Bidirectionalization for Free! *(Pearl)*

Janis Voigtländer

Technische Universität Dresden
01062 Dresden, Germany
janis.voigtlaender@acm.org

Abstract

A bidirectional transformation consists of a function get that takes a source (document or value) to a view and a function put that takes an updated view and the original source back to an updated source, governed by certain consistency conditions relating the two functions. Both the database and programming language communities have studied techniques that essentially allow a user to specify only one of get and put and have the other inferred automatically. All approaches so far to this bidirectionalization task have been syntactic in nature, either proposing a domain-specific language with limited expressiveness but built-in (and composable) backward components, or restricting get to a simple syntactic form from which some algorithm can synthesize an appropriate definition for put. Here we present a semantic approach instead. The idea is to take a general-purpose language, Haskell, and write a higher-order function that takes (polymorphic) get-functions as arguments and returns appropriate put-functions. All this on the level of semantic values, without being willing, or even able, to inspect the definition of get, and thus liberated from syntactic restraints. Our solution is inspired by relational parametricity and uses free theorems for proving the consistency conditions. It works beautifully.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures, Polymorphism; D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.2.4 [*Software Engineering*]: Software/Program Verification—Correctness proofs; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; H.2.3 [*Database Management*]: Languages—Data manipulation languages, Query languages

General Terms Design, Languages, Verification

Keywords generic programming, program transformation, view-update problem

1. Introduction

Imagine we have written the following Haskell function:

$$\begin{array}{l} halve :: [\alpha] \to [\alpha] \\ halve\ as = take\ (length\ as\ \text{`}div\text{`}\ 2)\ as \end{array}$$

Clearly, it outputs only an abstraction of its input list, as that list's second half is omitted. Now assume this abstracted value, or view,

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

is updated in some way, and we would like to propagate this update back to the original input list. Here is how to do so:

$$\begin{array}{l}
put_1 :: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha] \\
put_1\ as\ as' \mid length\ as' == n \\
\qquad\qquad\quad = as' +\!\!+\ drop\ n\ as \\
\qquad\qquad\qquad \mathbf{where}\ n = length\ as\ \text{`}div\text{`}\ 2
\end{array}$$

Note that the backwards propagation of the assumed updated view as' into the original source as is only possible if as' is itself also half as long as as. This is so because otherwise there is no consistent way to combine as' and the second half of as into an updated source from which $halve$ would indeed lead to as'.

Let us consider another example:

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

$$\begin{array}{l}
flatten :: \mathsf{Tree}\ \alpha \rightarrow [\alpha] \\
flatten\ (\mathsf{Leaf}\ a) \qquad = [a] \\
flatten\ (\mathsf{Node}\ t_1\ t_2) = flatten\ t_1 +\!\!+\ flatten\ t_2
\end{array}$$

Now the abstraction amounts to forgetting the tree structure of the input source. But if the list view is updated in any way preserving its length, the new content can be propagated back into the original tree as follows:

$$\begin{array}{l}
put_2 :: \mathsf{Tree}\ \alpha \rightarrow [\alpha] \rightarrow \mathsf{Tree}\ \alpha \\
put_2\ s\ v = \mathbf{case}\ go\ s\ v\ \mathbf{of}\ (t, [\,]) \rightarrow t \\
\ \ \mathbf{where}\ go\ (\mathsf{Leaf}\ a) \qquad (b : bs) = (\mathsf{Leaf}\ b, bs)
\end{array}$$

$$
\begin{aligned}
&go\ (\mathsf{Node}\ s_1\ s_2)\ bs \qquad = (\mathsf{Node}\ t_1\ t_2, ds)\\
&\quad \mathbf{where}\ (t_1, cs) = go\ s_1\ bs\\
&\qquad\qquad\ (t_2, ds) = go\ s_2\ cs
\end{aligned}
$$

Finally, consider a function that removes duplicate occurrences of elements from a list, with implementation taken over from a standard library:

$$
\begin{aligned}
&rmdups :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]\\
&rmdups = \mathsf{List}.nub
\end{aligned}
$$

An appropriate backwards propagation function looks as follows:

$$
\begin{aligned}
&put_3 :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]\\
&put_3\ s\ v\ \mid\ v == \mathsf{List}.nub\ v\ \ \&\&\ \ length\ v == length\ s'\\
&\qquad\quad = map\ (fromJust \circ flip\ lookup\ (zip\ s'\ v))\ s\\
&\qquad\qquad \mathbf{where}\ s' = \mathsf{List}.nub\ s
\end{aligned}
$$

For example, in a Haskell interpreter:

```
> put_3 "abcbabcbaccba" "aBc"
"aBcBaBcBaccBa"
```

Clearly, always having to explicitly write both forwards/backwards-related functions is not the ideal situation. Thus, there has been a lot of recent research into bidirectionalization (Hu et al. 2004; Bohannon et al. 2006; Foster et al. 2007; Matsuda et al. 2007; Bohannon et al. 2008; Foster et al. 2008). One approach is to design a domain-specific language, fencing in a certain subclass of transformations, in which a single specification denotes both a forward and a backward function. Another approach is to devise an algorithm that works on a syntactic representation of (restricted) forward functions and tries to find the missing backward components. In this paper we present a completely novel approach that works for polymorphic functions such as those above. We write, directly in Haskell, a higher-order function bff (named for an abbreviation of the paper's title). This function takes a source-to-view function as input and returns an appropriate backward function. For example, we expect, and will get,

$$\begin{aligned} \mathit{bff}\ \mathit{halve} &\equiv \mathit{put}_1 \\ \mathit{bff}\ \mathit{flatten} &\equiv \mathit{put}_2 \\ \mathit{bff}\ \mathit{rmdups} &\equiv \mathit{put}_3\,. \end{aligned}$$

Note that applying bff to halve, for example, will not return the exact syntactic definition of put_1 given above, but merely a functional value that is semantically equivalent to it. Hence the use of $\equiv$ instead of $=$ here. But this is absolutely enough from an application perspective. We want automatic bidirectionalization precisely because we do not want to be bothered with thinking about the backward function. So we do not care about its syntactic form, as long as the function serves its purpose. And the same level of syntactic ignorance applied to the input, rather than output, side of bff means that we can pass any Haskell function of appropriate type and obtain a good backward component for it. We are not restricted to drawing forward functions from some sublanguage only.

Of course, the concept of a "good backward component" needs to be addressed. As evaluation criteria we use the standard consistency conditions (Bancilhon and Spyratos 1981) that a forward/backward pair of functions get/put should satisfy the laws

$$put\ s\ (get\ s) \equiv s$$

and, if $put\ s\ v$ is defined,

$$get\ (put\ s\ v) \equiv v\,,$$

known as GetPut and PutGet, respectively. These consistency conditions are why all the put-functions given above are partial functions only. For example,

```
> put_3 "abcbabcbaccba" "aBB"
```

should, and does, fail, because a view with duplicate elements can never be in the image of $rmdups$. An alternative that is used in some of the related literature would be to statically describe, or even calculate, the domain on which a put-function is well-defined, thus capturing a notion of permitted updates. We have not yet investigated whether this way of recovering totality is possible for our purely semantic approach to bidirectionalization.

Even so, a natural question is how often a put-function obtained via bff will be undefined on some input. For example, a trivial put-function that is undefined whenever the v in $put\ s\ v$ is not equal to $get\ s$ would satisfy the GetPut and PutGet laws, but is clearly undesirable in practice. Our approach usually does better than that, but one significant limitation that it has in its current state is that any update that changes the "shape" of a view, say the length of a list, will lead to failure. Further discussion is contained in Section 7.

Instead of a single function bff, we will actually give three functions bff, bff_{Eq}, and bff_{Ord}, the choice from which depends on whether the source-to-view functions to be handled may involve equality and/or ordering tests on the elements contained in the data structures to be transformed. This reflects that for a function like $rmdups$ conceptually more involved conditions are required for safe bidirectionalization than for $halve$ or $tail$, or any other function of type $[\alpha] \to [\alpha]$ without an Eq-constraint. So $\mathit{bff}_{\mathsf{Eq}}$ will be used for $rmdups$ and its like, and $\mathit{bff}_{\mathsf{Ord}}$ for functions like the following one:

$$
\begin{aligned}
&top3 :: \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha] \\
&top3 = take\ 3 \circ \mathsf{List}.sort \circ \mathsf{List}.nub
\end{aligned}
$$

But it is indeed the case that the single function bff applies to both $halve$ and $flatten$, even though the former only deals with lists while the latter also involves trees. That is, bff, as well as $\mathit{bff}_{\mathsf{Eq}}$ and $\mathit{bff}_{\mathsf{Ord}}$, will be generic over both input and output structures. To get a rough idea of what kind of structures will be in reach, think of containers: shape plus data content (Abbott et al. 2003).

Proving that for any get a put obtained as $\mathit{bff}\ get$, $\mathit{bff}_{\mathsf{Eq}}\ get$, or $\mathit{bff}_{\mathsf{Ord}}\ get$ satisfies the GetPut and PutGet laws will be done by using free theorems (Reynolds 1983; Wadler 1989). Our formal reasoning there will be "morally correct" in the sense of Danielsson et al. (2006). That is, our proofs of the GetPut and PutGet laws will apply to total get-functions and total, finite data structures. In particular, we do not take into account Haskell intricacies like those studied by Johann and Voigtländer (2004). This simplification is done solely for the sake of exposition, not because of any fundamental problems with doing elsewise.

All code in this paper was developed and tested using the Glasgow Haskell Compiler 6.8.2. The final, generic version of the code is available as Hackage (`http://hackage.haskell.org`) package `bff-0.1`. An online tool based on it is also available at `http://linux.tcs.inf.tu-dresden.de/~bff/cgi-bin/bff.cgi`. Throughout, we sometimes use common Haskell functions and types without further comment, like *fromJust*, *lookup* (and thus Maybe), and *zip* above. Where these are not clear, Hoogle (`http://haskell.org/hoogle`) has the answer.

2. Getting Started

We first deal only with lists as both input and output structures, aiming at a bidirectionalizer of type

$$bff :: (\forall\alpha.[\alpha] \to [\alpha]) \to (\forall\alpha.[\alpha] \to [\alpha] \to [\alpha])\,.$$

Note that the local universal quantifications over α are essential here, and require compiler flag `-XRank2Types`.

Now, how can *bff* possibly learn anything about its input function, so as to exploit that information for producing a good backward function? The idea is to use the assumption that the input function *get* is polymorphic over the element type α. This entails that its behavior does not depend on any concrete list elements, but only on positional information. And 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 *bff* to see that the head element of the original source 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. And this observation can be transferred to other source lists than $[0..n]$ just as well, even to lists over non-integer types, thanks to parametric polymorphism (Strachey 1967; Reynolds 1983).

Let us develop this line of reasoning further, still on the $tail$ example. So $\mathit{bff}\ tail$ is supposed to return a good put. To do so, it must determine what this put should do when given an original source s and an updated view v. First, it would be good to find out to what element in s each element in v corresponds. Assume s has length $n + 1$. Then by applying $tail$ to the same-length list $[0..n]$, bff (or, rather, $\mathit{bff}\ tail \equiv put$) learns that the original view from which v was obtained by updating had length n, and also to what element in s each element in that original view corresponded. Being conservative, we will only accept v if it has retained that length n. For then, we also know directly the associations between elements in v and positions in the original source. Now, to produce the updated source, we can go over all positions in $[0..n]$ and fill

$$
\begin{array}{ll}
\mathit{fromAscList} & :: [(\mathsf{Int}, \alpha)] \rightarrow \mathsf{IntMap}\,\alpha \\
\mathit{empty} & :: \mathsf{IntMap}\,\alpha \\
\mathit{notMember} & :: \mathsf{Int} \rightarrow \mathsf{IntMap}\,\alpha \rightarrow \mathsf{Bool} \\
\mathit{insert} & :: \mathsf{Int} \rightarrow \alpha \rightarrow \mathsf{IntMap}\,\alpha \rightarrow \mathsf{IntMap}\,\alpha \\
\mathit{union} & :: \mathsf{IntMap}\,\alpha \rightarrow \mathsf{IntMap}\,\alpha \rightarrow \mathsf{IntMap}\,\alpha \\
\mathit{lookup} & :: \mathsf{Int} \rightarrow \mathsf{IntMap}\,\alpha \rightarrow \mathsf{Maybe}\,\alpha
\end{array}
$$

Figure 1. Functions from module Data.IntMap.

them with the associated values from v. For positions for which there is no corresponding value in v, because these positions were omitted when applying tail to $[0..n]$, we can look up the correct value in s rather than in v. For the concrete example, this will only concern position 0, for which we naturally take over the head element from s.

