead to an inconsistency within the pair.

A more promising solution to the challenge of bidirectional programming is to design languages that execute bidirectionally. In such languages, any user-defined program, mapping a source into a view, is coupled with an automatically generated 'backward' version, whose correctness with regards to certain bidirectional properties is guaranteed. An obvious bidirectional property is invertibility. For a given program *fwd*, its inverse *bwd* satisfies $fwd \circ bwd = id$ and $bwd \circ fwd = id$. While this invertibility is elegant, it restricts the expressiveness of the approach, since *fwd* must be bijective. Modulo information encoded in *fwd* itself, the source and view necessarily contain exactly the same information, probably with different presentations. This is an unrealistically strong assumption, and has driven research into more widely applicable frameworks:

> "More generally, bijective transformations are the exception rather than the rule: the fact that one model contains information not represented in the other is part of the reason for having separate models." [27]

1.1 Semi-Invertibility

To circumvent the bijectivity restriction, other languages have been designed with particular applications in mind, where less demanding constraints such as one-sided invertibility are imposed: left-invertibility ($fwd \circ bwd = id$) in [24], and right-invertibility ($bwd \circ fwd = id$) in [31]. A more dramatic diversion from the invertibility framework is the *lens* approach [10, 4, 12, 11, 3, 18], which originated from the study of the *view-update problem* of databases [7, 2, 14]. In the setting of lenses, (all or part of) the original source is copied and is used in the 'backward' computation: a 'get' function has type $s \to v$ from source to view, while a 'put' function has type $(v,s) \to s$. We make the following notational convention: for a 'get' function f, the counterpart 'put' function is written as $f^<$. The task of a 'put' function is to discover the connection between a view update and an appropriate corresponding source update. This 'appropriateness' is defined by the *definitional properties* [6, 27]:

Consistency $f\ (f^<\ (v,s)) = v$

Acceptability $f^<\ (f\ s,s) = s$

Here *consistency* (also known as the PutGet law) roughly corresponds to right-invertibility, basically ensuring that all updates on a view are captured by the updated source, and *acceptability* (also known as the GetPut law) roughly corresponds to left-invertibility, prohibiting changes to the source if no update has been made on the view. Bidirectional transformations satisfying the above two laws

ICFP'11, September 19–21, 2011, Tokyo, Japan.

are sometimes called *well-behaved* [10]. In addition to these laws, an optional *undoability* property is sometimes introduced:

Undoability $f^{<}\ (f\ s, f^{<}\ (v', s)) = s$

This property states that the result of an update can be undone through the view. For the sake of this paper, we will present our proposal within the lenses framework, with the understanding that the same technique applies to invertible languages as well.

1.2 State-Based and Operation-Based Approaches

The word 'update' is commonly found in the bidirectional transformation literature, and used to describe changes both in views and in sources. In the sequel, we restrict the use of the term 'update' to the transformed effect on a source, as a result of an *edit* to the view. Despite being fundamental to bidirectional transformation, there is no universal agreement on what constitutes an edit. Roughly speaking, opinions are divided as to whether one should look into the mechanism of an edit or simply its result. Translated into language design, one can either take an operation-based approach, considering an editing function that changes a view, or a state-based approach, which only sees the unedited and edited views. It happens that the majority of existing bidirectional frameworks take the latter approach, due to its mathematical simplicity and good compatibility. The bidirectional laws discussed above are specified from a state-based perspective. Since only the edited view is required for the 'put' function, it is easier to design a bidirectional framework independently of any editing system. On the other hand, a state-based approach necessarily discards information about an edit, and must reverse-engineer it later by performing some kind of difference analysis on the two view values. Consequently, the run-time performance of a state-based 'put' function is bound by the difference analysis that is required: even a small change to the view implies a complete re-traversal.

Meertens [23] observed that to maintain constraints between two structures, it is useful to know how a view is edited. Consider the scenario where two lists are connected by a mapping relation (i.e. each is the result of *map*ping some function over the other); an edit to one list, say deletion at a particular position, can be translated to a deletion at the same position in the other list. In this setting, a lot more information about the edit is made available to the bidirectional framework, including where (the location) and what (a deletion) has changed. As a result, the updating process could be more straightforward compared to a state-based approach—in the latter, all that is known is that one list is changed into another list that is one element shorter, which is fairly ambiguous. In addition, having an operation for source update potentially achieves better-than-linear runtime performance.

If run-time performance is the only concern, the 'where' part of the knowledge of an edit is the key. For a given edit-affected view fragment, once the corresponding source fragment is picked out, a state-based approach could perform the changes as efficiently as an operation-based counterpart, without the undesired complications that the latter brings. This idea is not new. As a matter of fact, for bidirectional systems in the neighbouring fields of databases and software engineering, incrementality of source updates is the norm rather than the exception [5, 8, 15]. With the more recent upsurge in (mostly functional) programming language approaches to bidirectional transformation [19, 10, 21, 28, 25, 29], fresh challenges emerge: structured data cannot easily be divided into separate parts to be transformed and updated independently; and recursions typically create interdependencies. As a result, in contrast to the situation with databases and model transformations, none of the existing bidirectional languages supports any kind of incremental behaviour.

1.3 A Change-Based Approach

In this paper, we propose a novel *change-based* framework for bidirectional transformation. Instead of inventing from first principles to add to the already burgeoning population of bidirectional languages, we focus on the preservation and propagation of user-provided editing information. In a sense, our proposal can be seen as a generic optimiser of certain given state-based bidirectional frameworks: we exploit any locality in the editing of views, and try to translate it into incrementality in the updating of sources. Obviously, such preservation of locality does not hold for arbitrary transformations. Identifying the semantic properties, or *conditions*, required forms one of the major technical contributions of this paper. The ultimate goal of our framework is to reduce an update of a (large) structure into one of a (small) delta, and then outsource the hopefully much smaller problem of actually translating the update to an existing state-based frameworks. This step positively impacts the run-time behaviour of 'put', due to the newly gained incrementality.

Our proposal aims at modularity: there is a clearly defined interface that decouples any editing system from our framework; and different state-based bidirectional approaches can be plugged in as black boxes, whatever their manifestation as purpose-built bidirectional languages or syntactic/semantic transformations of unidirectional programs. As a result, a change-based bidirectional framework arises automatically from a given state-based one, while preserving the bidirectional properties of the latter.

We choose Haskell [26] as the language of discourse, but no specific language feature other than algebraic datatypes is required for our technique to apply.

1.4 A Small Example

As a motivating example, consider the structure of a binary tree:

$$\begin{aligned}&\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty}\\ &\qquad\qquad \mid\ \mathit{Fork}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\end{aligned}$$

One possible 'get' function from this source is an inorder traversal, which produces a list, and can be defined as follows:

$$\begin{aligned}&\mathit{inorder} :: \mathit{Tree}\ a \to [a]\\ &\mathit{inorder}\ \mathit{Empty} \quad\ = [\,]\\ &\mathit{inorder}\ (\mathit{Fork}\ a\ t\ u) = \mathit{inorder}\ t +\!\!+ [a] +\!\!+ \mathit{inorder}\ u\end{aligned}$$

For instance, a traversal of the source

$$\begin{aligned}&\mathit{Fork}\ 5\ (\mathit{Fork}\ 6\ (\mathit{Fork}\ 7\ \mathit{Empty}\ \mathit{Empty})\ \mathit{Empty})\\ &\qquad(\mathit{Fork}\ 8\ (\mathit{Fork}\ 4\ \mathit{Empty}\ \mathit{Empty})\ (\mathit{Fork}\ 9\ \mathit{Empty}\ \mathit{Empty}))\end{aligned}$$

yields the list $[7,6,5,4,8,9]$. Now, suppose the number 4 is deleted from this view. A state-based 'put' function will take the edited view and try to construct a tree without the deleted element; and hopefully the new source remains similar to the original one, so that unnecessary changes are kept to a minimum. (Note that we have deliberately kept all the functions abstract, because our proposal is not dependent on any particular implementation.)

The method described above takes effort proportional to the size of the data, not to the size of the change. Assuming a functional cons-list, the deletion of 4 involves traversing the view list to the location of the deletion, and changing only the sublist $[4,8,9]$ rooted at that location; a more efficient approach is to update only the source subtree $(\mathit{Fork}\ 8\ (\mathit{Fork}\ 4\ \mathit{Empty}\ \mathit{Empty})\ (\mathit{Fork}\ 9\ \mathit{Empty}\ \mathit{Empty}))$ that is responsible for generating the view fragment $[4,8,9]$. That is to say, the bidirectional updating should be incremental. Better still, in the case of lists, where a deletion is local and does not induce subsequent changes to a substructure, a more refined analysis may discover that only the subtree $(\mathit{Fork}\ 4\ \mathit{Empty}\ \mathit{Empty})$ is really affected by the edit, and updating this is sufficient.

We were able to conclude above that the view sublist $[4,8,9]$ corresponds to the source subtree (*Fork* 8 (*Fork* 4 *Empty Empty*) (*Fork* 9 *Empty Empty*)), because all elements in the former can be found in the latter, and the 'get' function (inorder traversal) happens to have certain properties that allow structure correspondences to be derived from element correspondences. In the sequel, we will discuss in detail how 'get' functions that support incremental updates can be identified; and present a constructive method for performing the said update in a change-based framework. Assuming a fairly balanced source tree, the complexity of our algorithm is $O(m \times \log n + f\ m)$ where n is the size of the source tree, m is the size of the affected source part, and f is the complexity function of a state-based 'put' function. The update itself takes time proportional to m; but it takes $m \times \log n$ time to locate the target source location.

