JFP **23** (5): 515–551, 2013. © Cambridge University Press 2013
doi:10.1017/S0956796813000130 First published online 14 October 2013

Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins

JANIS VOIGTLÄNDER
University of Bonn, 53113 Bonn, Germany
(*e-mail:* `jv@informatik.uni-bonn.de`)

ZHENJIANG HU
National Institute of Informatics, Tokyo 101-8430, Japan
(*e-mail:* `hu@nii.ac.jp`)

KAZUTAKA MATSUDA
University of Tokyo, Tokyo 113-0033, Japan
(*e-mail:* `kztk@is.s.u-tokyo.ac.jp`)

MENG WANG
Chalmers University of Technology, 412 96 Gothenburg, Sweden
(*e-mail:* `wmeng@chalmers.se`)

Abstract

Matsuda *et al.* (Matsuda, K., Hu, Z., Nakano, K., Hamana, M. & Takeichi, M. (2007) Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proceedings of the International Conference on Functional Programming*. ACM Press, pp. 47–58) and Voigtländer (Voigtländer, J. (2009) Bidirectionalization for free! In *Proceedings of Principles of Programming Languages*. ACM Press, pp. 165–176) have introduced two techniques that given a source-to-view function provide an update propagation function mapping an original source and an updated view back to an updated source, subject to standard consistency conditions. Previously, we developed a synthesis of the two techniques, based on a separation of shape and content aspects (Voigtländer, J., Hu, Z., Matsuda, K. & Wang, M. (2010) Combining syntactic and semantic bidirectionalization. In *Proceedings of the International Conference on Functional Programming*. ACM Press, pp. 181–192). Here we carry that idea further, reworking the technique of Voigtländer such that *any* shape bidirectionalizer (based on the work of Matsuda *et al.* (2007) or not) can be used as a plug-in, to good effect. We also provide a data-type-generic account, enabling wider reuse, including the use of pluggable bidirectionalization itself as a plug-in.

1 Introduction

Bidirectionalization is the task of, given some function $get :: \tau_1 \to \tau_2$, producing a function $put :: \tau_1 \to \tau_2 \to \tau_1$ such that if get maps an *original source* s to an *original view* v, and v is somehow changed into an *updated view* v', then put applied to s and v' produces an *updated source* s' in a meaningful way.

$$\begin{tikzcd} s \arrow[rr, Rightarrow, "get"] \arrow[d, Rightarrow, bend right] & & v \arrow[d, dotted, "\text{Update}"] \\ s' & & v' \arrow[ll, Rightarrow, "put"'] \end{tikzcd}$$

Such *get*/*put*-pairs, called bidirectional transformations, play an important role in various application areas such as databases, file synchronization, structured editing, and model transformation. Czarnecki *et al.* (2009) survey relevant techniques and open problems. Functional programming approaches have had an important impact, with several ideas and solutions springing from this part of the programming languages field in particular (Bohannon *et al.*, 2006, 2008; Foster *et al.*, 2007, 2008, 2012; Matsuda *et al.*, 2007, 2009; Hu *et al.*, 2008; Voigtlander, 2009, 2012; Hidaka *et al.*, 2010; Pacheco & Cunha, 2010; Pacheco *et al.*, 2012; Wang *et al.*, 2011, 2013; Matsuda & Wang, 2013).

Automatic bidirectionalization is one approach to obtaining suitable *get*/*put*-pairs; others are domain-specific languages or more *ad hoc* programming techniques. Two different flavors of bidirectionalization have been proposed: syntactic and semantic. Syntactic bidirectionalization (Matsuda *et al.*, 2007) works on a syntactic representation of (somehow restricted) *get*-functions and synthesizes appropriate definitions for *put*-functions algorithmically. Semantic bidirectionalization (Voigtländer, 2009) does not inspect the syntactic definitions of *get*-functions at all, but instead provides a single definition of *put*, parameterized over *get* as a semantic object that does the job by invoking *get* in a kind of 'simulation mode'.

Both syntactic and semantic bidirectionalization have their strengths and weaknesses. Syntactic bidirectionalization depends heavily on syntactic restraints exercised when implementing the *get*-function. Basically, the technique of Matsuda *et al.* (2007) can only deal with programs in a custom first-order language subject to certain restrictions concerning variable use and nested function calls. Semantic bidirectionalization, in contrast, provides very easy access to bidirectionality within a general-purpose language, liberated from the syntactic corset as to how to write functions of interest. The price to pay for this in the case of the approach of Voigtländer (2009) is that it works for polymorphic functions only, and in the original form is unable to deal with view updates that change the shape of a data structure. The syntactic approach, on the other hand, is successful for many such shape-changing updates, and can deal with non-polymorphic functions.

Voigtländer *et al.* (2010) developed an approach for combining syntactic and semantic bidirectionalization. The resulting technique inherits the limitations in program coverage from *both* techniques, but gains improved updateability: more (s, v') pairs can successfully be mapped to a suitable s' by *put* (see the next section for a more formal conceptualization). Specifically, semantic bidirectionalization now gets a chance to deal with shape-changing updates, and the combined technique is superior to syntactic bidirectionalization on its own in many cases (and actually never worse than the better of the two original techniques). The combination strategy we pursued was essentially motivated by combining the specialties of the two approaches. Semantic bidirectionalization's specialty is to employ polymorphism to deal with the content elements of data structures in a very lightweight way. In fact, in the original technique, the shape and content aspects of a data structure are completely separated, updates affecting the shape are completely outlawed, arbitrary updates to content elements can be simply absorbed, and by recombining original shape with updated content the desired update consistency is guaranteed. Syntactic bidirectionalization's specialty is to have a more refined, and case-by-case, notion of what updates, including updates on the shape aspect, can be permitted. But it turns out that content elements often get in the way. In fact, by having to deal with both shape and content, at the same time, in the key step of syntactic bidirectionalization (namely 'view complement derivation'), updateability is hampered. In our combined approach we divided the labor: semantic bidirectionalization deals with content only, syntactic bidirectionalization deals with shape only. As a result, the reach of semantic bidirectionalization is expanded beyond shape-preserving updates, and syntactic bidirectionalization is invoked on a more specialized kind of program, on which it can yield better results, benefitting both.

In the current paper, we carry the idea further: since the combined approach essentially treats syntactic bidirectionalization as a black box, we can consider it as a completely external component, and indeed replace it by *other* approaches (than that of Matsuda *et al.* (2007)) for obtaining bidirectional transformations on shapes. Any such approach can be 'plugged into' the semantic technique (after suitable dissection/refactoring of the latter). We develop the details and consider some such plug-ins, including (in Section 5.3) of course the specific use case of combining the techniques of Matsuda *et al.* (2007) and Voigtländer (2009), thus covering the results of Voigtländer *et al.* (2010). We also generalize from lists (to which case the development of Voigtländer *et al.* (2010) was restricted) to more general data types (in Section 7). As a bonus, this enables a kind of bootstrapping in which pluggable bidirectionalization is itself used as a plug-in.

2 Consistency conditions and language setup

To explain what we mean by improved updateability, we have to elaborate on the phrase 'in a meaningful way' in the first sentence of the Introduction, and on 'suitable' at the start of its second paragraph. So, when is a *get*/*put*-pair 'good'? How should s, v, v', and s' in $get\ s \equiv v$ and $put\ s\ v' \equiv s'$ be related? One natural requirement is that if $v \equiv v'$, then $s \equiv s'$, or, put differently,

$$put\ s\ (get\ s) \equiv s\,. \tag{1}$$

Another requirement to expect is that s' and v' should be related in the same way as s and v are, or, again expressed as a round-trip property,

$$get\ (put\ s\ v') \equiv v'\,. \tag{2}$$

These are the standard consistency conditions (Bancilhon & Spyratos, 1981) known as GetPut and PutGet (Foster *et al.*, 2007). But the latter of the two is often too hard to satisfy in practice. For fixed *get*, it can be impossible to provide a *put*-function fulfilling Equation (2) for every choice of s and v', simply because v' may not even be in the range of *get*. One solution is to make the *put*-function partial and to only expect the PutGet law to hold in case $put\ s\ v'$ is actually defined. Of course, a trivially consistent *put*-function we could then always come up with is the one for which $put\ s\ v'$ is *only* defined if $get\ s \equiv v'$ and which simply returns s then. Clearly, this choice would satisfy both Equations (1) and (2), but would be utterly useless in terms of updateability. The very idea that v and v' can be different in the original scenario would be countermanded.