The same strategy works also for general $\mathit{bff}\ \mathit{get}$. In short, given s, produce a kind of template $s' = [0..n]$ of the same length, together with an association g between integer values in that template and the corresponding values in s. Then apply get to s' and produce a further association h by matching this template view versus the updated proper value view v. Combine the two associations into a single one h', giving precedence to h whenever an integer template index is found in both h and g. Thus, it is guaranteed that we will only resort to values from the original source s when the corresponding position did not make it into the view, and thus there is no way how it could have been affected by the update. Finally, produce an updated source by filling all positions in $[0..n]$ with their associated values according to h'. For maintaining the associations between integer values and values from s and v, we use the standard library $\mathsf{Data.IntMap}$. Concretely, we import from it the functions given in Figure 1. Their names and type signatures should be enough documentation here, the only necessary additions being that $\mathsf{IntMap}.\mathit{fromAscList}$ expects a list with integer keys in ascending order and that $\mathsf{IntMap}.\mathit{union}$ is left-biased for integers occurring as keys in both input maps. The latter will precisely realize the desired precedence of h over g. The described strategy is now easily implemented as follows:

$$
\begin{array}{l}
\mathit{bff} :: (\forall\alpha.[\alpha] \to [\alpha]) \to (\forall\alpha.[\alpha] \to [\alpha] \to [\alpha]) \\
\mathit{bff}\ \mathit{get} = \lambda s\ v \to \\
\quad \mathbf{let}\ \begin{array}[t]{l@{\ }l}
s' & = [0..\mathit{length}\ s - 1] \\
g & = \mathsf{IntMap}.\mathit{fromAscList}\ (\mathit{zip}\ s'\ s) \\
h & = \mathit{assoc}\ (\mathit{get}\ s')\ v \\
h' & = \mathsf{IntMap}.\mathit{union}\ h\ g
\end{array} \\
\quad \mathbf{in}\ \ \mathit{map}\ (\mathit{fromJust} \circ \mathit{flip}\ \mathsf{IntMap}.\mathit{lookup}\ h')\ s'
\end{array}
$$

$$
\begin{array}{l}
\mathit{assoc} :: [\mathsf{Int}] \to [\alpha] \to \mathsf{IntMap}\ \alpha \\
\mathit{assoc}\ [\,]\qquad\quad [\,]\qquad\ \ = \mathsf{IntMap}.\mathit{empty} \\
\mathit{assoc}\ (i : \mathit{is})\ (b : \mathit{bs})\ \mid\ \mathsf{IntMap}.\mathit{notMember}\ i\ m \\
\qquad\qquad\qquad\qquad\qquad\ = \mathsf{IntMap}.\mathit{insert}\ i\ b\ m \\
\qquad\qquad\qquad\qquad\qquad\quad\ \mathbf{where}\ m = \mathit{assoc}\ \mathit{is}\ \mathit{bs}
\end{array}
$$

Note that the function assoc, realizing the matching between the template view and the updated proper value view, needs to check that no index position is encountered twice, because otherwise it would not (yet) be clear how to deal with two potentially different update values.

Our current version of bff works quite nicely already. For example,

```
> bff tail "abcd" "bCd"
```

```
"abCd"
```

and for

$$
\begin{aligned}
&sieve :: [\alpha] \to [\alpha] \\
&sieve\ (a : b : cs) = b : sieve\ cs \\
&sieve\ \_ \qquad\qquad\quad\ = [\,]
\end{aligned}
$$

we automatically get

```
> bff sieve "abcdefg" "123"
"a1c2e3g"
```

(Note that $sieve$ "abcdefg" $\equiv$ "bdf".)

However, ultimately the current version is too weak. It fails as soon as a source-to-view function duplicates a list element. For example,

```
> bff (\s -> s ++ s) "a" "aa"
```

fails, defeating the GetPut law. (Note that the GetPut law would demand that $\mathit{bff}\ (\lambda s \to s +\!\!+\, s)$ "a" $((\lambda s \to s +\!\!+\, s)$ "a"$)$ $\equiv$ "a".) And also, a bit more subtly, the PutGet law is violated for empty source lists:

```
> bff halve "" "a"
""
```

(Note that the PutGet law would demand that, if $\mathit{bff}\ halve$ "" "a" is defined, $halve\ (\mathit{bff}\ halve$ "" "a"$)$ $\equiv$ "a", but it is not the case that $halve$ "" $\equiv$ "a".)

On the other hand, apart from this empty list weirdness we truly have $\mathit{bff}\ halve \equiv put_1$. So it seems we have made a good start, on which to extend in the next section.

3. Correct Bidirectionalization

In order to fix bff to adhere to the GetPut law, we need to deal with duplication of list elements. Consider again the source-to-view function $\lambda s \rightarrow s +\!\!+ s$. Applied to a template $[0..n]$, it will deliver the template view $[0, \ldots, n, 0, \ldots, n]$. Under what conditions should a match between this template view and an updated proper value view be considered successful? Clearly only when equal indices match up with equal values, because only then we can produce a meaningful association reflecting a legal update.

However, equality tests are not possible in Haskell at arbitrary types. So we will have to weaken the type of bff as follows:

$$\mathit{bff} :: (\forall \alpha. [\alpha] \rightarrow [\alpha]) \rightarrow (\forall \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha] \rightarrow [\alpha])$$

That is, the get-function given to bff will still (have to) be fully polymorphic, but the returned put-function will only be applicable to lists over an element type satisfying the Eq-constraint. This is not expected to cause any problems in practice, because application scenarios for view-update will typically involve data domains for which equality tests are naturally available (as opposed to, say, operating on lists of functions). And in any case, we could always recover the law-wise weaker but also type-wise slightly wider applicable version of bff from the previous section by simply defining bogus instances of Eq where the equality test $==$ invariably returns False.

Armed with equality tests, we can rewrite the function $assoc$ as follows. We also take the opportunity to introduce more useful error signaling than pattern-match errors as implicitly used before.

$$assoc :: \mathsf{Eq}\ \alpha \Rightarrow [\mathsf{Int}] \rightarrow [\alpha] \rightarrow \mathsf{Either\ String}\ (\mathsf{IntMap}\ \alpha)$$

$$
\begin{array}{lll}
\mathit{assoc}\ [\,] & [\,] & = \mathsf{Right}\ \mathsf{IntMap}.\mathit{empty} \\
\mathit{assoc}\ (i : \mathit{is}) & (b : \mathit{bs}) & = \mathit{either}\ \mathsf{Left}\ (\mathit{checkInsert}\ i\ b) \\
 & & \qquad\qquad\qquad\quad (\mathit{assoc}\ \mathit{is}\ \mathit{bs}) \\
\mathit{assoc}\ \_ & \_ & = \mathsf{Left}\ \text{“Update changes the length.”}
\end{array}
$$

$$
\begin{array}{l}
\mathit{checkInsert} :: \mathsf{Eq}\ \alpha \Rightarrow \mathsf{Int} \rightarrow \alpha \rightarrow \mathsf{IntMap}\ \alpha \\
\qquad\qquad\qquad\qquad\ \rightarrow \mathsf{Either}\ \mathsf{String}\ (\mathsf{IntMap}\ \alpha) \\
\mathit{checkInsert}\ i\ b\ m = \\
\ \ \mathbf{case}\ \mathsf{IntMap}.\mathit{lookup}\ i\ m\ \mathbf{of} \\
\quad \begin{array}{ll}
\mathsf{Nothing} & \rightarrow \mathsf{Right}\ (\mathsf{IntMap}.\mathit{insert}\ i\ b\ m) \\
\mathsf{Just}\ c & \rightarrow \mathbf{if}\ b == c \\
 & \quad\ \ \mathbf{then}\ \mathsf{Right}\ m \\
 & \quad\ \ \mathbf{else}\ \ \mathsf{Left}\ \text{“Update violates equality.”}
\end{array}
\end{array}
$$

From now on, we assume that every instance of Eq gives a definition for $==$ that makes it reflexive, symmetric, and transitive. Then, the following two lemmas hold.

Lemma 1. *For every $is :: [\mathsf{Int}]$, type τ that is an instance of Eq, and $f :: \mathsf{Int} \to \tau$, we have*

$$assoc\ is\ (map\ f\ is) \equiv \mathsf{Right}\ h$$

for some $h :: \mathsf{IntMap}\ \tau$ with

$$\mathsf{IntMap}.lookup\ i\ h \equiv \mathbf{if}\ elem\ i\ is\ \mathbf{then}\ \mathsf{Just}\ (f\ i)\ \mathbf{else}\ \mathsf{Nothing}$$

for every $i :: \mathsf{Int}$.

Lemma 2. *Let $is :: [\mathsf{Int}]$, let τ be a type that is an instance of Eq, and let $v :: [\tau]$ and $h :: \mathsf{IntMap}\ \tau$. We have that if*

$$\mathsf{Right}\ h \equiv assoc\ is\ v\,,$$

then

$$map\ (flip\ \mathsf{IntMap}.lookup\ h)\ is == map\ \mathsf{Just}\ v\,.$$

We do not explicitly prove either of the two lemmas here. Both are easily established by induction on the list is, taking the specifications of functions in $\mathsf{Data.IntMap}$ into account. Note that in the conclusion of Lemma 2 we cannot simply replace $==$ by $\equiv$, because the instance of Eq for τ may very well give $x == y$ for some $x \not\equiv y$. We will continue to be careful about this distinction in what follows. Of course, the instances of Eq used in practice will often have $==$ agree with semantic equivalence (such as for integers, characters, strings, ...).

The improved version of $assoc$ can now be used for an improved version of *bff* as follows:

$$
\begin{array}{l}
\mathit{bff} :: (\forall\alpha.[\alpha] \to [\alpha]) \to (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha]) \\
\mathit{bff}\ \mathit{get} = \lambda s\ v \to \\
\quad \mathbf{let}\ s' \ = [0..\mathit{length}\ s - 1] \\
\qquad\quad g\ \ = \mathsf{IntMap}.\mathit{fromAscList}\ (\mathit{zip}\ s'\ s) \\
\qquad\quad h\ \ = \mathit{either}\ \mathit{error}\ \mathit{id}\ (\mathit{assoc}\ (\mathit{get}\ s')\ v) \\
\qquad\quad h' = \mathsf{IntMap}.\mathit{union}\ h\ g \\
\quad \mathbf{in}\ \ \mathit{seq}\ h\ (\mathit{map}\ (\mathit{fromJust} \circ \mathit{flip}\ \mathsf{IntMap}.\mathit{lookup}\ h')\ s')
\end{array}
$$

Note that the use of *error* turns a potential failure in *assoc* (or, via *assoc*, in *checkInsert*) into an explicit runtime error with meaningful error message. The use of *seq* prevents such an error going unnoticed in the case that s, and thus s', is the empty list. (This solves the problem with the PutGet law observed at the end of Section 2.) Instead of the polymorphic strict evaluation primitive we could also have used an emptiness test or any other strict operation on h.

The new version of *bff* now does not only work for *halve*, *tail*, *sieve*, and the like, but also for *get*-functions that duplicate list elements. For example,

```
> bff (\s -> s ++ s) "a" "aa"
"a"
> bff (\s -> s ++ s) "a" "bb"
"b"
> bff (\s -> s ++ s) "a" "ab"
"*** Exception: Update violates equality.
```

Formally, we establish the GetPut and PutGet laws as follows.