The rest of the paper is structured as follows. Section 2 gives the overall setting of change-based bidirectional frameworks. Section 3 discusses the propagation of locality of view editing to the level of source updating, and Section 4 presents a constructive method for deriving change-based 'put' functions. We then refine the technique for list views (Section 5), and discuss related issues (Section 6), before surveying related work (Section 7) and concluding (Section 8).

2. The Overall Setting

We restrict our attention to polymorphically typed tree-structured data, and polymorphic 'get' functions. To be explicit about our assumptions, we express our requirements in terms of a type class *TypeFunctor*:

```
class TypeFunctor s where
  bimap :: (a → b) → (c → d) → s a c → s b d
  arity  :: s a b → Int
  select :: s a b → Int → b
  lab :: s (Label, a) Labels → Labels
  data Delta s :: ∗ → ∗ → ∗
  close :: Delta s a b → b → s a b
```

The remainder of Section 2 elaborates on these requirements.

2.1 Algebraic Datatypes

We restrict attention to regular datatypes—that is, defined in terms of sums, products, and least fixed points. We assume that the type functor is (as the name suggests) a bifunctor, satisfying the functor laws

```
bimap id id             = id
bimap (f ∘ g) (h ∘ k) = bimap f h ∘ bimap g k
```

The type functor determines the shape of the tree; polymorphic tree terms are then formed by taking the fixpoint (in the second argument) of this bifunctor.

```
data TypeFunctor s ⇒ Term s a = InT { outT :: s a (Term s a) }
```

For example, the binary tree type from Section 1.4 is represented by the following definitions:

```
data TreeF a b = EmptyF | ForkF a b b
type Tree a = Term TreeF a
```

with the obvious definition of *bimap*.

In the interests of brevity, we will usually write binary type constructors s in the type class *TypeFunctor* as single bold capitals **S**; and in an abuse of notation, we will often omit the element type, writing just μ**S** for *Term s a*.

As usual, the type functor induces a fold operator for consuming tree terms:

```
fold :: TypeFunctor s ⇒ (s a b → b) → Term s a → b
fold f t = f (bimap id (fold f) (outT t))
```

The idea is that 'get' functions should be written as polymorphic instances of *fold*. For example, with an appropriate type functor definition for lists:

```
data ListF a b = NilF | ConsF a b
type List a    = Term ListF a
```

we could implement the *inorder* transformation as follows:

```
inorder :: Tree a → List a
inorder t = fold step t (InT NilF) where
  step EmptyF       = id
  step (ForkF a f g) = f ∘ InT ∘ ConsF a ∘ g
```

2.2 Tree Navigation

We use functions

```
arity  :: TypeFunctor s ⇒ s a b → Int
select :: TypeFunctor s ⇒ s a b → Int → b
```

to capture the number of recursive components in an **S**-structure, and to select one of those components. For example, for binary trees we have

```
arity EmptyF       = 0
arity (ForkF _ _ _) = 2
select (ForkF _ t _) 0 = t
select (ForkF _ _ u) 1 = u
```

Note that *select* is a partial function, and *select x i* is defined iff $0 \leqslant i < arity\ x$. As a shorthand, we write x_i for *select x i*, and $x_{i \mapsto t}$ to denote the substitution of t for x_i in x. We define a function *child* to select an immediate subterm of a term:

```
child :: TypeFunctor s ⇒ Term s a → Int → Term s a
child t i = (outT t)_i
```

and a function *zoom* to select an arbitrarily deep subterm, following a path represented as a list of positions:

```
type Path = [Int]
zoom :: TypeFunctor s ⇒ Term s a → Path → Term s a
zoom = foldl child
```

2.3 Labelling

We assume that all tree elements are associated with unique labels; for this purpose we assume an abstract type *Label*, and a corresponding type *Labels* of sets of labels. (For example, the unique label associated with an element at a particular node might be formed out of the path from the root of the tree to that node, together with a disambiguating index in case there are multiple elements at the same node.)

```
type Label  = ...
type Labels = Set Label
```

Generally speaking, the labels should be thought of as being behind the scenes, inaccessible to normal functions; that is why we insist that 'get' functions should be polymorphic. In particular, we assume that the 'get' function cannot invent labels: all the labels that turn up in a view subterm originated from the corresponding source term. But for the purposes of discussing labels and labelsets themselves, we require the type functor to provide a mechanism for combining the labelsets from subterms to make the labelset for a term itself:

```
lab :: TypeFunctor s ⇒ s (Label, a) Labels → Labels
```

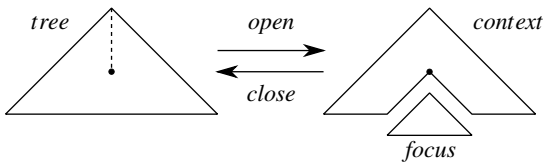

Figure 1. The context-focus representation.

For example, for binary trees we have

$$\begin{aligned} &\mathit{lab}\ \mathit{EmptyF} &&= \emptyset \\ &\mathit{lab}\ (\mathit{ForkF}\ (l,a)\ x\ y) &&= \mathit{singleton}\ l \cup x \cup y \end{aligned}$$

This forms an **S**-algebra, which can be exploited by a partly polymorphic fold to compute the labelset of a labelled term:

$$\begin{aligned} &\textbf{type}\ \mathit{LTerm}\ s\ a = \mathit{Term}\ s\ (\mathit{Label}, a) \\ &\mathit{labels} :: \mathit{TypeFunctor}\ s \Rightarrow \mathit{LTerm}\ s\ a \to \mathit{Labels} \\ &\mathit{labels} = \mathit{fold}\ \mathit{lab} \end{aligned}$$

We will often use the shorthand $\langle t \rangle$ to denote *labels t*; we extend this notation to **S**-structures of $\mu\mathbf{S}'$ trees $x :: \mathbf{S}\ \mu\mathbf{S}'$ by writing '$\langle x \rangle$' for '$\mathit{lab}_{\mathbf{S}}\ (\mathbf{S}\ \mathit{labels}_{\mu\mathbf{S}'}\ x)$', and, as a shorthand, write '$\langle x_{\neq i} \rangle$' for '$\mathit{lab}_{\mathbf{S}}\ (\mathbf{S}\ \mathit{labels}_{\mu\mathbf{S}'}\ x)_{i \mapsto \emptyset}$').

Splitting an **S**-structure into one component and the remainder yields an exhaustive and (at least for **S**-structures arising directly from a $\mu\mathbf{S}$-structure) disjoint partition of the label sets: for $x :: \mathbf{S}\ \mu\mathbf{S}'$ and for any valid i,

$$\langle x_{\neq i} \rangle \cup \langle x_i \rangle = \langle x \rangle$$

and for $s :: \mu\mathbf{S}$ and any valid i,

$$\langle (\mathit{outT}\ s)_{\neq i} \rangle \,\#\, \langle (\mathit{outT}\ s)_i \rangle$$

where # denotes disjointness, that is, $x \# y = (x \cap y = \emptyset)$. Note that $(x \subseteq y) \wedge (y \# z) \Rightarrow (x \# z)$ by monotonicity of intersection, so we allow ourselves to write derivations of the form '$w \subseteq x \# y \supseteq z$', with a chain of inclusions, a disjointness, and a chain of containments, and conclude from it that $w \# z$.

2.4 Tree Contexts

As implemented by function *zoom*, a subtree of interest can be accessed by navigating from the root of a tree and following a given path. Such a path need not represent a traversal from the root all the way to a leaf: paths may point to internal nodes in a tree. The traversal of a tree following a path 'opens' that tree into two disjoint structures: one represents the subtree below the node reached by the path, known as the current *focus*; the other represents the rest of the tree and is known as the *context* of the focus (see Figure 1). On typing grounds, we only consider as valid those paths that lead to recursive components; for example, the focus of the binary tree datatype above can descend to the left or the right subtree, but not to the element stored in a node.

We represent a context as a tree with a hole in it, denoting the location of the subtree given by the focus. This can be captured in terms of a type function *Delta*, such that *Delta s a b* is the type of one-hole contexts for *s a b*. That is, a value of type *Delta s a b* is equivalent to one of type *s a b* that is missing one value of type *b*; so the type functor should be equipped with the corresponding operation to 'close' a *Delta s a b* around the missing *b* to make an *s a b*.

$$\begin{aligned} &\textbf{data}\ \mathit{Delta}\ s :: * \to * \to * \to * \\ &\mathit{close} :: \mathit{TypeFunctor}\ s \Rightarrow \mathit{Delta}\ s\ a\ b \to b \to s\ a\ b \end{aligned}$$

Then a one-hole context for a tree consists of a list of one-hole contexts for nodes; we have chosen to represent that list outermost context first, so that 'closing' is a *foldr*.

$$\begin{aligned} &\textbf{type}\ \mathit{Context}\ s\ a = [\mathit{Delta}\ s\ a\ (\mathit{Term}\ s\ a)] \\ &(\lessdot) :: \mathit{TypeFunctor}\ s \Rightarrow \mathit{Context}\ s\ a \to \mathit{Term}\ s\ a \to \mathit{Term}\ s\ a \\ &cs \lessdot t = \mathit{foldr}\ (\lambda d\ u \to \mathit{InT}\ (\mathit{close}\ d\ u))\ t\ cs \end{aligned}$$

For example, for binary trees, there are two ways of making a *TreeF a b* that is missing a single *b*, one for each side of a *Fork* node—but there is no way for an *Empty* node to have a hole:

$$\begin{aligned} &\textbf{data}\ \mathit{Delta}\ \mathit{TreeF}\ a\ b = \mathit{LForkF}\ a\ b \mid \mathit{RForkF}\ a\ b \\ &\mathit{close}\ (\mathit{LForkF}\ a\ u)\ t = \mathit{ForkF}\ a\ t\ u \\ &\mathit{close}\ (\mathit{RForkF}\ a\ t)\ u = \mathit{ForkF}\ a\ t\ u \end{aligned}$$

In fact, the context type *Delta s* follows directly from the structure *s* of the tree; contexts are a type-indexed data type [17], and can be defined generically [16, 22].

