Three Complementary Approaches to Bidirectional Programming

Nate Foster[1], Kazutaka Matsuda[2], and Janis Voigtländer[3]

[1] Cornell University; jnfoster@cs.cornell.edu
[2] Tohoku University; kztk@kb.ecei.tohoku.ac.jp
[3] University of Bonn; jv@iai.uni-bonn.de

Abstract. This paper surveys three distinct approaches to bidirectional programming. The first approach, *syntactic bidirectionalization*, takes a program describing the forward transformation as input and calculates a well-behaved reverse transformation. The second approach, *semantic bidirectionalization*, is similar, but takes the forward transformation itself as input rather than a program describing it. It requires the transformation to be a polymorphic function and uses parametricity and free theorems in the proof of well-behavedness. The third approach, based on *bidirectional combinators*, focuses on the use of types to ensure well-behavedness and special constructs for dealing with alignment problems. In presenting these approaches, we pay particular attention to use of *complements*, which are structures that represent the information discarded by the transformation in the forward direction.

1 Introduction

Bidirectional transformations are a mechanism for converting data from one form to another, and vice versa. The forward transformation, often called *get*, maps a source structure to a view, while the backward transformation, often called *put*, maps a (possibly updated) view back to a source. The need for bidirectional transformations arises in a variety of areas including data management, software engineering, programming languages, and systems (Bancilhon and Spyratos 1981; Benton 2005; Berdaguer et al. 2007; Bohannon et al. 2006; Brabrand et al. 2008; Cunha et al. 2009; Czarnecki et al. 2009; Dayal and Bernstein 1982; Ennals and Gay 2007; Fisher and Gruber 2005; Foster et al. 2007a, 2009; Hu et al. 2008; Kawanaka and Hosoya 2006; Lutterkort 2008; Meertens 1998; Miller et al. 2001; Ramsey 2003; Schürr 1995; Stevens 2007; Xiong et al. 2007), as well as in generic programming frameworks where bidirectional transformations map between user-defined and canonical representations (*e.g.*, as a "sum of products") used by generic functions (Jeuring et al. 2009) or between an interface expressed using algebraic datatypes and an implementation using abstract datatypes (Wang et al. 2010).

In recent years, a number of programming language techniques for describing bidirectional transformations have been proposed. These techniques offer several advantages over the alternative—describing bidirectional transformations using

separate programs. First, because they make it possible to describe two transformations in a single program, bidirectional programming languages eliminate redundancy and make programs easier to maintain as formats evolve. Second, because the semantics of these languages typically offers guarantees about how the two transformations will operate together, they obviate the need for complicated pencil-and-paper proofs.

An important consideration in the design of a bidirectional language is the notion of what constitutes a "reasonable" pair of *get* and *put* functions. Several criteria for this have been discussed in the literature. Most of the conditions that have been adopted in existing languages are based on notions of correctness developed for the database view-update problem (Bancilhon and Spyratos 1981), but there are interesting and important variations between the semantic choices made in different techniques.

In this article, we survey three techniques developed in the programming language community to approach bidirectional programming. In the first two techniques, originally developed by Matsuda et al. (2007) and Voigtländer (2009), the programmer writes a program for the *get* function in an existing functional language, and a bidirectionalization technique is responsible for coming up with a suitable program for *put*. This can either be done using an algorithm that works on a syntactic representation of (somehow restricted) *get* functions and calculates appropriate *put* functions, or by exploiting the (higher-order and typed) abstractions and algorithmic methods available in the functional language itself. The third technique uses a domain-specific language approach, as exemplified in the series of languages developed by Foster et al. (2007b), in which a certain class of transformations of interest is covered by providing a collection of well-behaved *get* and *put* pairs—so called *lenses*—as well as systematic and sound ways of constructing bigger lenses out of smaller ones. A type system provides strong semantic guarantees.

All three techniques described in this paper are (ultimately) based on a notion of *complement*—*i.e.*, an explicit representation of the information discarded by the forward transformation. The technique developed by Matsuda et al. (2007) is fundamentally based on the classic *constant-complement* approach from the database literature (Bancilhon and Spyratos 1981). The key ingredient of the technique is a syntactic program transformation that takes a description of the *get* function and produces a function that computes a complement. The original presentation of the technique of Voigtländer (2009) was not in terms of complements but we show in this paper, for the first time, that it *can* also be formulated in terms of the constant-complement approach.[1] Likewise, the particular instance of the domain-specific language approach we describe (Barbosa et al. 2010) is presented here using a new and cleaner formulation that highlights the role of complements in that setting.

Section 2 discusses possible notions of "reasonable" pairs of *get* and *put* functions. Section 3 discusses the constant-complement approach, which is then used

[1] Also, we give an improved account of a generic programming generalization of the technique, in Section 5.4.

to present bidirectionalization via syntactic program transformations in Section 4 and bidirectionalization via semantic reasoning principles about polymorphic functions in Section 5. We then present bidirectional combinators, specifically *matching lens combinators*, in Section 6. We conclude with a comparative discussion and pointers to related work in Section 7.

2 Semantics

Let us begin by exploring the properties we might expect a pair of functions *get* and *put* to obey to qualify as a well-behaved bidirectional transformation, using a specific example to guide the discussion. Assume that the forward transformation is the following Haskell function,

$$\begin{array}{l}\texttt{get} :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha] \\ \texttt{get}\ s = \textbf{let}\ n = (\texttt{length}\ s)\ \texttt{`div`}\ 2\ \textbf{in}\ \texttt{take}\ n\ s\end{array}$$

which maps *source* lists (of arbitrary type) to *view* lists (of the same type), omitting some of the information contained in the input, namely the second half of the list. It should be clear that the *get* function is not injective, and so there is no hope of "simply" setting up a bijection between the set of source lists and the set of view lists. Instead, when the view (*i.e.*, the first half of the original list) is modified and we need to propagate the change back to the underlying source, we must supply the `put` transformation with the updated view as well as the original source:

$$\texttt{put} :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha] \to [\alpha]$$

One tempting implementation is as follows, combining the updated view with the list items deleted from the original source:

$$\texttt{put}\ v\ s = \textbf{let}\ n = (\texttt{length}\ s)\ \texttt{`div`}\ 2\ \textbf{in}\ v +\!\!+ (\texttt{drop}\ n\ s)$$

But is it any good?

A natural requirement on the `put` function is that it should fully reflect any changes made to the view in the underlying source. One way to express this requirement is as a "round-tripping" law which says: if we change the view in some way and then perform `put` followed by `get`, we should end up with the very same modified view. In general, if S is the set of source structures, V is the set of views, and the *get* and *put* functions have the following types,

$$\begin{array}{l}\mathit{get} \in S \to V \\ \mathit{put} \in V \to S \to S\end{array}$$

then we want the following law to hold for every $s \in S$ and $v \in V$:

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

There is also another natural law that constrains round-trips in the opposite direction. It stipulates that if the view is not modified at all, then the *put* function must not change the source. This condition is captured by the following law:

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

We will refer to a pair of *get* and *put* functions that satisfy these two laws as a *well-behaved lens*.[2] The concrete functions `get` and `put` shown above do not constitute such a pair: while GetPut is satisfied, PutGet is not (e.g., `get` (`put` [ ] $[a, b, c]$) = `get` $[b, c] = [b] \neq [\,]$). Further below, we will see a function `put` that *does* complete the above `get` towards a well-behaved lens.

We refer to well-behaved lenses that obey the following additional law,

$$put\ v'\ (put\ v\ s) = put\ v'\ s \qquad \text{(PutPut)}$$

as *very well-behaved*. This law ensures that the *put* function does not have "side-effects" on the source. It is not satisfied for the concrete function `put` above either, since `put` [ ] (`put` [ ] $[a, b, c]$) = `put` [ ] $[b, c] = [c] \neq [b, c] =$ `put` [ ] $[a, b, c]$.

Note that if PutPut *does* hold, then together with GetPut it implies the following equality,

$$put\ (get\ s)\ (put\ v\ s) = s$$

which means that updates made to the view can always be "undone."

A natural question to ask at this point is whether for every *get* function, there at least *exists* a corresponding *put* such that the two functions form a very well-behaved lens. Unfortunately, the answer to this question is negative. To see why, consider again the specific function `get` above and consider `put` [ ] $[a, b, c]$, for an *arbitrary* implementation of `put`. By the PutGet law, the new source produced by evaluating this function must either be the empty list [ ] or a singleton list $[x]$ for some x. However, by the PutPut and GetPut laws we must also have `put` $[a]$ (`put` [ ] $[a, b, c]$) = `put` $[a]$ $[a, b, c] = [a, b, c]$. That is, either `put` $[a]$ [ ] = $[a, b, c]$ (if `put` [ ] $[a, b, c] = [\,]$) or `put` $[a]$ $[x] = [a, b, c]$ (if `put` [ ] $[a, b, c] = [x]$), which is impossible for arbitrary a, b, and c! (Note that polymorphism prevents encoding b and c into x.)

To avoid such problems, many bidirectional programming languages allow the *put* function to fail on certain inputs. For the example, we can provide a partial solution by defining `put` to only accept inputs where the length of the view list is half the length (rounded down if necessary) of the source list.

```
put :: forall α. [α] → [α] → [α]
put v s = let n = (length s) `div` 2
          in  if (length v) == n then v ++ (drop n s)
                                 else  error "Shape mismatch."
```

[2] The definition of lenses often includes a third function *create* $\in V \to S$ and a law CreateGet, which is analogous to PutGet. See Section 6 for an alternative approach.

This definition (still) satisfies GetPut, and satisfies weakened versions of PutGet and PutPut (the hypotheses test the definedness of specific function calls):

$$\frac{(put\ v\ s)\downarrow}{get\ (put\ v\ s) = v} \qquad \text{(Partial-PutGet)}$$

$$\frac{(put\ v\ s)\downarrow}{put\ v'\ (put\ v\ s) = put\ v'\ s} \qquad \text{(Partial-PutPut)}$$

We will call a *get*/*put* pair satisfying the GetPut, Partial-PutGet, and Partial-PutPut laws a *partial* very well-behaved lens. Note, though, that even in a partial lens we require the forward transformation to be a *total* function—*i.e.*, we do not allow $get\ s = \bot$ for any s.

Summarizing the situation so far, for the given `get` function, there is no way to provide a `put` function such that `get`/`put` is a very well-behaved lens. But it *is* possible to complete it towards a partial very well-behaved lens (with the second implementation of `put` just given). It is *also* possible to complete it towards a (not very) well-behaved lens with the following implementation that combines the updated view with an appropriately long prefix of the second half of the original source (extended with undefined list items to handle cases where the update to the view makes the list longer):

```
put :: forall α. [α] → [α] → [α]
put v s = let l = length s
              k = length v
          in  v ++ take (k + l `mod` 2) (drop (l `div` 2) s ++ repeat ⊥)
```

The result is not a very well-behaved lens, and not even a partial very well-behaved lens. While GetPut and PutGet are satisfied (and so clearly Partial-PutGet is), neither PutPut nor Partial-PutPut (which are equivalent here, as `put` is total) is satisfied, since `put` $[a]$ (`put` $[\,]$ $[a, b]$) = `put` $[a]$ $[\,]$ $= [a, \bot] \neq [a, b] =$ `put` $[a]$ $[a, b]$.

Clearly, it is possible to abuse the admission of partiality in *put*, and the preconditions in Partial-PutGet and Partial-PutPut, to at least conceptually always manufacture a backward transformation leading to a partial very well-behaved lens as follows:

$$put\ v\ s = \begin{cases} s & \text{if } v = get\ s \\ \bot & \text{otherwise} \end{cases}$$

Such a backward transformation is rather useless, so our aim in manufacturing partial very well-behaved lenses must be to make *put* defined on as many inputs as possible. For example, for the specific function `get` from the beginning of this section, a slight improvement (in terms of definedness, while preserving partial very well-behavedness) to the second implementation of `put` above would be possible by weakening the condition

$$(\texttt{length}\ v) == n$$

to

$$\begin{aligned}(\texttt{length}\ v) == n\ \ ||\ \ ((\texttt{length}\ v) == n-1)\ \ \&\&\ \ \texttt{even}\ (\texttt{length}\ s)\\ ||\ \ ((\texttt{length}\ v) == n+1)\ \ \&\&\ \ \texttt{odd}\ (\texttt{length}\ s)\end{aligned}$$

In what follows, we will encounter more examples of well-behaved lenses, very well-behaved lenses, and partial very well-behaved lenses. Specifically, since the approach from Section 3 is tightly tied to PutPut or at least its partial variant, the techniques from Sections 4 and 5 always produce partial very well-behaved lenses. The technique from Section 6, on the other hand, always delivers total *put* functions, but sacrifices PutPut, thus yielding well-behaved lenses. In either setting, it is perfectly possible that for specific examples actually (total) very well-behaved lenses are obtained. We do *not* consider a notion of partial well-behaved lens here, though such lenses feature in the combined syntactic/semantic approach to bidirectionalization of Voigtländer et al. (2010).

3 The Constant-Complement Approach

In this section, we briefly review the *constant-complement* approach to view updating (Bancilhon and Spyratos 1981) which will serve as the basis of the bidirectionalization techniques (Matsuda et al. 2007; Voigtländer 2009) described in Sections 4 and 5.

Intuitively, a *complement* is a structure that preserves the information lost by the forward transformation. To define complements formally, we need to introduce the concept of function tupling. Given two total functions $f \in X \to Y$ and $g \in X \to Z$, the tupled function $\langle f, g\rangle \in X \to (Y, Z)$ is the function defined as follows:

$$\langle f, g\rangle\ x = (f\ x, g\ x)$$

That is, $\langle f, g\rangle$ duplicates the input x, passes one copy to f and the other to g, and places the results in a pair.

Definition 1. *Let* $\mathrm{get} \in S \to V$ *be a total function from* S *to* V*. A total function* $\mathrm{res} \in S \to C$ *computes a* complement *for* get *if and only if the tupled function* $\langle \mathrm{get}, \mathrm{res}\rangle \in S \to (V, C)$ *is injective.*

We will call *res* (abbreviation for "residue") a *complement function* for *get*.

As an example to illustrate, let $\texttt{add} :: (\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ be a function defined by $\texttt{add}\ (x, y) = x+y$. Then, the function $\texttt{fst} :: (\mathbb{R}, \mathbb{R}) \to \mathbb{R}$ defined by $\texttt{fst}\ (x, y) = x$ is a complement function for add. Note that the codomains of a function f and a complement function g for f can be different. This flexibility will be useful in Section 4 where we derive a complement function from a program defining the forward transformation automatically.