Theorem 1. *For every function $get :: \forall\alpha.[\alpha] \to [\alpha]$, type τ that is an instance of* Eq, *and $s :: [\tau]$, we have*

$$bff\ get\ s\ (get\ s) \equiv s\,.$$

Proof. By the function definition for bff we have

$$\begin{array}{c} bff\ get\ s\ (get\ s) \\ \equiv \\ seq\ h\ (map\ (fromJust \circ flip\ \mathsf{IntMap}.lookup\ h')\ s')\,, \end{array} \tag{1}$$

where:

$$s' \equiv [0..length\ s - 1] \tag{2}$$

$$g \equiv \mathsf{IntMap}.fromAscList\ (zip\ s'\ s) \tag{3}$$

$$h \equiv either\ error\ id\ (assoc\ (get\ s')\ (get\ s)) \tag{4}$$

$$h' \equiv \mathsf{IntMap}.union\ h\ g\,. \tag{5}$$

Clearly, (2) implies

$$s \equiv map\ (s\,!!)\ s'\,, \tag{6}$$

where the operator !! is used for extracting a list element at a given index position. Thus,

$$get\ s \equiv get\ (map\ (s\,!!)\ s')\,.$$

By a free theorem of Wadler (1989), every $get :: \forall\alpha.[\alpha] \to [\alpha]$ satisfies

$$get \circ map\ f \equiv map\ f \circ get$$

for every choice of f. Thus, in particular,

$$get\ s \equiv map\ (s\,!!)\ (get\ s')\,.$$

Together with (4) and Lemma 1, this gives that h is defined (i.e., not a runtime error) and that for every $i :: \mathsf{Int}$,

$$\begin{aligned}\mathsf{IntMap}.lookup\ i\ h \equiv \mathbf{if}\ elem\ i\ (get\ s')\ &\mathbf{then}\ \mathsf{Just}\ (s\,!!\,i)\\ &\mathbf{else}\ \ \mathsf{Nothing}\,.\end{aligned}$$

Since by (2), (3), and the specification of $\mathsf{IntMap}.fromAscList$, for every $i :: \mathsf{Int}$,

$$\begin{aligned}\mathsf{IntMap}.lookup\ i\ g \equiv \mathbf{if}\ elem\ i\ s'\ &\mathbf{then}\ \mathsf{Just}\ (s\,!!\,i)\\ &\mathbf{else}\ \ \mathsf{Nothing}\,,\end{aligned}$$

we have by (5) and the specification of $\mathsf{IntMap}.union$ that for every $i :: \mathsf{Int}$,

$$\begin{aligned}\mathsf{IntMap}.lookup\ i\ h' \equiv \mathbf{if}\ &elem\ i\ (get\ s')\\ &\mathbf{then}\ \mathsf{Just}\ (s\,!!\,i)\\ &\mathbf{else}\ \ \mathbf{if}\ elem\ i\ s'\ \mathbf{then}\ \mathsf{Just}\ (s\,!!\,i)\\ &\qquad\qquad\qquad\ \ \mathbf{else}\ \ \mathsf{Nothing}\,.\end{aligned}$$

Together with (1), the definedness of h, and (6), this gives the claim.

Theorem 2. *Let $get :: \forall\alpha.[\alpha] \to [\alpha]$, let τ be a type that is an instance of* Eq, *and let $v, s :: [\tau]$. We have that if $\mathit{bff}\ get\ s\ v$ is defined, then*

$$get\ (\mathit{bff}\ get\ s\ v) == v\,.$$

The proof of this second theorem, relying on Lemma 2 and using a similar style of reasoning as above, is given in Appendix A. Note that the theorem establishes the PutGet law only up to $==$, rather than for true semantic equivalence. As mentioned earlier, in practice $==$ will typically agree with $\equiv$ for the types of data under consideration, so this is no big issue.

4. Source-to-View Functions with Equality Tests

In the previous section we already used Eq-constraints for delivering good put-functions. On the other hand, the get-functions taken as input had to be fully polymorphic, and for good reason. Tempting as it may be to simply change the type of bff to

$$\begin{aligned} \mathit{bff} :: {} & (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]) \\ & \to (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])\,, \end{aligned}$$

so that it would also accept get-functions like the $rmdups ::$ $\mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ from the introduction, this would be inviting disaster:

```
> bff rmdups "abcbabcbaccba" "aBc"
"*** Exception: Update changes the length.
```

```
> bff rmdups "abcbabcbaccba" "abc"
"*** Exception: Update changes the length.
> bff rmdups "abc" "aaa"
"aaa"
> bff rmdups "aaa" "abc"
"abc"
```

All four experiments disagree with our expectations. For example, since $rmdups$ "abcbabcbaccba" $\equiv$ "abc", we would have expected in the first experiment that a view update into "aBc" leads to an update of the source into "aBcBaBcBaccBa". But instead, $bff\ rmdups$ fails. In the second experiment, where the view "abc" has not even been changed at all, we would have expected that $bff\ rmdups$ returns the original source "abcbabcbaccba". After all, that is what the GetPut law demands. But it does not happen. Similarly, the third experiment violates the PutGet law.

The main reason for failure here is that it is not necessarily true that one can always understand the behavior of a function $get :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$ on a source list s by simply observing its behavior on the template list $[0..n]$ of the same length. For this would completely lose track of potentially duplicated elements in s and how get might react to them. Note that this issue is nonexistent in Section 3, because a fully polymorphic function $get :: [\alpha] \rightarrow [\alpha]$ is *unable* to react to duplicated elements, as it cannot even detect them. Since here this is different, the first step towards a solution is a more intelligent template manufacture. For example, instead of $[0..12]$ the template for "abcbabcbaccba" should be $[0, 1, 2, 1, 0, 1, 2, 1, 0, 2, 2, 1, 0]$, together with an association of 0 to 'a', 1 to 'b', and 2 to 'c'. In writing a function to do this job, one needs to keep track of which elements have already been seen while going through the source list. Thus, it makes sense to use a state monad (Wadler 1992). And since for every element already seen one needs to be able to determine the template integer value to which it has been associated, it makes sense to extend the IntMap abstraction with a facility for "backwards" lookup. We have implemented such a new abstraction, with API as given in Figure 2. Of immediate interest here are only the functions IntMapEq.$empty$, IntMapEq.$insert$, and IntMapEq.$lookupR$. Using them, we obtain the following piece of code. The state that is carried around consists of an IntMapEq containing the elements that have already been encountered and an integer denoting the next available key. The function $number_{\mathsf{Eq}}$ describes the action to be performed for every element found in a source list, and by which integer key to replace it in the template list.

$$
\begin{array}{l}
template_{\mathsf{Eq}} :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \rightarrow ([\mathsf{Int}], \mathsf{IntMapEq}\ \alpha) \\
template_{\mathsf{Eq}}\ s = \mathbf{case}\ runState\ (go\ s)\ (\mathsf{IntMapEq}.empty, 0) \\
\qquad\qquad\qquad\quad \mathbf{of}\quad (s', (g, \_)) \rightarrow (s', g) \\
\quad \mathbf{where}\ go\ [\,] \qquad\quad = return\ [\,] \\
\qquad\qquad\ \ go\ (a : as) = \mathbf{do}\ i \leftarrow number_{\mathsf{Eq}}\ a \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ is \leftarrow go\ as \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ return\ (i : is)
\end{array}
$$

$$
\begin{array}{l}
number_{\mathsf{Eq}} :: \mathsf{Eq}\ \alpha \Rightarrow \alpha \rightarrow \mathsf{State}\ (\mathsf{IntMapEq}\ \alpha, \mathsf{Int})\ \mathsf{Int} \\
number_{\mathsf{Eq}}\ a = \\
\quad \mathbf{do}\ (m, i) \leftarrow \mathsf{State}.get \\
\qquad\quad \mathbf{case}\ \mathsf{IntMapEq}.lookupR\ a\ m\ \mathbf{of} \\
\qquad\qquad \mathsf{Just}\ j \quad\ \rightarrow return\ j \\
\qquad\qquad \mathsf{Nothing} \rightarrow \mathbf{do}\ \mathbf{let}\ m' = \mathsf{IntMapEq}.insert\ i\ a\ m \\
\qquad\qquad\qquad\qquad\qquad\quad \mathsf{State}.put\ (m', i + 1)
\end{array}
$$

$$
\begin{array}{ll}
 & \quad\quad\quad \textit{return } i \\
\textit{empty} & :: \mathsf{IntMapEq}\ \alpha \\
\textit{insert} & :: \mathsf{Int} \to \alpha \to \mathsf{IntMapEq}\ \alpha \to \mathsf{IntMapEq}\ \alpha \\
\textit{checkInsert} & :: \mathsf{Eq}\ \alpha \Rightarrow \mathsf{Int} \to \alpha \to \mathsf{IntMapEq}\ \alpha \\
 & \qquad\qquad \to \mathsf{Either\ String}\ (\mathsf{IntMapEq}\ \alpha) \\
\textit{union} & :: \mathsf{Eq}\ \alpha \Rightarrow \mathsf{IntMapEq}\ \alpha \to \mathsf{IntMapEq}\ \alpha \\
 & \qquad\qquad \to \mathsf{Either\ String}\ (\mathsf{IntMapEq}\ \alpha) \\
\textit{lookup} & :: \mathsf{Int} \to \mathsf{IntMapEq}\ \alpha \to \mathsf{Maybe}\ \alpha \\
\textit{lookupR} & :: \mathsf{Eq}\ \alpha \Rightarrow \alpha \to \mathsf{IntMapEq}\ \alpha \to \mathsf{Maybe\ Int}
\end{array}
$$

Figure 2. Functions from module IntMapEq.

Then, for example,

```
> template_Eq "transformation"
([0,1,2,3,4,5,6,1,7,2,0,8,6,3],fromList [(0,'t'),(
1,'r'),(2,'a'),(3,'n'),(4,'s'),(5,'f'),(6,'o'),(7,
'm'),(8,'i')])
```

More generally, the following lemma holds.

Lemma 3. *Let* τ *be a type that is an instance of* Eq *and let* $s :: [\tau]$, $s' :: [\mathsf{Int}]$, *and* $g :: \mathsf{IntMapEq}\ \tau$. *We have that if*

$$(s', g) \equiv \textit{template}_{\mathsf{Eq}}\ s\,,$$

then

- $\textit{map}\ (\textit{flip}\ \mathsf{IntMapEq}.\textit{lookup}\ g)\ s' == \textit{map}\ \mathsf{Just}\ s\,,$
- *for every* $i :: \mathsf{Int}$ *not in* s', $\mathsf{IntMapEq}.\textit{lookup}\ i\ g \equiv \mathsf{Nothing}\,,$
- *for every* $i \mathrel{/=} j$ *in* s',

$$\mathsf{IntMapEq}.\mathit{lookup}\ i\ g \mathrel{/{=}} \mathsf{IntMapEq}.\mathit{lookup}\ j\ g\,.$$

Here $/=$ is the complement of $==$. Again we refrain from giving an explicit proof of this auxiliary lemma. It is quite similar to an example of Hutton and Fulger (2008), and we have nothing conceptually new to contribute right here regarding proof techniques.

The final statement in Lemma 3, about different integers being mapped to different (according to $/=$ at type τ) values by g is very essential. The proofs of both Theorems 1 and 2 use the free theorem $get \circ map\ f \equiv map\ f \circ get$. But that was for $get :: [\alpha] \to [\alpha]$. For the $get :: \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$ of interest now, we know from Wadler (1989, Section 3.4) that f cannot be arbitrary anymore. Rather, it must respect Eq in the sense that $x == y$ if and only if $f\ x == f\ y$. And since ultimately the f for which we will want to apply the free theorem are connected to g and later h', we need an injectivity invariant for the $\mathsf{IntMapEq}$s under use. This is why both $\mathsf{IntMapEq}.\mathit{checkInsert}$ and $\mathsf{IntMapEq}.\mathit{union}$ have $\mathsf{Either\ String}\ (\mathsf{IntMapEq}\ \alpha)$ as return type in Figure 2, so that they can give a meaningful error message in case of a violation of this invariant. The $\mathsf{IntMapEq}.\mathit{insert}$ used in $\mathit{number}_{\mathsf{Eq}}$, on the other hand, has no such safeguards. But Lemma 3 tells us that everything is still okay with $\mathit{template}_{\mathsf{Eq}}$.