So our evaluation criteria for 'goodness' are that *get*/*put* should satisfy Equation (1), that they should satisfy Equation (2) whenever $put\ s\ v'$ is defined, and that $put\ s\ v'$ should be actually defined on a big part of its potential domain, indeed preferably for all s and v' of appropriate type. That measure, simply comparing the sizes of the applicability domains of *put*, is somewhat coarse, but we will also discuss finer distinctions (i.e., concerning *what* two different *put*-functions map a given (s,v') pair to) later.

Since our emphasis is on the updateability inherent in a *get*/*put*-pair, we make the partiality of *put* explicit in the type (and make the function itself total) via optionality of the return value, using the data type definition

$$\textbf{data}\ \mathsf{Maybe}\ \alpha = \mathsf{Nothing} \mid \mathsf{Just}\ \alpha$$

The following definition formulates the consistency conditions for this setting.

Definition 1. Let τ_1 and τ_2 be types. Let functions $get :: \tau_1 \to \tau_2$ and $put :: \tau_1 \to \tau_2 \to \mathsf{Maybe}\ \tau_1$ be given. We say that *put* is *consistent for get* if:

- For every $s :: \tau_1$, $put\ s\ (get\ s)\ \equiv\ \mathsf{Just}\ s$.
- For every $s, s' :: \tau_1$ and $v' :: \tau_2$, if $put\ s\ v'\ \equiv\ \mathsf{Just}\ s'$, then $get\ s'\ \equiv\ v'$.

We work in Haskell (Peyton Jones, 2003) with a few GHC extensions, almost: one deviation we make is that we assume that the type Int is replaced, throughout the language, by a type Nat, discarding all negative integers. In particular, the function *length* on lists will have type $[\alpha] \to \mathsf{Nat}$ instead of $[\alpha] \to \mathsf{Int}$, and we assume we are given a variant Data.NatMap of standard module Data.IntMap. We use $\equiv$ for semantic equivalence, but specialize notation to $=$ for natural numbers. All functions, from now on, are assumed to be total, except where partiality is explicitly mentioned.

3 The original semantic bidirectionalization technique

We briefly introduce the technique of Voigtländer (2009). For the moment, we only consider the case of lists, and throughout, only parametrically polymorphic functions to bidirectionalize (no *ad hoc* polymorphism). So we consider functions $get :: [\alpha] \to [\alpha]$.

The intuition underlying the method of Voigtländer (2009) is that *put* can gain information about the *get*-function by applying it to suitable inputs. The key is that *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 be observed explicitly by applying *get* to lists with fixed distinct elements. Particularly convenient are ascending lists over the natural numbers. Say *get* is *tail*, then every list $[1,\ldots,n]$ is mapped to $[2,\ldots,n]$, which allows *put* 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 $[1,\ldots,n]$ just as well, even to lists over non-number types, thanks to parametric polymorphism (Strachey, 2000; Reynolds, 1983). Specifically, it is easy to derive from a 'free theorem' (Wadler,

1989) that for every $get :: [\alpha] \to [\alpha]$ and every list s, over arbitrary type, it holds that with $n = length\ s$ and $t' \equiv get\ [1\,..\,n]$,

$$length\ (get\ s) = length\ t' \tag{3}$$

as well as for every $1 \leqslant j \leqslant length\ (get\ s)$,

$$(get\ s)\ !!\ (j-1) \equiv s\ !!\ ((t'\ !!\ (j-1)) - 1)\,. \tag{4}$$

(Note that !!, Haskell's operator for indexing into lists, starts counting from 0, hence the occurrences of -1.) Putting the above as a picture:

$$\begin{tikzcd}
{[1,\ldots,n]} \arrow[r, Rightarrow, "get"] & {[t'_1,\ldots,t'_m]} \arrow[d, Rightarrow, "\text{implies}"] \\
{[s_1,\ldots,s_n]} \arrow[r, Rightarrow, "get"'] & {[s_{t'_1},\ldots,s_{t'_m}]}
\end{tikzcd}$$

(where $[t'_1,\ldots,t'_m]$ could be the empty list, just as $[1\,..\,n] \equiv [1,\ldots,n]$ could be).

Let us further consider the *tail* example as in the middle of the previous paragraph. First, *put* should find out to what element in an original source s each element in an updated view v' corresponds. Assume s has length n. Then by applying *tail* to the same-length list $[1\,..\,n]$, *put* learns that the original view from which v' was obtained by updating had length $n-1$, and also to what element in s each element in that original view corresponded. Being conservative, the original semantic bidirectionalization method will only accept v' if it has retained that length $n-1$. 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 $[1\,..\,n]$ and fill each with the associated value from v'. For positions for which there is no corresponding value in v', because these positions were omitted when applying *tail* to $[1\,..\,n]$, we can look up the correct value in s rather than in v'. For the concrete example, this will only concern position 1, for which we naturally take over the head element from s (also see the picture below).

Actually, 'going over all positions in $[1\,..\,n]$ and filling each with *the* associated value from v' (or from s if non-existent in v')' can be a problematic task: what if *two* values in v' would associate with the same index position (as could easily happen if instead of *tail* we have a *get*-function that duplicates some of its list elements)? Ignoring that difficulty for the moment (but coming back to it soon in Section 4.1), the above strategy works for general *get*. In short, given s, produce a 'template' $t \equiv [1\,..\,n]$ of the same length, together with an association g between natural numbers in that template and the corresponding values in s. Then apply *get* to t and produce a further association h by matching this template view t' with the updated proper value view v'. Combine the two associations into a single one h', giving precedence to h whenever a natural numbers template index is found in both h and g (or first reduce g to g' by discarding all entries for index values that occur in $get\ [1\,..\,n]$; h and g' will then have disjoint domains and together exactly cover $\{1,\ldots,n\}$). Finally, produce an updated source by filling all positions in $[1\,..\,n]$ with their associated values according to h'. So for $s \equiv [s_1,\ldots,s_n]$ and $v' \equiv [x_1,\ldots,x_m]$, set $put\ s\ v' \equiv \mathsf{Just}\ [y_1,\ldots,y_n]$, where $y_i \equiv x_j$ if $i = t'_j$ (with $get\ [1\,..\,n] \equiv [t'_1,\ldots,t'_m]$), and $y_i \equiv s_i$ otherwise. On the *tail* example:

$$\begin{tikzcd}
 & {[s_1,\ldots,s_n]} \arrow[r, Rightarrow, "tail"] \arrow[dl, Rightarrow, bend left] & {[s_2,\ldots,s_n]} \arrow[d, dotted, "\text{Update}"] \\
{[s_1,x_1,\ldots,x_{n-1}]} & & {[x_1,\ldots,x_{n-1}]} \arrow[ll, Rightarrow, "put"']
\end{tikzcd} \tag{5}$$

since $tail\,[1\,..\,n] \equiv [2\,..\,n]$ and hence $2 = t'_1, 3 = t'_2, \ldots, n = t'_{n-1}$, and $1 \neq t'_j$ for all j.

The above strategy (plus proper handling of *get*-functions that duplicate list elements) is what Voigtländer (2009) implements (for the special case $get :: [\alpha] \to [\alpha]$). We recall the corresponding Haskell definitions, reformulating just a bit by writing the higher-order *bff*-function[1] that turns *get* into *put* in monadic style (Wadler, 1992) to provide for more convenient error handling.[2] The type class constraints Eq are for ensuring that list entries (of abstract type α) can be compared for equality using ==, as needed in *assoc*.[3]

```
bff :: Monad μ ⇒ (∀α.[α] → [α]) → (∀α.Eq α ⇒ [α] → [α] → μ [α])
bff get s v′ = do let n = length s
                  let t = [1..n]
                  let g = NatMap.fromDistinctAscList (zip t s)
                  let t′ = get t
                  let g′ = foldr NatMap.delete g t′
                  h ← assoc t′ v′
                  let h′ = NatMap.union h g′
                  return (map (fromJust ∘ flip NatMap.lookup h′) t)
assoc :: (Monad μ, Eq α) ⇒ [Nat] → [α] → μ (NatMap α)
assoc []     []     = return NatMap.empty
assoc (i:is) (b:bs) = do m ← assoc is bs
                         case NatMap.lookup i m of
                           Nothing → return (NatMap.insert i b m)
                           Just c  → if b == c
                                       then return m
                                       else fail "Update violates equality."
assoc _      _      = fail "Update changes the length."
```

We use (here and later) some functions from an assumed module Data.NatMap. Their type signatures, which should provide sufficient documentation, are given as follows[4]:

```
fromDistinctList    :: [(Nat, α)] → NatMap α
fromDistinctAscList :: [(Nat, α)] → NatMap α
empty               :: NatMap α
insert              :: Nat → α → NatMap α → NatMap α
delete              :: Nat → NatMap α → NatMap α
union               :: NatMap α → NatMap α → NatMap α
lookup              :: Nat → NatMap α → Maybe α
```

[1] The name *bff* is an abbreviation of the paper title 'Bidirectionalization for Free!'.