Complements provide a simple mechanism for *bidirectionalizing* an existing function: given a forward transformation, provided that we can compute a complement for it and invert the tupled function, we can obtain a very well-behaved reverse transformation mechanically (Bancilhon and Spyratos 1981). Let $get \in S \to V$ be a forward transformation function and let $res \in S \to C$ be a

complement function for it. (Note that both *get* and *res* must be total functions.) The function $put_{(get,res)}$ defined by

$$put_{(get,res)}\ v\ s = inv\ (v, res\ s), \text{ where } inv = \langle get, res\rangle^{-1} \qquad \text{(Upd)}$$

is a suitable backward transformation function. That is, when combined with *get*, it yields a very well-behaved lens. We have to be careful about definedness here. There are two cases to consider:

- The function $\langle get, res\rangle$ is not only injective, but also surjective, and *inv* is its (full) inverse, *i.e.*, for every $s \in S$, $v \in V$, and $c \in C$:

$$inv\ (\langle get, res\rangle\ s) = s \qquad \text{(LeftInv)}$$

$$\langle get, res\rangle\ (inv\ (v, c)) = (v, c) \qquad \text{(RightInv)}$$

  Then $put_{(get,res)}$ is a total function (*i.e.*, defined for every v and s) and *get* and $put_{(get,res)}$ constitute a very well-behaved lens—*i.e.*, they satisfy the GetPut, PutGet, and PutPut laws.
- The function $\langle get, res\rangle$ is not surjective, and *inv* is a left-inverse for it but only a partial right-inverse. That is, for every $s \in S$, $v \in V$, and $c \in C$ we have:

$$inv\ (\langle get, res\rangle\ s) = s \qquad \text{(LeftInv)}$$

$$\frac{(inv\ (v, c))\downarrow}{\langle get, res\rangle\ (inv\ (v, c)) = (v, c)} \qquad \text{(Partial-RightInv)}$$

  Then $put_{(get,res)}$ is partial, and *get* and $put_{(get,res)}$ constitute (only) a partial very well-behaved lens—*i.e.*, satisfy the laws GetPut, Partial-PutGet, and Partial-PutPut.[3]

In either case, the fact that the complement is kept constant can be readily seen since Upd and Partial-RightInv (or RightInv) imply:

$$\frac{(put_{(get,res)}\ v\ s)\downarrow}{res\ (put_{(get,res)}\ v\ s) = res\ s}$$

In general, there can be many possible complement functions for a given *get* function. For example, all of the functions below are valid complement functions for $\texttt{add} :: (\mathbb{R}, \mathbb{R}) \to \mathbb{R}$,

$$\texttt{fst}\ (x, y) = x$$
$$\texttt{sub}\ (x, y) = x - y$$
$$\texttt{id}_{\texttt{pair}}\ (x, y) = (x, y)$$

[3] For example, $put_{(get,id)}$ generally defines the trivial function *put* presented in Section 2, which is only defined on inputs (v, s) where $v = get\ s$.

and lead to the following backward transformation functions:

$$\begin{aligned}
\mathtt{put}_{(\mathtt{add},\mathtt{fst})}\ v\ (x,y) &= (x, v-x)\\
\mathtt{put}_{(\mathtt{add},\mathtt{sub})}\ v\ (x,y) &= ((v+(x-y))/2, (v-(x-y))/2)\\
\mathtt{put}_{(\mathtt{add},\mathtt{id}_{\mathtt{pair}})}\ v\ (x,y) &= \begin{cases} (x,y) \text{ if } v = x+y \\ \bot \quad \text{otherwise} \end{cases}
\end{aligned}$$

These backward transformation functions differ in the updates that they can handle. The first two functions handle arbitrary modifications to the view while the last does not allow any modifications—the view v must be equal to $x + y$. Bancilhon and Spyratos (1981) introduce the following preorder, under which smaller complement functions allow a larger set of updates to be propagated (*cf.*, Theorem 1 below).

Definition 2. *Let $f \in S \to C$, $g \in S \to C'$ be total functions. The* collapsing order*, $\preceq$, is the preorder defined by:*

$$f \preceq g \quad \Longleftrightarrow \quad \forall s, s' \in S.\, g\ s = g\ s' \Rightarrow f\ s = f\ s'$$

Intuitively, if $f \preceq g$ then f collapses the domain S at least as much as g. Minimal functions under this preorder are functions that collapse every element of the input to a single result—*i.e.*, constant functions. Maximal functions are those that collapse nothing—*i.e.*, injective functions. Among the above examples of complement functions for `add`, the $\mathtt{id}_{\mathtt{pair}}$ function is greater than the others, while `fst` and `sub` are incomparable.

Since a complement function preserves information that does not appear in the view obtained by a forward transformation, and since the backward transformation function derived from a complement function via equation Upd forbids any change in the information that the complement has kept, a smaller complement function under the preorder $\preceq$ gives a better backward transformation function, because it keeps less information. Formally, we have the following theorem (Bancilhon and Spyratos 1981).

Theorem 1. *Let $get \in S \to V$ be a forward transformation and $res_1 \in S \to C$ and $res_2 \in S \to C'$ be two complement functions for* get*. Then we have that*

$$\forall v \in V, s \in S.\ (put_{(get,res_2)}\ v\ s)\downarrow \Rightarrow put_{(get,res_1)}\ v\ s = put_{(get,res_2)}\ v\ s$$

if and only if $res_1 \preceq res_2$.

Even though the preorder $\preceq$ helps to tell which complement is better in terms of the definedness of *put*, note that it does not express *everything* about the precedence between complements. Usually, there are some pragmatic reasons to prefer one complement over another. For example, the following function, `biasedSub`, is also a complement for `add`:

$$\mathtt{biasedSub}\ (x,y) = 3x - y$$

The complement functions `sub` and `biasedSub` are incomparable under $\preceq$. But, it may happen that one prefers `sub` over `biasedSub` because of the simplicity of the definition or the more intuitive update behavior. Some in the literature prefer time- or space-efficient complement functions (Perumalla and Fujimoto 1999) but do not care about $\preceq$, while others prefer a more restricted class of complement functions for their intended requirements (*e.g.*, complement functions in terms of poset morphisms for uniqueness of *put* (Hegner 2004)).

The general bidirectionalization framework presented in this section has been used to bidirectionalize relational queries in the context of databases (Cosmadakis and Papadimitriou 1984; Laurent et al. 2001; Lechtenbörger and Vossen 2003). Sections 4 and 5 present methods for deriving complement functions for functional programs that manipulate algebraic data structures such as lists and trees (Matsuda et al. 2007; Voigtländer 2009).

4 Syntactic Bidirectionalization

In the remainder of the paper, we review the three techniques for development of bidirectional programs mentioned in the introduction. All three use complements in some sense. We begin in this section by introducing the syntactic bidirectionalization method originally proposed by Matsuda et al. (2007). It is the method most obviously based on complements, as it directly constructs complement functions to obtain bidirectional programs. Indeed, it precisely follows the constant-complement approach as outlined in the previous section; it takes a program describing a forward transformation and generates a program describing a backward transformation in three steps:

1. *Derivation of a Complement Function.* From a given program describing a forward transformation f, the method syntactically derives a program describing a complement function f^{res} for f.
2. *Tupling and Program Inversion.* From the program of the forward transformation and that of the derived complement function, the method derives a program of the partial inverse $\langle f, f^{\mathrm{res}} \rangle^{-1}$ of their tupling by using a syntactic tupling transformation (Hu et al. 1997) and syntactic program inversion. The inverse is partial in the sense that it satisfies LeftInv and PartialRightInv from the previous section.
3. *Construction of a Backward Transformation.* From the programs of the complement function f^{res} and the partial inverse $\langle f, f^{\mathrm{res}} \rangle^{-1}$ of the tupled function, the method constructs a program of a backward transformation using Upd. It can be optimized using syntactic fusion (Wadler 1990) or partial evaluation. Since fusion can remove "intermediate data" produced by the complement function, a fused backward transformation becomes monolithic and looks more like one a programmar would write.

Since in all three steps, syntactic transformations are performed on the program definitions of functions, the method itself is called *syntactic*. One of the main advantages of syntactic bidirectionalization is that we can apply program analyses

to obtain "better" backward transformation functions. For example, Matsuda et al. (2007) show how to use a range analysis to produce smaller complement functions (Section 4.4). On the other hand, even a small syntactic difference in forward transformations may affect the bidirectionalization results, which reduces the predictability of the method from a user's point of view.

4.1 Describing Forward Transformations

The input programs of the method must be given by functions in *affine* and *treeless* form (Wadler 1990) defined by a constructor-based first-order functional language with pattern matching. As a simple example, consider a transformation that takes a list of pairs and returns the list containing all the first components of those pairs. This forward transformation function can be defined in our language as follows:

$$\begin{aligned} &\texttt{mapfst}\ [\,] &&= [\,] \\ &\texttt{mapfst}\ ((a, b) : x) &&= a : (\texttt{mapfst}\ x) \end{aligned}$$

It decomposes the input data by pattern matching and constructs new data via data constructors.

Intuitively, being *affine* means that a function must not copy any data, and being *treeless* means that there is no function composition. Formally, a function is in *affine form* if, for any branch, every variable from the left-hand side occurs at most once in the corresponding right-hand side,[4] and a function is in *treeless form* if, for any function call, all arguments are variables. A simple example of a non-affine program is `dup` defined by $\texttt{dup}\ x = (x, x)$. A simple example of a non-treeless program is $\texttt{fstHd}\ x = \texttt{fst}\ (\texttt{head}\ x)$ where $\texttt{head}\ (x : xs) = x$.

Even though the language is restricted, it has enough expressive power to describe many useful basic functions such as `head`, `tail`, `init`, `last`, `fst`, `snd`, `zip`, `concat`, and first-order specializations of `map` like `mapfst`. With a small extension on patterns, it also can describe some first-order specializations of `filter` (Matsuda et al. 2009).

4.2 Deriving Complement Functions

Given the function definition of a forward transformation, the method starts by automatically deriving a *small* (with respect to the preorder from Definition 2) complement function so that tupling the two functions gives an injective function. For example, the complement function automatically derived for `mapfst` is as follows:

$$\begin{aligned} &\texttt{mapfst}^{\text{res}}\ [\,] &&= \mathsf{C}_1 \\ &\texttt{mapfst}^{\text{res}}\ ((a, b) : x) &&= \mathsf{C}_2\ b\ (\texttt{mapfst}^{\text{res}}\ x) \end{aligned}$$

[4] For simplicity, we do not consider **case** and **let**; thus, every expression in the language must be either a variable use, a constructor application, or a function application. Typical uses of **case** can be replaced by pattern matching, but typical uses of **let** correspond to the creation of intermediate results, *i.e.*, to function composition, which is disallowed.

One can see that the variable b present but unused in the second defining equation of `mapfst` is kept in the corresponding right-hand side of the complement function, and that different constructors C_1 and C_2 are added to trace which branch was taken. Also, for a function call (`mapfst` x), the corresponding complement-function call ($\texttt{mapfst}^{\text{res}}\ x$) occurs in the corresponding branch of the derived program. A close look at the definition in the above example reveals that the derived complement function actually computes the list containing all the second components of the pairs in the input list, *i.e.*, `mapsnd` (modulo constructor names). Hence, one can easily see that although `mapfst` is non-injective, the tupled function $\langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle$ is injective. Note that it is *not surjective* onto its potential range $\mathsf{Range}(\texttt{mapfst}) \times \mathsf{Range}(\texttt{mapfst}^{\text{res}})$ as it always returns pairs of lists with the same length. For example, there is no x such that

$$\langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle\ x = ([3], \mathsf{C}_1)$$

Later on, we will see how this non-surjectivity leads to a non-total *put* function.

In general, the syntactic bidirectionalization method uses the following three principles to derive complement functions. They are all guided by eliminating spurious sources of non-injectivity.

- *Branch Tags.* Constructors are used in the complement function to trace which branch would be taken by the forward transformation. For example,

$$\begin{array}{l} \texttt{true}\ \mathsf{True}\ = \mathsf{True} \\ \texttt{true}\ \mathsf{False} = \mathsf{True} \end{array} \quad \text{leads to} \quad \begin{array}{l} \texttt{true}^{\text{res}}\ \mathsf{True}\ = \mathsf{C}_1 \\ \texttt{true}^{\text{res}}\ \mathsf{False} = \mathsf{C}_2 \end{array}$$

- *Unused Variables.* Unused variables, which occur in a left-hand side of the forward transformation but not in the corresponding right-hand side, must be used in the complement function. For example,

$$\texttt{fst}\ (x, y) = x \qquad \text{leads to} \qquad \texttt{fst}^{\text{res}}\ (x, y) = \mathsf{C}\ y$$

- *Complement Function Calls.* For every function call $(f\ x_1\ x_2\ \ldots\ x_n)$ in the definition of the forward transformation, there is a corresponding call of the complement function, $(f^{\text{res}}\ x_1\ x_2\ \ldots\ x_n)$, in the complement definition. For example,

$$\begin{array}{ll} \texttt{fstHd}\ (x : xs) = \texttt{fst}\ x \\ \texttt{fst}\ (x, y) \quad\ = x \end{array} \quad \text{leads to} \quad \begin{array}{ll} \texttt{fstHd}^{\text{res}}\ (x : xs) = \mathsf{C}_1\ (\texttt{fst}^{\text{res}}\ x)\ xs \\ \texttt{fst}^{\text{res}}\ (x, y) \quad\ = \mathsf{C}_2\ y \end{array}$$

A formal algorithm working on the syntax description of the input functions is given in the original paper describing syntactic bidirectionalization (Matsuda et al. 2007).

4.3 Deriving Backward Transformation Functions

After obtaining the complement function, the method generates a backward transformation function via equation Upd, using two syntactic program transformations: tupling and inversion.

For the example `mapfst`, the method first automatically derives the following definition for the tupled function $\langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle$:

$$\begin{array}{l} \langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle\ [\,] \qquad\quad = ([\,],\ \ \mathsf{C}_1) \\ \langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle\ ((a,b) : x) = (a : y,\ \mathsf{C}_2\ b\ z) \\ \qquad \textbf{where}\ (y,z) = \langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle\ x \end{array}$$

Tupling of the forward function and its derived complement function is always possible, because they have the same recursion structure, by construction. The formal transformation follows the approach developed by Hu et al. (1997). Note that tupling preserves totality, because also the domain of a derived complement function is always the same as that of the forward transformation.

Then, the method derives the partial inverse of the tupled function, basically by exchanging the roles of left- and right-hand sides in function definitions (and adjusting recursive calls). In the specific example, we obtain:

$$\begin{array}{l} \langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle^{-1}\ ([\,],\ \ \mathsf{C}_1) \qquad = [\,] \\ \langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle^{-1}\ (a : y,\ \mathsf{C}_2\ b\ z) = (a,b) : x \\ \qquad \textbf{where}\ x = \langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle^{-1}\ (y,z) \end{array}$$

Note that $\langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle^{-1}$ is not defined for all elements of its potential domain $\mathsf{Range}(\texttt{mapfst}) \times \mathsf{Range}(\texttt{mapfst}^{\text{res}})$, because, as already observed earlier, the tupled function $\langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle$ is not surjective onto that set. As a consequence of the partiality of $\langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle^{-1}$, the `put` function obtained from equation UPD:

$$\texttt{put}_{(\texttt{mapfst},\texttt{mapfst}^{\text{res}})}\ v\ s = \langle \texttt{mapfst}, \texttt{mapfst}^{\text{res}} \rangle^{-1}\ (v, \texttt{mapfst}^{\text{res}}\ s)$$

is only partial. For example,

$$\texttt{put}_{(\texttt{mapfst},\texttt{mapfst}^{\text{res}})}\ [3]\ [\,] = \bot$$

To more clearly see what the derived backward transformation function actually is, and in general to make it more efficient by eliminating intermediate results, we can apply the fusion/deforestation transformation of Wadler (1990). In the example, this leads to the following definition, where we rename $\texttt{put}_{(\texttt{mapfst},\texttt{mapfst}^{\text{res}})}$ to $\texttt{mapfst}_{\text{B}}$:

$$\begin{array}{l} \texttt{mapfst}_{\text{B}}\ [\,] \qquad [\,] \qquad\quad = [\,] \\ \texttt{mapfst}_{\text{B}}\ (a : y)\ ((\_,b) : x) = (a,b) : (\texttt{mapfst}_{\text{B}}\ y\ x) \end{array}$$

That is, $\texttt{mapfst}_{\text{B}}$ is a function accepting (being defined for) a new view v and the original source s precisely when they are of same length, then returning a new source s' obtained from s by replacing the first component of each pair with the item from v at the corresponding list position. The call $\texttt{mapfst}_{\text{B}}\ v\ s$ fails if

the lengths of v and s differ! For example, let s be $[(1,\mathsf{A}),(2,\mathsf{B})]$. We have:[5]

$$\begin{array}{ll}
\texttt{mapfst}\ s = [1,2] & \\
\texttt{mapfst}_{\mathrm{B}}\ [11,22] & s = [(11,\mathsf{A}),(22,\mathsf{B})] \\
\texttt{mapfst}_{\mathrm{B}}\ [11] & s = \bot \\
\texttt{mapfst}_{\mathrm{B}}\ [11,22,33] & s = \bot
\end{array}$$

One issue that is not visible from the above example is that syntactic inversion is not always so easy. For $\langle \texttt{mapfst}, \texttt{mapfst}^{\mathrm{res}} \rangle$, exchanging the left- and right-hand sides led to a program with non-overlapping patterns on the (new) left-hand sides, and thus to deterministic branching. In general, though, once we apply some of the optimizations discussed in the next subsection to make the complement smaller, the syntactically inverted program $\langle \texttt{get}, \texttt{get}^{\mathrm{res}} \rangle^{-1}$ can require a full non-deterministic search to find, for a given pair (v, c), the unique (if any) s' with $\langle \texttt{get}, \texttt{get}^{\mathrm{res}} \rangle\ s' = (v, c)$.