Of course, we also need to adapt assoc, but only slightly. Basically, we just switch from operations on IntMaps to operations on $\mathsf{IntMapEq}$s, the most important change being that $\mathsf{IntMapEq}.\mathit{checkInsert}$ does not only prevent insertion of two different update values for the same integer key, but does also prevent insertion of equal update values for different integer keys (so as to prevent the $\mathit{bff}\ \mathit{rmdups}$ “abc” “aaa” $\equiv$ “aaa” disaster with its violation of the PutGet law). The variant of assoc to use is then as follows:

$$
\begin{array}{l}
assoc_{\mathsf{Eq}} :: \mathsf{Eq}\ \alpha \Rightarrow [\mathsf{Int}] \to [\alpha] \to \mathsf{Either\ String}\ (\mathsf{IntMapEq}\ \alpha) \\
assoc_{\mathsf{Eq}}\ [\,]\qquad\quad [\,]\qquad\quad = \mathsf{Right\ IntMapEq}.empty \\
assoc_{\mathsf{Eq}}\ (i : is)\ (b : bs) = either\ \mathsf{Left} \\
\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{IntMapEq}.checkInsert\ i\ b) \\
\qquad\qquad\qquad\qquad\qquad\qquad (assoc_{\mathsf{Eq}}\ is\ bs) \\
assoc_{\mathsf{Eq}}\ \_\qquad\quad \_\qquad\quad = \mathsf{Left}\ \text{“Update changes the length.”}
\end{array}
$$

For it, we claim the following two lemmas. The notion of a function $f :: \mathsf{Int} \to \tau$, for a type τ that is an instance of Eq, being injective on a list $is :: [\mathsf{Int}]$ is defined as "for every $i \mathrel{/=} j$ in is, also $f\ i \mathrel{/=} f\ j$".

Lemma 4. *Let* $is :: [\mathsf{Int}]$*, let* τ *be a type that is an instance of* Eq*, and let* $f :: \mathsf{Int} \to \tau$ *and* $v :: [\tau]$*. We have that if* $map\ f\ is == v$ *and* f *is injective on* is*, then*

$$assoc_{\mathsf{Eq}}\ is\ v \equiv \mathsf{Right}\ h$$

for some $h :: \mathsf{IntMapEq}\ \tau$ *with*

$$\mathsf{IntMapEq}.lookup\ i\ h == \mathbf{if}\ elem\ i\ is\ \mathbf{then}\ \mathsf{Just}\ (f\ i)\ \mathbf{else}\ \mathsf{Nothing}$$

for every $i :: \mathsf{Int}$.

Lemma 5. *Let* $is :: [\mathsf{Int}]$*, let* τ *be a type that is an instance of* Eq*, and let* $v :: [\tau]$ *and* $h :: \mathsf{IntMapEq}\ \tau$*. We have that if*

$$\mathsf{Right}\ h \equiv assoc_{\mathsf{Eq}}\ is\ v\,,$$

then

- $map\ (flip\ \mathsf{IntMapEq}.lookup\ h)\ is == map\ \mathsf{Just}\ v\,,$
- *for every* $i :: \mathsf{Int}$ *not in* is*,* $\mathsf{IntMapEq}.lookup\ i\ h \equiv \mathsf{Nothing}\,,$
- $flip\ \mathsf{IntMapEq}.lookup\ h$ *is injective on* is*.*

Like for Lemmas 1 and 2, the proofs are by induction on the list is, but now relying on the correct implementation (in particular, regarding the injectivity invariant) of the operations in module $\mathsf{IntMapEq}$.

Now we are prepared to give a correct bidirectionalizer for source-to-view functions potentially involving equality tests:

$$
\begin{aligned}
&\mathit{bff}_{\mathsf{Eq}} :: (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]) \\
&\qquad\quad \to (\forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha]) \\
&\mathit{bff}_{\mathsf{Eq}}\ \mathit{get} = \lambda s\ v \to \\
&\quad \mathbf{let}\ (s', g) = \mathit{template}_{\mathsf{Eq}}\ s \\
&\qquad\quad h \qquad = \mathit{either}\ \mathit{error}\ \mathit{id}\ (\mathit{assoc}_{\mathsf{Eq}}\ (\mathit{get}\ s')\ v) \\
&\qquad\quad h' \qquad = \mathit{either}\ \mathit{error}\ \mathit{id}\ (\mathsf{IntMapEq}.\mathit{union}\ h\ g) \\
&\quad \mathbf{in}\ \ \mathit{seq}\ h'\ (\mathit{map}\ (\mathit{fromJust} \circ \mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ h')\ s')
\end{aligned}
$$

Let us do some sanity checks:

```
> bff_Eq rmdups "abcbabcbaccba" "aBc"
"aBcBaBcBaccBa"
> bff_Eq rmdups "abcbabcbaccba" "abc"
"abcbabcbaccba"
> bff_Eq rmdups "abc" "aaa"
"*** Exception: Update violates differentness.
> bff_Eq rmdups "aaa" "abc"
"*** Exception: Update changes the length.
```

This looks much better than what we saw at the beginning of the current section. Indeed, we now truly have $\mathit{bff}_{\mathsf{Eq}}\ \mathit{rmdups} \equiv \mathit{put}_3$ for put_3 as given in the introduction (except that $\mathit{bff}_{\mathsf{Eq}}\ \mathit{rmdups}$ gives more meaningful error messages).

A small, but important, detail in the definition of $\mathit{bff}_{\mathsf{Eq}}$ is that the computation of h' via $\mathsf{IntMapEq}.\mathit{union}$ may now also lead to an error being raised. This is essential for properly dealing with examples like the following one:

```
> bff_Eq (tail . rmdups) "abcbabcbaccba" "ba"
"*** Exception: Update violates differentness.
```

Note that the view obtained from "abcbabcbaccba" by applying $tail \circ rmdups$ is "bc". Updating "bc" to "ba" does not yet introduce a differentness violation on the view level. But blindly propagating this change from 'c' to 'a' back into the original source would give "abababababaaaba". And this would contradict the PutGet law, because $tail \circ rmdups$ applied to "abababababaaaba" gives "b", which is different from the supposed "ba". The solution employed to detect such late conflicts (arising when the updates learned by comparing the template view with the updated proper value view encounter those values from the original source that did not make it into the view and thus are simply kept unchanged) is to make sure that no unwarranted equalities occur when combining the associations h and g into h'. Our implementation of $\mathsf{IntMapEq}.union$ takes care of that. This does not change its left-biased nature. That is, an error is only reported if a pair (i, b) in h conflicts with a pair (j, a) in g in the sense that $a == b$ and there is no pair (j, c) in h that renders (j, a) irrelevant.

Before proving the GetPut and PutGet laws for bff_{Eq}, let us clarify the situation of free theorems for functions of the type $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$. The general form, as obtained for example from our online free theorems generator `http://linux.tcs.inf.tu-dresden.de/~voigt/ft`, is that for every choice of types τ_1 and τ_2 that are instances of Eq, relation $\mathcal{R}$ between them that respects Eq, and lists $l_1 :: [\tau_1]$ and $l_2 :: [\tau_2]$ of the same length and element-wise related by $\mathcal{R}$, the lists $get\ l_1$ and $get\ l_2$ are also of the same length and element-wise related by $\mathcal{R}$. The notion of $\mathcal{R}$ respecting Eq here means that for every (a, b) and (c, d) in $\mathcal{R}$, $a == c$ if and only if $b == d$. This general free theorem easily gives the following specialized version.

Lemma 6. *Let* $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$*, let* τ *be a type that is an instance of* Eq*, and let* $f :: \mathsf{Int} \to \tau$*,* $s' :: [\mathsf{Int}]$*, and* $s :: [\tau]$*. We have that if* $map\ f\ s' == s$ *and* f *is injective on* s'*, then* $map\ f\ (get\ s') == get\ s$ *and every* i *in* $get\ s'$ *is also in* s'*.*

The relation $\mathcal{R}$ used for this specialization is the one which contains exactly all pairs (i, a) with $i :: \mathsf{Int}$, $a :: \tau$, i in s', and $f\ i == a$.

Now, we can go about proving the GetPut and PutGet laws for bff_{Eq}. The former is now also established only up to $==$.

Theorem 3. *For every* $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$*, type* τ *that is an instance of* Eq*, and* $s :: [\tau]$*, we have*

$$bff_{\mathsf{Eq}}\ get\ s\ (get\ s) == s\,.$$

Theorem 4. *Let* $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$*, let* τ *be a type that is an instance of* Eq*, and let* $v, s :: [\tau]$*. We have that if* $bff_{\mathsf{Eq}}\ get\ s\ v$ *is defined, then*

$$get\ (bff_{\mathsf{Eq}}\ get\ s\ v) == v\,.$$

The proofs of these two theorems, relying on Lemmas 3–6, are given in Appendices B and C.

5. Source-to-View Functions with Ordering Tests

Having dealt with equality tests, how about ordering tests? Can we produce a correct bidirectionalizer of type

$$\begin{aligned} \mathit{bff}_{\mathsf{Ord}} &:: (\forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha]) \\ &\quad \to (\forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to [\alpha] \to [\alpha])\ ? \end{aligned}$$

The roadmap to follow should be relatively clear from the previous section. First, we need an appropriate template manufacture. Now the template integer values should not only reflect which elements in the original source are equal, but also need to reflect their relative order. Since this means that we cannot assign integer values until we have seen the full source list, it turns out that the "monadic traversal" used in $\mathit{template}_{\mathsf{Eq}}$ is not sufficient anymore. Instead, we use the framework of applicative functors, or idioms (McBride and Paterson 2008). It is captured by the following Haskell type constructor class, defined in the standard library Control.Applicative:

$$\begin{aligned} &\mathbf{class}\ \mathsf{Functor}\ \phi \Rightarrow \mathsf{Applicative}\ \phi\ \mathbf{where} \\ &\quad \mathit{pure} :: \alpha \to \phi\ \alpha \\ &\quad (<\!*\!>) :: \phi\ (\alpha \to \beta) \to \phi\ \alpha \to \phi\ \beta \end{aligned}$$

For ordered template generation we conceptually need two phases, a first to collect all values occurring in the original source list, so that after sorting them a second phase can assign appropriate integer values. It turns out that for both tasks there already exist predefined applicative functors. For the collection of values, we can simply use the constant functor (Control.Applicative.Const) mapping to the monoid (Data.Monoid) of sets (Data.Set):

$$\begin{aligned} &\mathit{collect} :: \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \to \mathsf{Const}\ (\mathsf{Set}\ \alpha)\ [\beta] \\ &\mathit{collect}\ s = \mathit{traverse}\ (\lambda a \to \mathsf{Const}\ (\mathsf{Set}.\mathit{singleton}\ a))\ s \end{aligned}$$

$$
\begin{aligned}
&\mathit{traverse} :: \mathsf{Applicative}\ \phi \Rightarrow (\alpha \rightarrow \phi\ \beta) \rightarrow [\alpha] \rightarrow \phi\ [\beta]\\
&\mathit{traverse}\ f\ [\,] \qquad\quad = \mathit{pure}\ [\,]\\
&\mathit{traverse}\ f\ (a : as) = \mathit{pure}\ (:)\ <\!\!*\!\!>\ f\ a\ <\!\!*\!\!>\ \mathit{traverse}\ f\ as
\end{aligned}
$$

To build a proper association between integer values and (ordered) source values, we need an abstraction similar to $\mathsf{IntMapEq}$ but maintaining an order-preservation invariant as well. We provide this in module $\mathsf{IntMapOrd}$, with API as given in Figure 3. Note that $\mathit{fromAscPairList}$ expects a list with both keys and values in ascending order. Together with the function $\mathsf{Set}.\mathit{toAscList}$ that transforms a set into a sorted list, we can define

$$
\begin{aligned}
&\mathit{set2map} :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Set}\ \alpha \rightarrow \mathsf{IntMapOrd}\ \alpha\\
&\mathit{set2map}\ as =\\
&\quad \mathsf{IntMapOrd}.\mathit{fromAscPairList}\ (\mathit{zip}\ [0..]\ (\mathsf{Set}.\mathit{toAscList}\ as))
\end{aligned}
$$

and then have, for example:

```
> set2map . getConst $ collect "transformation"
fromList [(0,'a'),(1,'f'),(2,'i'),(3,'m'),(4,'n'),
(5,'o'),(6,'r'),(7,'s'),(8,'t')]
```