2.5 Subterms

We prohibit trees with 'junk' structures that are not labelled, because such structures break the one-to-one correspondence between subtrees of a given tree and their labelsets, a property required for tracing structure transformations using labels.

REQUIREMENT 1 (No Junk). *Given a tree t, $\langle \mathit{child}\ t\ i \rangle \subset \langle t \rangle$, for any valid index i.* □

The no-junk condition enforces that the labelset of any subtree tree must be a proper subset of that of its parent. For many datatypes, similar to the representation of binary trees above, this no-junk condition is enforced by the constructors. But for other datatypes such as leaf-labelled trees

$$\begin{aligned} &\textbf{data}\ \mathit{LTree}\ a = \mathit{EmptyL} \\ &\qquad\qquad \mid\ \mathit{Leaf}\ a \\ &\qquad\qquad \mid\ \mathit{Bin}\ (\mathit{LTree}\ a)\ (\mathit{LTree}\ a) \end{aligned}$$

one can construct invalid values such as $\mathit{Bin}\ \mathit{EmptyL}\ (\mathit{Leaf}\ 1)$, where *Leaf* 1 is a strict subterm of $\mathit{Bin}\ \mathit{EmptyL}\ (\mathit{Leaf}\ 1)$, but has the same labelset. In this paper, we rule out datatypes like this that do not enforce the no-junk requirement.

Tree navigation induces a subterm ordering on trees.

DEFINITION 2 (Subterm Ordering). *Given trees r and t, we say r is a subterm of t, written $r \preccurlyeq t$, if $r = \mathit{zoom}\ t\ p$ for some p. We say that a subterm r is trivial if $\langle r \rangle = \emptyset$.* □

Throughout this paper, we assume non-trivial subterms unless otherwise stated.

COROLLARY 3 (Distinct Subterms). *Because tree elements are uniquely labelled, when $r \preccurlyeq t$ (and r is non-trivial), then r is in fact at the end of a unique path; that is, there is a (partial) function $\mathit{locate} :: \mu\mathbf{S} \to \mu\mathbf{S} \to \mathit{Path}$ satisfying*

$$\mathit{locate}\ t\ r = p \quad \Leftrightarrow \quad r = \mathit{zoom}\ t\ p$$

We say that 'r is at depth n in t' when $n = \mathit{length}\ (\mathit{locate}\ t\ r)$. □

DEFINITION 4 (Orderedness). *We say r and t are ordered, written as $r \sim t$, if $r \preccurlyeq t \vee t \preccurlyeq r$.* □

A consequence of the no-junk requirement is a close correlation between the subterm relation and inclusion among labelsets.

COROLLARY 5 (Labels of Subterms). *The subterm relationship is a refinement of inclusion of labelsets: $r \preccurlyeq t \Rightarrow \langle r \rangle \subseteq \langle t \rangle$. And for ordered trees $r \sim t$, the converse holds too: $r \preccurlyeq t \Leftarrow \langle r \rangle \subseteq \langle t \rangle$.* □

Another consequence is that the labelsets of two ordered trees intersect.

COROLLARY 6. *We have* $r \sim t \Rightarrow (\langle r \rangle \cap \langle t \rangle \neq \emptyset)$ *for non-trivial trees r,t.* □

The operator $(\lessdot)$ that closes a context around a tree has a partial inverse $(/)$, in the sense that

$$c \lessdot r = t \quad \Leftrightarrow \quad t / r = c$$

when $r \preccurlyeq t$; in particular, $(t / r) \lessdot r = t$ and $(c \lessdot r) / r = c$. One might think of t / r as the result of subtracting subtree r from tree t. The two operators $(\lessdot)$ and $(/)$ associate to the right, and $(/)$ has a higher precedence than $(\lessdot)$; we can therefore nest them:

PROPOSITION 7 (Nesting). *Given that* $r \preccurlyeq s \preccurlyeq t$, *then* $t / s \lessdot s / r \lessdot q = t / r \lessdot q$. □

Subtraction is related to substition:

PROPOSITION 8. *For trees r and t and position i,* $InT\ (outT\ r)_{i \mapsto t} = r / (outT\ r)_i \lessdot t$. □

Closing extends the input tree, as captured by the following monotonicity condition.

PROPOSITION 9 (Monotonicity). $t \preccurlyeq (c \lessdot t)$ □

Lastly, all holes are equal.

REQUIREMENT 10 (Hole Equality). $s / s = t / t$. □

COROLLARY 11 (Left Unit). $s / s \lessdot t = t$. □

2.6 Local Editing

An *editing function* is an endofunction $edit :: v \to v$; we will consider editing functions only on views. We require all editing functions to be total, so that they can always be applied to a subterm of a view. We treat editing functions as being in some sense location-independent: they can be applied to any superterm enclosing the subterm that is actually affected by the edit.

DEFINITION 12 (Locality). *We say an editing function e is local to a subterm* u_0 *of a view v if*

$$\forall u.\ u_0 \preccurlyeq u \preccurlyeq v \Rightarrow e\ v = v / u \lessdot e\ u$$

□

In the above definition, applying the local editing function e to any subterm u of v enclosing the affected subterm u_0 has the same effect as applying the function to v. For example, as we have seen in the binary tree example, deleting 4 from the sublist $[4,8,9]$ and combining the result with the context $[7,6,5]$ is the same as deleting 4 from the complete view $[7,6,5,4,8,9]$. From the equality of holes, we can conclude that the trivial locality always holds.

COROLLARY 13. *Given a view v, any editing function is local to subterm v of v.* □

Beyond the trivial one, there is certainly an ordering among different levels of locality, based on the subterm ordering, which falls out from the above definition. In this sense, a context-sensitive (path-based) editing function, always requiring traversing from the root, fixes u_0 to be v, which implies very poor locality characteristics. We will discuss an option for remedying this in Section 6.1.

In our proposal, the subterm u_0 to which an editing function is local is user-provided; our approach is based on the assumption that u_0 is significantly smaller than v. We pair the editing function with an additional function that returns the affected subterm.

data $Edit\ a = E\ \{edit :: a \to a, affect :: a \to a\}$

(The above declaration creates a polymorphic record type *Edit* with named fields *edit* and *affect*, each of which is a function. The field extractors are named after the fields; so given a value $e :: Edit\ a$, the two functions encapsulated in it can be retrieved as $edit\ e :: a \to a$ and $affect\ e :: a \to a$. We require that *edit e* is local to *affect e*.)

2.7 Change-Based Bidirectional Frameworks

A change-based bidirectional framework consists of two functions: a 'get' function $f :: s \to v$ from source to view, and a change-based 'put' function $f^{<}_{ch} :: Edit\ v \to s \to s$. We only consider 'get' functions that are regular structural recursions, because they are more likely to benefit from our proposed improvement. We will discuss this choice in detail in Section 3.2. We also rule out 'get' functions involving duplication of labels, so that uniqueness of identifiers is preserved. The function $f^{<}_{ch}$ is higher-order, in contrast to $f^{<}_{st} :: (v,s) \to s$ in a state-based setting. Thus $f^{<}_{ch}$ no longer constructs an updated source from an edited view, but from the original source; any information in the edited view can be derived from the editing function and the original source. In contrast to an operation-based approach, $f^{<}_{ch}$ is not dependent on the actual editing functions.

Bidirectional laws semantically equivalent to those developed for state-based bidirectional frameworks can be specified in the new setting.

Consistency $f\ (f^{<}_{ch}\ e\ s) = edit\ e\ (f\ s)$

Acceptability $f^{<}_{ch}\ (E\ \{edit = id\}) = id$

Undoability $f^{<}_{ch}\ (e\ \{edit = (edit\ e)^{\circ}\}) \circ f^{<}_{ch}\ e = id$.

The relationships between different view values are expressed through explicit editing functions. For acceptability, we construct a record with the identity edit (and leave the *affect* field unspecified). For undoability, a record is updated with the left-inverse $((edit\ e)^{\circ})$ of its editing function to cancel its effect on the source.

Moving from a state-based framework to a change-based framework potentially improves run-time performance, as we exploit the locality of updating. We look into the details in the next section.

3. Locality Preservation

Incremental updates can be achieved if the locality of an editing function is propagated to the source level. Figure 2 shows how a 'get' function may relate subterms in the source to subterms in the view. The idea is that the subterm v of the view depends only on the subterm s of the source. Furthermore, the sequences of source contexts $sc_1, ..., sc_n$ and view contexts $vc_1, ..., vc_m$ maintain this relationship, so that $vc_1 \lessdot v$ depends only on $sc_1 \lessdot s$, and so on, until $vc_m \lessdot ... \lessdot vc_1 \lessdot v$ depends only on $sc_n \lessdot ... \lessdot sc_1 \lessdot s$. Note that we can always arrange the two columns in a way that both have the same length (i.e., $m = n$); when one subterm on one side matches with multiple ones on the other side, we only need to insert a few empty contexts (since $[] \lessdot t = t$) to realign the two sides. This kind of locality preservation is determined by the 'get' function, which defines the connection between a view and its source.