As already mentioned, *put* functions obtained by the syntactic bidirectionalization method are non-total in general. Thus, it is important to provide a way for users to know when *put* v s succeeds. To tackle this problem, Matsuda et al. (2007) generate, given an initial source s_0, an *update checker* represented by a tree automaton (Comon et al. 2007) that can check for a given v whether *put* v s_0 will succeed, before and independent of actually executing the call to *put*. The law PARTIAL-PUTPUT guarantees that this tree automaton is invariant under successive application of *put*, and thus reusable through backward transformations.

4.4 Optimizing Complement Functions to be Small

Sometimes the complement functions obtained as in Section 4.2 are too large (with respect to the preorder from Definition 2) to be useful—the backward transformations obtained from them are defined for only a narrow range of arguments. This subsection presents syntactic techniques for obtaining smaller complement functions.

Removing Constructors. As an example, consider the function `zip`, which transforms a pair of lists into a list of pairs, and its derived complement $\texttt{zip}^{\mathrm{res}}$, given as:

$$\begin{array}{lll}
\texttt{zip}\ ([\,],\ \ y) & = [\,] & \texttt{zip}^{\mathrm{res}}\ ([\,],\ \ y) = \mathsf{C}_1\ y \\
\texttt{zip}\ (a:x,\ [\,]) & = [\,] & \texttt{zip}^{\mathrm{res}}\ (a:x,\ [\,]) = \mathsf{C}_2\ a\ x \\
\texttt{zip}\ (a:x,\ b:y) & = (a,b) : (\texttt{zip}\ (x,y)) & \texttt{zip}^{\mathrm{res}}\ (a:x,\ b:y) = \mathsf{C}_3\ (\texttt{zip}^{\mathrm{res}}\ (x,y))
\end{array}$$

[5] Note that even though we use Haskell syntax, we assume a strict functional language here. That is, we do not consider partially defined lists: if one item or tail is undefined, the whole list is (as opposed to Section 2, where we considered $[a, \bot]$ to be different from $\bot$).

Because the tupled function $\langle \texttt{zip}, \texttt{zip}^{\text{res}} \rangle$ is not a surjective function onto the product $\mathsf{Range}(\texttt{zip}) \times \mathsf{Range}(\texttt{zip}^{\text{res}})$, the backward transformation that is derived, namely $\texttt{zip}_{\text{B}} = \texttt{put}_{(\texttt{zip},\texttt{zip}^{\text{res}})}$, is partial: it rejects any view update that changes the length of the view. For example, let s be $([1,2,0],[\mathsf{A},\mathsf{B}])$. We have:

$$
\begin{array}{ll}
\texttt{zip}\ s = [(1,\mathsf{A}),(2,\mathsf{B})] & \\
\texttt{zip}_{\text{B}}\ [(11,\mathsf{D}),(22,\mathsf{E})] & s = ([11,22,0],\ [\mathsf{D},\mathsf{E}]) \\
\texttt{zip}_{\text{B}}\ [(11,\mathsf{D})] & s = \bot \\
\texttt{zip}_{\text{B}}\ [(11,\mathsf{D}),(22,\mathsf{E}),(33,\mathsf{F})] & s = \bot
\end{array}
$$

Performing *range analysis*, which approximates the set of results an expression can possibly evaluate to, sometimes helps us to obtain a smaller complement. For example, we can observe that the possible evaluation results of the right-hand side expression $(a,b) : (\texttt{zip}\ (x,y))$ of the third branch in the definition of `zip` above do not overlap those of the first and second branches. Thus, we do not need to use C_3 in the third branch of the complement function, because even without it the tupled function $\langle \texttt{zip}, \texttt{zip}^{\text{res}} \rangle$ would be injective. If we do remove it, thus creating a complement function that is smaller with respect to $\preceq$ than the one above, $\langle \texttt{zip}, \texttt{zip}^{\text{res}} \rangle$ becomes surjective onto $\mathsf{Range}(\texttt{zip}) \times \mathsf{Range}(\texttt{zip}^{\text{res}})$ and we obtain a new, now total, backward transformation $\texttt{zip}_{\text{B}} = \texttt{put}_{(\texttt{zip},\texttt{zip}^{\text{res}})}$ equivalent to the following definition:

$$
\begin{array}{l}
\texttt{zip}_{\text{B}}\ v\ (x,y) = (s +\!\!+ r, t +\!\!+ u) \\
\quad \textbf{where}\ (s,t) = \texttt{unzip}\ v \\
\qquad\qquad r \quad = \texttt{drop}\ m\ x \\
\qquad\qquad u \quad = \texttt{drop}\ m\ y \\
\qquad\qquad m \quad = \texttt{min}\ (\texttt{length}\ x)\ (\texttt{length}\ y)
\end{array}
$$

For example, let s be $([1,2,0],[\mathsf{A},\mathsf{B}])$ again. We now have:

$$
\begin{array}{lll}
\texttt{zip}\ s = [(1,\mathsf{A}),(2,\mathsf{B})] & & \\
\texttt{zip}_{\text{B}}\ [(11,\mathsf{D}),(22,\mathsf{E})] & s = ([11,22,0], & [\mathsf{D},\mathsf{E}]) \\
\texttt{zip}_{\text{B}}\ [(11,\mathsf{D})] & s = ([11,0], & [\mathsf{D}]) \\
\texttt{zip}_{\text{B}}\ [(11,\mathsf{D}),(22,\mathsf{E}),(33,\mathsf{F})] & s = ([11,22,33,0], & [\mathsf{D},\mathsf{E},\mathsf{F}])
\end{array}
$$

and similarly for $s = ([1,2],[\mathsf{A},\mathsf{B},\mathsf{Z}])$. Such behavior of $\texttt{zip}_{\text{B}}$ would probably be the expected intuitive one to users.

Matsuda et al. (2007) use tree automata (Comon et al. 2007) to analyze the ranges of functions. In fact, due to the restrictions imposed on the functional language, the ranges of functions can be described in exact form this way. Moreover, a similar approach enables one to check whether a function is injective or not in a sound and complete way; thus, the method can derive a constant function as the complement for an injective function.

The injectivity analysis also enables us to remove calls of the corresponding complement function for an injective function; they do not contribute to the injectivity of the tupled functions. In addition, removing complement-function calls sometimes creates more opportunities for applying the constructor removal method (which only removes singleton constructors) discussed above.

Unifying Constructors. As another example, consider the function `even`, which checks whether a given natural number is even, and its derived complement function $\texttt{even}^{\text{res}}$:

$$\begin{array}{lll} \texttt{even}\ \mathsf{Z} & = \mathsf{True} & \texttt{even}^{\text{res}}\ \mathsf{Z} & = \mathsf{C}_1 \\ \texttt{even}\ (\mathsf{S}\ \mathsf{Z}) & = \mathsf{False} & \texttt{even}^{\text{res}}\ (\mathsf{S}\ \mathsf{Z}) & = \mathsf{C}_2 \\ \texttt{even}\ (\mathsf{S}\ (\mathsf{S}\ x)) & = \texttt{even}\ x & \texttt{even}^{\text{res}}\ (\mathsf{S}\ (\mathsf{S}\ x)) & = \mathsf{C}_3\ (\texttt{even}^{\text{res}}\ x) \end{array}$$

Since $\texttt{even}^{\text{res}}$ (not `even`!) is an injective function, no update on a view can be propagated back to the source by the backward transformation $\texttt{put}_{(\texttt{even},\texttt{even}^{\text{res}})}$ obtained from the above. Moreover, it is not possible here to remove the constructor C_3 in the third branch of the complement function, because then the tupled function $\langle \texttt{even}, \texttt{even}^{\text{res}} \rangle$ would not be injective anymore. However, since the return values of the first and the second branch of `even` differ, one does not actually need different constructors C_1 and C_2 in the complement function. Even if we replace the two by a single constructor, the tupled function $\langle \texttt{even}, \texttt{even}^{\text{res}} \rangle$ remains injective. Indeed, the following function is also a complement function for `even` and smaller with respect to $\preceq$ than the above one:

$$\begin{array}{ll} \texttt{even}^{\text{res}}\ \mathsf{Z} & = \mathsf{C}_1 \\ \texttt{even}^{\text{res}}\ (\mathsf{S}\ \mathsf{Z}) & = \mathsf{C}_1 \\ \texttt{even}^{\text{res}}\ (\mathsf{S}\ (\mathsf{S}\ x)) & = \mathsf{C}_3\ (\texttt{even}^{\text{res}}\ x) \end{array}$$

Intuitively, this new definition of $\texttt{even}^{\text{res}}$ computes $\lfloor x/2 \rfloor$ for a given natural number x. Now that the tupled function $\langle \texttt{even}, \texttt{even}^{\text{res}} \rangle$ has become surjective onto $\mathsf{Range}(\texttt{even}) \times \mathsf{Range}(\texttt{even}^{\text{res}})$, the corresponding backward transformation $\texttt{even}_{\text{B}} = \texttt{put}_{(\texttt{even},\texttt{even}^{\text{res}})}$ is total and is able to propagate any view changes to source changes.

The formal way to soundly unify constructors again relies on the range analysis mentioned earlier. Criteria for making complement functions smaller by removing constructors, unifying constructors, and exploiting injectivity analysis, are incorporated into an algorithm by Matsuda et al. (2007).

4.5 Summary

Syntactic bidirectionalization (Matsuda et al. 2007) directly follows the constant-complement approach to bidirectionalization. From a given definition of a forward transformation function, the method generates the definition of a complement function and then constructs the backward transformation function based on equation Upd. Sometimes range analysis and injectivity analysis help to obtain smaller complement functions (as shown for the `zip` and `even` examples).

5 Semantic Bidirectionalization

This section presents a semantic bidirectionalization technique. The idea is to define a higher-order function that takes the forward transformation as an argument and produces a suitable backward transformation as a result. This function

invokes the forward function as a subroutine but does not (indeed, cannot) otherwise inspect it. Since there is no dependence on the syntactic definition of the forward function whatsoever, and it is only used as a semantic entity, the technique can be used with functions that have already been compiled or whose source is otherwise not available.

The way this is done depends crucially on having suitable abstraction mechanisms available in the functional language at hand. In particular, we will stipulate that the forward transformation must be a *polymorphic* function, because this will allow us to learn something about its behavior without having access to its defining equations.

5.1 Leveraging Polymorphism

The technical mechanism we use exploits "free theorems" (Wadler 1989)—formal statements about the behavior of functions that do not depend on their definitions, just their types. For example, assume that we are given a function $\texttt{get} :: \textbf{forall}\ \alpha.\ [\alpha] \to [\alpha]$. Since it is polymorphic, there are certain restrictions on what the function can do. In particular, it cannot manufacture new list items or manipulate the existing ones. Essentially, the function can only drop, move around, or duplicate items from the input list to produce the output list. That still leaves considerable room for the function's behavior, but some aspects are fixed, for example that the length of the output list only depends on the length of the input list.

Wadler's free theorems are a way to make explicit such constraints on the behavior of functions imposed by their (polymorphic) type. For the above type of `get`, a free theorem states that for any list l and (type-appropriate) function h, we have

$$\texttt{get}\ (\texttt{map}\ h\ l) = \texttt{map}\ h\ (\texttt{get}\ l) \tag{1}$$

where

$$\begin{aligned}
&\texttt{map} :: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ (\alpha \to \beta) \to [\alpha] \to [\beta] \\
&\texttt{map}\ h\ [\,] \quad\ = [\,] \\
&\texttt{map}\ h\ (a : as) = (h\ a) : (\texttt{map}\ h\ as)
\end{aligned}$$

This implies that the behavior of `get` must not depend on the values of the list items, but only on positional information. This positional information can even be observed explicitly, for example by applying `get` to ascending lists over integer values. Say `get` is `tail`, then every list $[0..n]$ is mapped to $[1..n]$, which allows us to see (without inspecting the syntactic definition of `tail`, or its suggestive name) that the head item of the original source list is absent from the view, hence cannot be affected by an update on the view, and hence should remain unchanged when propagating an updated view back into the source. Even more important, this observation can be transferred to other source lists than $[0..n]$ just as well, thanks to statement (1) above. In particular, that statement allows us to establish that for every list s of the same length as $[0..n]$, but over arbitrary

type, we have

$$\texttt{get}\ s = \texttt{get}\ (\texttt{map}\ (s\,!!)\ [0..n]) = \texttt{map}\ (s\,!!)\ (\texttt{get}\ [0..n]) \tag{2}$$

where (!!) :: **forall** α. $[\alpha] \to \mathsf{Int} \to \alpha$ is the Haskell operator for extracting a list item at a given index position, starting from 0.

Statement (2) means that the behavior of `get` is fully determined by its behavior on initial segments of the naturals (or, if we want, by its behavior on finite lists of distinct items). Now we "only" need to make good use of that observation to provide an appropriate backward transformation `put`. We do not insist on totality, but instead aim for a `get`/`put` pair that constitutes a partial very well-behaved lens. The original paper by Voigtländer (2009) gives a direct construction of the `put` function. Here we instead lay out its construction in terms of complements. We consider first the case of lists as input and output. Section 5.4 describes a generic extension that allows the technique to be used with other structures besides lists.

5.2 Using the Constant-Complement Approach

Assume a fixed `get` :: **forall** α. $[\alpha] \to [\alpha]$. What should a complement function `res` look like, so that the tupled function $\langle \texttt{get}, \texttt{res} \rangle$ becomes injective? Clearly, `res` needs to record all the information (about the input list) that is discarded by `get`. Natural candidates are the input list length and the positions and values in it that get discarded. For example, if `get` = `tail`, then `res` may record the input list length as well as that the first item is missing from the view and what its value was. Using statement (2), we can learn such information about which items are missing in the view, for a concrete source s, without inspecting the definition of `get`. Namely, we can apply `get` to the list $[0..n]$ of same length as s, and observe which of the values $0, \ldots, n$ are missing from the result. If we count from 1 instead of from 0, this idea leads to the following implementation,

```
res :: forall α. [α] → (Int, IntMap α)
res s = let n  = length s
            t  = [1..n]
            g  = IntMap.fromDistinctAscList (zip t s)
            g' = foldr IntMap.delete g (get t)
        in  (n, g')
```

which uses some Haskell functions from the standard Prelude and from the Data.IntMap module. Figure 1 gives the names and the type signatures for those from Data.IntMap, as well as some other functions from the same module that will be used later.