[2] We will only ever specialize μ to Maybe in the paper, but when running the code it is convenient to be able to (even silently) use an arbitrary monad. For example, just running examples in the interpreter can directly use the IO monad and thus give unwrapped outputs – unless there is a failure, of course. Monads in Haskell are not plain monads; they include a *fail* method. In the Maybe monad, we have $fail\ s =$ Nothing.

[3] Note that == is programmed equality, not in general semantic equivalence $\equiv$.

[4] The function *fromDistinctAscList* can assume that its argument has Nat keys in ascending order, and thus can work more efficiently than plain *fromDistinctList*.

Actually, Voigtländer (2009), and also Voigtländer *et al.* (2010), use slight variants (conceptually) of *bff* above where $g' \equiv g$, i.e., the line **let** $g' = \mathit{foldr}\ \mathsf{NatMap}.\mathit{delete}\ g\ t'$ is not present. The variation to use the reduced g' comes from Foster *et al.* (2012). This version is semantically equivalent to the earlier versions (since $\mathsf{NatMap}.\mathit{union}$ is assumed to be left-biased for natural numbers occurring as keys in both its input maps), but a difference appears when one starts to refactor *bff* to deal with view updates that change the shape. Then, in this paper from Section 4.4 onwards, the choice of g' versus g becomes relevant. We will return to this discussion there, after Theorem 6.

The following theorem is essentially (up to the different way of expressing partiality of $\mathit{put} \equiv \mathit{bff}\ \mathit{get}$) what is proved by Voigtländer (2009) in his Theorems 1 and 2, based on the key statements (3) and (4).

Theorem 1. Let $\mathit{get} :: [\alpha] \to [\alpha]$ and let τ be a type that is an instance of the type class Eq in such a way that the definition given for $==$ makes it reflexive, symmetric, and transitive.

- For every $s :: [\tau]$, $\mathit{bff}\ \mathit{get}\ s\ (\mathit{get}\ s) :: \mathsf{Maybe}\ [\tau] \equiv \mathsf{Just}\ s$.
- For every $s, v', s' :: [\tau]$, if $\mathit{bff}\ \mathit{get}\ s\ v' :: \mathsf{Maybe}\ [\tau] \equiv \mathsf{Just}\ s'$, then $\mathit{get}\ s' == v'$.

Corollary 1. Let $\mathit{get} :: [\alpha] \to [\alpha]$ and let τ be a type that is an instance of Eq in such a way that the definition given for $==$ agrees with $\equiv$. Then

$$\mathit{bff}\ \mathit{get} :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is consistent (in the sense of Definition 1) for $\mathit{get} :: [\tau] \to [\tau]$.

Applying semantic bidirectionalization is very easy. We simply call *bff* with the *get*-function we want to bidirectionalize. The following two examples will also be used for later discussions.

Running Example 1. Assume our *get*-function is such that it sieves a list to keep only every second element, as exemplified with the following calls:

s	`""`	`"a"`	`"ab"`	`"abc"`	`"abcd"`	`"abcde"`	$[1,2,3,4,5]$
$\mathit{get}_1\ s$	`""`	`""`	`"b"`	`"b"`	`"bd"`	`"bd"`	$[2,4]$

Then here are the results of a few representative calls to $\mathit{bff}\ \mathit{get}_1$ (the results of the relevant calls to get_1 are all the same):

s	$\mathit{get}_1\ s$	v'	$\mathit{bff}\ \mathit{get}_1\ s\ v'$
`"abcd"`	`"bd"`	`"x"`	Nothing
`"abcd"`	`"bd"`	`"xy"`	Just `"axcy"`
`"abcd"`	`"bd"`	`"xyz"`	Nothing
`"abcde"`	`"bd"`	`"x"`	Nothing
`"abcde"`	`"bd"`	`"xy"`	Just `"axcye"`
`"abcde"`	`"bd"`	`"xyz"`	Nothing

We see (and indeed it holds in general) that $\mathit{bff}\ \mathit{get}_1\ s\ v'$ fails if and only if $\mathit{length}\ (\mathit{get}_1\ s) \neq \mathit{length}\ v'$. If it succeeds, it mixes the elements of s and v' in an appropriate fashion. In a similar fashion as earlier for the *tail* example, see (5), the behavior here (specifically, the last but one line of the second table above) can be explained as follows:

$$\begin{tikzcd}
{[s_1,\ldots,s_5]} \arrow[r, Rightarrow, "get_1"] \arrow[d, Rightarrow, bend left] & {[s_2,s_4]} \arrow[d, dotted, "\text{Update}"] \\
{[s_1,x_1,s_3,x_2,s_5]} & {[x_1,x_2]} \arrow[l, Rightarrow, "put"']
\end{tikzcd} \tag{6}$$

due to $get_1\ [1\,..\,5] \equiv [2,4]$ and hence $2 = t'_1$, $4 = t'_2$, and $1,3,5 \neq t'_j$ for all j.

Running Example 2. Assume our *get*-function is such that it keeps every element of a list except for the last one, e.g.:

s	`""`	`"a"`	`"ab"`	`"abc"`	`"abcd"`	`"abcde"`	$[1,2,3,4,5]$
$get_2\ s$	`""`	`""`	`"a"`	`"ab"`	`"abc"`	`"abcd"`	$[1,2,3,4]$

Again, semantic bidirectionalization allows no view updates that change the shape, so $bff\ get_2\ s\ v'$ will only be successful if $length\ (get_2\ s) = length\ v'$, e.g.:

s	$get_2\ s$	v'	$bff\ get_2\ s\ v'$
`""`	`""`	`""`	`Just ""`
`""`	`""`	`"x"`	`Nothing`
`"a"`	`""`	`""`	`Just "a"`
`"a"`	`""`	`"x"`	`Nothing`
`"ab"`	`"a"`	`""`	`Nothing`
`"ab"`	`"a"`	`"x"`	`Just "xb"`
`"ab"`	`"a"`	`"xy"`	`Nothing`
`"abc"`	`"ab"`	`"x"`	`Nothing`
`"abc"`	`"ab"`	`"xy"`	`Just "xyc"`

4 Refactoring semantic bidirectionalization to enable 'plug-ins'

In order to motivate our next moves, let us consider the case $bff\ get_1$ `"abcde"` `"xyz"` $\equiv$ Nothing from Example 1 in the previous section. What makes

$$\begin{tikzcd}
{[s_1,\ldots,s_5]} \arrow[r, Rightarrow, "get_1"] \arrow[d, Rightarrow, bend left] & {[s_2,s_4]} \arrow[d, dotted, "\text{Update}"] \\
? & {[x_1,x_2,x_3]} \arrow[l, Rightarrow, "put"']
\end{tikzcd}$$

fail, in contrast to (6), is that we cannot simply (as there) set $put\ [s_1,\ldots,s_5]\ [x_1,x_2,x_3] \equiv$ Just $[y_1,\ldots,y_5]$ with the y_i appropriately chosen from among the s_i and x_j. After all, no such choice(s) will ever make $get_1\ [y_1,\ldots,y_5] \equiv [x_1,x_2,x_3]$ true, as the second of our consistency conditions would demand (cf. Definition 1). The problem is that length $n = 5$ on the input side does not fit length $m = 3$ on the (updated) output side. If, however, from