3.1 Alignment

DEFINITION 14 (Alignment). *We say 'get' function f aligns at subterm r of s if for all t we have*

$$f\ (s / r \lessdot t) = f\ s / f\ r \lessdot f\ t$$

We call r an alignment position in s with respect to f. □

When a 'get' function f and a source s are unambiguous, the term r may be referred to as an alignment position, where f is said to align at r. The above definition not only characterizes the matching of source subterms r to corresponding view subterms $f\ r$, but also a kind of isolation between them. An alignment position can be seen as a 'resistive barrier' between the construction of a subterm and its

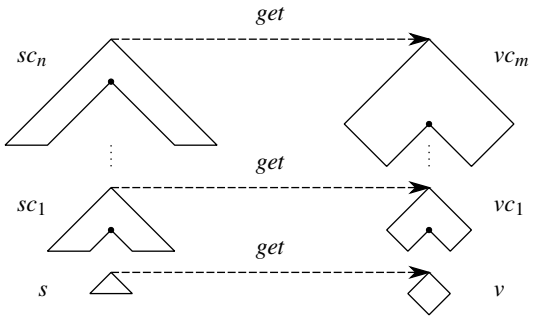

Figure 2. Source-view alignment.

context, through which information does not flow. At an alignment position, $f\ r$ is independent of $s \,/\, r$ and $f\ s \,/\, f\ r$ is independent of r.

The significance of alignment positions is that they capture the mapping between the locality to $f\ r$ in the view and the locality to r in the source. As a result, if $f\ r$ can be locally edited, then r can be locally updated:

$$f^{\leq}_{st'}\ (f\ s \,/\, f\ r \lessdot v', s) = s \,/\, r \lessdot f^{\leq}_{st}\ (v', r)$$

The above defines an optimization of an existing 'put' function. Basically, to process an edited view $f\ s \,/\, f\ r \lessdot v'$, we only need to process v' (the edited $f\ r$), provided f aligns at r. To show that the above transformation is correct, we prove the consistency of $f^{\leq}_{st'}$.

$$\begin{array}{ll} & f\ (f^{\leq}_{st'}\ (f\ s \,/\, f\ r \lessdot v', s)) \\ = & \quad \{\text{ definition of } f^{\leq}_{st'} \} \\ & f\ (s \,/\, r \lessdot f^{\leq}_{st}\ (v', r)) \\ = & \quad \{f \text{ aligns at } r\} \\ & f\ s \,/\, f\ r \lessdot f\ (f^{\leq}_{st}\ (v', r)) \\ = & \quad \{\text{ consistency of } f^{\leq}_{st} \} \\ & f\ s \,/\, f\ r \lessdot v' \end{array}$$

Other bidirectional properties hold as well; we postpone their proofs until Section 4, where the complete solution is presented.

Not all view subterms match exactly with a source subterm; sometimes we need to resort to a looser fit.

DEFINITION 15. *Given a 'get' function f, we say an alignment position s covers v if $v \preccurlyeq f\ s$.*

We now show some example 'get' functions that preserve different degrees of alignment. Consider a function that returns the mirror image of a tree.

$$\begin{array}{l} mirror :: Tree\ a \to Tree\ a \\ mirror\ Empty \quad\ \ = Empty \\ mirror\ (Fork\ a\ l\ r) = Fork\ a\ (mirror\ r)\ (mirror\ l) \end{array}$$

Every subtree in the source is an alignment position, because the constructions of the view and of the source coincide.

Another such function is *inorder*, defined in Section 1.4. In this case, only the right subtrees are alignment positions, because the left subtrees do not correspond to subterms in the view.

Yet another example is the function *spine* that extracts the elements on the spine of a tree:

$$\begin{array}{l} spine :: Tree\ a \to [a] \\ spine\ Empty \quad\ \ = [\,] \\ spine\ (Fork\ a\ l\ r) = a : spine\ r \end{array}$$

In this case, all subterms of the source are alignment positions, though the left subtrees, which always correspond to the empty list in the view, are not very interesting.

3.2 Exploiting Regularity

Change-based 'put' functions are only interesting when there are plenty of alignment positions to choose from, so that view subterms can be covered 'tightly'. As alignment positions represent matches between source and view constructions, for a recursive 'get' function, this matching of constructions suggests a kind of structural recursion pattern. Though not a sufficient condition, regularity of the recursion pattern is likely to positively impact the availability of alignment positions. Thus, we focus on regular structural recursions—functions that can be implemented as folds. For a regular structural recursion, a source is deconstructed in a uniform way, which leaves the fold body to determine whether the construction of a view matches up.

To explain the intuition behind how 'get' functions determine alignment positions, let us revisit the function *spine*. A fold deconstructs a non-empty input tree into two source subterms l and r and a single element a; for the *spine* computation, the fold body discards l and adds the element a to *spine r*. The recursive calls always produce a view recursive component (such as *spine r*) from a source subterm (r); whether the view recursive component so produced is made into a view subterm by the fold body determines the possibility of alignment. For example, recursive component *spine r* is a subterm of $a : spine\ r$, which makes r an alignment position. In this case, any edit local to *spine r* can be addressed by updating r. In contrast, if we define *spine* as

$$spineRev\ (Fork\ a\ l\ r) = spineRev\ r +\!\!+ [a]$$

then the view construction is the 'opposite' of the source construction, with the parent a at the bottom (tail end) of the list. This misalignment manifests itself by causing *spineRev r* not to form a subterm in the view. Any edit to the view affects a sublist including a, which implies an update to the complete source tree.

In the case of *spine*, a view subterm not only has a recursive component as its origin, but also as its exclusive origin: *spine r* is copied to the view without modification. This exclusivity is necessary for producing alignment positions. Consider a variant of *spine* that breaks this rule:

$$spineRot\ (Fork\ a\ l\ r) = a : reverse\ (spineRot\ r)$$

The recursive component *spineRot r* is changed by *reverse*; and its manifestation in the view depends on its context, which decides how many times *reverse* is applied to it. Though at each individual recursive step *reverse* (*spineRot r*) is a subterm of $a :$ *reverse* (*spineRot r*), a subsequent step does not preserve this property: *reverse* (*spineRot r*) ceases to be a subterm of $b : (reverse\ (a :$ *reverse* (*spineRot r*))). In this case, only three alignment positions (the complete source and the root's two immediate children) exist, which is not very interesting.

3.2.1 The Well-Aligning Condition

We formalize the above observation into a condition on the bodies of 'get' functions defined as folds that guarantees the availability of alignment positions.

DEFINITION 16 (Well-aligning). *We say a fold body b is well-aligning if for all x such that arity $x \neq 0$, and for all non-trivial subterms u of $b\ x$, we have*

$$\exists i.\ u \sim x_i \wedge \forall w.\ b\ x_{i \mapsto w} = b\ x \,/\, x_i \lessdot w$$

Further, we say that $f = fold\ b$ is well-aligning if its body b is. □

We do not worry about the case when there are no recursive components in x (i.e., *arity* $x = 0$), as they are terminals in construction, and will not affect alignment. There are two parts to the condition: the first part ($u \sim x_i$) enforces that each non-trivial view subterm u has a recursive component as its origin; the second part

$(\forall w.\ b\ x_{i\mapsto w} = b\ /\ x_i \lessdot w)$ guarantees the exclusivity of the origins (with no influence by external factors)—the recursive component in question is copied unchanged to the view. It is important for the expressiveness of 'get' functions that this copying requirement only applies to selected recursive components; some, that will appear as subterms of $b\ x$, are taken as opaque blocks, leaving the rest to be broken up for gluing the blocks together.

For example, consider the following functions:

$$\begin{aligned} vlr\ (v,ls,rs) &= [v] +\!\!+ ls\ +\!\!+ rs \\ lvr\ (v,ls,rs) &= ls\ +\!\!+ [v] +\!\!+ rs \\ lrv\ (v,ls,rs) &= ls\ +\!\!+ rs\ +\!\!+ [v] \end{aligned}$$

The functions *vlr*, *lvr*, and *lrv* (standing for 'visit, left, right', etc) correspond to individual cases of the fold bodies for traversing binary trees in pre-, in-, and post-order, respectively. There are two inputs to the functions that are recursive view components, namely *ls* and *rs*. Functions *vlr* and *lvr* are well-aligning, as *rs* is ordered with respect to all the view subterms that are visited, whereas *lrv* is not.

Generalizing the definition to the semantics of transformations, we say that a 'get' function is well-aligning if all cases of its body are well-aligning. By that definition, *preorder*, *inorder*, *unzip*, *mirror*, *spine*, *filter*, *map* are examples of well-aligning 'get' functions, while *postorder* is not. (However, we will discuss how this function can be made well-aligning in Section 5.)

3.2.2 Availability of Alignment Positions

The well-aligning property guarantees the availability of alignment positions; and we can state a declarative result about how they may be found.

THEOREM 17. *Given a well-aligning 'get' function f such that $f\ s = v$, we have that f aligns at subterm r of s if there exists a non-trivial subterm u of v such that $u \preccurlyeq f\ r$.* □

The well-aligning condition tells us clearly that some selected recursive components become subterms in the view; and the source subterms producing the selected subcomponents are alignment positions. The key to proving Theorem 17 is to establish the fact that recursive component $f\ r$ is among those selected due to the premise $u \preccurlyeq f\ r$; this can be achieved by connecting the unique labels in u with those in $f\ r$.

As preparation for formally proving Theorem 17, we state some properties regarding labels of trees under transformation. As mentioned at the beginning of Section 2, one important requirement of 'get' functions is that they do not invent labels.