Next, we need a (partial) function `inv` such that for every type τ, source $s :: [\tau]$, view $v :: [\tau]$, and complement $c :: (\mathsf{Int}, \mathsf{IntMap}\ \tau)$, the laws LeftInv and Partial-RightInv hold. It is tempting to write something like (using the

```
fromDistinctAscList :: forall α. [(Int, α)] → IntMap α
empty               :: forall α. IntMap α
insert              :: forall α. Int → α → IntMap α → IntMap α
delete              :: forall α. Int → IntMap α → IntMap α
union               :: forall α. IntMap α → IntMap α → IntMap α
lookup              :: forall α. Int → IntMap α → Maybe α
keys                :: forall α. IntMap α → [Int]
elems               :: forall α. IntMap α → [α]
```

Fig. 1. Some functions from module Data.IntMap

`fromJust` function from the Data.Maybe module):

```
inv :: forall α. ([α], (Int, IntMap α)) → [α]
inv (v, (n, g')) = let t  = [1..n]
                       h  = fromList (zip (get t) v)
                       h' = IntMap.union h g'
                   in  map (λi → fromJust (IntMap.lookup i h')) t

fromList :: forall α. [(Int, α)] → IntMap α
fromList = foldl (λm (i, b) → IntMap.insert i b m) IntMap.empty
```

For `get = tail` and the case that `inv` is called with a list v of length $n-1$, with n, and with g' representing a finite mapping with exactly $\{1\}$ as domain, h will associate the "indices" $2, \ldots, n$ with the first, second, and so on, item of v, and so the overall result will be the value stored for index 1 in g' followed by the whole of v. So far, so good for this specific example. But in general, we have to be careful, because:

1. The function `inv` may be called with arguments v and n where `get` $[1..n]$ and v are *not* lists of the same length. In this case we would also have that `get` (`map` $(\lambda i \to \cdots)$ $[1..n]$) and v are lists of different lengths, due to statement (1), which contradicts the requirement derived from law PARTIAL-RIGHTINV that

$$\frac{(\texttt{inv}\ (v, (n, g')))\downarrow}{\texttt{get}\ (\texttt{inv}\ (v, (n, g'))) = v}\ (\textsc{Partial-RightInv-Get})$$

2. The function `inv` may be called with arguments v and n such that `get` $[1..n]$ contains duplicate items at positions where the corresponding items of v do not agree. In this case, only one of these two items of v would be associated with such an index (that occurred twice in `get` $[1..n]$) in h, and hence would be used for the thus indexed position of the overall result of the call to `inv`, which in turn would again cause `get` (`inv` $(v, (n, g'))$) to differ from v.
3. The function `inv` may be called with arguments n and g' such that the domain of g' contains integers other than those of $1, \ldots, n$ not occurring in

`get` $[1..n]$, which would lead to a contradiction to the requirement derived from law PARTIAL-RIGHTINV that

$$\frac{(\texttt{inv}\ (v,(n,g')))\downarrow}{\texttt{res}\ (\texttt{inv}\ (v,(n,g'))) = (n,g')}\ (\textsc{Partial-RightInv-Res})$$

To alleviate all these problems, we implement (using (\\) from the Data.List module and `guard` and `foldM` from the Control.Monad module):

```
inv :: forall α. Eq α ⇒ ([α], (Int, IntMap α)) → [α]
inv (v, (n, g')) = fromJust (do let t = [1..n]
                                let t' = get t
                                guard (length t' == length v)
                                h ← assoc (zip t' v)
                                guard (null (IntMap.keys g' \\ (t \\ t')))
                                let h' = IntMap.union h g'
                                mapM (λi → IntMap.lookup i h') t)


assoc :: forall α. Eq α ⇒ [(Int, α)] → Maybe (IntMap α)
assoc = foldM (λm (i, b) → checkInsert i b m) IntMap.empty

checkInsert :: forall α. Eq α ⇒ Int → α → IntMap α → Maybe (IntMap α)
checkInsert i b m = case IntMap.lookup i m of
                      Nothing → Just (IntMap.insert i b m)
                      Just c  → if (b == c) then Just m else Nothing
```

Note that we use monadic error handling, and in particular the two calls to `guard` to prevent the first and third potential problems mentioned in the list above. The second potential problem is prevented by replacing the simple call to `fromList` in the previous definition of `inv` with a possibly failing call to `assoc`, which checks that if there *are* duplicates in `get` $[1..n]$ then the corresponding items of v *do* agree, at least up to programmed (though not necessarily, semantic) equivalence ==. This use of == leads to a slightly different type of `inv` than before, namely a type class constraint Eq has to be added. Finally, note that the last line, `mapM` $(\lambda i \to$ `IntMap.lookup` $i\ h')\ t$, can also lead to a failure, namely if one of $1,\ldots,n$ occurs neither in `get` $[1..n]$ nor in the domain of g'.

Now, using statement (2), actually its variant for lists starting from 1, we can prove that law LEFTINV holds for `inv`, `get`, and `res`, but instead of law PARTIAL-RIGHTINV we can only prove a slightly weaker variant in that PARTIAL-RIGHTINV-RES *does* hold, but instead of `get` (`inv` $(v,(n,g'))) = v$ in PARTIAL-RIGHTINV-GET only

$$\frac{(\texttt{inv}\ (v,(n,g')))\downarrow}{(\texttt{get}\ (\texttt{inv}\ (v,(n,g'))) == v) = \mathsf{True}}$$

holds. We will henceforth abbreviate statements $(x == y) = \mathsf{True}$ to $x == y$, but keep the distinction between $==$ and $=$. We do assume, however, that every instance of Eq defines an $==$ that is reflexive, symmetric, and transitive.

Using the facts we already have, we can prove that `get` and

$$\begin{aligned}&\texttt{put} :: \textbf{forall}\ \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha]\\&\texttt{put}\ v\ s = \texttt{inv}\ (v, \texttt{res}\ s)\end{aligned}$$

constitute a partial very well-behaved lens, except that we have to replace law PARTIAL-PUTGET by the following slightly weaker variant:[6]

$$\frac{(put\ v\ s)\!\downarrow}{get\ (put\ v\ s) == v} \qquad \text{(PARTIAL-EQ-PUTGET)}$$

In terms of providing a suitable backward function we are done.[7] It is interesting, though, to inline the definitions of `inv` and `res` into that of `put`, because it allows some optimization as well as connecting to the formulation (not based on constant complements) given by Voigtländer (2009). We obtain:

```
put :: forall α. Eq α ⇒ [α] → [α] → [α]
put v s = fromJust (do let n = length s
                       let t = [1..n]
                       let t' = get t
                       let g = IntMap.fromDistinctAscList (zip t s)
                       let g' = foldr IntMap.delete g t'
                       guard (length t' == length v)
                       h ← assoc (zip t' v)
                       guard (null (IntMap.keys g' \\ (t \\ t')))
                       let h' = IntMap.union h g'
                       mapM (λi → IntMap.lookup i h') t)
```

Given this, we can observe that:

- the second call to `guard` is superfluous, because in the context in which it now appears it is guaranteed that the domain of g' consists exactly of those integers of $1, \ldots, n$ that to not occur in `get` $[1..n]$;
- no failure can happen in the line computing `mapM` $(\lambda i \to \ldots)\ t$, because every element of $[1..n]$ occurs in the domain of (exactly) one of h and g', and thus in the domain of h';

[6] For the typical instances of Eq used in practice, $==$ and $=$ totally agree, so the difference would be immaterial. Note also that Voigtländer (2009) assumed that also GETPUT and PARTIAL-PUTPUT need to be weakened to use $==$ instead of $=$, which was overly pessimistic.

[7] For example, if `get` is the function from the beginning of Section 2, then the above definition of `put` behaves exactly like the second implementation of `put` given in that earlier section.

– indeed, the domain of h' is exactly $\{1, \ldots, n\}$, so instead of looking up the elements $[1..n]$, in this order, we might as well simply return all elements of the map in the ascending order of their keys.

Hence, we can simplify as follows, while at the same time abstracting from a fixed `get` to a variable one, thus providing the higher-order function alluded to earlier (named for an abbreviation of "Bidirectionalization for Free"):

```
bff :: (forall α. [α] → [α]) → (forall α. Eq α ⇒ [α] → [α] → [α])
bff get v s = fromJust (do let t = [1..(length s)]
                           let t' = get t
                           let g = IntMap.fromDistinctAscList (zip t s)
                           let g' = foldr IntMap.delete g t'
                           guard (length t' == length v)
                           h ← assoc (zip t' v)
                           let h' = IntMap.union h g'
                           Just (IntMap.elems h'))
```

This version conceptually differs from the one originally published (Voigtländer 2009) in no essential way, except for the role of g', which is avoided in the original version (building h' directly as the union of h and g, which makes a potential difference only in terms of efficiency, either way, but not in terms of semantics).

The original paper by Voigtländer (2009) does not stop there—it goes on to develop semantic bidirectionalization for other functions besides fully polymorphic functions on lists. The rest of this section reviews these generalizations.

5.3 Generalizing

One dimension of generalization is to consider functions that are not *fully* polymorphic, but may actually perform *some* operations on list items. For example, the following function uses equality (or rather inequality) tests to remove duplicate list items:

```
get :: forall α. Eq α ⇒ [α] → [α]
get []       = []
get (a : as) = a : (get (filter (a /=) as))
```

Unfortunately, this function is not handled by the semantic bidirectionalization strategy described thus far. It cannot be given the type **forall** α. $[\alpha] \to [\alpha]$, and indeed the essential statement (2) does not hold for it.[8] By working with refined free theorems (Wadler 1989, Section 3.4) it is possible to treat `get`-functions of type **forall** α. Eq $\alpha \Rightarrow [\alpha] \to [\alpha]$ as well, to implement a higher-order function

```
bff_Eq :: (forall α. Eq α ⇒ [α] → [α]) → (forall α. Eq α ⇒ [α] → [α] → [α])
```

[8] Consider $s =$ "abcbabcbaccba" and $n = 12$. Then on the one hand, `get` $s =$ "abc", but on the other hand, `map` $(s\,!!)$ (`get` $[0..n]$) = `map` $(s\,!!)$ $[0..n] = s$.

and to prove that every pair $\texttt{get} :: \textbf{forall}\ \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ and $\texttt{put} = \texttt{bff}_{\mathsf{Eq}}\ \texttt{get}$ satisfies the laws PARTIAL-PUTPUT and PARTIAL-EQ-PUTGET and the following variant of the law GETPUT:[9]

$$put\ (get\ s)\ s == s$$

The same goes for the type class Ord capturing ordering tests (assuming that the provided $<$ is transitive, $x < y$ implies $x \mathrel{/}= y$, and $x \mathrel{/}= y$ implies $x < y$ or $y < x$), a new higher-order function

$$\texttt{bff}_{\mathsf{Ord}} :: (\textbf{forall}\ \alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]) \to (\textbf{forall}\ \alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])$$

and forward transformations like the following one:

$$\begin{aligned}&\texttt{get} :: \textbf{forall}\ \alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]\\ &\texttt{get} = (\texttt{take}\ 3) \circ \mathsf{List}.\texttt{sort}\end{aligned}$$

Another dimension of generalization is to consider functions that deal with data structures other than lists. By employing polymorphism over type constructor classes and type-generic programming techniques, Voigtländer (2009) provides one implementation of each $\texttt{bff}$, $\texttt{bff}_{\mathsf{Eq}}$, and $\texttt{bff}_{\mathsf{Ord}}$ that applies to functions involving a wide range of type constructors, on both the source and the view sides. For example, the very same $\texttt{bff}$ can be used to bidirectionalize the $\texttt{get}$-function shown at the beginning of Section 2 as well as the following function:

$$\begin{aligned}&\texttt{flatten} :: \textbf{forall}\ \alpha.\ \mathsf{Tree}\ \alpha \to [\alpha]\\ &\texttt{flatten}\ (\mathsf{Leaf}\ a) \quad\ = [a]\\ &\texttt{flatten}\ (\mathsf{Node}\ t_1\ t_2) = (\texttt{flatten}\ t_1) +\!\!+ (\texttt{flatten}\ t_2)\end{aligned}$$

where

$$\textbf{data}\ \mathsf{Tree}\ \alpha = \mathsf{Node}\ (\mathsf{Tree}\ \alpha)\ (\mathsf{Tree}\ \alpha) \mid \mathsf{Leaf}\ \alpha$$

In the next subsection we give a somewhat more streamlined account of data-type genericity for $\texttt{bff}$ than originally provided by Voigtländer (2009). The main benefit is that the new version uses only standard type constructor classes, rather than a specifically introduced new one. As a consequence, the generic $\texttt{bff}$ is now much more readily applicable to new data types, because no instance definitions have to be implemented by hand—the Glasgow Haskell Compiler can automatically derive them.

5.4 Going Generic via Container Representations

Instead of bidirectionalizing functions of type $\textbf{forall}\ \alpha.\ [\alpha] \to [\alpha]$, we now want to more generally treat functions of type $\textbf{forall}\ \alpha.\ \theta\ \alpha \to \theta'\ \alpha$ for some type

[9] Again, Voigtländer (2009) actually assumed that also PARTIAL-PUTPUT needs to be weakened to use $==$ instead of $=$, which is not necessary. But for GETPUT it is indeed necessary in the case of $\texttt{bff}_{\mathsf{Eq}}$ (and $\texttt{bff}_{\mathsf{Ord}}$ below).

constructors θ and θ'. In fact, we want `bff` to be polymorphic over those type constructors. Clearly, the operations we previously performed on lists now need to be somehow generalized to other data types. For example, we previously compared lists by their lengths, but now we have to consider more complex shapes. Also, we previously manufactured a "template" $[1..n]$ for every source list s of length n, and now need to do something similar for fairly arbitrary tree structures. Our strategy here is to reuse as much as possible of `bff`'s operation on lists, by first separating other data structures into their shape and content aspects, much like the *shape calculus* (Jay 1995) and *container representations* (Abbott et al. 2003) do. In fact, we can largely follow a generic programming account of these ideas due to Gibbons and Oliveira (2009) here.

The general idea of container representations is to explicitly represent, for a given type constructor, a type of underlying shapes:

$$\textbf{type}\ \mathsf{Shape}\ \kappa = \ldots$$

as well as a type of associated positions:

$$\textbf{type}\ \mathsf{Pos}\ \kappa = \ldots$$

and to provide functions (potentially with dependent types actually more precise than those given here):

$$\begin{array}{ll}
\texttt{positions} & :: \mathsf{Shape}\ \kappa \to \mathsf{Set}\ (\mathsf{Pos}\ \kappa) \\
\texttt{shape} & :: \textbf{forall}\ \alpha.\ \kappa\ \alpha \to \mathsf{Shape}\ \kappa \\
\texttt{content} & :: \textbf{forall}\ \alpha.\ \kappa\ \alpha \to (\mathsf{Pos}\ \kappa \to \alpha) \\
\texttt{fill} & :: \textbf{forall}\ \alpha.\ (\mathsf{Shape}\ \kappa, \mathsf{Pos}\ \kappa \to \alpha) \to \kappa\ \alpha
\end{array}$$

connected by some natural laws. If one agrees to always represent positions by natural numbers and to use as set of positions for a given shape always a prefix of the natural numbers, one can replace `positions` by a function

$$\texttt{arity} :: \mathsf{Shape}\ \kappa \to \mathsf{Int}$$

and replace $\mathsf{Pos}\ \kappa \to \alpha$ by $[\alpha]$ in the types of `content` and `fill`:

$$\begin{array}{ll}
\texttt{content} & :: \textbf{forall}\ \alpha.\ \kappa\ \alpha \to [\alpha] \\
\texttt{fill} & :: \textbf{forall}\ \alpha.\ (\mathsf{Shape}\ \kappa, [\alpha]) \to \kappa\ \alpha
\end{array}$$

The natural laws mentioned above then become

$$\texttt{arity}\ (\texttt{shape}\ x) = \texttt{length}\ (\texttt{content}\ x) \tag{3}$$

and

$$\texttt{fill}\ (\texttt{shape}\ x, \texttt{content}\ x) = x \tag{4}$$

in this formulation.[10]

[10] Section 6 also employs this formulation.

With

$$\textbf{type}\ \mathsf{Shape}\ \kappa = \kappa\ ()$$

and requiring κ to be an instance of the type constructor class Data.Traversable, Gibbons and Oliveira (2009) give generic implementations of essentially the latter two functions `content` and `fill` under the names `runContents` and `runReassemble`, respectively. For our purposes, it is convenient to replace the second of the two by a function that does not necessarily take a $\mathsf{Shape}\ \kappa = \kappa\ ()$ as first argument, but actually *any* κ-structure. Moreover, we make this new function `decorate` curried and flip its arguments. Specifically, we assume given three functions

$$\begin{aligned}
&\texttt{shape} &&:: \textbf{forall}\ \alpha.\ \kappa\ \alpha \to \kappa\ ()\\
&\texttt{content} &&:: \textbf{forall}\ \alpha.\ \kappa\ \alpha \to [\alpha]\\
&\texttt{decorate} &&:: \textbf{forall}\ \alpha.\ \textbf{forall}\ \beta.\ [\alpha] \to \kappa\ \beta \to \kappa\ \alpha
\end{aligned}$$

(eventually each implemented by relying on a "$\mathsf{Traversable}\ \kappa \Rightarrow$" context), which satisfy the laws

$$\texttt{decorate}\ (\texttt{content}\ x)\ x = x \tag{5}$$

and

$$\texttt{decorate}\ y\ x = \texttt{decorate}\ y\ (\texttt{shape}\ x) \tag{6}$$

Note that (5) is GetPut for $get = \texttt{content}$ and $put = \texttt{decorate}$. Reasonable definitions will also satisfy the corresponding Partial-PutGet, namely that if $\texttt{decorate}\ y\ x$ is defined, then $\texttt{content}\ (\texttt{decorate}\ y\ x) = y$. Together with the above and the types, Partial-PutPut follows as well. In fact, reasonable definitions of the above three functions let `content`/`decorate` be a partial very well-behaved lens, with `shape` in the role of the chosen complement function *res* for $get = \texttt{content}$, à la Section 3! However, we will only rely on the statements (5) and (6) below.