For propagating knowledge about such a proper assignment between integer values and ordered source values, we can use a partially applied function arrow functor:[1]

$$
\begin{aligned}
&\mathit{propagate} :: \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \rightarrow ((\rightarrow)\ (\mathsf{IntMapOrd}\ \alpha))\ [\mathsf{Int}]\\
&\mathit{propagate}\ s =\\
&\quad \mathit{traverse}\ (\lambda a \rightarrow \mathit{fromJust} \circ \mathsf{IntMapOrd}.\mathit{lookupR}\ a)\ s
\end{aligned}
$$

For example, with m being the $\mathsf{IntMapOrd}\ \mathsf{Char}$ returned above, we have:

```
> propagate "transformation" m
[8,6,0,4,7,1,5,6,3,0,8,2,5,4]
```

Since we do not want to spend two traversals on the collection and propagation phases, we pair the involved applicative functors together with a lifted product bifunctor. Altogether, we realize the new template generator as follows:

$$
\begin{array}{l}
\mathit{template}_{\mathsf{Ord}} :: \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \rightarrow ([\mathsf{Int}], \mathsf{IntMapOrd}\ \alpha) \\
\mathit{template}_{\mathsf{Ord}}\ s = \mathbf{case}\ \mathit{traverse}\ \mathit{number}_{\mathsf{Ord}}\ s\ \mathbf{of} \\
\qquad\qquad\qquad \mathsf{Lift}\ (\mathsf{Const}\ \mathit{as}, f) \rightarrow \mathbf{let}\ m = \mathit{set2map}\ \mathit{as} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \ (f\ m, m)
\end{array}
$$

$$
\begin{array}{l}
\mathit{number}_{\mathsf{Ord}} :: \mathsf{Ord}\ \alpha \Rightarrow \alpha \rightarrow \mathsf{Lift}\ (,)\ (\mathsf{Const}\ (\mathsf{Set}\ \alpha)) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad ((\rightarrow)\ (\mathsf{IntMapOrd}\ \alpha))\ \mathsf{Int} \\
\mathit{number}_{\mathsf{Ord}}\ a = \mathsf{Lift}\ (\mathsf{Const}\ (\mathsf{Set}.\mathit{singleton}\ a), \\
\qquad\qquad\qquad\qquad \mathit{fromJust} \circ \mathsf{IntMapOrd}.\mathit{lookupR}\ a)
\end{array}
$$

Note that $\mathit{number}_{\mathsf{Ord}}$, which serves as argument to *traverse* in the definition of $\mathit{template}_{\mathsf{Ord}}$, is essentially obtained as a "split"

---

[1] Note that the type of *propagate* could equivalently be written as follows: $\mathsf{Ord}\ \alpha \Rightarrow [\alpha] \rightarrow \mathsf{IntMapOrd}\ \alpha \rightarrow [\mathsf{Int}]$.

$$
\begin{array}{ll}
\mathit{fromAscPairList} & :: \mathsf{Ord}\ \alpha \Rightarrow [(\mathsf{Int}, \alpha)] \rightarrow \mathsf{IntMapOrd}\ \alpha \\
\mathit{empty} & :: \mathsf{IntMapOrd}\ \alpha \\
\mathit{checkInsert} & :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Int} \rightarrow \alpha \rightarrow \mathsf{IntMapOrd}\ \alpha \\
& \qquad\qquad \rightarrow \mathsf{Either}\ \mathsf{String}\ (\mathsf{IntMapOrd}\ \alpha) \\
\mathit{union} & :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{IntMapOrd}\ \alpha \rightarrow \mathsf{IntMapOrd}\ \alpha \\
& \qquad\qquad \rightarrow \mathsf{Either}\ \mathsf{String}\ (\mathsf{IntMapOrd}\ \alpha) \\
\mathit{lookup} & :: \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Int} \rightarrow \mathsf{IntMapOrd}\ \alpha \rightarrow \mathsf{Maybe}\ \alpha \\
\mathit{lookupR} & :: \mathsf{Ord}\ \alpha \Rightarrow \alpha \rightarrow \mathsf{IntMapOrd}\ \alpha \rightarrow \mathsf{Maybe}\ \mathsf{Int}
\end{array}
$$

Figure 3. Functions from module IntMapOrd.

of the corresponding arguments in the definitions of *collect* and *propagate* above. This kind of tupling is an old trick to avoid multiple traversals of data structures (Pettorossi 1987). An alternative approach to ordered template generation would be to use an order-maintenance data structure (Dietz and Sleator 1987).

Under the assumption that in addition to the conditions we have already imposed on instances of Eq every instance of Ord satisfies that the provided $<$ is transitive, that $x < y$ implies $x \mathbin{/\!\!=} y$, and that $x \mathbin{/\!\!=} y$ implies $x < y$ or $y < x$, the following analogue of Lemma 3 now holds. The notion of a function $f :: \mathsf{Int} \to \tau$, for a type τ that is an instance of Ord, being order-preserving on a list $s' :: [\mathsf{Int}]$ is defined as "for every $i < j$ in s', also $f\ i < f\ j$".

Lemma 7. *Let τ be a type that is an instance of* Ord *and let $s :: [\tau]$, $s' :: [\mathsf{Int}]$, and $g :: \mathsf{IntMapOrd}\ \tau$. We have that if*

$$(s', g) \equiv template_{\mathsf{Ord}}\ s\,,$$

then

- $map\ (\mathit{flip}\ \mathsf{IntMapOrd}.lookup\ g)\ s' == map\ \mathsf{Just}\ s\,,$
- *for every* $i :: \mathsf{Int}$ *not in* s', $\mathsf{IntMapOrd}.lookup\ i\ g \equiv \mathsf{Nothing}\,,$
- $\mathit{flip}\ \mathsf{IntMapOrd}.lookup\ g$ *is order-preserving on* s'*.*

We omit a formal proof, but the following example should be reassuring:

```
> template_Ord "transformation"
([8,6,0,4,7,1,5,6,3,0,8,2,5,4],fromList [(0,'a'),(
```

```
1,'f'),(2,'i'),(3,'m'),(4,'n'),(5,'o'),(6,'r'),(7,
's'),(8,'t')])
```

On the view association side, the changes from $assoc_{\mathsf{Eq}}$ to $assoc_{\mathsf{Ord}}$ are almost trivial:

$$
\begin{array}{lll}
assoc_{\mathsf{Ord}} :: \mathsf{Ord}\ \alpha \Rightarrow [\mathsf{Int}] \rightarrow [\alpha] \rightarrow \mathsf{Either\ String}\ (\mathsf{IntMapOrd}\ \alpha) \\
assoc_{\mathsf{Ord}}\ [\,]\ \ \ \ \ \ [\,] & = \mathsf{Right\ IntMapOrd}.empty \\
assoc_{\mathsf{Ord}}\ (i : is)\ (b : bs) & = either\ \mathsf{Left} \\
& \qquad\quad (\mathsf{IntMapOrd}.checkInsert\ i\ b) \\
& \qquad\quad (assoc_{\mathsf{Ord}}\ is\ bs) \\
assoc_{\mathsf{Ord}}\ \_\ \ \ \ \ \ \_ & = \mathsf{Left}\ \text{“Update changes the length.”}
\end{array}
$$

and analogues of Lemmas 4 and 5 for $assoc_{\mathsf{Ord}}$ instead of $assoc_{\mathsf{Eq}}$ are obtained by simply replacing Eq by Ord, “injective” by “order-preserving”, and $\mathsf{IntMapEq}$ by $\mathsf{IntMapOrd}$.

Finally, our bidirectionalizer for source-to-view functions potentially involving ordering tests takes the following, by now probably expected, form:

$$
\begin{array}{l}
\mathit{bff}_{\mathsf{Ord}} :: (\forall \alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]) \\
\qquad\qquad \rightarrow (\forall \alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]) \\
\mathit{bff}_{\mathsf{Ord}}\ get = \lambda s\ v \rightarrow \\
\quad \mathbf{let}\ (s', g) = template_{\mathsf{Ord}}\ s \\
\qquad\quad h \qquad = either\ error\ id\ (assoc_{\mathsf{Ord}}\ (get\ s')\ v) \\
\qquad\quad h' \qquad = either\ error\ id\ (\mathsf{IntMapOrd}.union\ h\ g) \\
\quad \mathbf{in}\ \ seq\ h'\ (map\ (fromJust \circ flip\ \mathsf{IntMapOrd}.lookup\ h')\ s')
\end{array}
$$

Showing off its power, for the function $top3$ from the introduction:

```
> bff_Ord top3 "transformation" "abc"
"transbormatcon"
> bff_Ord top3 "transformation" "xyz"
"*** Exception: Update violates relative order.
```

For proving the GetPut and PutGet laws for $\mathit{bff}_{\mathsf{Ord}}$, we need an appropriate free theorem that holds for every function of type $\mathit{get} :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$. Again consulting the online free theorems generator `http://linux.tcs.inf.tu-dresden.de/~voigt/ft`, we obtain that for every choice of types τ_1 and τ_2 that are instances of Ord, relation $\mathcal{R}$ between them that respects Ord, and lists $l_1 :: [\tau_1]$ and $l_2 :: [\tau_2]$ of the same length and element-wise related by $\mathcal{R}$, the lists $\mathit{get}\ l_1$ and $\mathit{get}\ l_2$ are also of the same length and element-wise related by $\mathcal{R}$. The notion of $\mathcal{R}$ respecting Ord here means that for every (a, b) and (c, d) in $\mathcal{R}$, $a < c$ if and only if $b < d$ (and assuming that the other operations of the Ord type class relate to the definitions for $==$ and $<$ in the natural way). Setting $\tau_1 = \mathsf{Int}$, $\tau_2 = \tau$, $\mathcal{R} = \{(i, a) \mid \mathit{elem}\ i\ s'\ \ \&\&\ \ f\ i == a\}$, $l_1 = s'$, and $l_2 = s$, we obtain the following specialized version.

Lemma 8. *Let* $\mathit{get} :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$*, let* τ *be a type that is an instance of* Ord*, and let* $f :: \mathsf{Int} \rightarrow \tau$*,* $s' :: [\mathsf{Int}]$*, and* $s :: [\tau]$*. We have that if* $\mathit{map}\ f\ s' == s$ *and* f *is order-preserving on* s'*, then* $\mathit{map}\ f\ (\mathit{get}\ s') == \mathit{get}\ s$ *and every* i *in* $\mathit{get}\ s'$ *is also in* s'*.*

Now, quite pleasingly, the proofs of the following two theorems are exact replays of the proofs given for Theorems 3 and 4 in Appendices B and C, respectively, except that Lemma 7 is used instead of Lemma 3, that Lemma 8 is used instead of Lemma 6, and that the analogues of Lemmas 4 and 5 mentioned above are used instead of those two lemmas themselves.

Theorem 5. *For every* $get :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$*, type* τ *that is an instance of* Ord*, and* $s :: [\tau]$*, we have*

$$\mathit{bff}_{\mathsf{Ord}}\ get\ s\ (get\ s) == s\,.$$

Theorem 6. *Let* $get :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow [\alpha] \rightarrow [\alpha]$*, let* τ *be a type that is an instance of* Ord*, and let* $v, s :: [\tau]$*. We have that if* $\mathit{bff}_{\mathsf{Ord}}\ get\ s\ v$ *is defined, then*

$$get\ (\mathit{bff}_{\mathsf{Ord}}\ get\ s\ v) == v\,.$$

6. Going Generic

So far, we have focused on list data structures for sources and views. In this section, we lift this restriction, both on the input and output sides of get-functions. Let us start with the input side, and with $\mathit{bff}_{\mathsf{Ord}}$.