REQUIREMENT 18 (Conservation of Labels). *Given a 'get' function $f = fold\ b$, we have that b does not invent labels:*

$$\forall x.\ \langle b\ x\rangle \subseteq lab\ x$$

and hence, neither does $f = fold\ b$ invent labels:

$$\forall s.\ \langle f\ s\rangle \subseteq \langle s\rangle$$

□

An important consequence of the fact that 'get' functions do not invent labels is that labels cannot reappear after they have been dropped during the construction of a view. Conversely, if a label set $\langle v\rangle$ has been generated after processing subterm r of t by the 'get' function f (that is, $r \preccurlyeq t$ and $\langle v\rangle \subseteq \langle f\ r\rangle$), and $\langle v\rangle$ is still present after processing t itself (that is, $\langle v\rangle \subseteq \langle f\ t\rangle$), then $\langle v\rangle$ is present at every intermediate stage too ($\langle v\rangle \subseteq \langle f\ s\rangle$ for every s such that $r \preccurlyeq s \preccurlyeq t$). This is a kind of 'convexity' property of label sets.

More importantly in what follows, a similar result holds for subterms, rather than their projections to label sets; but for this, we need the additional assumption that the 'get' function f is well-aligning. The primary result (Corollary 20) is a convexity property for terms: given sources r,t with $r \preccurlyeq t$ such that $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$, then also $v \preccurlyeq f\ s$ for any s such that $r \preccurlyeq s \preccurlyeq t$. (In fact, $v \preccurlyeq f\ t$ is not strictly required; $\langle v\rangle \subseteq \langle f\ t\rangle$ suffices.) The essential step (Lemma 19) is the one from the outermost term t to one of its immediate children: if view subterm v shows up after processing a subterm r within the ith child $(outT\ t)_i$ of t, and v is still present after processing t, then v must have come from the ith child: $v \preccurlyeq f\ (outT\ t)_i$. Note that, for both of these results, we make use of our implicit assumption that v is non-trivial.

LEMMA 19 (Maintaining terms). *Suppose a well-aligning 'get' function $f = fold\ b$. For source terms r,t with $r \preccurlyeq (outT\ t)_i$, if $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$, then also $v \preccurlyeq f\ (outT\ t)_i$.*

PROOF. Let $x = \mathbf{S}f\ (outT\ t)$, so that $f\ t = b\ x$ and $f\ (outT\ t)_i = x_i$. Since b is well-aligning, and v is a non-trivial subterm of $b\ x$, there exists a j such that $v \sim x_j = f\ (outT\ t)_j$. In fact, this j must be i:

$$\begin{array}{ll} & \langle v\rangle \\ \subseteq & \quad \{\text{ labels of subterms (Corollary 5); } v \preccurlyeq f\ r\text{, by assumption}\} \\ & \langle f\ r\rangle \\ \subseteq & \quad \{f \text{ does not invent labels}\} \\ & \langle r\rangle \\ \subseteq & \quad \{\ r \preccurlyeq (outT\ t)_i\text{; Corollary 5 again}\} \\ & \langle (outT\ t)_i\rangle \\ \# & \quad \{\text{ disjointness of labels}\} \\ & \langle (outT\ t)_{\neq i}\rangle \\ \supseteq & \quad \{f \text{ does not invent labels; monotonicity of intersection}\} \\ & \langle (\mathbf{S}f\ (outT\ t))_{\neq i}\rangle \\ = & \quad \{\text{ definition of } x\} \\ & \langle x_{\neq i}\rangle \end{array}$$

and so $\langle v\rangle \,\#\, \langle x_{\neq i}\rangle$, and hence $\langle v\rangle \subseteq \langle x_i\rangle$ by disjointness of labels. Finally, $v \sim x_i$ and $\langle v\rangle \subseteq \langle x_i\rangle$ imply $v \preccurlyeq x_i$, by Corollary 5. □

COROLLARY 20 (Term convexity). *Suppose a well-aligning 'get' function $f = fold\ b$. For source terms r,t with $r \preccurlyeq t$, if $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$, then also $v \preccurlyeq f\ s$ for every s such that $r \preccurlyeq s \preccurlyeq t$.*

PROOF. The proof is by induction over the length of *locate t r*. The base case is when the path is empty, so $r = t$; then the lemma is trivially true. For the inductive case, assume the statement is valid for paths of length n. Suppose that $r \preccurlyeq t$, and that r is at depth $n+1$ in t (so that $r \preccurlyeq (outT\ t)_i$ for some unique index i, and *locate* $(outT\ t)_i\ r$ has length n), and that $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$. By Lemma 19, we get $v \preccurlyeq f\ (outT\ t)_i$; then by induction we get $v \preccurlyeq f\ s$ for every s with $r \preccurlyeq s \preccurlyeq (outT\ t)_i$ too; and the final case $s = t$ trivially holds. □

Theorem 17 follows directly from the following result: given a well-aligning 'get' function f, and sources r,s such that $r \preccurlyeq s$, if there exists any view subterm v such that $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ s$, then f aligns at subterm r of s. (Again, we assume that v is non-trivial.)

LEMMA 21 (Get alignment). *Given a well-aligning 'get' function f, sources r,t with $r \preccurlyeq t$, and view v such that $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$, then f aligns at subterm r of t.*

PROOF. Again, by induction over the length of *locate t r*. The base case is when the path is empty; then $t = r$ and the theorem is trivially true (since f necessarily aligns at the root t of source t). For the inductive case, we assume that the statement is valid for paths of length n; we are given terms r,t with $r \preccurlyeq t$ and r at depth $n+1$ in t, and a term v with $v \preccurlyeq f\ r$ and $v \preccurlyeq f\ t$, and we have to show that f aligns at subterm r of t.

Suppose that r is within the i'th child of t, that is, $r \preccurlyeq s$ where $s = (outT\ t)_i$. Then by Corollary 20, we have $v \preccurlyeq f\ s$, and by induction, f aligns at subterm r of s. Let $x = \mathbf{S} f\ (outT\ t)$, so that $f\ t = b\ x$ and $f\ (outT\ t)_i = x_i$. Because b is well-aligning and $v \preccurlyeq b\ x$, there exists a j such that $v \sim x_j$ and $b\ x_{j \mapsto w} = b\ x \,/\, x_j \lessdot w$ for any w. In fact, that j must be i, by the same argument as in the proof of Lemma 19. In particular, $f\ s = x_i \preccurlyeq b\ x$, a fact that we shall use below. Finally, we show that f aligns at subterm r of t. For an arbitrary source term p, we have:

$$
\begin{array}{ll}
 & f\ (t \,/\, r \lessdot p) \\
= & \quad \{\text{ since } r \preccurlyeq s \preccurlyeq t\} \\
 & f\ (t \,/\, s \lessdot (s \,/\, r \lessdot p)) \\
= & \quad \{\text{ Proposition 8—} s = (outT\ t)_i\} \\
 & f\ (InT\ (outT\ t)_{i \mapsto (s/r \lessdot p)}) \\
= & \quad \{\text{ evaluation rule for } f = fold\ b\} \\
 & b\ (\mathbf{S} f\ (outT\ t)_{i \mapsto (s/r \lessdot p)}) \\
= & \quad \{\text{ naturality of } select\} \\
 & b\ (\mathbf{S} f\ (outT\ t))_{i \mapsto (f\ (s/r \lessdot p))} \\
= & \quad \{ b \text{ is well-aligning; discussion above}\} \\
 & b\ (\mathbf{S} f\ (outT\ t))\ /\ f\ (outT\ t)_i \lessdot f\ (s \,/\, r \lessdot p) \\
= & \quad \{\text{ evaluation rule for } f \text{ again; } s = (outT\ t)_i\} \\
 & f\ t \,/\, f\ s \lessdot f\ (s \,/\, r \lessdot p) \\
= & \quad \{\text{ induction}\} \\
 & f\ t \,/\, f\ s \lessdot (f\ s \,/\, f\ r \lessdot f\ p) \\
= & \quad \{\text{ nesting}\} \\
 & f\ t \,/\, f\ r \lessdot f\ p
\end{array}
$$

□

So far, we have established well-alignment as a sufficient condition for the availability of alignment positions (Definition 16), and proved a declarative result about how alignment positions can be found (Theorem 17). Next, we move on to devise a constructive method of finding alignment positions, and deriving a change-based 'put' function based on this method.

4. Change-Based 'Put' Functions

The derivation of a 'put' function is divided into three steps: (i) finding an alignment position covering the edited view subterm; (ii) using a state-based 'put' function to map the view subterm to a source subterm; and (iii) merging the original source context with the updated source subterm. The key part of this process is step (i); the other two follow on quite naturally. Taking the previous result, we know that a source subterm is an alignment position if there is a corresponding subterm in the view. A standard way of establishing the source/view correspondence semantically is to trace the uniquely identifying labels.

4.1 Labelling and Reflecting

Unique identifiers are created using paths in the source, from the root to the node constructors of the elements. As a result, a node is at the root position of the subterm identified by the path. As far as 'get' functions are concerned, an element and its label form an atomic unit; an element in the view originates from an element in the source associated with the same label. (It is worth noting that the labels only represent paths in the source, not those in the view.) Given an edit-affected view subterm v, a sensible alignment position should include all the labels in $\langle v \rangle$; the path leading to such a source subterm is the maximum common prefix $mcp\ \langle v \rangle$ of all the paths to nodes with these labels.

Consider a simple example with *mirror* as the 'get' function (see Figure 3). Note that the labels in the source—the list component of each pair in the diagram—are copied over to the view. Suppose we insert a new node at the location labelled $[1]$ in the view, affecting the element 68 at location $[1]$ and the locations below it $[[1,1],[1,2]]$. (Note that we don't require a concrete label for the newly inserted node since it will not contribute to the identification of the affected source.) The maximum common prefix of the affected labels $[[1],[1,1],[1,2]]$ is $[1]$. Now tracing the pedigree of the path $[1]$ back to the source, we conclude that it is the subtree with root element 68 that needs to be changed.