Suitable implementations of `shape`, `content`, and `decorate` are given in Appendix A, and can be used to make the function `bff` from Section 5.2 more generic as follows:

```
bff :: (Traversable κ, Traversable κ', Eq (κ' ())) ⇒
       (forall α. κ α → κ' α) → (forall α. Eq α ⇒ κ' α → κ α → κ α)
bff get v s = fromJust (do let l = content s
                           let t = [1..(length l)]
                           let t' = get (decorate t s)
                           let g = IntMap.fromDistinctAscList (zip t l)
                           let l' = content t'
                           let g' = foldr IntMap.delete g l'
                           guard (shape t' == shape v)
                           h ← assoc (zip l' (content v))
                           let h' = IntMap.union h g'
                           Just (decorate (IntMap.elems h') s))
```

Instead of directly constructing a template $[1..n]$ from a list, we first "reduce" a more general data structure to its list of content items, construct a template from that, use it to redecorate the actual data structure, and work from there. On the view side, we again work with the separation into content and shape, in particular constructing g' from the *content* of the outcome of the subcall to *get*, and instead of comparing the lengths of lists, comparing the *shapes* of t' and v. In the end, instead of directly returning the elements of h', we use them to redecorate the actual source data structure once more, but now with (some) items updated according to the content of v. In essence, lists provide an interface here for enumerating, collecting, comparing, and replacing data items in a fairly arbitrary structure, and the functions `shape`/`content`/`decorate` are used to go back and forth between such arbitrary structures and lists.

We postulate that in order for the laws GetPut, Partial-Eq-PutGet, and Partial-PutPut to hold for any functions `get` :: **forall** α. Eq $\alpha \Rightarrow \theta\ \alpha \to \theta'\ \alpha$ and `put` = `bff get` (for θ, θ' satisfying the type (constructor) class constraints imposed in the type of `bff` above), it is enough to have (5) and (6) plus that for every $t, t' :: \theta'\ ()$,

$$t = t' \iff t == t'$$

and that in fact Eq-instances are always such that data structures with the same shape and ==-equivalent content are themselves ==-equivalent (a condition which could be formalized via `shape` and `content`). All these conditions can reasonably be expected to hold of the implementations from Appendix A, together with the Traversable- and Eq-instances a programmer would write (or that the compiler would derive automatically).

For $\texttt{bff}_{\mathsf{Eq}}$ and $\texttt{bff}_{\mathsf{Ord}}$, a similar development is possible, though not given here. It can be obtained by applying similar simplifications as above to their generic versions from the original paper (Voigtländer 2009).

5.5 Summary

Semantic bidirectionalization (Voigtländer 2009) exploits the abstraction mechanisms—in particular, polymorphic typing—of a higher-order functional language to implement a backward transformation function without inspecting the syntactic form of the forward transformation. The key idea is to use the forward transformation function as a subroutine "in simulation mode" to learn important information about its behavior, to be used in complement generation and tupled function inversion. Generic programming techniques allow the realization of this approach for a wide range of data types.

6 Bidirectional Combinators

This section describes an approach to building bidirectional transformations using domain-specific *bidirectional combinators*. Unlike the techniques developed in the preceding sections, which calculate a well-behaved *put* function from a

given *get* function (or the program describing it), the technique presented here allows programmers to describe a pair of *get* and *put* functions simultaneously.

Using combinators has several advantages over other approaches:

- They make it easy to develop type systems that guarantee strong behavioral properties, such as round-tripping laws and totality.
- They allow programmers to choose an appropriate *put* function for a given *get* function (unlike approaches such as bidirectionalization, which calculate a single *put* function for a particular *get* function).
- They are easy to extend with special constructs for dealing with issues such as alignment (Barbosa et al. 2010; Bohannon et al. 2008), ignorable information (Foster et al. 2008), and confidential data (Foster et al. 2009).

Of course, using combinators also has a significant disadvantage—it does not allow programmers to describe lenses using programs in existing languages. But often the syntax of the combinators can be designed to closely resemble familiar languages so that this is not a major burden. Boomerang, a bidirectional language for processing textual data, is based on combinators (Foster and Pierce 2009) as is Augeas, a language that extends Boomerang's core constructs with combinators for processing trees (Lutterkort 2008).

This section focuses on the special case of *matching lens combinators*, which are designed to deal with the problems that come up when ordered data are manipulated using bidirectional transformations. Lenses and their associated behavioral laws capture important conditions on the handling of data in the source and view. But they do not address an important issue that comes up in many practical applications: *alignment*. As we have seen, the *get* component of a lens may discard some of the information in the source. So to correctly propagate updates to the view, the *put* function needs to combine the pieces of the view with the corresponding pieces of the source (or complement). In particular, when the source and view are ordered (e.g., lists, strings, XML trees, etc.), doing this correctly requires re-aligning the pieces of each structure with each other. Unfortunately, the laws given in Section 2 do not include any properties involving alignment. Hence, they consider a *put* function that operates in the simplest possible way—by position—to be correct.

6.1 Alignment Problems

To illustrate the problems that come up when lenses that are used with ordered structures, consider an example where the source is a list,

$$s = [(\text{“Alice”}, \text{“Anchorage, AK”}), (\text{“Bob”}, \text{“Boston, MA”}),\\ (\text{“Carol”}, \text{“Chicago, IL”}), (\text{“Dave”}, \text{“Detroit, MI”})]$$

and the view is obtained by projecting the name from each source item (`mapfst` from Section 4):

$$v = [\text{“Alice”}, \text{“Bob”}, \text{“Carol”}, \text{“Dave”}]$$

If we modify the view by replacing "Dave" with "David", adding "Eve" to the beginning of the list, and deleting "Carol", we would like the *put* function to take the updated view,

$$v' = [\text{“Eve”}, \text{“Alice”}, \text{“Bob”}, \text{“David”}],$$

together with the complement computed from the original source,

$$c = [\text{“Anchorage, AK”}, \text{“Boston, MA”}, \text{“Chicago, IL”}, \text{“Detroit, MI”}],$$

and produce a new source that reflects all three updates,

$$\begin{aligned} s' = [&(\text{“Eve”}, \text{“”}), (\text{“Alice”}, \text{“Anchorage, AK”}),\\ &(\text{“Bob”}, \text{“Boston, MA”}), (\text{“David”}, \text{“Detroit, MI”})] \end{aligned}$$

using the empty string as the default city and state for "Eve", who was newly added.

Unfortunately, if the lens matches pieces of the view and complement by their absolute position in each list, this is not what will happen. Instead, the first name in the view will be matched up with the first city and state in the complement, the second name with the second city and state, and so on, yielding a mangled source,

$$\begin{aligned} s' = [&(\text{“Eve”}, \text{“Anchorage, AK”}), (\text{“Alice”}, \text{“Boston, MA”}),\\ &(\text{“Bob”}, \text{“Chicago, IL”}), (\text{“David”}, \text{“Detroit, MI”})] \end{aligned}$$

where the city and state for "Alice" have been restored to the pair for "Eve", the city and state for "Bob" to the pair for "Alice", and so on.

And yet, most existing bidirectional languages use this very strategy (Foster et al. 2007b; Matsuda et al. 2007; Voigtländer 2009). Although it works in some simple cases—*e.g.*, when the source and view are unordered, or when updates only modify items in-place—it fails dramatically in many others. Addressing this deficiency is the goal of the matching lenses presented in this section, which is based on papers by Bohannon et al. (2008) and Barbosa et al. (2010), but presented here with a cleaner and streamlined semantics.

6.2 Lenses with Complements

As a first step toward matching lenses, let us generalize the standard definition of well-behaved lenses as described in Section 2 by adding complements. Let S be a set of source structures, V a set of views, and C a set of complements. A *basic lens* on S, V, and C comprises three functions,

$$\begin{aligned} &\mathit{get} \in S \to V\\ &\mathit{res} \in S \to C\\ &\mathit{put} \in V \times \mathsf{Maybe}\ C \to S \end{aligned}$$

obeying the following laws for every s in S, v in V, c in C, and mc in $\mathsf{Maybe}\ C$:

$$\frac{get\ s = v \qquad res\ s = c}{put\ (v, \mathsf{Just}\ c) = s} \tag{GetPut}$$

$$\frac{put\ (v, mc) = s}{get\ s = v} \tag{PutGet}$$

We will write $S \overset{C}{\Longleftrightarrow} V$ for the set of all basic lenses on S, V, and C.

Note that the definition of basic lenses requires the *put* component to be a *total* function. Totality is a simple, powerful condition which ensures that basic lenses are capable of doing something reasonable with every view and every complement, even when the view has been modified significantly. Insisting that the *put* function be total is a strong constraint, especially in combination with the other lens laws imposed. In particular, totality is often in tension with the PutPut law:

$$\frac{}{put\ (v', \mathsf{Just}\ (res\ (put\ (v, mc)))) = put\ (v', mc)} \tag{PutPut}$$

For example, the total versions of the union and iteration operators (defined later in this section), which are needed in many practical examples, do not obey it. Therefore, in this section, we will not require that every lens obey PutPut.

Readers familiar with previously published descriptions of lenses may notice some minor differences (Bohannon et al. 2008; Foster et al. 2007b):

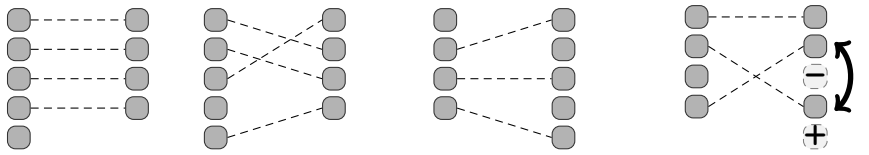
- The *put* function takes a complement rather than a source, and a new function *res* computes a complement from a source.
- The *put* component takes an *optional* value as its second argument, instead of having a separate *create* function of type $V \to S$ (see footnote 2). To map a view to a source, one can invoke *put* with Nothing.
- Finally, the *put* function has an uncurried type: $V \times \mathsf{Maybe}\ C \to S$ instead of $V \to \mathsf{Maybe}\ C \to S$. This simplifies several of the definitions that follow.

To see that these changes do not affect the semantics of lenses in any significant way, observe that, given a "classic" lens l, we can build a basic lens as follows:

$$\begin{array}{l} get\ s \quad\quad\ = l.get\ s \\ res\ s \quad\quad\ = s \\ put\ (v, ms) = \mathbf{case}\ ms\ \mathbf{of} \\ \qquad\qquad\qquad\quad \mathsf{Just}\ s \quad \to l.put\ v\ s \\ \qquad\qquad\qquad\quad \mathsf{Nothing} \to l.create\ v \end{array}$$

The notation $l.get$ refers to the *get* function of l. Similarly, given a basic lens l, we can build a classic lens as follows:

$$\begin{array}{l} get\ s \quad\ = l.get\ s \\ put\ v\ s \ = l.put\ (v, \mathsf{Just}\ (l.res\ s)) \\ create\ v = l.put\ (v, \mathsf{Nothing}) \end{array}$$



(a) positional (b) best match (c) best non-crossing (d) actual operations

Fig. 2. Alignment strategies.

6.3 Matching Lenses

Matching lenses address the alignment problems that arise in basic lenses by separating the two tasks performed by the *put* function: matching up pieces of the updated view with the corresponding pieces of the complement, and weaving the view and complement together to produce an updated source. To achieve this separation, they structure the complement as a pair consisting of a rigid component and a list component. This makes it easy to realign the complement after an update because the list can be used to supply the lens with explicit alignment information. Matching lenses also include additional behavioral laws that stipulate how items in the list component of the complement must be handled by the *put* function—*e.g.*, they require the *put* function to combine pieces of the view with the corresponding pieces of the complement.

The matching lens framework can be instantiated with arbitrary heuristic alignment strategies while still enjoying a simple and intuitive semantics. In practice, we often use matching lenses with a variety of different strategies, such as the heuristics depicted in Figure 2,

(a) simple positional alignment,
(b) "best match" alignment, which tries to match chunks without regard to ordering,
(c) a variant of best-match that only considers "non-crossing" matches, like the longest common subsequence heuristic used by `diff`, and
(d) edit-based alignment, which uses the actual edit operations performed by the user (if available) to calculate the intended alignment.

Boomerang (Foster and Pierce 2009), which implements matching lenses for textual data, supports a number of such alignment heuristics.

6.4 Structures with Chunks

Matching lenses assume that the source and view are made up of reorderable pieces, which we will call *chunks*. Formally, we model structures with chunks as containers, as defined in Section 5.4. To review, containers support the following functions,

– *shape*, which computes the shape of a structure with chunks,

- *content*, which computes the contents of a structure with chunks (represented concretely as a list),
- *arity*, which computes the arity of a shape, and
- *fill*, which computes the structure obtained by filling a shape with a given list of chunks.

We assume that these functions satisfy some natural laws, corresponding to (3) and (4) in Section 5.4. Many types—including pairs, sums, lists, trees, matrices, etc.—can be defined as containers satisfying these laws. In this section, we describe the container representations using standard datatypes, using a type constructor $\langle\cdot\rangle$ to indicate the locations of chunks. For instance, the type

$$\mathit{Unit} + \langle(\mathit{Int} \times \mathit{String})\rangle$$

where the "+" operator builds a (tagged) disjoint union and "×" builds a product, denotes the set of structures with chunks whose *shape* function either returns $\mathsf{Inl}\ ()$ (which has arity 0) or $\mathsf{Inr}\ \Box$ (which has arity 1), and whose *content* function either returns the empty list $[]$ or a singleton $[(n, s)]$ containing a pair (n, s) where n is an integer and s a string.

6.5 Semantics

With this notation in place, we can now define matching lenses precisely. In a matching lens, the top-level lens processes the information in the shape of the source and view, while a subordinate basic lens processes the chunks. To simplify the technicalities, we will assume that chunks only appear at the top level (*i.e.*, they are not nested), that the same basic lens is used to process each chunk, and that matching lenses themselves do not delete, duplicate, or reorder chunks. Each of these assumptions can be relaxed—see Section 6 of the original paper by Barbosa et al. (2010) for details.

Let S and V be sets of structures with chunks, C a set of structures ("rigid complements"), and k a basic lens in $S_k \overset{C_k}{\Longleftrightarrow} V_k$ such that the type of chunks in S is S_k and the type of chunks in V is V_k. A *matching lens* l on S, C, k, and V comprises three functions,

$$\begin{aligned} &\mathit{get} \in S \to V \\ &\mathit{res} \in S \to C \times [C_k] \\ &\mathit{put} \in V \times (\mathsf{Maybe}\ C \times [\mathsf{Maybe}\ C_k]) \to S \end{aligned}$$

that obey the laws shown in Figure 3 for every s and s' in S, v and v' in V, p in $(\mathsf{Maybe}\ C \times [\mathsf{Maybe}\ C_k])$, c in C, r in $[C_k]$, and mr in $[\mathsf{Maybe}\ C_k]$. We write $S \overset{C,k}{\Longleftrightarrow} V$ for the set of all matching lenses on S, C, k, and V.

Architecturally, the most important change in a matching lens is that the complement is structured as a pair $(C \times [C_k])$. We call the first component of the complement the *rigid complement* and the second the *resource*. Intuitively, the rigid complement records any information in the source shape discarded by the *get* function as it computes the view shape, while the resource records the

$$\frac{shape\ s = shape\ s'}{shape\ (get\ s) = shape\ (get\ s')} \quad \text{(GetShape)}$$

$$\frac{shape\ v = shape\ v'}{shape\ (put\ (v, p)) = shape\ (put\ (v', p))} \quad \text{(PutShape)}$$

$$\frac{res\ s = (c, r) \qquad p = (\mathsf{Just}\ c, map\ \mathsf{Just}\ r)}{shape\ (put\ (get\ s, p)) = shape\ s} \quad \text{(GetPutShape)}$$

$$\frac{}{shape\ (get\ (put\ (v, p))) = shape\ v} \quad \text{(PutGetShape)}$$

$$\frac{}{content\ (get\ s) = map\ k.get\ (content\ s)} \quad \text{(GetContent)}$$

$$\frac{(c, r) = res\ s}{r = map\ k.res\ (content\ s)} \quad \text{(ResContent)}$$

$$\frac{(\_, mr) = p \qquad arity\ (shape\ v) = length\ mr}{content\ (put\ (v, p)) = map\ k.put\ (zip\ (content\ v)\ mr)} \quad \text{(PutContent)}$$

Fig. 3. Matching lens laws.

information in the source chunks discarded by $k.get$ as it computes the view chunks. Structuring the complement in this way provides a uniform interface for applying various alignment heuristics—just rearrange the list of complements in the resource, using Nothing to handle situations where a chunk in the view is not aligned with any source chunk. It also makes it possible to state additional laws constraining the handling of data in the resource.