Where in the definition of $\mathit{bff}_{\mathsf{Ord}}$ is the fact important that the input data structure is a list? The answer is: at two places; once in the definition of $traverse$ as used in $template_{\mathsf{Ord}}$ and thus in $\mathit{bff}_{\mathsf{Ord}}$, and once when using map inside $\mathit{bff}_{\mathsf{Ord}}$ itself. But note that even though we have provided our own definition of $traverse$ in the previous section, a function with that name already exists in the standard library $\mathsf{Data.Traversable}$, where it is a method of the type constructor class $\mathsf{Traversable}$ and has the following type:

$$\begin{aligned} traverse :: {} & (\mathsf{Applicative}\ \phi, \mathsf{Traversable}\ \kappa) \\ & \Rightarrow (\alpha \to \phi\ \beta) \to \kappa\ \alpha \to \phi\ (\kappa\ \beta)\,. \end{aligned}$$

Moreover, there is also a predefined instance of $\mathsf{Traversable}$ for the type of lists, and the definition for $traverse$ in that instance is exactly the one seen in the previous section. So we could have avoided defining our own $traverse$ and instead used the predefined one. But more importantly, we can give $template_{\mathsf{Ord}}$ the following more general type, without changing anything about its definition:

$$\begin{aligned} template_{\mathsf{Ord}} :: {} & (\mathsf{Traversable}\ \kappa, \mathsf{Ord}\ \alpha) \\ & \Rightarrow \kappa\ \alpha \to (\kappa\ \mathsf{Int}, \mathsf{IntMapOrd}\ \alpha)\,. \end{aligned}$$

Providing an instance definition for the data type Tree from the introduction as follows:

$$\begin{array}{l} \mathbf{instance}\ \mathsf{Traversable}\ \mathsf{Tree}\ \mathbf{where} \\ \quad traverse\ f\ (\mathsf{Leaf}\ a) \qquad\ = pure\ \mathsf{Leaf}\ <\!\!*\!\!>\ f\ a \\ \quad traverse\ f\ (\mathsf{Node}\ t_1\ t_2) = pure\ \mathsf{Node}\ <\!\!*\!\!>\ traverse\ f\ t_1 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad <\!\!*\!\!>\ traverse\ f\ t_2 \end{array}$$

we then have, for example:

```
> template_Ord (Node (Leaf 'a') (Leaf 'b'))
(Node (Leaf 0) (Leaf 1),fromList [(0,'a'),(1,'b')])
```

Actually, for dependency reasons, we also need to add the following two instance definitions:

$$\begin{array}{l} \mathbf{instance}\ \mathsf{Foldable}\ \mathsf{Tree}\ \mathbf{where} \\ \quad foldMap = foldMapDefault \end{array}$$

$$\mathbf{instance}\ \mathsf{Functor}\ \mathsf{Tree}\ \mathbf{where}$$

$$fmap = fmapDefault$$

But these will always be the same for every new data type, and so do not impose any real burden. And even the burden of having to write $\mathsf{Traversable}$ instances can be avoided. Namely, by using the modules Data.DeriveTH and Data.Derive.Traversable of Hackage package `derive-0.1.1` (authored by N. Mitchell and S. O'Rear), as well as compiler flag `-XTemplateHaskell`, we could instead of the above manual instance definition for Tree simply have written

$$\$(\, derive\ makeTraversable\ ''\mathsf{Tree}\,)$$

Back to bff_{Ord} itself. Since every $\mathsf{Traversable}$ is also a $\mathsf{Functor}$, it now suffices to replace map by

$$fmap :: \mathsf{Functor}\ \kappa \Rightarrow (\beta \to \alpha) \to \kappa\ \beta \to \kappa\ \alpha$$

in bff_{Ord}'s definition to allow a generalization of its type as well:

$$\begin{aligned} bff_{\mathsf{Ord}} :: \mathsf{Traversable}\ \kappa \Rightarrow (&\forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow \kappa\ \alpha \to [\alpha]) \\ &\to (\forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow \kappa\ \alpha \to [\alpha] \to \kappa\ \alpha)\,. \end{aligned}$$

This means that we can now bidirectionalize functions of type $get :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow \kappa\ \alpha \to [\alpha]$ for any instance κ of $\mathsf{Traversable}$, not just for lists. For example, we can use bff_{Ord} on functions $get :: \forall\alpha.\ \mathsf{Ord}\ \alpha \Rightarrow \mathsf{Tree}\ \alpha \to [\alpha]$ just as well.

Can we profit from the same kind of genericity also for bff_{Eq} and bff? Concentrating on bff_{Eq} first, it seems that we cannot readily use the generic $traverse$, because $template_{\mathsf{Eq}}$ is based on a monad, not on an applicative functor. But actually every monad can be wrapped to form an applicative functor, and there are even predefined facilities for this in Control.Applicative. So without changing anything at all about $number_{\mathsf{Eq}}$ we can rewrite $template_{\mathsf{Eq}}$ as follows:

$$\begin{aligned}
&template_{\mathsf{Eq}} :: (\mathsf{Traversable}\ \kappa, \mathsf{Eq}\ \alpha) \\
&\qquad\qquad \Rightarrow \kappa\ \alpha \to (\kappa\ \mathsf{Int}, \mathsf{IntMapEq}\ \alpha) \\
&template_{\mathsf{Eq}}\ s = \mathbf{case}\ runState\ (go\ s)\ (\mathsf{IntMapEq}.empty, 0) \\
&\qquad\qquad\quad \mathbf{of}\quad (s', (g, _)) \to (s', g) \\
&\quad \mathbf{where}\ go = unwrapMonad \\
&\qquad\qquad\qquad \circ\ traverse\ (\mathsf{WrapMonad} \circ number_{\mathsf{Eq}})
\end{aligned}$$

and then obtain a generic bidirectionalizer

$$\begin{aligned}
&bff_{\mathsf{Eq}} :: \mathsf{Traversable}\ \kappa \Rightarrow (\forall \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow \kappa\ \alpha \to [\alpha]) \\
&\qquad\qquad\qquad\qquad \to (\forall \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow \kappa\ \alpha \to [\alpha] \to \kappa\ \alpha)
\end{aligned}$$

simply by replacing map by $fmap$ in the definition of bff_{Eq}.

And even for bff we can replace the template generation via $s' = [0..length\ s - 1]$ and $g = \mathsf{IntMap}.fromAscList\ (zip\ s'\ s)$ by a more streamlined one amenable to Traversable. Again we use a state monad, wrapped up as an applicative functor. In full:[2]

$$\begin{aligned}
&bff :: \mathsf{Traversable}\ \kappa \Rightarrow (\forall \alpha.\ \kappa\ \alpha \to [\alpha]) \\
&\qquad\qquad\qquad\qquad \to (\forall \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow \kappa\ \alpha \to [\alpha] \to \kappa\ \alpha) \\
&bff\ get = \lambda s\ v \to \\
&\quad \mathbf{let}\ (s', g) = template\ s \\
&\qquad\quad h \quad\ = either\ error\ id\ (assoc\ (get\ s')\ v) \\
&\qquad\quad h' \quad = \mathsf{IntMap}.union\ h\ g \\
&\quad \mathbf{in}\ \ seq\ h\ (fmap\ (fromJust \circ flip\ \mathsf{IntMap}.lookup\ h')\ s')
\end{aligned}$$

$$\begin{aligned}
&template :: \mathsf{Traversable}\ \kappa \Rightarrow \kappa\ \alpha \to (\kappa\ \mathsf{Int}, \mathsf{IntMap}\ \alpha) \\
&template\ s = \\
&\quad \mathbf{case}\ runState\ (go\ s)\ ([\,], 0) \\
&\quad \mathbf{of}\quad (s', (l, _)) \to (s', \mathsf{IntMap}.fromAscList\ (reverse\ l))
\end{aligned}$$

$$\mathbf{where}\ go = unwrapMonad \\ \circ\ traverse\ (\mathsf{WrapMonad} \circ number)$$

$$number :: \alpha \rightarrow \mathsf{State}\ ([(\mathsf{Int}, \alpha)], \mathsf{Int})\ \mathsf{Int}$$
$$number\ a = \mathbf{do}\ (l, i) \leftarrow \mathsf{State}.get \\ \mathsf{State}.put\ ((i, a) : l, i + 1) \\ return\ i$$

This version is now also applicable to *get*-functions with source data structures other than lists. For example, for the function *flatten* from the introduction we obtain:

```
> bff flatten (Node (Leaf 'a') (Leaf 'b')) "xy"
Node (Leaf 'x') (Leaf 'y')
```

Indeed, $bff\ flatten \equiv put_2$.

Clearly, a similar generalization from lists to other data structures is desirable for the output sides of *get*-functions as well. The key task then is to replace $assoc$, $assoc_{\mathsf{Eq}}$, and $assoc_{\mathsf{Ord}}$ by generic versions that are not anymore specific to lists. Unfortunately, there is no predefined class like Traversable that we can simply use here. But there *is* a common core to the different *assoc*-functions. Namely, they all traverse two lists in lock-step, pairing up the elements found in corresponding positions, and inserting those pairs into some variation of integer maps. It is very natural to capture the first aspect, of traversing two data structures in a synchronized fashion and collecting pairs of corresponding elements, by a new type constructor class as follows:

$$\mathbf{class}\ \mathsf{Zippable}\ \kappa\ \mathbf{where} \\ tryZip :: \kappa\ \alpha \rightarrow \kappa\ \beta \rightarrow \mathsf{Either\ String}\ (\kappa\ (\alpha, \beta))$$

Since such a zipping can also fail, for example if two lists have unequal length, we provide for potential error messages in the return type of $tryZip$. Now, for example, instances of $\mathsf{Zippable}$ for lists and for the data type Tree can be given as follows:

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Zippable}\ []\ \textbf{where} \\
\quad tryZip\ []\ \ []\ = \mathsf{Right}\ [] \\
\quad tryZip\ (i : is)\ (b : bs) = \mathsf{Right}\ (:)\ <\!\!*\!\!>\ \mathsf{Right}\ (i, b) \\
\qquad\qquad\qquad\qquad\qquad\qquad <\!\!*\!\!>\ tryZip\ is\ bs \\
\quad tryZip\ \_\ \ \_\ = \mathsf{Left}\ \text{“Update changes the length.”}
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathsf{Zippable}\ \mathsf{Tree}\ \textbf{where} \\
\quad tryZip\ (\mathsf{Leaf}\ i)\ (\mathsf{Leaf}\ b) = \mathsf{Right}\ (\mathsf{Leaf}\ (i, b)) \\
\quad tryZip\ (\mathsf{Node}\ t_1\ t_2)\ (\mathsf{Node}\ v_1\ v_2) = \mathsf{Right}\ \mathsf{Node} \\
\qquad\qquad\qquad\qquad\qquad\qquad <\!\!*\!\!>\ tryZip\ t_1\ v_1 \\
\qquad\qquad\qquad\qquad\qquad\qquad <\!\!*\!\!>\ tryZip\ t_2\ v_2 \\
\quad tryZip\ \_\ \_ = \mathsf{Left}\ \text{“Update changes the shape.”}
\end{array}
$$

[2] The use of $reverse$ in the definition of $template$ is necessary to ensure that $\mathsf{IntMap}.fromAscList$ indeed receives a list with keys in ascending order.

Note that for convenient propagation of potential errors we use an appropriate instance of $\mathsf{Applicative}$ for $\mathsf{Either\ String}$.