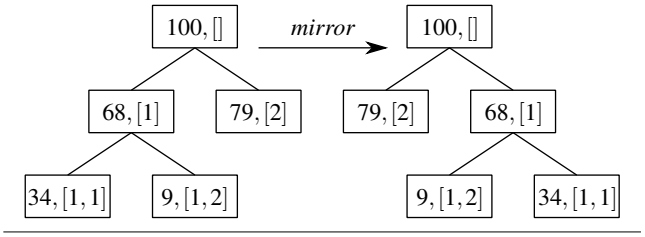

Figure 3. Using *mirror* as a 'get' function.

The key to demonstrating the correctness of the above process is to show that the subset relation between label sets corresponds to the subterm relation between trees.

LEMMA 22. *Given a well-aligning 'get' function $f = fold\ b$, and source terms s, t with $s \preccurlyeq t$, and view term v, if $v \preccurlyeq f\ t$ and $\langle v \rangle \subseteq \langle s \rangle$ then $v \preccurlyeq f\ s$.*

PROOF. Again, by induction over the length of *locate t s*. The base case is when the path is empty; then $s = t$, and the result trivially holds. For the inductive case, assume that the result holds for paths of length n, and that s is at depth $n+1$ in t. Let i be such that $s \preccurlyeq (outT\ t)_i$, so that $locate\ (outT\ t)_i\ s$ has length n. We will show that $v \preccurlyeq f\ (outT\ t)_i$; then we can conclude $v \preccurlyeq f\ s$ by appeal to the inductive hypothesis.

Let $x = \mathbf{S} f\ (outT\ t)$, so $f\ t = b\ x$ and $f\ (outT\ t)_i = x_i$. Since v is a non-trivial subterm of $b\ x$, and b is well-aligning, there exists a j such that $v \sim x_j$. By the usual argument, that j must be i:

$$
\begin{array}{ll}
 & \langle v \rangle \\
\subseteq & \quad \{\text{ assumption}\} \\
 & \langle s \rangle \\
\subseteq & \quad \{\text{ hypothesis, and Corollary 5}\} \\
 & \langle (outT\ t)_i \rangle \\
\# & \quad \{\text{ disjointness of labels}\} \\
 & \langle (outT\ t)_{\neq i} \rangle \\
\supseteq & \quad \{ f \text{ does not invent labels}\} \\
 & \langle (\mathbf{S} f\ (outT\ t))_{\neq i} \rangle \\
= & \quad \{\text{ definition}\} \\
 & \langle x_{\neq i} \rangle
\end{array}
$$

So $v \sim x_i$. Moreover, $\langle v \rangle \subseteq \langle x_i \rangle$, by disjointness of labels, since by assumption we have $v \preccurlyeq f\ t$ and hence $\langle v \rangle \subseteq \langle f\ t \rangle = \langle b\ x \rangle \subseteq \langle x \rangle$, and we have just shown that $\langle v \rangle \,\#\, \langle x_{\neq i} \rangle$. Therefore, by Corollary 5 we conclude $v \preccurlyeq x_i = f\ (outT\ t)_i$. □

As a side remark, so far we have been oblivious to the fact that the source and view nodes are now labelled, and have assumed that the 'get' and 'put' functions work uniformly on them. This is certainly correct given parametrically polymorphic datatypes and functions; free theorems [30] provide the guarantee we need. It is straightforward to relax this fully-parametric type restriction to constrained polymorphic types. For example, we can use equality in the transformation by introducing the following:

$$
\begin{array}{l}
f :: Eq\ a \Rightarrow s\ a \to v\ a \\
f^{<}_{st} :: Eq\ a \Rightarrow (v\ a, s\ a) \to s\ a
\end{array}
$$

and a generic instance to bypass the labels:

instance $Eq\ a \Rightarrow Eq\ (Label, a)$ **where**
$(\equiv)\ (\_, a)\ (\_, b) = a \equiv b$

This kind of generic definition is all that is required to introduce constrained polymorphism. For example, consider a function that filters the labels of a tree and returns them as a list.

$filterT :: Eq\ a \Rightarrow (a, Tree\ a) \to [a]$
$filterT\ (x, Empty) \quad = [\,]$
$filterT\ (x, Fork\ a\ l\ r) = \textbf{if}\ a \equiv x\ \textbf{then}\ lr\ \textbf{else}\ a : lr$
where $lr = filterT\ l +\!\!+ filterT\ r$

Function *filterT* is well-aligning, and our technique is applicable without modification.

It is worth mentioning that labelling tree elements uniquely and exploiting parametricity to establish connections between source and view is not limited to finding subterm correspondence. In [28], 'get' functions are applied to source values with elements replaced by labels, which allows one to conduct a kind of forensic examination of the transformation, determining its effect without examining its implementation; with such information a 'put' function can be constructed. We will discuss in more detail in Section 7 the connection between our technique and the approach in [28].

4.2 The Change-Based 'Put' Function

We are now ready to present the change-based 'put' function. For any $f_{st}^{<}$, a generic $f_{ch}^{<}$ function can be defined as follows:

$f_{ch}^{<} :: Edit\ v \to s \to s$
$f_{ch}^{<}\ e\ s = s \,/\, r \lessdot (f_{st}^{<} \circ ((edit\ e \circ f) \vartriangle id))\ r$
where $r = zoom\ i\ s$
$i = mcp\ \langle affect\ e\ (f\ s) \rangle$

The function $f_{ch}^{<}$ maps an edit operation on views, e, into an update operation on sources. The evaluation of $f_{ch}^{<}$ is illustrated in Figure 4, which shows how an updated source is obtained by an indirect route. A source is firstly mapped into a view via f, with the affected view subterm of the edit extracted via *affect e*. After that, the labels in the affected view subterm are collected and are used to identify an alignment position, r, covering the affected view subterm. A view of the alignment position is then constructed by applying f, and is edited before the state-based 'put' function $f_{st}^{<}$ maps it into an updated source subterm. Finally, the standard split operator, $\vartriangle$, with type $(a \to b) \to (a \to c) \to (a \to (b, c))$, is used to produce a pair containing the result of applying two functions to a value. Here it combines the newly created source subterm with the original context that remains unchanged. (The straightforward re-labelling of the newly generated source subterm with i as the root label is omitted.)

As shown in Figure 4, there are several passes across the source/view boundary, and apart from the initial 'get' function application, all of them concern only the edit-affected subterms. Assuming the editing and 'get' functions have no worse runtime performance than the 'put' function, the complexity of $f_{ch}^{<}$ is $O\ (m \times \log n + c\ m)$ where n and m are the sizes of the source s and the change r, and c is the complexity function for $f_{st}^{<}$. Note that *affect e* $(f\ s)$ will have been executed prior to the 'put' function execution, and is not included in the performance analysis. The cost of computing $s \,/\, r$ is not considered because the context $s \,/\, r$ can be computed together with the focus r when the zooming is performed. The $m \times \log n$ part of the above complexity function comes from the computation of *mcp*, where m labels of size $\log n$ need to be processed.

The function $f_{ch}^{<}$ is expected to preserve the bidirectional properties of $f_{st}^{<}$. This is established when $f_{ch}^{<}$ operates at alignment positions.

THEOREM 23. *Given a well-aligning 'get' function f such that $f\ s = v$, then for all source subterms r of s and view subterms u of v, zoom $(mcp\ \langle u \rangle)\ s$ is the smallest alignment position covering u.*

PROOF. By definition, we have $\langle u \rangle \subseteq \langle zoom\ (mcp\ \langle u \rangle)\ s \rangle$.

$\langle u \rangle \subseteq \langle zoom\ (mcp\ \langle u \rangle)\ s \rangle \wedge u \preccurlyeq f\ s \wedge$
$(zoom\ (mcp\ \langle u \rangle)\ s) \preccurlyeq s$
$\Rightarrow$ { lemma 22 }
$u \preccurlyeq f\ (zoom\ (mcp\ \langle u \rangle)\ s) \wedge u \preccurlyeq f\ s \wedge$
$\langle zoom\ (mcp\ \langle u \rangle)\ s \rangle \preccurlyeq s$
$\Rightarrow$ { theorem 17 }
f aligns at subterm $zoom\ (mcp\ \langle u \rangle)\ s$ of s

Also from the definition of *mcp*, there exists no r such that $r \prec zoom\ (mcp\ \langle u \rangle)\ s$ and $\langle u \rangle \subseteq \langle r \rangle$. Thus, $zoom\ (mcp\ \langle u \rangle)\ s$ is the smallest alignment position covering u. □

We can state the bidirectional properties of $f_{ch}^{<}$.

THEOREM 24 (Consistency). $f\ (f_{ch}^{<}\ e\ s) = edit\ e\ (f\ s)$

PROOF.

$f\ (f_{ch}^{<}\ e\ s)$
$=$ { definition of $f_{ch}^{<}$ }
$f\ (s \,/\, r \lessdot (f_{st}^{<} \circ ((edit\ e \circ f) \vartriangle id))\ r)$
$=$ { r is an alignment position }
$f\ s \,/\, f\ r \lessdot (f \circ f_{st}^{<} \circ ((edit\ e \circ f) \vartriangle id))\ r)$
$=$ { consistency of $f_{st}^{<}$ }
$f\ s \,/\, f\ r \lessdot (edit\ e \circ f)\ r$
$=$ { r covers *affect* $e\ (f\ s)$, and locality of *edit* }
$edit\ e\ (f\ s \,/\, f\ r \lessdot f\ r)$
$=$ { r is an alignment position }
$edit\ e\ (f\ (s \,/\, r \lessdot r)))$
$=$ { cancellation }
$edit\ e\ (f\ s)$

□

For acceptability, we need an 'identity' edit that does not change the view.