$n = 5$ and $m = 3$ we could deduce an appropriate choice of length for the updated input list, say $n' = 6$ (or alternatively, $n' = 7$), then we could set the desired s' to $[y_1, \ldots, y_6]$ with $y_i \equiv x_j$ if $i = t'_j$, where $get_1\ [1\,..\,6] \equiv [t'_1, t'_2, t'_3]$, and $y_i \equiv s_i$ otherwise.[5]

Determining n' – given *get*, n, and m – can be considered as a separate problem, which is not solved (or solvable) by the semantic bidirectionalization technique itself. The idea now, already of Voigtländer *et al.* (2010), is to outsource this separate problem. If we abstract from the concrete list elements, instead considering (in the case of the above example) the following problem on the shape level:

$$\begin{tikzcd}
{[\cdot,\cdot,\cdot,\cdot,\cdot]} \arrow[r, Rightarrow, "get'_1"] \arrow[d, Rightarrow, bend left] & {[\cdot,\cdot]} \arrow[d, dotted, "\text{Update}"] \\
? & {[\cdot,\cdot,\cdot]} \arrow[l, Rightarrow, "put'"']
\end{tikzcd}$$

then maybe we are in better luck. Indeed, Voigtländer *et al.* (2010) showed that the syntactic bidirectionalization technique of Matsuda *et al.* (2007) can profit from such abstraction, and successfully generate *put′* (on the shape level) even in cases where it fails to generate (an as useful) *put* for the original problem. Next, we show how to go about this separation of concerns in general, and to prepare integration of arbitrary shape bidirectionalizer plug-ins (that of Matsuda *et al.* (2007) or others). But beforehand, there is one more issue to consider.

The issue is that we have not yet discussed here, although of course we did so in the original work on semantic bidirectionalization (Voigtländer, 2009, in detail at the end of Section 2 and start of Section 3), whether/how to deal with *get*-functions that duplicate input list elements. Here we shall shortly outlaw any duplication of list elements. Formally, we will consider only functions $get :: [\alpha] \to [\alpha]$ such that for every $n :: \mathsf{Nat}$, $get\ [1\,..\,n]$ contains no duplicates. We call such a function *semantically affine*. The property will clearly be fulfilled if *get*'s syntactic definition is affine (i.e., if no variable occurs more than once on a single right-hand side), but it can also hold in other cases. In the next section we explain why we need this restriction; the reader less interested in the theoretical argument may want to skip that and go directly to Section 4.2. On the practical side, the restriction to semantically affine *get*-functions does not cost us all too much *additionally* in terms of reducing reach. In particular, the syntactic bidirectionalization technique of Matsuda *et al.* (2007), with which we combine semantic bidirectionalization in Section 5.3 as one chief result (Voigtländer *et al.*, 2010) of our overall approach, is itself already unable to deal with *syntactically* non-affine functions.

4.1 Semantic bidirectionalization and duplication of list elements

So how can semantic bidirectionalization deal with *get*-functions that duplicate input list elements? First of all, this issue is exactly what leads to the somewhat complicated

[5] If we were to, alternatively, choose $n' = 7$, then some default value would have to be brought into play, because for $i = 7$ there is no x_j with $7 = t'_j$, but also no s_7.

definition of *assoc* in the previous section and to the references to Eq and $==$ in the function definitions and in Theorem 1 and Corollary 1. More specifically, the need arises because we want to ensure that *bff* satisfies some form of the PutGet law (ultimately, expressed as the second of the two points stated in Theorem 1). The problem for *bff* is to come up with an s' such that

$$length\ (get\ s') = length\ v'$$

and for every $1 \leqslant j \leqslant length\ (get\ s')$, ideally

$$(get\ s')\,!!\,(j-1) \ \equiv\ v'\,!!\,(j-1)\,,$$

though we would actually be content with $==$ instead of $\equiv$. What do we have to go by for proving these two statements? Well, of course Equations (3) and (4), applied to s'. For $length\ (get\ s') = length\ v'$, Equation (3) already does half the job, leaving us with the proof obligation

$$length\ t' = length\ v'\ . \tag{7}$$

But this equality is ensured if the *assoc*-call on t' and v' inside *bff* succeeded (and only then do we have to prove anything about s' at all, cf. the exact formulation of the second point stated in Theorem 1). For $(get\ s')\,!!\,(j-1) \ \equiv\ v'\,!!\,(j-1)$, Equation (4) looks quite useful, leaving us with the proof obligation

$$s'\,!!\,((t'\,!!\,(j-1))-1) \ \equiv\ v'\,!!\,(j-1)\,, \tag{8}$$

where $n = length\ s'$ and $t' \ \equiv\ get\ [1\,..\,n]$. And indeed, the fact that when filling up s' in the last line of the definition of *bff*, lookup for index positions that are elements of t' leads to lookup in h – which was obtained through associating, by position, elements of t' and of v' – seems to indicate that we are fine. *But*, actually Equation (8) is treacherous as a 'definition' (implemented in *bff*) for elements of s' at positions corresponding to elements of t'. It is not actually well-defined in general! What if there are j and j' such that $t'\,!!\,(j-1) = t'\,!!\,(j'-1)$? Then Equation (8) tries to 'assign' two potentially different values $v'\,!!\,(j-1)$ and $v'\,!!\,(j'-1)$ to the same position of s'. The *assoc*-function as given in the previous section prevents this from happening, at the price of additionally performing equality checks (using $==$).

As a concrete example, consider a function *get* that maps every singleton list s to $s +\!\!+ s$ (and all other lists to $[\,]$, say). Let us look at a specific case $s =$ `"a"`, and suppose the view `"aa"` is updated to `"bc"`. Should we take this as suggesting a replacement of `'a'` by `'b'` or by `'c'`, i.e., do we want *bff get* `"a"` `"bc"` to be Just `"b"` or Just `"c"`? Neither makes sense, since neither *get* `"b"` nor *get* `"c"` is `"bc"`. So the only possibility for *bff* is to return Nothing.

Restricting to semantically affine *get*-functions is not just the easiest way out of the complications described above. At a deeper level, it is really fundamental to a successful separate treatment of shapes as we have in mind. If the problem of coming up, for given v' (and s), with an s' such that *get* s' equals v' is to be decomposed in such a way that we first try to determine the length of s' from only the length of v' (and that of s), then we cannot afford to have to be concerned about the inner structure of $get\ [1\,..\,n]$ (or actually of $get\ [1\,..\,n']$ for some new n' potentially different from $n = length\ s$) in order to eventually make sense of Equation (8). Instead, we should only have to be concerned about the shape aspects, as in the other proof obligation (7). And indeed, if given the length of v' (and that of s) we manage to find an n' such that Equation (7) holds for $t' \ \equiv\ get\ [1\,..\,n']$, then only under the semantic affineness assumption will we be guaranteed – no matter how v' looks internally – to be able to fill the list positions of s' (of length n') in such a way that *get* s' equals v'. This is because Equation (8) is not only sufficient but actually necessary, and as soon as t' contains duplicates an adversary could come up with list elements for v' such that s' cannot exist (not even after replacing $\equiv$ by $==$ in Equation (8); assuming the type of elements of v' has at least two non-equal values).

4.2 Specializing semantic bidirectionalization to semantically affine get-functions

We define

$$\mathit{bff}_{\text{affine}} :: \mathsf{Monad}\ \mu \Rightarrow (\forall\alpha.[\alpha] \to [\alpha]) \to (\forall\alpha.[\alpha] \to [\alpha] \to \mu\ [\alpha])$$

like *bff* (but note the different type), except that the call to *assoc* is replaced by a call, with the same arguments, to the following function, not performing any equality checks:

```
$assoc' :: \mathsf{Monad}\ \mu \Rightarrow [\mathsf{Nat}] \to [\alpha] \to \mu\ (\mathsf{NatMap}\ \alpha)$
$assoc'\ [\,]\qquad [\,]\qquad = return\ \mathsf{NatMap}.empty$
$assoc'\ (i:is)\ (b:bs) = \mathbf{do}\ m \leftarrow assoc'\ is\ bs$
                                  $return\ (\mathsf{NatMap}.insert\ i\ b\ m)$
$assoc'\ \_\qquad \_\qquad = fail$ "Update changes the length."
```

The proof of the following theorem is very similar to that of Theorem 1, additionally using semantic affineness of *get* in a straightforward way.

Theorem 2. Let $get :: [\alpha] \to [\alpha]$ be semantically affine. For every type τ,

$$\mathit{bff}_{\text{affine}}\ get :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is consistent for $get :: [\tau] \to [\tau]$.