Now, the $assoc$-functions can be factorized into applications of $tryZip$ followed by folding some insertion functions over the zipped structure containing pairs of integers and updated view values. By “folding”, we of course mean a generic operation not specific to lists, and fortunately there is already a type constructor class for just this purpose in the standard library $\mathsf{Data.Foldable}$. The class method of interest here is the following one:

$$\mathsf{Data.Foldable}.\mathit{foldr} :: \mathsf{Foldable}\ \kappa \Rightarrow (\alpha \to \beta \to \beta) \to \beta \\ \to \kappa\ \alpha \to \beta$$

Using it, we get for example:

$$\begin{aligned}
&\mathit{assoc} :: (\mathsf{Zippable}\ \kappa, \mathsf{Foldable}\ \kappa, \mathsf{Eq}\ \alpha) \\
&\qquad \Rightarrow \kappa\ \mathsf{Int} \to \kappa\ \alpha \to \mathsf{Either}\ \mathsf{String}\ (\mathsf{IntMap}\ \alpha) \\
&\mathit{assoc} = \mathit{makeAssoc}\ \mathit{checkInsert}\ \mathsf{IntMap}.\mathit{empty}
\end{aligned}$$

$$\begin{aligned}
&\mathit{makeAssoc}\ \mathit{checkInsert}\ \mathit{empty}\ s''\ v = \\
&\quad \mathit{either}\ \mathsf{Left}\ f\ (\mathit{tryZip}\ s''\ v) \\
&\quad\quad \textbf{where}\ f = \mathsf{Data.Foldable}.\mathit{foldr} \\
&\qquad\qquad\qquad (\mathit{either}\ \mathsf{Left} \circ \mathit{uncurry}\ \mathit{checkInsert}) \\
&\qquad\qquad\qquad (\mathsf{Right}\ \mathit{empty})
\end{aligned}$$

Then we can change the type of *bff* into

$$\begin{aligned}
\mathit{bff} &:: (\mathsf{Traversable}\ \kappa, \mathsf{Zippable}\ \kappa', \mathsf{Foldable}\ \kappa') \\
&\Rightarrow (\forall \alpha.\ \kappa\ \alpha \to \kappa'\ \alpha) \\
&\quad \to (\forall \alpha.\ \mathsf{Eq}\ \alpha \Rightarrow \kappa\ \alpha \to \kappa'\ \alpha \to \kappa\ \alpha)
\end{aligned}$$

without having to change anything at all about the function's current definition. Analogously, with

$$\begin{aligned}
&\mathit{assoc}_{\mathsf{Eq}} :: (\mathsf{Zippable}\ \kappa, \mathsf{Foldable}\ \kappa, \mathsf{Eq}\ \alpha) \\
&\qquad \Rightarrow \kappa\ \mathsf{Int} \to \kappa\ \alpha \to \mathsf{Either}\ \mathsf{String}\ (\mathsf{IntMapEq}\ \alpha) \\
&\mathit{assoc}_{\mathsf{Eq}} = \mathit{makeAssoc}\ \mathsf{IntMapEq}.\mathit{checkInsert} \\
&\qquad\qquad\qquad \mathsf{IntMapEq}.\mathit{empty}
\end{aligned}$$

and

$$\begin{aligned}
&\mathit{assoc}_{\mathsf{Ord}} :: (\mathsf{Zippable}\ \kappa, \mathsf{Foldable}\ \kappa, \mathsf{Ord}\ \alpha) \\
&\qquad \Rightarrow \kappa\ \mathsf{Int} \to \kappa\ \alpha \to \mathsf{Either}\ \mathsf{String}\ (\mathsf{IntMapOrd}\ \alpha) \\
&\mathit{assoc}_{\mathsf{Ord}} = \mathit{makeAssoc}\ \mathsf{IntMapOrd}.\mathit{checkInsert}
\end{aligned}$$

$$\mathsf{IntMapOrd}.\mathit{empty}$$

and without any changes to the current function definitions of $\mathit{bff}_{\mathsf{Eq}}$ and $\mathit{bff}_{\mathsf{Ord}}$, we get more generic types for them in the spirit of the final type for bff given above, that is, generalizing $[\alpha]$ to $\kappa'\ \alpha$ for any κ' that is an instance of both Zippable and Foldable.

Note that instances of Foldable are already automatically derivable in the same fashion using Data.DeriveTH as instances of Traversable are, or alternatively can be obtained from Traversable instances using the kind of default definition seen earlier in this section. Thus, all the remaining effort required to make bff, $\mathit{bff}_{\mathsf{Eq}}$, and $\mathit{bff}_{\mathsf{Ord}}$ successfully deal with a new data type on both the input and output sides of get-functions is to provide an appropriate Zippable instance. This could be done manually, but Hackage package `bff-0.1` also contains an automatic deriver (contributed by J. Breitner) that generalizes the Zippable instances seen earlier in this section.[3]

What about the correctness of the generic versions? Of course, for their specific instantiations to the case of lists our proofs as given previously continue to apply. For the generic case some generalization effort is required. For example, Lemmas 3 and 7 need to be replaced by versions involving fmap instead of map, and a similar statement is necessary for $\mathit{template}$ in order to replace the

[3] Actually, it produces slightly different definitions using an efficiency improvement trick inspired by Voigtländer (2008). Also, it became necessary to add a Traversable class restriction as precondition to the definition of class Zippable.

use of (2) and (3) in the proof of Theorem 1. We need to derive generic versions of the free theorems we have used, and we need to replace the lemmas about $assoc$-functions (i.e., Lemmas 1, 2, 4, 5, and the analogues of Lemmas 4 and 5 for the Ord-setting as mentioned in Section 5) by corresponding generic versions. Actually, these lemmas can now be factorized into statements about instances of Zippable and statements about the $checkInsert$- and $empty$-functions being folded over the zipped structures. We refrain here from exercising all this through.

7. Discussion and Evaluation

We have presented a new bidirectionalization technique for a wide range of polymorphic functions. One might wonder whether what we achieve is "true" bidirectionalization. After all, for a given forward function, we do not really obtain a backward component that is somehow tailored to it in the sense that it is based on a deep analysis of the forward function's innards. Rather, the put-function we obtain will, at runtime, observe the get-function in forward mode, and draw conclusions from this kind of "simulation". Is not that cheating?

While this might first appear to be a serious objection casting our overall approach in doubt, we think it is ultimately unnecessary concern. At the end of the day, what counts is whether or not the obtained put-function is extensionally the one we want and need, and its genesis and intensional, syntactic aspects are completely irrelevant for this evaluation. So how good are our $bff\ get$ and so on, under such impartial judgment? Having established the GetPut and PutGet laws is one step towards an answer. Moreover, even though we have not included the additional proofs here, also the PutPut law holds. That is, for every pair get/put with $put \equiv \mathit{bff}\ get$, $put \equiv \mathit{bff}_{\mathsf{Eq}}\ get$, or $put \equiv \mathit{bff}_{\mathsf{Ord}}\ get$, we have that if $put\ s\ v$ and $put\ (put\ s\ v)\ v'$ are defined, then

$$put\ (put\ s\ v)\ v' == put\ s\ v'\,.$$

And undoability is also a given; i.e., if $put\ s\ v$ is defined, then

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

And even beyond just those base requirements, the put-functions returned by our bidirectionalizers often do exactly The Right Thing. Examples for this can be seen in the introduction and throughout the paper, and more are easy to come by. Of course, it should not be expected that an automatic approach can always deliver the absolutely best backward component one could write by hand. For example, for the function $halve$ from the introduction a slight improvement to put_1 would be possible by weakening the condition

$$length\ as' == n$$

to

$$length\ as' == n\ \ ||\ \ odd\ (length\ as)\ \ \&\&\ \ length\ as' == n{+}1\,.$$

Our technique does not detect this, i.e., $\mathit{bff}\ halve$ is semantically equivalent to the original version of put_1 without this small improvement. But that much comes for free, and is arguably sufficient in most cases.

Maybe a good way to think about possible application scenarios for our technique is to consider it as a very useful tool for rapid prototyping. Imagine one wants to build some system with built-in bidirectionality support, such as the structured document editor of Hu et al. (2004). Would not it be nice to have at one's command much of the Haskell Prelude and polymorphic functions from other standard libraries, all with backward components obtained at no cost? Even if the automatically provided backward components are not perfect in each and every case, they give an initial solution and enable progress to be made quickly on the overall design without getting lost in the bidirectionality aspect. And once that design has solidified, it is possible to see which forward functions are actually going to be used, which of them are critical and did not get assigned a sufficiently good backward component the free and easy way, and then to provide fine-tuned versions for those by hand.

For programming in the large, it would also be worthwhile to look at connecting our technique to the combinatory approach pioneered by Foster et al. (2007). Their framework provides for systematic and sound ways of assembling bigger bidirectional transformations from smaller ones, but naturally depends on a supply of correctly behaving get/put-pairs being available on the lowest level of granularity. Our free bidirectionalizers promise to provide a rich and safe source to be used in this context. It would also be interesting to investigate how our development relates to recent extensions of the combinatory approach for ordered data (Bohannon et al. 2008) and for correctness modulo equivalence relations (Foster et al. 2008).

Other pragmatic questions worth investigating include whether it is possible to use a similar technique to ours for deriving $create$-functions that produce a new source from a given view without having access to an original source, and whether it is possible to meaningfully augment bff, and its two variants, with additional parameters that steer its choice of a backward component. The latter may be useful, for example, when an update changes the shape of a view, causing the current regime to report failure.

A somewhat secondary concern is that about the efficiency of the obtained put- (and potentially $create$-) functions. Clearly, a purely semantic approach like ours here cannot in general hope to produce as efficient backward components as a more syntactic, but also more restricted, approach might achieve. After all, detached from the realm of syntax, no intensional knowledge about the get-function's underlying algorithm can be gained and thus used. But this does not impair the prototyping scenario sketched above. And dumping premature optimization, the safety and programmer (rather than program) productivity boon offered by free bidirectionalization may often be more essential in practice than efficiency differences that may only show up at rather large scales of data.

That said, there *is* room for improving the efficiency of put-functions as obtained by our technique. For one thing, the variations of integer maps used are currently implemented rather naively. Some data structure and algorithm engineering would likely have a beneficial impact here. Also, even though our bidirectionalizers are, by design, ignorant of the definition of the get-function provided as argument, nothing stops us, or a compiler, from inlining that function definition in a particular application like $\mathit{bff}\ get$ for a concrete get-function. Then, the door is open to applying any of the program specialization and fusion methods that abound in the field of functional languages. In combination with rules about the integer map interface functions, it might even be possible in some cases to thus transform the automatically obtained put-functions into ones with efficiency close to hand-coded versions.

And yet, just how bad is the current performance? To evaluate this, we have run a few simple experiments on a 2.2 GHz AMD Opteron 248 processor (core) with 2GB memory. Every experiment consists of comparing the efficiency of one of the hand-coded put-functions from the introduction to that of the corresponding automatically obtained version, on input data structures of varying sizes and with views that actually represent permitted updates. The elements contained in source and view data structures are integers, so that each equality test on them takes constant time only. To make asymptotic behavior more apparent, runtimes are plotted normalized through division by input size. The results can be seen in Figures 4–7.

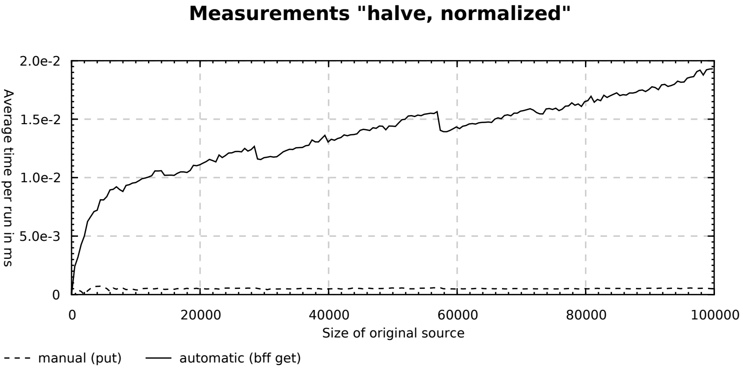

Figure 4. Evaluation of put_1 vs. $\mathit{bff}\ \mathit{halve}$.

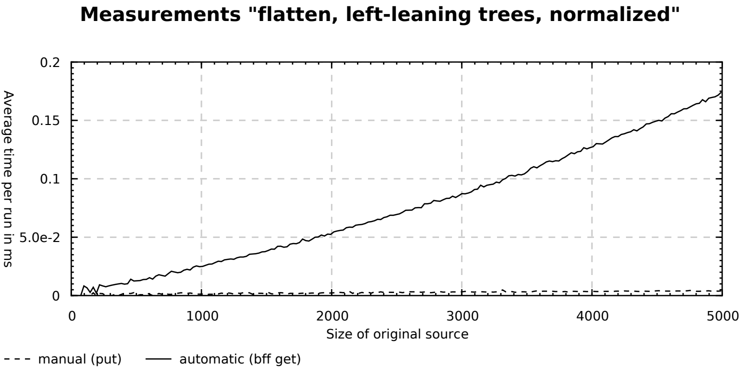

Figure 5. Evaluation of put_2 vs. *bff flatten*, on nasty input.