THEOREM 25 (Acceptability). $f_{ch}^{<}\ (E\ \{edit = id\}) = id$.

PROOF.

$f_{ch}^{<}\ (E\ \{edit = id\})\ s$
$=$ { definition of $f_{ch}^{<}$ }
$s \,/\, r \lessdot (f_{st}^{<} \circ ((id \circ f) \vartriangle id))\ r$
$=$ { acceptability of $f_{st}^{<}$ }
$s \,/\, r \lessdot r$
$=$ { cancellation }
s

□

Undoability involves inverting an edit as a function.

THEOREM 26 (Undoability). $f_{ch}^{<}\ (e\ \{edit = (edit\ e)^{\circ}\}) \circ f_{ch}^{<}\ e = id$.

PROOF.

$(f_{ch}^{<}\ (e\ \{edit = (edit\ e)^{\circ}\}) \circ f_{ch}^{<}\ e)\ s$
$=$ { definition of $f_{ch}^{<}$, and constant *affect* in e }
$s \,/\, r \lessdot (f_{st}^{<} \circ ((edit\ e)^{\circ} \circ f \vartriangle id))\ ((f_{st}^{<} \circ (edit\ e \circ f \vartriangle id))\ r)$
$=$ { definition of $\vartriangle$ }
$s \,/\, r \lessdot f_{st}^{<}\ (((edit\ e)^{\circ} \circ f \circ f_{st}^{<})\ ((edit\ e \circ f)\ r, r),$
$f_{st}^{<}\ ((edit\ e \circ f)\ r, r))$

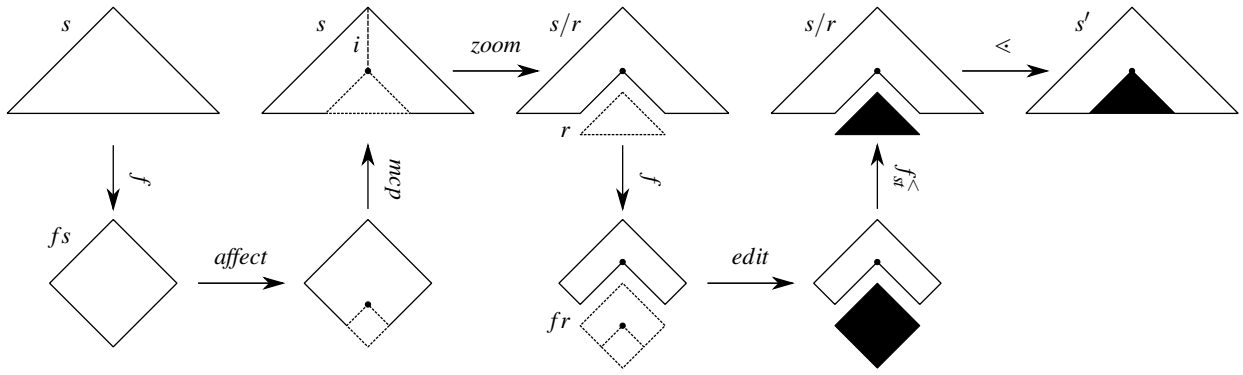

Figure 4. Change-based 'put' execution, showing relationship between source (top) and associated view (bottom) during transformation.

$$
\begin{array}{ll}
= & \{\text{ consistency of } f_{st}^{<}\ \} \\
 & s\,/\,r \lessdot f_{st}^{<}\ (((edit\ e)^{\circ} \circ edit\ e \circ f)\ r, f_{st}^{<}\ ((edit\ e \circ f)\ r, r)) \\
= & \{\ (edit\ e)^{\circ} \circ edit\ e = id\ \} \\
 & s\,/\,r \lessdot f_{st}^{<}\ (f\ r, f_{st}^{<}\ ((edit\ e \circ f)\ r, r)) \\
= & \{\ \text{undoability of } f_{st}^{<}\ \} \\
 & s\,/\,r \lessdot r \\
= & \{\ \text{cancellation}\ \} \\
 & s
\end{array}
$$

□

5. More Refined Locality

As we have seen, the performance of our proposal depends on the height of the source tree and the size of the affected region (i.e., the degree of locality of the edit). The former is clearly beyond the control of any bidirectional framework, and the latter is largely decided by the structure of the view. For fairly balanced trees, the majority of nodes are deep in the structure, so it is reasonable to suppose that the majority of edits will be too; given structure alignment, this implies a good degree of locality. A problem arises when the view tree is skewed, such as in a list, since the likelihood that a node appears at any depth is the same. If a node high in the structure is affected by an edit, such as a deletion, the affected subtree could be rather large.

This problem has already manifested itself in our binary-tree traversal example (see Section 1.4, where it is excessive to mark the whole sublist $[4,8,9]$ as affected). A better alternative is to recognize the sublist $[8,9]$ as unaffected context too.

Another example is post-order tree traversal, where any non-empty sublist of the view contains the head of the source, which results in very poor locality preservation. As a matter of fact, post-order traversal is excluded through the well-aligning condition.

Nevertheless, being a special kind of tree, lists enjoy a number of unique properties. We notice that unlike general trees, where a separate datatype is needed for contexts, the context type for lists is isomorphic to the list type itself, and so we can simply use lists as both contexts and foci, and use the append function ($+\!\!+$) as the close function ($\lessdot$). Given the symmetry of ($+\!\!+$), either the context or the focus can be edited, and all the definitions and results dualize. For example, consider the *reverse* function. Editing the front of the list view can be localized to a prefix of the view and mapped back to a suffix of the source.

As a result, it makes sense to try to capture a lower (right) bound of an edit-affected sublist, in addition to the upper (left) bound. Instead of splitting a list view into a prefix (context) and a suffix (focus), we can now see it as $l_1 +\!\!+ l_2 +\!\!+ l_3$. To reflect this specialization, we overload the infix operator $\lessdot$ and define its list version as $(l_1, l_3) \lessdot l_2 = l_1 +\!\!+ l_2 +\!\!+ l_3$. (In a sense, this is treating lists as semi-structured data, similar to traditional treatments of relational databases and graphs, and not as an algebraic datatype.) All the other definitions developed for general trees remain valid.

Now instead of always picking out a complete suffix or prefix, we can mark an interior list segment as affected by editing, and the same definition of $f_{ch}^{<}$ directly applies. For example, deleting 4 from $[7,6,5,4,8,9]$ only affects the interior segment $[4]$, leaving both contexts $[7,6,5]$ and $[8,9]$ unaffected. Correspondingly, the alignment positions now match subterms in the source with segments (rather than with tails) in the view.

6. Discussion

6.1 Context-Sensitive Editing

The editing system we have looked at so far is context-independent; this is particularly convenient for local editing, since the same edit can be applied both to a structure and to its subterms. For tree-structured views, it is sometimes useful to provide a full (or partial) path to the intended editing location, to narrow down the search. In this case, the editing becomes context sensitive, because the starting point of the path matters. Consider a path-based editing system.

$$\textbf{type}\ EditP\ t = \{\ edit :: Path \to t \to t, affect :: Path \to t \to Path\ \}$$

An edit operation now finds its target in a structure following a path, and produces the edited structure together with the path leading to the affected subterm. Note that these paths in the view should not be confused with labels of nodes that represent paths in the source.

The definition of $f_{ch}^{<}$ can be adapted for the new editing system. We separate all interesting steps into **where** clauses to facilitate explanation.

$$
\begin{array}{l}
f_{ch}^{<} :: Edit\ v \to Path \to s \to s \\
f_{ch}^{<}\ e\ p\ s = (s\,/\,r) \lessdot f_{st}^{<}\ (edit\ e\ p_3\ u', r)\ \textbf{where} \\
\quad v = f\ s \\
\quad p_1 = affect\ e\ p\ v \\
\quad u = zoom\ p_1\ v \qquad\qquad \text{-- affected subterm} \\
\quad i = mcp\ \langle u \rangle \\
\quad r = zoom\ i\ s \\
\quad u' = f\ r \\
\quad p_2 = travelUntil\ (p_1, v)\ u' \quad \text{-- path to } u' \\
\quad Just\ p_3 = stripPrefix\ p_2\ p_1 \quad \text{-- relative editing path}
\end{array}
$$

Note that we keep the path information of the edit explicit, so that it can be modified along with the shifting of focus. Compared with the context-independent version, there are a few additional steps. As the editing function returns a path locating the affected subterm, we need to open the view to get to it (u in the third clause above). A bigger challenge posed by this context-sensitivity is to find a

relative editing path when the starting point is moved to the root of subterm u'. We denote the path from the root of a structure x to its subterm y as $x \rightsquigarrow y$. Since we know the subterm relations $u \preccurlyeq u' \preccurlyeq v$ among the affected subterm, the view of the alignment position and the complete view, the path $p_3 = u' \rightsquigarrow u$ is the difference between $p_1 = v \rightsquigarrow u$ and $p_2 = v \rightsquigarrow u'$. We already know p_1; traversing p_1 until the root of u' gives us p_2, before we can perform path arithmetic to recover the correspondence between the path and structure inputs of *edit e*. Function *travelUntil* follows a path down a tree until it reaches a given subtree; the part of the path travelled is returned as the output. Function *stripPrefix* is a standard Haskell function of type $Eq\ a \Rightarrow [a] \rightarrow [a] \rightarrow Maybe\ [a]$ that strips the first input from the second one; since we know that p_2 is a prefix of p_1, the execution of $stripPrefix\ p_2\ p_1$ is always going to succeed. A concern here is that the additional computation does impose a performance overhead: travelling the path takes time linear in the height of the view tree. Clearly, the multiple traversals in the above code can be combined; we have presented them in separate steps for clarity.