The matching lens laws are straightforward generalizations of the basic lens laws. The first two laws, GetShape and PutShape force the lens to map sources with identical shapes to views with identical shapes, and vice versa. The GetPutShape and PutGetShape laws are just the basic lens laws restricted to shapes. The GetContent law states that the contents of the view must be identical to the list obtained by mapping $k.get$ over the source contents. The ResContent law states an analogous condition for the resource produced by the *res* function. Taken together, these laws capture the intuition that the top-level matching lens should handle the processing of the source and view shape, and use k to process their chunks. The final law, PutContent, is the most important matching lens law. It states that if the arity of view shape is equal to the length of the resource mr, then the contents of the source produced by the *put* function must be equal to the list obtained by mapping $k.put$ over the list $(zip\ (content\ v)\ mr)$. Note that we can always truncate the resource, or extend it with additional Nothing items, to satisfy the condition on the arity of the shape and length of the resource.

6.6 Using A Matching Lens

To see how matching lenses make it possible to use arbitrary alignment heuristics, consider the same example we did before, where the source is a list of pairs and the view is obtained by projecting the first component of each item:

$$\begin{aligned} s &= [(\text{“Alice”}, \text{“Anchorage, AK”}), (\text{“Bob”}, \text{“Boston, MA”}), \\ &\quad\ (\text{“Carol”}, \text{“Chicago, IL”}), (\text{“Dave”}, \text{“Detroit, MI”})] \\ v &= [\text{“Alice”}, \text{“Bob”}, \text{“Carol”}, \text{“Dave”}] \end{aligned}$$

Also, for the sake of the example, suppose that each item in the source and view lists is a chunk. Given s, the *res* function produces the following rigid complement and resource:

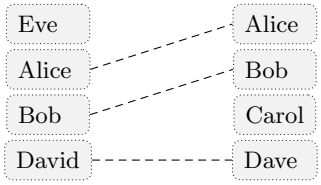
$$\begin{aligned} c &= [\square, \square, \square, \square] \\ r &= [\text{“Anchorage, AK”}, \text{“Boston, MA”}, \text{“Chicago, IL”}, \text{“Detroit, MI”}] \end{aligned}$$

The rigid complement c records the position of the source contents, while the resource r records the pieces of the contents not reflected in the view.

Now suppose that we modify “Dave” to “David”, delete “Carol”, and add “Eve” to the beginning of the list, as before. But before we invoke the *put* function to propagate these changes back to the source, we *align* the original and updated views,

$$\begin{aligned} v &= [\text{“Alice”}, \text{“Bob”}, \text{“Carol”}, \text{“Dave”}] \\ v' &= [\text{“Eve”}, \text{“Alice”}, \text{“Bob”}, \text{“David”}] \end{aligned}$$

using a heuristic function. For example, we could use a heuristic that minimizes the sum of the edit distances between contents, obtaining an *alignment* g between the locations of contents in the new and old views,



$$g = \left\{\begin{matrix} 2 \mapsto 1 \\ 3 \mapsto 2 \\ 4 \mapsto 4 \end{matrix}\right\}$$

Formally, we represent an alignment using a partial injective mapping between the locations of contents. That is, each location is associated with at most one location on the other.

Next we apply the alignment to the resource, discarding and reordering complements as specified in g, and inserting Nothing as the complement for any newly created chunks in v'. In this case, realigning the resource r using the alignment g yields the following pre-aligned resource:

$$\begin{aligned} \mathit{realign}\ (\mathit{length}\ (\mathit{content}\ v'))\ r\ g = [&\mathsf{Nothing}, \mathsf{Just}\ \text{“Anchorage, AK”}, \\ &\mathsf{Just}\ \text{“Boston, MA”}, \mathsf{Just}\ \text{“Detroit, MI”}] \end{aligned}$$

Note that the length of this resource is equal to the arity of the updated view. Finally, we run *put* on the updated view, rigid complement, and the pre-aligned resource. The PUTCONTENT law ensures that each complement in the re-aligned resource is *put* back with the corresponding chunk in the updated view,

$$s' = [(\text{“Eve”}, \text{“”}), (\text{“Alice”}, \text{“Anchorage, AK”}),\\ (\text{“Bob”}, \text{“Boston, MA”}), (\text{“David”}, \text{“Detroit, MI”})]$$

as desired.

6.7 Coercing a Matching Lens to a Basic Lens

The steps described in the previous subsection can be packaged up into a coercion $\lfloor \cdot \rfloor$ (pronounced “lower”) that takes a matching lens l in $S \overset{C,k}{\Longleftrightarrow} V$ and converts it into a basic lens in $S \overset{S}{\Longleftrightarrow} V$. Let *align* be a function that takes two views and computes an *alignment* (*i.e.*, a partial injective mapping from integers to integers). The only requirement we impose on *align* to ensure that the basic lens produced by $\lfloor \cdot \rfloor$ is well-behaved, is that it yield the identity alignment when supplied with identical lists.

The lower coercion is defined in the following two boxes:

$$\frac{l \in S \overset{C,k}{\Longleftrightarrow} V}{\lfloor l \rfloor \in S \overset{S}{\Longleftrightarrow} V}$$

$$\begin{aligned}
&\mathit{get}\ s &&= l.\mathit{get}\ s\\
&\mathit{res}\ s &&= s\\
&\mathit{put}\ (v, \mathsf{Nothing}) &&= l.\mathit{put}\ (v, (\mathsf{Nothing}, [\,]))\\
&\mathit{put}\ (v, \mathsf{Just}\ s) &&= l.\mathit{put}\ (v, (c, \mathit{realign}\ (\mathit{length}\ (\mathit{content}\ v))\ r\ g))\\
&\quad \text{where } (c, r) &&= l.\mathit{res}\ s\\
&\quad\ \text{and } g &&= \mathit{align}\ v\ (l.\mathit{get}\ s)
\end{aligned}$$

The top box states a typing rule that can be read as a lemma asserting that, if l is a matching lens in $S \overset{C,k}{\Longleftrightarrow} V$, then $\lfloor l \rfloor$ is a basic lens in $S \overset{S}{\Longleftrightarrow} V$. The bottom box defines the components of $\lfloor l \rfloor$. The *get* function is just $l.\mathit{get}$. The *res* function uses the whole source as the basic lens complement. The *put* function takes a view v and an optional basic lens complement as arguments. If the complement is Nothing, it invokes $l.\mathit{put}$ with Nothing as the rigid complement and the empty resource. If the complement is Just s, it first uses $l.\mathit{res}$ to calculate a rigid complement c and a resource r from s. Next, it uses *align* to calculate a correspondence g between the locations of chunks in the updated view v and chunks in the original view $l.\mathit{get}\ s$ and applies the *realign* function, which interprets the alignment g on r, discarding and reordering items as indicated in g, and adding Nothing for unaligned chunks. To finish the job, it passes v, c, and the pre-aligned resource $(\mathit{realign}\ (\mathit{length}\ (\mathit{content}\ v))\ r\ g)$ to $l.\mathit{put}$, which computes the updated source.

6.8 Matching Lens Combinators

We now define matching lens combinators for a number of useful transformations on datatypes, with typing rules that ensure the behavioral laws.

Lifting Intuitively, it should be clear that matching lenses generalize basic lenses. This fact is witnessed by the *lift* operator, which takes a basic lens k as an argument:

$$\frac{k \in A \overset{C}{\Longleftrightarrow} B}{\mathit{lift}\ k \in A \overset{C,k'}{\Longleftrightarrow} B}$$

$$\begin{array}{ll} \mathit{get}\ a & = k.\mathit{get}\ a \\ \mathit{res}\ a & = (k.\mathit{res}\ a, [\,]) \\ \mathit{put}\ (b, (\mathit{co}, \_)) & = k.\mathit{put}\ (b, \mathit{co}) \end{array}$$

The *get* function simply invokes $k.\mathit{get}$ on the source. The *res* function computes the rigid complement using $k.\mathit{res}$ and produces the empty resource (as it must to satisfy ResContent). The *put* function invokes $k.\mathit{put}$ and ignores its resource argument. To ensure the other matching lens laws, the typing rule for *lift* requires that the source and view types must not contain chunks. (We use metavariables A and B to range over sets of structures without chunks.) The basic lens k' mentioned in the type of $\mathit{lift}\ k$ can be arbitrary. Using *lift*, we can obtain matching lenses versions of many useful basic lenses including the identity lens $\mathit{id}\ A \in A \overset{\mathit{Unit}}{\Longleftrightarrow} A$, which copies elements of A in both directions; the rewriting lens $A \leftrightarrow \{b\} \in A \overset{A}{\Longleftrightarrow} \{b\}$, which rewrites an element of A to b in the *get* direction and restores the discarded A in the *put* direction; and the lenses $\pi_1 \in A \times B \overset{B}{\Longleftrightarrow} A$ and $\pi_2 \in A \times B \overset{A}{\Longleftrightarrow} B$, which project away one component of a pair in the *get* direction, and restore it in the *put* direction.

Match Another way to lift a basic lens to a matching lens is to place it in a chunk.

$$\frac{k \in A \overset{C}{\Longleftrightarrow} B}{\langle k \rangle \in \langle A \rangle \overset{\{\Box\},k}{\Longleftrightarrow} \langle B \rangle}$$

$$\begin{array}{ll} \mathit{get}\ a & = k.\mathit{get}\ a \\ \mathit{res}\ a & = (\Box, [k.\mathit{res}\ a]) \\ \mathit{put}\ (b, (\_, c : \_)) & = k.\mathit{put}\ (b, c) \\ \mathit{put}\ (b, (\_, [\,])) & = k.\mathit{put}\ (b, \mathsf{Nothing}) \end{array}$$

The lens $\langle k \rangle$ (pronounced "match k") is perhaps the most important matching lens. The *get* function invokes $k.\mathit{get}$. The *res* function takes a source a as an argument and yields $\Box$ as the rigid complement and $[k.\mathit{res}\ a]$ as the resource. The *put* function accesses the complement for the chunk through the resource

r, invoking $k.put$ on the view and head of r if r is non-empty or Nothing if r is empty. The elements of the source type $\langle A \rangle$ and the view type $\langle B \rangle$ have a single shape and contents that consists of a single reorderable chunk. Also note that the basic lens mentioned in the type of $\langle k \rangle$ is k itself.

Composition The next combinator puts two matching lenses in sequence:

$$\frac{l_1 \in S \overset{C_1,k_1}{\Longleftrightarrow} U \qquad l_2 \in U \overset{C_2,k_2}{\Longleftrightarrow} V \qquad C = C_1 \times C_2 \qquad k = k_1; k_2}{l_1; l_2 \in S \overset{C,k}{\Longleftrightarrow} V}$$

$$\begin{aligned}
&\mathit{get}\ s &&= l_2.\mathit{get}\ (l_1.\mathit{get}\ s)\\
&\mathit{res}\ s &&= ((c_1, c_2), \mathit{zip}\ r_1\ r_2)\\
&\quad \text{where } (c_1, r_1) = l_1.\mathit{res}\ s\\
&\quad\quad \text{and } (c_2, r_2) = l_2.\mathit{res}\ (l_1.\mathit{get}\ s)\\
&\mathit{put}\ (v, (\mathsf{Just}\ (c_1, c_2), r)) &&= s\\
&\quad \text{where } s = l_1.\mathit{put}\ (u, (\mathsf{Just}\ c_1, r_1))\\
&\quad\quad \text{and } u = l_2.\mathit{put}\ (v, (\mathsf{Just}\ c_2, r_2))\\
&\quad\quad \text{and } (r_1, r_2) = \mathit{unzip}\ (\mathit{map}\ \mathit{split_maybe}\ r)\\
&\mathit{put}\ (v, (\mathsf{Nothing}, r)) &&= s\\
&\quad \text{where } s = l_1.\mathit{put}\ (u, (\mathsf{Nothing}, r_1))\\
&\quad\quad \text{and } u = l_2.\mathit{put}\ (v, (\mathsf{Nothing}, r_2))\\
&\quad\quad \text{and } (r_1, r_2) = \mathit{unzip}\ (\mathit{map}\ \mathit{split_maybe}\ r)
\end{aligned}$$

Composition is especially interesting as a matching lens because it handles alignment in two sequential phases of computation. The *get* function applies $l_1.\mathit{get}$ and $l_2.\mathit{get}$ in sequence. The *res* function applies $l_1.\mathit{res}$ to the source s, yielding a rigid complement c_1 and resource r_1, and $l_2.\mathit{res}$ to $l_1.\mathit{get}\ s$, yielding c_2 and r_2. It merges the rigid complements into a pair (c_1, c_2) and combines the resources by zipping them together. Note that the two resources must have the same length by GetContent and ResContent, so $\mathit{zip}\ r_1\ r_2$ loses no information. The *put* function maps *split_maybe* over the resource, unzips the result, and applies the $l_2.\mathit{put}$ and $l_1.\mathit{put}$ functions in that order. The *split_maybe* function is defined as follows,

$$\begin{aligned}
\mathit{split_maybe} = \lambda mc \to \mathbf{case}\ mc\ \mathbf{of}\\
\qquad \mathsf{Nothing} &\to (\mathsf{Nothing}, \mathsf{Nothing})\\
\qquad \mathsf{Just}\ (c_1, c_2) &\to (\mathsf{Just}\ c_1, \mathsf{Just}\ c_2)
\end{aligned}$$

Because the zipped resource represents the resources generated by l_1 and l_2 together, rearranging the resource has the effect of pre-aligning the resources for both phases of computation. The typing rule for the composition lens combinator requires the view type of l_1 to be identical to the source type of l_2. In particular, it requires that the chunks in these types must be identical. Intuitively, this makes sense—the only way that the *put* function can reasonably translate alignments on the view back through both phases of computation to the source is if the chunks in the types of each lens agree.

Product The next combinator takes lenses l_1 and l_2 as arguments and produces a lens that operates on pairs.

$$\frac{l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1 \qquad l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2 \qquad C = C_1 \times C_2}{l_1 \otimes l_2 \in S_1 \times S_2 \overset{C,k}{\Longleftrightarrow} V_1 \times V_2}$$

$$\begin{aligned}
&\mathit{get}\ (s_1, s_2) &&= (l_1.\mathit{get}\ s_1,\ l_2.\mathit{get}\ s_2)\\
&\mathit{res}\ (s_1, s_2) &&= ((c_1, c_2), r_1 +\!\!+ r_2)\\
&\quad \text{where } (c_1, r_1) = l_1.\mathit{res}\ s_1\\
&\quad\ \ \text{and } (c_2, r_2) = l_2.\mathit{res}\ s_2\\
&\mathit{put}\ ((v_1, v_2), (\mathsf{Just}\ (c_1, c_2), r)) &&= (s_1, s_2)\\
&\quad \text{where } s_1 = l_1.\mathit{put}\ (v_1, (\mathsf{Just}\ c_1, r_1))\\
&\quad\ \ \text{and } s_2 = l_2.\mathit{put}\ (v_2, (\mathsf{Just}\ c_2, r_2))\\
&\quad\ \ \text{and } (r_1, r_2) = (\mathit{take}\ n\ r, \mathit{drop}\ n\ r)\\
&\quad\ \ \text{and } n = \mathit{length}\ (\mathit{content}\ v_1)\\
&\mathit{put}\ ((v_1, v_2), (\mathsf{Nothing}, r)) &&= (s_1, s_2)\\
&\quad \text{where } s_1 = l_1.\mathit{put}\ (v_1, (\mathsf{Nothing}, r_1))\\
&\quad\ \ \text{and } s_2 = l_2.\mathit{put}\ (v_2, (\mathsf{Nothing}, r_2))\\
&\quad\ \ \text{and } (r_1, r_2) = (\mathit{take}\ n\ r, \mathit{drop}\ n\ r)\\
&\quad\ \ \text{and } n = \mathit{length}\ (\mathit{content}\ v_1)
\end{aligned}$$

The *get* function applies $l_1.\mathit{get}$ and $l_2.\mathit{get}$ to the components of the source pair. The *res* function takes a source (s_1, s_2) and applies $l_1.\mathit{res}$ to s_1 and $l_2.\mathit{res}$ to s_2, yielding rigid complements c_1 and c_2 and resources r_1 and r_2. It then merges the rigid complements into a pair (c_1, c_2) and the resources into a single resource $r_1 +\!\!+ r_2$. Because the same basic lens k is mentioned in the types of l_1 and l_2, the resources r_1, r_2, and $r_1 +\!\!+ r_2$ all have type $[C_k]$. This is essential—it ensures that we can freely reorder the resource and pass arbitrary portions of it to l_1 and l_2. It is tempting to relax this condition and allow l_1 and l_2 to be defined over different basic lenses, as long as the resources produced by those lenses have the same type. Unfortunately, this would require weakening the matching lens laws—see the paper by Barbosa et al. (2010) for details and a concrete example. The *put* function of the product lens applies the *put* functions of l_1 and l_2 to the appropriate pieces of the view. To create the resource for the calls to *put*, it splits the resource into two pieces using the number of chunks in the first component of the view. Note that although this appears to be biased toward the left component of the pair, it is not in the case where the resource has been pre-aligned so that it contains the same number of items as chunks in the view.