But semantic affineness gives us more. It rules out one important cause (namely potential equality mismatch in v') for a potential failure of view update. As a consequence, we can now formulate a sufficient condition for a successful update.

Definition 2. We say that a function $put :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$ (for some type τ) is *fixed-shape-friendly for get* $:: [\tau] \to [\tau]$ if for every $s, v' :: [\tau]$, if $length\ (get\ s) = length\ v'$, then $put\ s\ v' \ \equiv\ \mathsf{Just}\ s'$ for some $s' :: [\tau]$.

Note that the original $\mathit{bff}\ get :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$ from Section 3 is not in general fixed-shape-friendly for *get*-functions that are not semantically affine. On the other hand, $\mathit{bff}_{\text{affine}}\ get :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$ is not even generally *consistent* for *get*-functions that are not semantically affine. But when we *do* restrict *get*-functions to be semantically affine, we have consistency by the above theorem, and can moreover prove the following one.

Theorem 3. Let $get :: [\alpha] \to [\alpha]$ be semantically affine. For every type τ,

$$\mathit{bff}_{\text{affine}}\ get :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is fixed-shape-friendly for $get :: [\tau] \to [\tau]$.

For the proof, we basically just observe that the last defining equation of $assoc'$ will never be reached if the argument lists are of the same length.

We can also give a negative statement about updateability (which also holds for the *bff* from Section 3, of course).

Theorem 4. Let $get :: [\alpha] \to [\alpha]$. For every type τ and $s, v' :: [\tau]$, if $length\ (get\ s) \neq length\ v'$, then $bff_{\mathrm{affine}}\ get\ s\ v' :: \mathsf{Maybe}\ [\tau] \ \equiv\ \mathsf{Nothing}$.

For the proof, we observe that the last defining equation of $assoc'$ (or *assoc*) *is* reached if the argument lists are of different lengths.

4.3 Decomposing to expose the shape aspect

We refactor bff_{affine} to make the treatment of shapes (list lengths) more explicit. To that end, we first define a function $sput_{\mathrm{naive}}$, depending on $get :: [\alpha] \to [\alpha]$, as follows:

```
sput_naive :: Monad μ ⇒ (∀α.[α] → [α]) → (Nat → Nat → μ Nat)
sput_naive get l_s l_v' = if length (get [1..l_s]) == l_v'
                              then return l_s
                              else fail "Update changes the length."
```

Using that function, we then define bff_{refac} as follows:

```
bff_refac :: Monad μ ⇒ (∀α.[α] → [α]) → (∀α.[α] → [α] → μ [α])
bff_refac get s v' = do let n = length s
                        let t = [1..n]
                        let g = NatMap.fromDistinctAscList (zip t s)
                        let g' = foldr NatMap.delete g (get t)
                        n' ← sput_naive get n (length v')
                        let t = [1..n']
                        let h = NatMap.fromDistinctList (zip (get t) v')
                        let h' = NatMap.union h g'
                        return (map (fromJust ∘ flip NatMap.lookup h') t)
```

The refactoring consists of

- making the check for equal length of $get\ [1\,..\,(length\ s)]$ and v', otherwise performed inside $assoc'$, explicit, and outsourcing it to $sput_{\mathrm{naive}}$, and
- realizing that once this check was successful, the role of $assoc'$ can be taken over by *zip* and NatMap.*fromDistinctList*.

Note that the second local binding for t inside bff_{refac} shadows the earlier one, but that actually the two values bound will be identical here, since due to the behavior of $sput_{\mathrm{naive}}$, if the second binding is reached at all, then n' will be identical to n. (That will change in the next section.)

The following lemma establishes that the above refactoring is indeed correct, and thus transports the (good and bad) properties of bff_{affine} to bff_{refac}.

Lemma 1. Let $get :: [\alpha] \to [\alpha]$. For every type τ and $s, v' :: [\tau]$, we have

$$bff_{\mathrm{affine}}\ get\ s\ v' :: \mathsf{Maybe}\ [\tau] \ \equiv\ bff_{\mathrm{refac}}\ get\ s\ v' :: \mathsf{Maybe}\ [\tau]\,.$$

Corollary 2. Let $get :: [\alpha] \to [\alpha]$ be semantically affine. For every type τ,

$$bff_{\mathrm{refac}}\ get :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is consistent for $get :: [\tau] \to [\tau]$.

Corollary 3. Let $get :: [\alpha] \to [\alpha]$ be semantically affine. For every type τ,

$$bff_{\text{refac}}\ get :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is fixed-shape-friendly for $get :: [\tau] \to [\tau]$.

Corollary 4. Let $get :: [\alpha] \to [\alpha]$. For every type τ and $s, v' :: [\tau]$, if $length\ (get\ s) \neq length\ v'$, then $bff_{\text{refac}}\ get\ s\ v' :: \mathsf{Maybe}\ [\tau] \ \equiv\ \mathsf{Nothing}$.

The motivation for our refactoring above is that we make explicit, in $sput_{\text{naive}}$, what happens on the shape level, namely that only updated views with the same length as the original view can be accepted, and that the length of the source will never be changed. By 'playing' with $sput_{\text{naive}}$, or rather replacing it, we can change that behavior.

4.4 Enabling 'plug-ins'

The key idea in the previous section is abstraction: from lists to list lengths (generally, from data structures to their shapes). We can define a function *shapify* as follows:

$$\begin{aligned}&shapify :: (\forall \alpha.[\alpha] \to [\alpha]) \to (\mathsf{Nat} \to \mathsf{Nat})\\ &shapify\ get\ n = length\ (get\ [1\,..\,n])\end{aligned}$$

Actually, one can often directly derive, from *get*, a simple syntactic definition for a function *sget* semantically equivalent to *shapify get*. That will be an important point in Section 5.3. But for the moment, we simply take the above definition.

Next, we assume that some function *sput* is given, with the following type:

$$sput :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat}\,,$$

and that *sput* is consistent for *shapify get*. Of course, $sput\ \equiv\ sput_{\text{naive}}\ get$ is always a valid choice, but for many *get*-functions there will be better alternatives!

We now define bff_{plug} as below. There are three differences from bff_{refac}: instead of calling out to $sput_{\text{naive}}\ get$, we call out to an (additional, besides *get*) function argument *sput* (that is again itself a function), we generate an error message in case that *sput* fails (previously this was done directly in $sput_{\text{naive}}$), and we drop the *fromJust* from the last (*return*-) line. The latter change introduces an extra Maybe-type constructor in the output list type, and is done to deal with list positions for which no data is known, neither from the original source nor from the updated view,

```
bff_plug :: Monad μ ⇒ (∀α.[α] → [α]) → (Nat → Nat → Maybe Nat)
                      → (∀α.[α] → [α] → μ [Maybe α])
bff_plug get sput s v' = do let n = length s
                            let t = [1..n]
                            let g = NatMap.fromDistinctAscList (zip t s)
                            let g' = foldr NatMap.delete g (get t)
                            n' ← case sput n (length v') of
                                   Nothing → fail "Could not handle shape change."
                                   Just n'  → return n'
                            let t = [1..n']
                            let h = NatMap.fromDistinctList (zip (get t) v')
                            let h' = NatMap.union h g'
                            return (map (flip NatMap.lookup h') t)
```

Note that now the second local binding for t, shadowing the first one, can really yield a different list because it is no longer given that n' is identical to n. Also at this point, it becomes relevant that we assume $\mathsf{NatMap}.\mathit{union}$ to be left-biased for natural numbers occurring as keys in both its input maps. That is important to guarantee precedence of h over g' for positions of the output list that are represented in the domain of both h (comprising all natural numbers that occur in $\mathit{get}\ [1\,..\,n']$) and g' (comprising all natural numbers that occur in $[1\,..\,n]$ but not in $\mathit{get}\ [1\,..\,n]$).

The proof of the following theorem is then very similar to that of Theorem 1, but, of course, in addition uses the assumption about the relationship between *sput* and *shapify get*.

Theorem 5. Let $\mathit{get} :: [\alpha] \rightarrow [\alpha]$ be semantically affine. Let *sput* be consistent for *shapify get*. Let τ be a type.