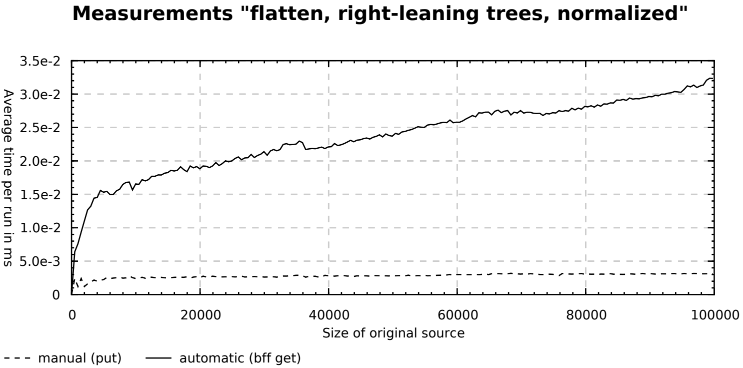

Figure 6. Evaluation of put_2 vs. *bff flatten*, on nice input.

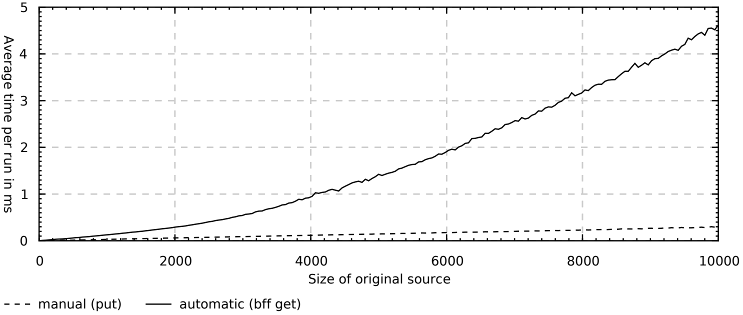

Figure 7. Evaluation of put_3 vs. $\mathit{bff}_{\mathsf{Eq}}\ rmdups$.

Acknowledgments

I thank Edward A. Kmett and Stuart Cook for additions to their Hackage packages `category-extras-0.53.5` and `bimap-0.2.3` that made reuse easier for me. I thank Joachim Breitner for his work on the automatic deriver for `Zippable` instances, the implementation of the online tool, his assistance with efficiency measurements, and general release support. Finally, I thank the reviewers for their enthusiasm about the paper. I am sorry that I could not realize all their suggestions for addressing remaining shortcomings in the presentation.

References

M. Abbott, T. Altenkirch, and N. Ghani. Categories of containers. In *Foundations of Software Science and Computational Structures, Proceedings*, volume 2620 of *LNCS*, pages 23–38. Springer-Verlag, 2003.

F. Bancilhon and N. Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(3):557–575, 1981.

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

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*, pages 407–419. ACM Press, 2008.

N.A. Danielsson, R.J.M. Hughes, P. Jansson, and J. Gibbons. Fast and loose reasoning is morally correct. In *Principles of Programming Languages, Proceedings*, pages 206–217. ACM Press, 2006.

P.F. Dietz and D.D. Sleator. Two algorithms for maintaining order in a list. In *Symposium on Theory of Computing, Proceedings*, pages 365–372. ACM Press, 1987.

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, 2007.

J.N. Foster, A. Pilkiewicz, and B.C. Pierce. Quotient lenses. In *International Conference on Functional Programming, Proceedings*, pages 383–395. ACM Press, 2008.

Z. Hu, S.-C. Mu, and M. Takeichi. A programmable editor for developing structured documents based on bidirectional transformations. In *Partial Evaluation and Semantics-Based Program Manipulation, Proceedings*, pages 178–189. ACM Press, 2004.

G. Hutton and D. Fulger. Reasoning about effects: Seeing the wood through the trees. In *Trends in Functional Programming, Draft Proceedings*, 2008.

P. Johann and J. Voigtländer. Free theorems in the presence of seq. In *Principles of Programming Languages, Proceedings*, pages 99–110. ACM Press, 2004.

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*, pages 47–58. ACM Press, 2007.

C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008.

A. Pettorossi. Derivation of programs which traverse their input data only once. In *Advanced School on Programming Methodologies, Proceedings*, pages 165–184. Academic Press, 1987.

J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing, Proceedings*, pages 513–523. Elsevier, 1983.

C. Strachey. Fundamental concepts in programming languages. In *International Summer School in Computer Programming, Lecture Notes*, 1967. Reprint appeared in *Higher-Order and Symbolic Computation*, 13(1–2): 11–49, 2000.

J. Voigtländer. Asymptotic improvement of computations over free monads. In *Mathematics of Program Construction, Proceedings*, volume 5133 of *LNCS*, pages 388–403. Springer-Verlag, 2008.

P. Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture, Proceedings*, pages 347–359. ACM Press, 1989.

P. Wadler. The essence of functional programming (Invited talk). In *Principles of Programming Languages, Proceedings*, pages 1–14. ACM Press, 1992.

A. Proof of Theorem 2

Let $get :: \forall\alpha.[\alpha] \to [\alpha]$, let τ be a type that is an instance of Eq, and let $v, s :: [\tau]$. If $\mathit{bff}\ get\ s\ v$ is defined, then we necessarily have

$$\mathit{bff}\ get\ s\ v \equiv map\ f\ s'\,,$$

where

$$\begin{aligned}\mathsf{Right}\ h &\equiv assoc\ (get\ s')\ v && (7)\\ h' &\equiv \mathsf{IntMap}.union\ h\ g && (8)\\ f &\equiv fromJust \circ \mathit{flip}\ \mathsf{IntMap}.lookup\ h' && (9)\end{aligned}$$

(and the values of s' and g are unimportant in what follows). Thus, by the free theorem mentioned in the proof of Theorem 1,

$$get\ (\mathit{bff}\ get\ s\ v) \equiv map\ f\ (get\ s')\,. \tag{10}$$

By (7) and Lemma 2 we have

$$map\ (\mathit{flip}\ \mathsf{IntMap}.lookup\ h)\ (get\ s') == map\ \mathsf{Just}\ v\,. \tag{11}$$

In particular, for every i in $get\ s'$, we have $\mathsf{IntMap}.lookup\ i\ h \equiv \mathsf{Just}\ b$ for some $b :: \tau$. But then by (8) and the specifications of $\mathsf{IntMap}.union$ and $\mathsf{IntMap}.lookup$,

$$map\ (\mathit{flip}\ \mathsf{IntMap}.lookup\ h')\ (get\ s')$$

$$\equiv$$
$$map\ (flip\ \mathsf{IntMap}.lookup\ h)\ (get\ s')\,.$$

Together with (9), the well-known anti-fusion law $map\ (f_1 \circ f_2) \equiv map\ f_1 \circ map\ f_2$, and (11), this implies

$$map\ f\ (get\ s') == map\ fromJust\ (map\ \mathsf{Just}\ v)\,,$$

which gives

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

by (10).

B. Proof of Theorem 3

Let $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$, let τ be a type that is an instance of Eq, and let $s :: [\tau]$. By the function definition for bff_{Eq} we have

$$\begin{array}{c} bff_{\mathsf{Eq}}\ get\ s\ (get\ s) \\ \equiv \\ seq\ h'\ (map\ (fromJust \circ flip\ \mathsf{IntMapEq}.lookup\ h')\ s')\,, \end{array} \tag{12}$$

where:

$$(s', g) \equiv template_{\mathsf{Eq}}\ s \tag{13}$$

$$h \equiv either\ error\ id\ (assoc_{\mathsf{Eq}}\ (get\ s')\ (get\ s)) \tag{14}$$

$$h' \equiv either\ error\ id\ (\mathsf{IntMapEq}.union\ h\ g)\,. \tag{15}$$

By (13) and Lemma 3, we have

$$map\ (flip\ \mathsf{IntMapEq}.lookup\ g)\ s' == map\ \mathsf{Just}\ s\,, \tag{16}$$

as well as that

- for every $i :: \mathsf{Int}$ not in s', $\mathsf{IntMapEq}.\mathit{lookup}\ i\ g \equiv \mathsf{Nothing}$, and that
- $\mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ g$ is injective on s'.

Consequently, setting

$$f \equiv \mathit{fromJust} \circ \mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ g\,, \tag{17}$$

we have

$$\mathit{map}\ f\ s' == s \tag{18}$$

and that f is injective on s'. By Lemma 6, this gives

$$\mathit{map}\ f\ (\mathit{get}\ s') == \mathit{get}\ s$$

and that every i in $\mathit{get}\ s'$ is also in s'. Together with (14) and Lemma 4, we can conclude that h is defined and that for every $i :: \mathsf{Int}$,

$$\begin{aligned}\mathsf{IntMapEq}.\mathit{lookup}\ i\ h == \mathbf{if}\ \mathit{elem}\ i\ (\mathit{get}\ s')\ &\mathbf{then}\ \mathsf{Just}\ (f\ i)\\ &\mathbf{else}\ \ \mathsf{Nothing}\,.\end{aligned}$$

On the other hand, we have by (16), (17), and the fact (derived above) that for every $i :: \mathsf{Int}$ not in s', $\mathsf{IntMapEq}.\mathit{lookup}\ i\ g \equiv \mathsf{Nothing}$, that for every $i :: \mathsf{Int}$,

$$\begin{aligned}\mathsf{IntMapEq}.\mathit{lookup}\ i\ g \equiv \mathbf{if}\ \mathit{elem}\ i\ s'\ &\mathbf{then}\ \mathsf{Just}\ (f\ i)\\ &\mathbf{else}\ \ \mathsf{Nothing}\,.\end{aligned}$$

Hence, by (15), the injectivity of $\mathit{flip}\ \mathsf{IntMapEq}.\mathit{lookup}\ g$ on s' (derived above), the fact (also derived above) that every i in $\mathit{get}\ s'$ is also in s', and the specification of $\mathsf{IntMapEq}.\mathit{union}$, we have that h' is defined and that for every i in s',

$$\mathsf{IntMapEq}.lookup\ i\ h' == \mathsf{Just}\ (f\ i)\,.$$

Together with (12) and (18), this gives

$$bff_{\mathsf{Eq}}\ get\ s\ (get\ s) == s\,.$$

C. Proof of Theorem 4

Let $get :: \forall\alpha.\ \mathsf{Eq}\ \alpha \Rightarrow [\alpha] \to [\alpha]$, let τ be a type that is an instance of Eq, and let $v, s :: [\tau]$. If $bff_{\mathsf{Eq}}\ get\ s\ v$ is defined, then we necessarily have

$$bff_{\mathsf{Eq}}\ get\ s\ v \equiv map\ f\ s'\,, \tag{19}$$

where

$$(s', g) \equiv template_{\mathsf{Eq}}\ s \tag{20}$$

$$\mathsf{Right}\ h \equiv assoc_{\mathsf{Eq}}\ (get\ s')\ v \tag{21}$$

$$\mathsf{Right}\ h' \equiv \mathsf{IntMapEq}.union\ h\ g \tag{22}$$

$$f \equiv fromJust \circ flip\ \mathsf{IntMapEq}.lookup\ h'\,. \tag{23}$$

By (20) and Lemma 3 we have that for every i in s', it holds $\mathsf{IntMapEq}.lookup\ i\ g \equiv \mathsf{Just}\ a$ for some $a :: \tau$. Moreover, by (21) and Lemma 5 we have

$$map\ (flip\ \mathsf{IntMapEq}.lookup\ h)\ (get\ s') == map\ \mathsf{Just}\ v\,, \tag{24}$$

as well as that

- for every $i :: \mathsf{Int}$ not in $get\ s'$, $\mathsf{IntMapEq}.lookup\ i\ h \equiv \mathsf{Nothing}$, and that
- $flip\ \mathsf{IntMapEq}.lookup\ h$ is injective on $get\ s'$.

Putting all these facts together with (22), the specification of $\mathsf{IntMapEq}.union$, and (23), we get that f is injective on s'. Thus, by (19) and Lemma 6,

$$get\ (bff_{\mathsf{Eq}}\ get\ s\ v) == map\ f\ (get\ s')\,. \tag{25}$$

The remainder of the proof is analogous to the second half of that for Theorem 2 in Appendix A, where now (22)–(25) take the roles of (8)–(11).