Iteration The iteration combinator applies a lens to a list of items.

$$\frac{l \in S \overset{C_1,k}{\Longleftrightarrow} V \qquad C = [C_1]}{l^* \in [S] \overset{C,k}{\Longleftrightarrow} [V]}$$

$$\begin{aligned}
&\mathit{get}\ [s_1, \ldots, s_n] &&= [l.\mathit{get}\, s_1, \ldots, l.\mathit{get}\, s_n] \\
&\mathit{res}\ [s_1, \ldots, s_n] &&= ([c_1, \ldots, c_n], r_1 +\!\!+ \ldots +\!\!+ r_n) \\
&\quad \text{where } (c_i, r_i) = l.\mathit{res}\ s_i \text{ for } i \in \{1, \ldots, n\} \\
&\mathit{put}\ ([v_1 \ldots v_n], (mc, r'_0)) = [s'_1, \ldots, s'_n] \\
&\quad \text{where } s'_i = \begin{cases} l.\mathit{put}\ (v_i, (\mathsf{Just}\ c_i, r_i)) & i \in \{1, \ldots, \min(n, m)\} \\ l.\mathit{put}\ (v_i, (\mathsf{Nothing}, r_i)) & i \in \{m+1, \ldots, n\} \end{cases} \\
&\qquad \text{and } [c_1, \ldots, c_m] = \begin{cases} c & \text{if } mc = \mathsf{Just}\ c \\ [\,] & \text{if } mc = \mathsf{Nothing} \end{cases} \\
&\qquad \text{and } r_i = \mathit{take}\ (\mathit{length}\ (\mathit{content}\ v_i))\ r'_{(i-1)} \text{ for } i \in \{1, \ldots, n\} \\
&\qquad \text{and } r'_i = \mathit{drop}\ (\mathit{length}\ (\mathit{content}\ v_i))\ r'_{(i-1)} \text{ for } i \in \{1, \ldots, n\}
\end{aligned}$$

The *get* and *res* components are straightforward generalizations of the corresponding components of the product lens. The *put* function, however, is different—it handles cases where the view and the rigid complement have different lengths. When the rigid complement is longer, it discards the extra complements; when the view is longer, it processes the extras using Nothing.

Union The final combinator forms the union of two matching lenses.

$$\frac{l_1 \in S_1 \overset{C_1,k}{\Longleftrightarrow} V_1 \qquad l_2 \in S_2 \overset{C_2,k}{\Longleftrightarrow} V_2 \qquad C = C_1 + C_2 \qquad \mathit{compatible}(V_1, V_2)}{l_1 \mid l_2 \in S_1 + S_2 \overset{C,k}{\Longleftrightarrow} (V_1 \cap V_2) + (V_1 \setminus V_2 + V_2 \setminus V_1)}$$

$$\begin{aligned}
&\mathit{get}\ (\mathsf{Inl}\ s_1) &&= \begin{cases} \mathsf{Inl}\ (l_1.\mathit{get}\ s_1) & \text{if } l_1.\mathit{get}\ s_1 \in V_2 \\ \mathsf{Inr}\ (\mathsf{Inl}\ (l_1.\mathit{get}\ s_1)) & \text{if } l_1.\mathit{get}\ s_1 \notin V_2 \end{cases} \\
&\mathit{get}\ (\mathsf{Inr}\ s_2) &&= \begin{cases} \mathsf{Inl}\ (l_2.\mathit{get}\ s_2) & \text{if } l_2.\mathit{get}\ s_2 \in V_1 \\ \mathsf{Inr}\ (\mathsf{Inr}\ (l_2.\mathit{get}\ s_2)) & \text{if } l_2.\mathit{get}\ s_2 \notin V_1 \end{cases} \\
&\mathit{res}\ (\mathsf{Inl}\ s_1) &&= (\mathsf{Inl}\ c_1, r), \quad \text{where } (c_1, r) = l_1.\mathit{res}\ s_1 \\
&\mathit{res}\ (\mathsf{Inr}\ s_2) &&= (\mathsf{Inr}\ c_2, r), \quad \text{where } (c_2, r) = l_2.\mathit{res}\ s_2 \\
&\mathit{put}\ (\mathsf{Inl}\ v, (\mathsf{Just}\ (\mathsf{Inl}\ c_1), r)) &&= \mathsf{Inl}\ (l_1.\mathit{put}\ (v, (\mathsf{Just}\ c_1, r))) \\
&\mathit{put}\ (\mathsf{Inl}\ v, (\mathsf{Just}\ (\mathsf{Inr}\ c_2), r)) &&= \mathsf{Inr}\ (l_2.\mathit{put}\ (v, (\mathsf{Just}\ c_2, r))) \\
&\mathit{put}\ (\mathsf{Inl}\ v, (\mathsf{Nothing}, r)) &&= \mathsf{Inl}\ (l_1.\mathit{put}\ (v, (\mathsf{Nothing}, r))) \\
&\mathit{put}\ (\mathsf{Inr}\ (\mathsf{Inl}\ v_1), (\mathsf{Just}\ (\mathsf{Inl}\ c_1), r)) &&= \mathsf{Inl}\ (l_1.\mathit{put}\ (v_1, (\mathsf{Just}\ c_1, r))) \\
&\mathit{put}\ (\mathsf{Inr}\ (\mathsf{Inl}\ v_1), (\mathsf{Just}\ (\mathsf{Inr}\ c_2), r)) &&= \mathsf{Inl}\ (l_1.\mathit{put}\ (v_1, (\mathsf{Nothing}, r))) \\
&\mathit{put}\ (\mathsf{Inr}\ (\mathsf{Inl}\ v_1), (\mathsf{Nothing}, r)) &&= \mathsf{Inl}\ (l_1.\mathit{put}\ (v_1, (\mathsf{Nothing}, r))) \\
&\mathit{put}\ (\mathsf{Inr}\ (\mathsf{Inr}\ v_2), (\mathsf{Just}\ (\mathsf{Inr}\ c_2), r)) &&= \mathsf{Inr}\ (l_2.\mathit{put}\ (v_2, (\mathsf{Just}\ c_2, r))) \\
&\mathit{put}\ (\mathsf{Inr}\ (\mathsf{Inr}\ v_2), (\mathsf{Just}\ (\mathsf{Inl}\ c_1), r)) &&= \mathsf{Inr}\ (l_2.\mathit{put}\ (v_2, (\mathsf{Nothing}, r))) \\
&\mathit{put}\ (\mathsf{Inr}\ (\mathsf{Inr}\ v_2), (\mathsf{Nothing}, r)) &&= \mathsf{Inr}\ (l_2.\mathit{put}\ (v_2, (\mathsf{Nothing}, r)))
\end{aligned}$$

The union combinator implements a bidirectional conditional operator on lenses. Its *get* function selects one of $l_1.get$ or $l_2.get$ by testing the tag on the source. It tags the result, injecting it into the type $(V_1 \cap V_2) + (V_1 \setminus V_2 + V_2 \setminus V_1)$, which is a disjoint union representing values in the intersection of V_1 and V_2 and values that only belong to V_1 or V_2. Its *res* function is similar. It places the rigid complement in a tagged sum, producing $\mathsf{Inl}\ c$ if the source belongs to S_1 and $\mathsf{Inr}\ c$ if it belongs to S_2. It does not tag the resource however—because l_1 and l_2 are defined over the same basic lens k for chunks, we can safely pass a resource computed by $l_1.res$ to $l_2.put$ and vice versa.

The *put* function of the union lens first tries to select one of $l_1.put$ or $l_2.put$ using the tag on the view, and only uses the rigid complement to disambiguate cases where the view belongs to $(V_1 \cap V_2)$. Note that because *put* is a total function, it needs to handle cases where the view has the form $\mathsf{Inr}\ (\mathsf{Inl}\ v_1)$ (*i.e.*, v_1 belongs to $V_1 \setminus V_2$) but the complement is of the form $\mathsf{Just}\ (\mathsf{Inr}\ c_2)$. To satisfy the PutGetShape law, it must invoke one of l_1's component functions, but it cannot invoke $l_1.put$ with the rigid complement c_2 because c_2 does not belong to C_1. Thus, it discards c_2 and uses $\mathsf{Nothing}$ instead. The *put* function arbitrarily uses $l_1.put$ in the case where the view belongs to both V_1 and V_2 and the complement is $\mathsf{Nothing}$. The condition $compatible(V_1, V_2)$ mentioned in the hypothesis of the typing rule stipulates that *shape*, *content*, etc. must return identical results for structures in the intersection $V_1 \cap V_2$. This ensures that the type of the view is a well-formed container.

6.9 Matching Lens Example

Let us finish this section by defining a matching lens that implements the transformation from sources consisting of pairs of names and cities to views consisting of just names. For the sake of the example, and to illustrate the use of sequential composition, we will implement a transformation that works in two steps.

Assume that we have a basic lens *delete_city* whose *get* function takes a source string of the form "City, XY" and produces a view of the form "XY". Also assume that we have a type *Name* that describes the set of name strings. Both of these can be easily defined in the Boomerang language. The matching lens l_1 copies the name from each item in the source list and deletes the city:

$$l_1 = \langle id\ Name \otimes delete\_city \rangle^*$$

The basic lens inside of the match combinator in l_1 uses the (basic lens version of) the product operator to combine ($id\ Name$) and *delete_city*. Its *get* function maps the following source

$$s = [(\text{“Alice”}, \text{“Anchorage, AK”}), (\text{“Bob”}, \text{“Boston, MA”}),\\ (\text{“Carol”}, \text{“Chicago, IL”}), (\text{“Dave”}, \text{“Detroit, MI”})]$$

to the view v_1 by deleting the cities:

$$v_1 = [(\text{“Alice”}, \text{“AK”}), (\text{“Bob”}, \text{“MA”}), (\text{“Carol”}, \text{“IL”}), (\text{“Dave”}, \text{“MI”})]$$

Now consider the matching lens l_2,

$$l_2 = \langle \pi_1 \rangle^*$$

Its *get* function projects away the second component of each item in its source list. Returning to the example, it takes the view v_1 computed by l_1 and produces a view v_2:

$$v_2 = [\text{“Alice”}, \text{“Bob”}, \text{“Carol”}, \text{“Dave”}]$$

Thus, the *get* function for our running example is just $\lfloor l_1; l_2 \rfloor$. In fact, this lens implements the *put* function too. To see how *put* works, first consider the rigid complement and resource computed by the *res* from the source s. By the definition of the sequential composition lens, these structures record the information produced by $l_1.res$ and $l_2.res$:

$$\begin{aligned} c &= ([\Box, \Box, \Box, \Box], [\Box, \Box, \Box, \Box]) \\ r &= [(\text{“Anchorage”}, \text{“AK”}), (\text{“Boston”}, \text{“MA”}), \\ &\quad\ (\text{“Chicago”}, \text{“IL”}), (\text{“Detroit”}, \text{“MI”})] \end{aligned}$$

If we edit the final view v_2 to v_2' by inserting “Eve”, deleting “Carol”, and modifying “Dave” to “David”,

$$v_2' = [\text{“Eve”}, \text{“Alice”}, \text{“Bob”}, \text{“David”}],$$

and then realign the resource using an alignment such as g from Section 6.6, which minimizes the sum of the total edit distances between aligned chunks, then the resulting resource will contain the pre-aligned chunk complements for both phases of computation:

$$\begin{aligned} &\mathit{realign}\ (\mathit{length}\ (\mathit{content}\ v_2'))\ r\ g = \\ &\quad [\mathsf{Nothing}, \mathsf{Just}\ (\text{“Anchorage”}, \text{“AK”}), \\ &\quad\ \mathsf{Just}\ (\text{“Boston”}, \text{“MA”}), \mathsf{Just}\ (\text{“Detroit”}, \text{“MI”})] \end{aligned}$$

Evaluating the *put* function on the updated view, rigid complement, and this resource first splits the complement into two pieces and unzips the resource. Next, it invokes $l_2.put$ on v_2' and

$$\begin{aligned} c_2 &= [\Box, \Box, \Box, \Box] \\ r_2 &= [\mathsf{Nothing}, \mathsf{Just}\ \text{“AK”}, \mathsf{Just}\ \text{“MA”}, \mathsf{Just}\ \text{“MI”}] \end{aligned}$$

which produces an intermediate view:

$$\begin{aligned} v_1' = [&(\text{“Eve”}, \text{“”}), (\text{“Alice”}, \text{“AK”}), \\ &(\text{“Bob”}, \text{“MA”}), (\text{“David”}, \text{“MI”})] \end{aligned}$$

Finally, it invokes $l_1.put$ on v_1' and

$$\begin{aligned} c_1 &= [\Box, \Box, \Box, \Box] \\ r_1 &= [\mathsf{Nothing}, \mathsf{Just}\ \text{“Anchorage”}, \mathsf{Just}\ \text{“Boston”}, \mathsf{Just}\ \text{“Detroit”}] \end{aligned}$$

yielding the final result,

$$\begin{aligned} s' = [&(\text{“Eve”}, \text{“”}), (\text{“Alice”}, \text{“Anchorage, AK”}), \\ &(\text{“Bob”}, \text{“Boston, MA”}), (\text{“David”}, \text{“Detroit, MI”})] \end{aligned}$$

as desired.

6.10 Summary

Matching lenses address some important problems that come up when bidirectional transformations are used to manipulate ordered structures. By separating the handling of the reorderable chunks and the rigidly ordered parts of the source and view, they provide a framework that can be instantiated with arbitrary alignment heuristics. A number of useful primitives and combinators can be interpreted as matching lenses.

7 Discussion and Related Work

This section summarizes the three techniques described in this paper and compares the relative advantages and disadvantages of each approach. At the end of the section, we briefly discuss related work on languages for describing bidirectional transformations.

The first technique, syntactic bidirectionalization (Matsuda et al. 2007), constructs a *put* function from a (program describing a) *get* function using a combination of syntactic program transformations. The overall transformation goes in three steps:

- First, it constructs a complement function $res \in S \to C$ from the definition of $get \in S \to V$.
- Second, it combines *get* and *res* into a single function $\langle get, res \rangle \in S \to V \times C$ amd inverts that one to obtain function $\langle get, res \rangle^{-1} \in V \times C \to S$.
- Finally, it uses *res* and $\langle get, res \rangle^{-1}$ to construct $put \in V \to S \to S$:

$$put = \lambda v \to \lambda s \to \langle get, res \rangle^{-1}\ (v, res\ s)$$

By construction, the lens consisting of the *get* and *put* functions is guaranteed to be (partially) very well-behaved. Syntactic bidirectionalization is attractive for several reasons. Most importantly, it makes it possible to express bidirectional transformations using a standard language. The programmer simply writes the *get* function in a (restricted) functional language and the technique constructs a suitable *put* function automatically. Moreover, because it is based on the constant-complement approach, the *put* function is guaranteed to obey the PutPut law, in its partial form. The most significant disadvantage of syntactic bidirectionalization is indeed that, in general, the *put* function is not total. This can be mitigated, to some extent, using optimizations that produce "smaller" complements. But the heuristics used for optimization can be unpredictable. Another issue is that syntactic bidirectionalization produces one *put* function, while in general there are many *put* functions that can be combined with a given *get* function to form a reasonable lens—and because the transformation is automatic, the programmer has no influence over which of them is chosen.

The second technique, semantic bidirectionalization (Voigtländer 2009), is similar to the syntactic approach. But instead of taking a program describing the *get* function as input, it takes the function itself. Besides being elegant, this

approach has a significant advantage: because it does not operate on the syntax of programs, it can be used to bidirectionalize arbitrary functions, including ones whose source code cannot be analyzed. The only condition semantic bidirectionalization requires is that the *get* function be a polymorphic function. Exploiting parametricity, the technique manufactures a *put* function by simulating the *get* function on a canonical template and interpreting the results to infer the mapping from source to view (and then reverse it). Like syntactic bidirectionalization, it guarantees that the *put* function will satisfy a form of PutPut, does not guarantee that the *put* function will be total, and only generates a single *put* function for a given *get*. Finally, because the *put* function is implemented by simulating the behavior of *get*, it is not very efficient.