- For every $s :: [\tau]$, $\mathit{bff}_{\mathrm{plug}}\ \mathit{get}\ \mathit{sput}\ s\ (\mathit{get}\ s) :: \mathsf{Maybe}\ [\mathsf{Maybe}\ \tau] \equiv \mathsf{Just}\ (\mathit{map}\ \mathsf{Just}\ s)$.
- For every $s, v' :: [\tau]$ and $s' :: [\mathsf{Maybe}\ \tau]$, if

$$\mathit{bff}_{\mathrm{plug}}\ \mathit{get}\ \mathit{sput}\ s\ v' :: \mathsf{Maybe}\ [\mathsf{Maybe}\ \tau] \equiv \mathsf{Just}\ s',$$

  then $\mathit{get}\ s' \equiv \mathit{map}\ \mathsf{Just}\ v'$.

The following theorem can also be shown to hold.

Theorem 6. Let $\mathit{get} :: [\alpha] \rightarrow [\alpha]$ be semantically affine. Let *sput* be consistent for *shapify get*. For every type τ and $s, v' :: [\tau]$, if $\mathit{length}\ (\mathit{get}\ s) = \mathit{length}\ v'$, then $\mathit{bff}_{\mathrm{plug}}\ \mathit{get}\ \mathit{sput}\ s\ v' :: \mathsf{Maybe}\ [\mathsf{Maybe}\ \tau] \equiv \mathsf{Just}\ (\mathit{map}\ \mathsf{Just}\ s')$ for some $s' :: [\tau]$.

The proof is basically by observing that if we have $\mathit{length}\ (\mathit{get}\ s) = \mathit{length}\ v'$, then also $\mathit{shapify}\ \mathit{get}\ (\mathit{length}\ s) = \mathit{length}\ v'$, and thus, by consistency of *sput* for *shapify get*, inside the $\mathit{bff}_{\mathrm{plug}}$-definition n' will be successfully assigned the value n, and subsequently every index position from t will lead to a successful lookup in h', because h covers all such positions that also occur in $\mathit{get}\ t$ while g' covers exactly all the rest.

Neither Theorem 5 nor Theorem 6 says anything about when a $\mathsf{Nothing}$ can become manifest for the 'inner' Maybe in the result type of $\mathit{bff}_{\mathrm{plug}}\ \mathit{get}\ \mathit{sput}\ s\ v'$. This is so because such $\mathsf{Nothing}$-values can only appear on the updated source side, and only if the shape was changed. We already mentioned earlier that $\mathit{bff}_{\mathrm{plug}}$ uses the extra Maybe type constructor to deal with positions in the output list for which no data is known, neither from the original source nor from the updated view. Let us discuss this in a bit more detail now, also returning to the discussion of the 'choice of g' versus g' as promised directly after introducing *bff* in Section 3. (The reader less interested in the subtleties may want to skip over directly to the paragraph before Corollary 5.)

So what happens with $\mathit{bff}_{\mathrm{plug}}\ \mathit{get}\ \mathit{sput}\ s\ v'$ if v' does not have the same length as $\mathit{get}\ s$? Then n' will be different from n in $\mathit{bff}_{\mathrm{plug}}$, and when going through the index positions from $t \equiv [1\,..\,n']$ to create s', some positions might not be found in the domain of h'. Obviously (since the domain of h' is the union of the domains of h and g'), this will happen exactly for all natural numbers from 'set' $S' = [1\,..\,n']$ that occur neither in $V' = \mathit{get}\ [1\,..\,n']$ (the domain of h) nor in, with $S = [1\,..\,n]$ and $V = \mathit{get}\ [1\,..\,n]$, $S \setminus\setminus V$ (the domain of g', computed using the 'set difference' operator $\setminus\setminus$). We may picture the setup/connections here as follows (although we will never have reason to actually compute $put\ [1\,..\,n]\ (get\ [1\,..\,n'])$):

$$\begin{tikzcd}
{[1\,..\,n]} \arrow[rr, Rightarrow] \arrow[d, Rightarrow, bend left] & & get\ [1\,..\,n] \arrow[d, dotted] \\
{[1\,..\,n']} & & get\ [1\,..\,n'] \arrow[ll, Rightarrow]
\end{tikzcd}$$

So, following the above observations about the domains of h', h, and g', all positions of s' that correspond to a natural number from

$$X = S' \mathbin{\backslash\backslash} (V' \cup (S \mathbin{\backslash\backslash} V))$$

will be filled with Nothing. Justifiedly so? Well, first of all, it is ensured that this will not happen for positions that correspond, after performing *get*, to elements/positions of v'.[6] Moreover, for various elements of X, namely for all elements of

$$Y = S' \mathbin{\backslash\backslash} (V' \cup S)\,,$$

it is hardly conceivable what Just-value to provide for them. After all, Y will only be non-empty if $n' > n$, i.e., the update on the view (shape) triggered an update on the source that forced it to become longer, and any element of Y will be a position for which we have no meaningful way to pick an element from the original source (precisely because it will be a position beyond the length n of the original source) and for which we also cannot justify picking an element from the updated view (because it will be a position not occurring in V', hence not corresponding to an element of the view list).

But one doubt may remain: what about elements of $V \cap S' \mathbin{\backslash\backslash} V'$? This set is obtained as the difference $X \mathbin{\backslash\backslash} Y$ (taking into account that $S \supseteq V$). So the elements in question are exactly the positions for which – since they appear in X – we assign Nothing in s' even though they do *not* appear in Y. For them, it might seem tempting to lookup a value in g, which would correspond to accessing a position of the original source s (which after all has length n, so we are guaranteed to find some value). Indeed, that is exactly what we did previously (Voigtländer *et al.*, 2010, by not having the line **let** $g' = foldr$ NatMap.$delete\ g\ t'$ in *bff* and bff_{plug}, instead using $g' \equiv g$). But morally it is actually wrong: we would fill a position in the updated source for which we have no support from the updated view, with a value from the original view. This seems too arbitrary. After all, the *get*-function will have a ‘reason’ for omitting that position when going from a source of length n' to a corresponding view. If at all attempting to fill the position with an element from the situation before any update happens, we should attempt to explain it from the original source, not from the original view. If we are unable to do so, we are better off returning Nothing. We will again consider this issue, based on a concrete example – materializing the difference in this respect between what we did previously and what we do in this paper – toward the end of Section 5.1.

[6] This, which is guaranteed by V' being covered by h, is reflected in the second point stated in Theorem 5, which implies that all elements of $get\ s'$ are Just-values. Of course, that point does not prevent other positions of s' from containing Nothing-values.

Instead of producing Just- and Nothing-values, it is usually more convenient to simply use a default value for positions in the output list for which no data is known, neither from the original source nor from the updated view. Hence, we define a function *dbff* as follows:

$$\begin{aligned}
&\mathit{dbff} :: \mathsf{Monad}\ \mu \Rightarrow (\forall\alpha.[\alpha] \to [\alpha]) \to (\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat})\\
&\qquad\qquad\qquad\qquad \to (\forall\alpha.\alpha \to [\alpha] \to [\alpha] \to \mu\ [\alpha])\\
&\mathit{dbff}\ \mathit{get}\ \mathit{sput}\ d\ s\ v' = \mathbf{do}\ s' \leftarrow \mathit{bff}_{\text{plug}}\ \mathit{get}\ \mathit{sput}\ s\ v'\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{return}\ (\mathit{map}\ (\lambda\mathbf{case}\ \{\mathsf{Nothing} \to d; \mathsf{Just}\ y \to y\})\ s')
\end{aligned}$$

The following two statements are then relatively direct consequences of Theorems 5 and 6.

Corollary 5. Let $\mathit{get} :: [\alpha] \to [\alpha]$ be semantically affine. Let *sput* be consistent for *shapify get*. For every type τ and $d :: \tau$,

$$\mathit{dbff}\ \mathit{get}\ \mathit{sput}\ d :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is consistent for $\mathit{get} :: [\tau] \to [\tau]$.

Corollary 6. Let $\mathit{get} :: [\alpha] \to [\alpha]$ be semantically affine. Let *sput* be consistent for *shapify get*. For every type τ and $d :: \tau$,

$$\mathit{dbff}\ \mathit{get}\ \mathit{sput}\ d :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is fixed-shape-friendly for $\mathit{get} :: [\tau] \to [\tau]$. (Moreover, the default value d is not actually used in $\mathit{dbff}\ \mathit{get}\ \mathit{sput}\ d\ s\ v'$ if $\mathit{length}\ (\mathit{get}\ s) = \mathit{length}\ v'$.)