6.2 Totality of 'Put' Functions

The proofs of bidirectional properties in Section 4.2 are done in a total setting, where the state-based transformations are assumed to execute successfully when given well-defined terms. Since the 'get' functions are regular structural recursions, it is reasonable to expect that if the execution over a complete source is successful, then the evaluation over subterms will also succeed. Establishing a similar safety property for 'put' functions is much harder; 'put' functions are often partial due to the conflicts between the edited view and the original source (though advanced type systems may help to specify their domains [10]), and usually do not expose their semantics other than through the bidirectional laws.

Since our approach employs state-based bidirectional frameworks as black-boxes, it is impossible to conclude that the approach is universally safe. Nevertheless, there are patterns to follow: most bidirectional frameworks create 'put' functions that try to trace the original 'get' execution backwards, and recursively deconstruct their view and source inputs in parallel, until a discrepancy is encountered, which is the point where the 'put' function has to either resolve the conflict or fail. Our change-based approach reduces the scope of a 'put' function by removing a context that is known to be unaffected by the edit. Consequently, we expect that failures, if there are any, happen within the processing of the alignment position, and so moving from a state-based approach to the change-based approach preserves the safety property.

7. Related Work

Incremental updates have been studied in the context of model transformation, for improving speed [13], and for achieving more refined semantics [8]. Similar to our design, both of these approaches also require additional specification of the effect of an edit. In contrast to tree-like datatypes, models are loosely-connected untyped graphs, which are more easily divided into independent fragments to be updated separately; whereas our well-aligning property of typed and overlapping subtrees is much harder to establish. In their work on graph transformations via structural recursion, Hidaka et al. [15] use a simplified assumption that different parts of the graph are always independent. As a result, the structural recursion is effectively reduced to a *concatMap* operation. However, this assumption is not valid in general; and as a result, the acceptability property does not hold in their framework.

The concept of an alignment position is also known as an *exclusive data source* in the database literature [9], and is used to prevent 'side effects' on the view that is edited (similar to our consistency requirement). Exclusive data sources are commonly computed by establishing *lineages* between a database and its view through tracing identifiers of data [5]. Databases are typically large, so it is never practical to process them completely for an update; as a result, identifying exclusive data sources is not considered as an optimization, but the core part of the update. Despite the obvious similarity of these ideas, techniques developed for databases are not applicable in our setting, due to the very different representations of data and transformations.

Despite being in a unidirectional setting, the concept of *adaptive programming* [1] is closely related to incremental updates. The basic idea of adaptive programming is to build up a complete input-output dependency graph for a given input, from some syntactic annotations to the program. Based on the dependency, a corresponding output change can be derived from an input change, which hopefully has a much better run-time performance compared to re-executing the program with the new input. However, it is not obvious how the technique can be applied in a bidirectional setting, where we need to derive an input change from an output change. Nevertheless, this would be an interesting future direction to explore.

Explicit caching of intermediate computations is another way of achieving incremental execution. If an input change can be described as a loop increment, an incremental version of the program under the change can be constructed and benefit from previously computed results [20]. In a sense, the contexts in our approach can be seen as cached values, and the focuses can be seen as increments. Since in our case the closing operation ($\lessdot$) that combines the cached result and the newly computed increment is not dependent on the transformations, we do not need to 'improve' the transformations to achieve incrementality.

Indexed elements in source structures and parametricity arising from polymorphic 'get' functions are the key components of *semantic bidirectionalization* [28] – deriving a 'put' function without inspecting the syntactic definition of the 'get' function. Similar to our approach, the indices in [28] are unique and cannot be created by 'get' transformations, so that individual elements in the view can be mapped back to their source origins, as can any edits. Nevertheless, there has not been any attempt to derive structure correspondences from the element correspondences, as we do in this paper. Consequently, editing view structures is not permitted in semantic bidirectionalization, while our approach only optimizes a given 'put' function instead of creating one.

Maintaining proper alignment between ordered source and view is itself an important semantic issue of bidirectional programming, as view editing may cause mismatchings between view/source data. *Matching lenses* [3] aims at addressing this issue without hard-wiring alignment strategies into bidirectional systems. Though our use of alignment in this paper is intended for optimization purposes, we note that a change-based 'put' function should preserve the alignment semantics of the state-based 'put' function it uses, because the unaffected context that is not processed will have no impact on alignment.

The connection between alignment and incrementality is speculated in [18]. However, as far as we are aware, there has not been any concrete proposal before.

8. Conclusions

We have developed a change-based bidirectional transformation framework that focuses on changes rather than data. The technique we have presented is very general, and most existing state-based frameworks may draw benefits from its adoption, so long as the 'get' function is well-aligning. This condition is semantic; there is no restriction on the language that is used for implementation.

In the future, we plan to look at ways of dealing with monomorphic functions, which are widely used in XML transformations. In

contrast to the polymorphic case presented in this paper, the labelling of tree nodes does require some adjustment of a state-based bidirectional framework. It will be interesting to see whether such adjustments can be made in a systematic way.

Acknowledgements

We are grateful to Ralf Hinze for his valuable comments on an early draft of the paper; and the ICFP reviewers for their detailed and insightful reviews. The work was partly conducted when the first author was at the University of Oxford supported by the UK EPSRC grant *Generic and Indexed Programming* (EP/E02128X).

References

[1] U. A. Acar, G. E. Blelloch, and R. Harper. Adaptive functional programming. *ACM Transactions on Programming Languages and Systems*, 28:990–1034, November 2006.

[2] F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

[3] D. M. Barbosa, J. Cretin, N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *International Conference on Functional Programming (ICFP)*, pages 193–204, New York, NY, USA, 2010. ACM.

[4] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *Principles of Programming Languages*, pages 407–419, New York, NY, USA, Jan. 2008. ACM.

[5] Y. Cui, J. Widom, and J. L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Transactions on Database Systems*, 25:179–227, June 2000.

[6] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, pages 260–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[7] U. Dayal and P. A. Bernstein. On the correct translation of update operations on relational views. *ACM Transactions on Database Systems*, 7(3):381–416, 1982.

[8] Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations. In *Theory and Practice of Model Transformations*, ICMT'10, pages 61–76, Berlin, Heidelberg, 2010. Springer-Verlag.

[9] L. Fegaras. Propagating updates through XML views using lineage tracing. In *International Conference on Data Engineering*, pages 309–320, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[10] J. N. Foster, M. B. Greenwald, J. T. Moore, B. C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007. Preliminary version in POPL '05.

[11] J. N. Foster, B. C. Pierce, and S. Zdancewic. Updatable security views. In *Computer Security Foundations*, pages 60–74, Washington, DC, USA, 2009. IEEE Computer Society.

[12] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *International Conference on Functional Programming (ICFP)*, pages 383–396, New York, NY, USA, 2008. ACM.

[13] H. Giese and R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 8:21–43, 2009. 10.1007/s10270-008-0089-9.

[14] G. Gottlob, P. Paolini, and R. Zicari. Properties and update semantics of consistent views. *ACM Transactions on Database Systems*, 13(4):486–524, 1988.

[15] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *International Conference on Functional Programming (ICFP)*, ICFP '10, pages 205–216, New York, NY, USA, 2010. ACM.

[16] R. Hinze and J. Jeuring. Functional Pearl: Weaving a web. *Journal of Functional Programming*, 11(6):681–689, nov 2001.

[17] R. Hinze, J. Jeuring, and A. Löh. Type-indexed data types. *Science of Computer Programming*, 51(1-2):117–151, 2004.

[18] M. Hofmann, B. Pierce, and D. Wagner. Symmetric lenses. In *Principles of Programming Languages (POPL)*, POPL '11, pages 371–384, New York, NY, USA, 2011. ACM.

[19] Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Workshop on Partial Evaluation and Program Manipulation (PEPM)*, pages 178–189, New York, NY, USA, 2004. ACM.

[20] Y. A. Liu, S. D. Stoller, and T. Teitelbaum. Static caching for incremental computation. *ACM Transactions on Programming Languages and Systems*, 20:546–585, May 1998.

[21] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *International Conference on Functional Programming (ICFP)*, pages 47–58, New York, NY, USA, 2007. ACM.

[22] C. McBride. The derivative of a regular type is its type of one-hole contexts (extended abstract). `http://strictlypositive.org/diff.pdf`, University of Durham, 2001.

[23] L. Meertens. Designing constraint maintainers for user interaction. `ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps`, CWI, Amsterdam, 1998.

[24] S.-C. Mu, Z. Hu, and M. Takeichi. An injective language for reversible computation. In *Mathematics of Program Construction*, volume 3125 of *Lecture Notes in Computer Science*, pages 289–313. Springer, 2004.

[25] H. Pacheco and A. Cunha. Generic point-free lenses. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*, pages 331–352. Springer Berlin / Heidelberg, 2010.

[26] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[27] P. Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and Systems Modeling*, 9:7–20, 2010.

[28] J. Voigtländer. Bidirectionalization for free! (Pearl). In *Principles of Programming Languages (POPL)*, pages 165–176, New York, NY, USA, 2009. ACM.

[29] J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Combining syntactic and semantic bidirectionalization. In *International Conference on Functional Programming (ICFP)*, pages 181–192, New York, NY, USA, 2010. ACM.

[30] P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359, New York, NY, USA, 1989. ACM.

[31] M. Wang, J. Gibbons, K. Matsuda, and Z. Hu. Gradual refinement: Blending pattern matching with data abstraction. In C. Bolduc, J. Desharnais, and B. Ktari, editors, *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*, pages 397–426. Springer Berlin / Heidelberg, 2010.