Recent work by (among others) two of the authors of this paper combines the syntactic and semantic approaches to bidirectionalization in a single system (Voigtländer et al. 2010). A prototype implementation of all three forms of bidirectionalization—syntactic, semantic, and combined—as a web interface can be found at the following url:

`http://www-ps.iai.uni-bonn.de/cgi-bin/b18n-combined-cgi`

Combining syntactic and semantic bidirectionalization allows more updates to be handled (*i.e.*, the *put* function is defined on more inputs) than in the individual approaches. In particular, the combined approach gracefully handles updates to the shape of the view using a mechanism based on syntactic bidirectionalization, while it uses semantic bidirectionalization to manage the polymorphic data in the view. The separation of shape from content is conceptually similar to the treatment of generic data structures discussed in Section 5.4 and the structures with chunks used in the matching lenses described in Section 6. In addition, the combined technique allows programmers to select different *put* functions for a given *get* by specifying a *bias*, which controls the handling of extra (or deleted) data values in the view.

The third and final technique described in this paper uses domain-specific lens combinators to describe a *get* and *put* function simultaneously (Barbosa et al. 2010; Bohannon et al. 2008; Foster et al. 2007b). Several full-blown bidirectional programming languages have been built using combinators, including Boomerang (Foster and Pierce 2009) and Augeas (Lutterkort 2008). Unlike the pure syntactic and semantic bidirectionalization techniques, which both follow the constant-complement approach but produce partial *put* functions, lens combinators sacrifice the PutPut law but guarantee that *put* is a total function.[11] The failure of the PutPut law can be easily seen in the iteration and union lenses, which do not always preserve the complement. The combinators ensure well-behavedness using a type system—every well-typed program in $S \overset{C}{\Longleftrightarrow} V$ denotes a well-behaved lens on S, V, and C. An advantage of the combinator approach is that operators are guaranteed to satisfy strong properties such as totality, as they are derived directly from the semantics. In addition, because

[11] The combined syntactic/semantic bidirectionalization technique of Voigtländer et al. (2010) gives up both constant-complement/PutPut and totality of *put*.

each lens combinator describes a *get* function and a *put* function, programmers have a means to select between the possible well-behaved *put* functions for a given *get* function. Finally, working with combinators makes it easy to extend the language with new constructs for dealing with important issues such as alignment, as in the matching lenses described in this paper. The existing bidirectionalization techniques, even the combined approach, are limited to positional alignment. The main disadvantage of using combinators is, of course, that it requires bidirectional transformations to be expressed using special-purpose language constructs.

The three approaches presented in this paper are not comprehensive. Numerous other techniques for describing bidirectional transformations that have been proposed in the literature. We briefly summarize some of the most recent related work in this area from a programming language perspective in the rest of this section. For more comprehensive, and broader, overviews we direct interested readers to the original paper on lens combinators (Foster et al. 2007b) and the GRACE workshop report (Czarnecki et al. 2009).

A project by Pacheco and Cunha (2010, 2011) proposes an extensive collection of point-free generic lens combinators. The authors have implemented these combinators as a Haskell library and investigated the issue of optimization for lens programs using many of the same algebraic equivalences that are commonly used to optimize programs in standard functional languages. Wang et al. (2011) are also concerned about efficiency, and study incremental updating in a bidirectional setting. Fegaras (2010) proposes a technique for propagating updates to XML views using lineage—metadata that tracks the relationship between a piece of the view and the pieces of the source that generated it—to guide the translation of updates. The reliance on polymorphic type information to achieve this correctly is closely related to what happens in the semantic bidirectionalization technique. Hofmann et al. (2011) describe a variant of lenses in which the *get* and *put* functions have symmetric types $S \times C \to V$ and $V \times C \to S$. They develop a number of useful combinators in this symmetric setting. Another recent paper by Hidaka et al. (2010) defines a bidirectional semantics for the UnCAL graph transformation language. The reverse semantics of the language uses traces, which are similar to the lineage artifacts mentioned above. Finally, a paper by Diskin et al. (2010) proposes a system in which the *put* functions take update operations instead of whole states (of the view) as inputs. Among other things, this approach makes it possible to solve the alignment problems described in Section 6.1 in an elegant way.

Acknowledgments. The authors wish to thank Jeremy Gibbons and Jeremy Siek for their comments and suggestions for improving the paper. Foster's work was supported in part by the ONR under grant N00014-09-1-0652. Any opinions, findings, and recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ONR. Matsuda's work was supported in part by Japan Society for the Promotion of Science, Grant-in-Aid for Research Activity Start-up 22800003.

References

M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Foundations of Software Science and Computation Structures, Proceedings*, volume 2620 of *LNCS*, pages 23–38. Springer, 2003. doi: 10.1007/3-540-36576-1_2.

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981. doi: 10.1145/319628.319634.

D.M.J. Barbosa, J. Cretin, J.N. Foster, M. Greenberg, and B.C. Pierce. Matching lenses: Alignment and view update. In *International Conference on Functional Programming, Proceedings*, volume 45(9) of *SIGPLAN Notices*, pages 193–204. ACM, 2010. doi: 10.1145/1932681.1863572.

N. Benton. Embedded interpreters. *Journal of Functional Programming*, 15(4): 503–542, 2005. doi: 10.1017/S0956796804005398.

P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser. Coupled schema transformation and data conversion for XML and SQL. In *Practical Aspects of Declarative Languages, Proceedings*, volume 4354 of *LNCS*, pages 290–304. Springer, 2007. doi: 10.1007/978-3-540-69611-7_19.

A. Bohannon, J.A. Vaughan, and B.C. Pierce. Relational lenses: A language for updateable views. In *Principles of Database Systems, Proceedings*, pages 338–347. ACM, 2006. doi: 10.1145/1142351.1142399.

A. Bohannon, J.N. Foster, B.C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: Resourceful lenses for string data. In *Principles of Programming Languages, Proceedings*, volume 43(1) of *SIGPLAN Notices*, pages 407–419. ACM, 2008. doi: 10.1145/1328897.1328487.

C. Brabrand, A. Møller, and M.I. Schwartzbach. Dual syntax for XML languages. *Information Systems*, 33(4–5):385–406, 2008. doi: 10.1016/j.is.2008.01.006.

H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available from `http://tata.gforge.inria.fr/`, 2007. Release October, 12th 2007.

S.S. Cosmadakis and C.H. Papadimitriou. Updates of relational views. *Journal of the ACM*, 31(4):742–760, 1984. doi: 10.1145/1634.1887.

J. Cunha, J. Saraiva, and J. Visser. From spreadsheets to relational databases and back. In *Partial Evaluation and Program Manipulation, Proceedings*, pages 179–188. ACM, 2009. doi: 10.1145/1480945.1480972.

K. Czarnecki, J.N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J.F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. GRACE meeting notes, state of the art, and outlook. In *International Conference on Model Transformations, Proceedings*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009. doi: 10.1007/978-3-642-02408-5_19.

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. doi: 10.1145/319732.319740.

Z. Diskin, Y. Xiong, and K. Czarnecki. From state- to delta-based bidirectional model transformations. In *International Conference on Model Transformations, Proceedings*, volume 6142 of *LNCS*, pages 61–76. Springer, 2010. doi: 10.1007/978-3-642-13688-7_5.

R. Ennals and D. Gay. Multi-language synchronization. In *European Symposium on Programming, Proceedings*, volume 4421 of *LNCS*, pages 475–489. Springer, 2007. doi: 10.1007/978-3-540-71316-6_32.

L. Fegaras. Propagating updates through XML views using lineage tracing. In *International Conference on Data Engineering, Proceedings*, pages 309–320. IEEE, 2010. doi: 10.1109/ICDE.2010.5447896.

K. Fisher and R. Gruber. PADS: A domain-specific language for processing ad hoc data. In *Programming Language Design and Implementation, Proceedings*, volume 40(6) of *SIGPLAN Notices*, pages 295–304. ACM, 2005. doi: 10.1145/1064978.1065046.

J.N. Foster and B.C. Pierce. *Boomerang Programmer's Manual*, 2009. Available from `http://www.seas.upenn.edu/~harmony/`.

J.N. Foster, M.B. Greenwald, C. Kirkegaard, B.C. Pierce, and A. Schmitt. Exploiting schemas in data synchronization. *Journal of Computer and System Sciences*, 73(4):669–689, 2007a. doi: 10.1016/j.jcss.2006.10.024.

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):17, 2007b. doi: 10.1145/1232420.1232424.

J.N. Foster, A. Pilkiewicz, and B.C. Pierce. Quotient lenses. In *International Conference on Functional Programming, Proceedings*, volume 43(9) of *SIGPLAN Notices*, pages 383–395. ACM, 2008. doi: 10.1145/1411203.1411257.

J.N. Foster, B.C. Pierce, and S. Zdancewic. Updatable security views. In *Computer Security Foundations, Proceedings*, pages 60–74. IEEE, 2009. doi: 10.1109/CSF.2009.25.

J. Gibbons and B.C.d.S. Oliveira. The essence of the iterator pattern. *Journal of Functional Programming*, 19(3–4):377–402, 2009. doi: 10.1017/S0956796809007291.

S.J. Hegner. An order-based theory of updates for closed database views. *Annals of Mathematics and Artificial Intelligence*, 40(1–2):63–125, 2004. doi: 10.1023/A:1026158013113.

S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *International Conference on Functional Programming, Proceedings*, volume 45(9) of *SIGPLAN Notices*, pages 205–216. ACM, 2010. doi: 10.1145/1932681.1863573.

M. Hofmann, B.C. Pierce, and D. Wagner. Symmetric lenses. In *Principles of Programming Languages, Proceedings*, volume 46(1) of *SIGPLAN Notices*, pages 371–384. ACM, 2011. doi: 10.1145/1925844.1926428.

Z. Hu, H. Iwasaki, M. Takeichi, and A. Takano. Tupling calculation eliminates multiple data traversals. In *International Conference on Functional Programming, Proceedings*, volume 32(8) of *SIGPLAN Notices*, pages 164–175. ACM, 1997. doi: 10.1145/258949.258964.

Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order and Symbolic Computation*, 21(1–2):89–118, 2008. doi: 10.1007/s10990-008-9025-5.

C.B. Jay. A semantics for shape. *Science of Computer Programming*, 25(2–3): 251–283, 1995. doi: 10.1016/0167-6423(95)00015-1.

J. Jeuring, S. Leather, J.P. Magalhães, and A. Rodriguez Yakushev. Libraries for generic programming in Haskell. In *Advanced Functional Programming 2008, Revised Lectures*, volume 5832 of *LNCS*, pages 165–229. Springer, 2009. doi: 10.1007/978-3-642-04652-0_4.

S. Kawanaka and H. Hosoya. biXid: a bidirectional transformation language for XML. In *International Conference on Functional Programming, Proceedings*, volume 41(9) of *SIGPLAN Notices*, pages 201–214. ACM, 2006. doi: 10.1145/1160074.1159830.

D. Laurent, J. Lechtenbörger, N. Spyratos, and G. Vossen. Monotonic complements for independent data warehouses. *The VLDB Journal*, 10(4):295–315, 2001. doi: 10.1007/s007780100055.

J. Lechtenbörger and G. Vossen. On the computation of relational view complements. *ACM Transactions on Database Systems*, 28(2):175–208, 2003. doi: 10.1145/777943.777946.

D. Lutterkort. Augeas—A configuration API. In *Linux Symposium, Proceedings*, pages 47–56, 2008.

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, Proceedings*, volume 42(9) of *SIGPLAN Notices*, pages 47–58. ACM, 2007. doi: 10.1145/1291220.1291162.

K. Matsuda, Z. Hu, and M. Takeichi. Type-based specialization of XML transformations. In *Partial Evaluation and Program Manipulation, Proceedings*, pages 61–72. ACM, 2009. doi: 10.1145/1480945.1480955.

L. Meertens. Designing constraint maintainers for user interaction, 1998. Manuscript, available from `ftp://ftp.kestrel.edu/pub/papers/meertens/dcm.ps`.

R.J. Miller, M.A. Hernandez, L.M. Haas, L. Yan, C.T.H. Ho, R. Fagin, and L. Popa. The Clio project: Managing heterogeneity. *SIGMOD Record*, 30(1): 78–83, 2001. doi: 10.1145/373626.373713.

H. Pacheco and A. Cunha. Generic point-free lenses. In *Mathematics of Program Construction, Proceedings*, volume 6120 of *LNCS*, pages 331–352. Springer, 2010. doi: 10.1007/978-3-642-13321-3_19.

H. Pacheco and A. Cunha. Calculating with lenses: Optimising bidirectional transformations. In *Partial Evaluation and Program Manipulation, Proceedings*, pages 91–100. ACM, 2011. doi: 10.1145/1929501.1929520.

K. Perumalla and R. Fujimoto. Source-code transformations for efficient reversibility. Technical Report GIT-CC-99-21, College of Computing, Georgia Tech, 1999.

N. Ramsey. Embedding an interpreted language using higher-order functions and types. In *Interpreters, Virtual Machines and Emulators, Proceedings*, pages 6–14. ACM, 2003. doi: 10.1145/858570.858571.

A. Schürr. Specification of graph translators with triple graph grammars. In *Graph-Theoretic Concepts in Computer Science 1994, Proceedings*, volume 903 of *LNCS*, pages 151–163. Springer, 1995. doi: 10.1007/3-540-59071-4_45.

P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Model Driven Engineering Languages and Systems, Proceedings*, volume 4735 of *LNCS*, pages 1–15. Springer, 2007. doi: 10.1007/978-3-540-75209-7_1.

J. Voigtländer. Bidirectionalization for free! In *Principles of Programming Languages, Proceedings*, volume 44(1) of *SIGPLAN Notices*, pages 165–176. ACM, 2009. doi: 10.1145/1594834.1480904.

J. Voigtländer, Z. Hu, K. Matsuda, and M. Wang. Combining syntactic and semantic bidirectionalization. In *International Conference on Functional Programming, Proceedings*, volume 45(9) of *SIGPLAN Notices*, pages 181–192. ACM, 2010. doi: 10.1145/1932681.1863571.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404.

P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990. doi: 10.1016/0304-3975(90)90147-A.

M. Wang, J. Gibbons, K. Matsuda, and Z. Hu. Gradual refinement: Blending pattern matching with data abstraction. In *Mathematics of Program Construction, Proceedings*, volume 6120 of *LNCS*, pages 397–425. Springer, 2010. doi: 10.1007/978-3-642-13321-3_22.

M. Wang, J. Gibbons, and N. Wu. Incremental updates for efficient bidirectional transformations. In *International Conference on Functional Programming, Proceedings*, volume 46(9) of *SIGPLAN Notices*, pages 392–403. ACM, 2011. doi: 10.1145/2034574.2034825.

Y. Xiong, D. Liu, Z. Hu, H. Zhao, M. Takeichi, and H. Mei. Towards automatic model synchronization from model transformations. In *Automated Software Engineering, Proceedings*, pages 164–173. ACM, 2007. doi: 10.1145/1321631.1321657.

A Container Implementations

This appendix presents suitable implementations of `shape`, `content`, and `decorate` using Data.Traversable:

```
shape :: Traversable κ ⇒ forall α. κ α → κ ()
shape = fmapDefault (const ())

content :: Traversable κ ⇒ forall α. κ α → [α]
content = foldMapDefault (λa → [a])

decorate :: Traversable κ ⇒ forall α. forall β. [α] → κ β → κ α
decorate l t = case State.runState (unwrapMonad (traverse f t)) l
                 of (t', []) → t'
  where f _ = WrapMonad (do (n : ns) ← State.get
                            State.put ns
                            return n)
```

In addition to the Data.Traversable module (for Traversable/`traverse` itself, but also the `fmapDefault` and `foldMapDefault` functions), these definitions use several (types and) functions from the Control.Monad.State (for the State.`get`, State.`put`, and State.`runState` functions) and Control.Applicative modules (for the data constructor WrapMonad and the function `unwrapMonad`).

That (5) and (6) hold for the above implementations relies on laws put forward by Gibbons and Oliveira (2009, Sections 5.2 and 5.3), about sequential and parallel composition of traversals.