It is important to note that no general negative statement like Theorem 4 or Corollary 4 holds for *dbff* (or for $\mathit{bff}_{\text{plug}}$). It all depends on the argument *sput*! If we find a good *sput* that is consistent for *shapify get*, then *dbff get sput* will also be good for *get*. This is where we can now plug in arbitrary ‘shape bidirectionalizers’.

5 Some concrete ‘plug-ins’

5.1 Manual shape-bidirectionalization

In principle, a reasonable stance to take is that the programmer, who has programmed *get*, should also provide *sput*. After all, the programmer can be often expected to have a very good idea of how shape-changing updates should be dealt with.

Running Example 1 (continued, with manual provision of *sput*). Recall that get_1 sieves a list to keep only every second element. On the shape level this means to halve the length of the list, so an intuitive backward transformation seems to be to double the length of any provided updated view list:

$$\begin{aligned}
&\mathit{sput} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat}\\
&\mathit{sput}\ l_s\ l_{v'} = \mathsf{Just}\ (2 * l_{v'})
\end{aligned}$$

But this violates the condition that *sput* should be consistent for $\mathit{shapify}\ \mathit{get}_1$. Indeed, $\mathit{sput}\ l_s\ (\mathit{shapify}\ \mathit{get}_1\ l_s) \equiv \mathsf{Just}\ l_s$ does not hold for any odd natural number l_s. After all, $\mathit{shapify}\ \mathit{get}_1$ is not *exact* halving, but actually halving with truncation. A natural remedy is to refine *sput* as follows:

$$\begin{aligned}&\mathit{sput} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat}\\&\mathit{sput}\ l_s\ l_{v'} = \mathsf{Just}\ (2 * l_{v'} + l_s\ \text{`}\mathit{mod}\text{`}\ 2)\end{aligned}$$

Then *sput* is consistent for *shapify* get_1, and can thus be used as follows, with the guarantees from Corollaries 5 and 6:

s	$get_1\ s$	v'	$dbff\ get_1\ sput$ ' ' $s\ v'$
"abcd"	"bd"	"x"	Just "ax"
"abcd"	"bd"	"xy"	Just "axcy"
"abcd"	"bd"	"xyz"	Just "axcy z"
"abcd"	"bd"	"xyzv"	Just "axcy z v"
"abcde"	"bd"	"x"	Just "axc"
"abcde"	"bd"	"xy"	Just "axcye"
"abcde"	"bd"	"xyz"	Just "axcyez "
"abcde"	"bd"	"xyzv"	Just "axcyez v "

Note that when $length\ (get_1\ s) \neq length\ v'$, $dbff\ get_1\ sput$ ' ' $s\ v'$ extends – making use of the default value – or shrinks the source list by a number of elements that is a multiple of two. All updates can now be successfully handled, much in contrast to earlier, when we used $bff\ get_1$ instead of $dbff\ get_1\ sput$ ' '. The moderate price to pay is that the programmer has to come up with *sput*.

Running Example 2 (continued, with manual provision of *sput*). Recall that get_2 keeps every element of a list except for the last one, and maps the empty list to itself. On the shape level this means that *shapify* get_2 maps 0 to 0, and every positive number to its predecessor. One possible choice for a consistent *sput* is thus:

$$\begin{aligned}&\mathit{sput} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat}\\&\mathit{sput}\ 0\ 0 \qquad\qquad\qquad\quad = \mathsf{Just}\ 0\\&\mathit{sput}\ l_s\ l_{v'} \mid l_s > 0 \lor l_{v'} > 0 = \mathsf{Just}\ (l_{v'} + 1)\end{aligned}$$

With it, we get:

s	$get_2\ s$	v'	$dbff\ get_2\ sput$ ' ' $s\ v'$
""	""	""	Just ""
""	""	"x"	Just "x "
""	""	"xy"	Just "xy "
"a"	""	""	Just "a"
"a"	""	"x"	Just "x "
"ab"	"a"	""	Just " "
"ab"	"a"	"x"	Just "xb"
"ab"	"a"	"xy"	Just "xy "
"abc"	"ab"	""	Just " "
"abc"	"ab"	"x"	Just "x "
"abc"	"ab"	"xy"	Just "xyc"
"abc"	"ab"	"xyz"	Just "xyz "

This is better than what we saw for this example in Section 3, but still not perfect, since at some places the default value gets used where intuitively a specific value from the source list (namely the last element of s) would be appropriate instead. We will return to this aspect in Section 6.

A related issue is that in the above table we see a manifest effect of the 'choice of g' versus g' issue that was already mentioned after introducing *bff* in Section 3 and that was discussed after Theorem 6 in Section 4.4. Had we concerning that choice proceeded as previously (Voigtländer *et al.*, 2010) , we would have got *dbff* get_2 *sput* `' '` `"ab"` `""` $\equiv$ Just `"a"`, *dbff* get_2 *sput* `' '` `"abc"` `""` $\equiv$ Just `"a"`, and *dbff* get_2 *sput* `' '` `"abc"` `"x"` $\equiv$ Just `"xb"`, but those `'a'` and `'b'` have no business appearing in the updated sources, since they are *not* (even) the last elements of the respective original sources.

It is worth pointing out that when writing *sput*, the programmer may well profit from a structured/combinator-based approach such as generic point-free lenses (Pacheco & Cunha, 2010). Where writing *get* using the point-free combinators is hard (due to having to worry about the projection of elements and the inventing of them in the backward direction), writing *sget* (and thus, due to the lens framework, immediately also *sput*) using the combinators could be much simpler.

5.2 *Shape-bidirectionalization by search*

Another viable option is to discover appropriate new source shapes by search. Specifically, one can change the last line of the definition of $sput_{\text{naive}}$ in Section 4.3 to

$$\textbf{else}\ return\ (head\ [l_{s'} \mid l_{s'} \leftarrow [0\,..], length\ (get\ [1\,..\,l_{s'}]) == l_{v'}])$$

(which may of course lead to non-termination that is unavoidable in general) and use that version to obtain (partial function) *sput* from *get*. Actually, thanks to semantic affineness of *get*, it is sufficient to start the search for $l_{s'}$ (after l_s itself has been ruled out[7]) at $l_{v'}$, i.e., one can replace $[0\,..]$ by $[l_{v'}\,..]$ above, or equivalently have altogether (and specialized to the Maybe monad, but nevertheless still returning a non-total *sput* in general):

$$\begin{aligned}
&sput_{\text{search}} :: (\forall\alpha.[\alpha] \to [\alpha]) \to (\mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat})\\
&sput_{\text{search}}\ get\ l_s\ l_{v'} = \mathsf{Just}\ (head\ [l_{s'} \mid l_{s'} \leftarrow l_s : ([l_{v'}\,..] \mathbin{\backslash\backslash} [l_s]),\\
&\qquad\qquad\qquad\qquad\qquad\qquad length\ (get\ [1\,..\,l_{s'}]) == l_{v'}])
\end{aligned}$$

One could even give the user some control over the 'perfect updateability' achieved using pure search, enabling them to provide guidance via heuristics expressed as reorderings of the candidate list $[l_{v'}\,..]$. Here we instead only consider the most basic search approach on our two running examples.

Running Example 1 (with search instead of manual provision of *sput*). Applying $sput_{\text{search}}$ to get_1 yields a function semantically equivalent to the following one:

$$\begin{aligned}
&sput :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat}\\
&sput\ l_s\ l_{v'} = \textbf{if}\ l_s\ \text{`}div\text{`}\ 2 == l_{v'}\ \textbf{then}\ \mathsf{Just}\ l_s\ \textbf{else}\ \mathsf{Just}\ (2 * l_{v'})
\end{aligned}$$

It is (by construction) consistent for *shapify* get_1, and behaves like the second *sput* given for this example in Section 5.1, except when l_s is odd and $l_{v'}$ is not its (truncated) half.

[7] We always need to check l_s first, to guarantee the first of our consistency conditions (cf. Definition 1).

Concretely, this implies that (only) the following lines change compared with the corresponding input/output table from Section 5.1.

s	$get_1\ s$	v'	$dbff\ get_1\ (sput_{\text{search}}\ get_1)$ ' ' $s\ v'$
"abcde"	"bd"	"x"	Just "ax"
"abcde"	"bd"	"xyz"	Just "axcyez"
"abcde"	"bd"	"xyzv"	Just "axcyez v"

Running Example 2 (with search instead of manual provision of *sput*). Applying $sput_{\text{search}}$ to get_2 yields a function semantically equivalent to the following one:

$$\begin{aligned}
&sput :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat}\\
&sput\ l_s\ 0\ \mid l_s \leqslant 1 = \mathsf{Just}\ l_s\\
&sput\ l_s\ 0\ \mid l_s > 1\ = \mathsf{Just}\ 0\\
&sput\ \_\ l_{v'} \mid l_{v'} > 0 = \mathsf{Just}\ (l_{v'} + 1)
\end{aligned}$$

It differs from the one given for this example in Section 5.1 exactly when $l_s > 1$ and $l_{v'} = 0$, so that we get changed behavior as follows:

s	$get_2\ s$	v'	$dbff\ get_2\ (sput_{\text{search}}\ get_2)$ ' ' $s\ v'$
"ab"	"a"	""	Just ""
"abc"	"ab"	""	Just ""

5.3 Combining syntactic and semantic bidirectionalization

The search approach from the previous section is attractive in its simplicity, and works reasonably well for our two running examples, but it is certainly not a panacea. Besides possible concerns about its efficiency in finding solutions, there is the problem that even if there is no solution (appropriate source shape for a given update configuration) at all, that fact will not be discovered (by leading to return value Nothing) in finite time. Also, formally legitimate updates found by search may be less meaningful to the user than ones obtained from more ‘intelligent’ or ‘intuition-guided’ shape bidirectionalizers. One possibility of the latter kind is to employ an existing bidirectionalization approach (Matsuda *et al.*, 2007) based on constant complements (Bancilhon & Spyratos, 1981).

The basic idea of Matsuda *et al.*’s (2007) technique is that for a function

$$get :: \tau_1 \to \tau_2$$

one finds a function

$$compl :: \tau_1 \to \tau_3$$

such that the pairing of the two,

$$\begin{aligned}
&paired :: \tau_1 \to (\tau_2, \tau_3)\\
&paired\ s = (get\ s, compl\ s)
\end{aligned}$$

is an injective function. Given a ‘partial inverse’ $inv :: (\tau_2, \tau_3) \to \mathsf{Maybe}\ \tau_1$ of *paired*, satisfying the requirements that

- for every $s :: \tau_1$,
$$inv\ (paired\ s) \equiv \mathsf{Just}\ s\,,$$
and
- for every $s' :: \tau_1$, $v' :: \tau_2$, and $c :: \tau_3$, if $inv\ (v', c) \equiv \mathsf{Just}\ s'$, then
$$paired\ s' \equiv (v', c)\,,$$

one obtains that

$$\begin{aligned}&put :: \tau_1 \to \tau_2 \to \mathsf{Maybe}\ \tau_1\\&put\ s\ v' = inv\ (v', compl\ s)\end{aligned}$$

is consistent for *get*.

The approach of Matsuda *et al.* (2007) is to perform all the above by syntactic program transformations. For a certain class of programs, they give an algorithm that automatically derives *compl* from *get* in such a way that *paired* is indeed injective. Then instead of the definition for *paired* above they produce one using a tupling transformation (Pettorossi, 1977) that avoids the two independent traversals of *s* with *get* and *compl*. They syntactically invert *paired* to obtain *inv* (although they make *inv* implicitly a partial function, not explicitly in the type as above), and subsequently fuse the computations of *inv* and *compl* in the definition of *put*, again using a syntactic transformation (Wadler, 1990).

We illustrate the syntactic approach based on our two running examples.

Running Example 1 (syntactically). The function alluded to in the first running example, sieving a list to keep only every second element, could have been defined as follows:

$$\begin{aligned}&get_1 :: [\alpha] \to [\alpha]\\&get_1\ [\,] \qquad\quad = [\,]\\&get_1\ [x] \qquad\ \ = [\,]\\&get_1\ (x : y : zs) = y : (get_1\ zs)\end{aligned}$$

That function definition fulfills the syntactic prerequisites imposed by Matsuda *et al.* (2007). They are (necessary[8] and sufficient): that functions must be first-order, must be affine (i.e., no variable occurs more than once on a single right-hand side), and that there must be no function call with anything else than variables in its arguments.

Given the above function definition, the following complement function is automatically derived[9]:

$$\begin{aligned}&\mathbf{data}\ \mathsf{Compl}\ \alpha = \mathsf{C}_1 \mid \mathsf{C}_2\ \alpha \mid \mathsf{C}_3\ \alpha\ (\mathsf{Compl}\ \alpha)\\&compl :: [\alpha] \to \mathsf{Compl}\ \alpha\\&compl\ [\,] \qquad\quad = \mathsf{C}_1\\&compl\ [x] \qquad\ \ = \mathsf{C}_2\ x\\&compl\ (x : y : zs) = \mathsf{C}_3\ x\ (compl\ zs)\end{aligned}$$

[8] At least for the original method of Matsuda *et al.* (2007). Later work (Matsuda *et al.*, 2009, in Japanese) relaxes the restrictions somewhat.

[9] Matsuda *et al.* (2007) work in an untyped language, so they have no need to explicitly introduce the data type Compl, but as we formulate our ideas in Haskell, we will be careful to introduce appropriate types as we go along.

Tupling of get_1 and *compl* gives the following definition for the paired function:

$$\begin{aligned}
&\mathit{paired} :: [\alpha] \to ([\alpha], \mathsf{Compl}\ \alpha)\\
&\mathit{paired}\ [\,] &&= ([\,]\quad, \mathsf{C}_1)\\
&\mathit{paired}\ [x] &&= ([\,]\quad, \mathsf{C}_2\ x)\\
&\mathit{paired}\ (x : y : zs) &&= (y : v, \mathsf{C}_3\ x\ c)\\
& &&\quad\ \ \mathbf{where}\ (v, c) = \mathit{paired}\ zs
\end{aligned}$$

Syntactic inversion, (here) basically just exchanging left- and right-hand sides, plus introduction of monadic error propagation, gives:

$$\begin{aligned}
&\mathit{inv} :: \mathsf{Monad}\ \mu \Rightarrow ([\alpha], \mathsf{Compl}\ \alpha) \to \mu\ [\alpha]\\
&\mathit{inv}\ ([\,]\quad, \mathsf{C}_1) &&= \mathit{return}\ [\,]\\
&\mathit{inv}\ ([\,]\quad, \mathsf{C}_2\ x) &&= \mathit{return}\ [x]\\
&\mathit{inv}\ (y : v, \mathsf{C}_3\ x\ c) &&= \mathbf{do}\ zs \leftarrow \mathit{inv}\ (v, c)\\
& &&\qquad\ \ \mathit{return}\ (x : y : zs)\\
&\mathit{inv}\ \_ &&= \mathit{fail}\ \texttt{"Update violates complement."}
\end{aligned}$$

Finally,

$$\begin{aligned}
&\mathit{put} :: \mathsf{Monad}\ \mu \Rightarrow [\alpha] \to [\alpha] \to \mu\ [\alpha]\\
&\mathit{put}\ s\ v' = \mathit{inv}\ (v', \mathit{compl}\ s)
\end{aligned}$$

can be fused to:

$$\begin{aligned}
&\mathit{put} :: \mathsf{Monad}\ \mu \Rightarrow [\alpha] \to [\alpha] \to \mu\ [\alpha]\\
&\mathit{put}\ [\,]\qquad\quad [\,] &&= \mathit{return}\ [\,]\\
&\mathit{put}\ [x]\qquad\ \ [\,] &&= \mathit{return}\ [x]\\
&\mathit{put}\ (x : y : zs)\ (y' : v') &&= \mathbf{do}\ zs' \leftarrow \mathit{put}\ zs\ v'\\
& &&\qquad\ \ \mathit{return}\ (x : y' : zs')\\
&\mathit{put}\ \_\qquad\quad\ \_ &&= \mathit{fail}\ \texttt{"Update violates complement."}
\end{aligned}$$

Note that, just as was the case for the original semantic bidirectionalization technique here, $\mathit{put}\ s\ v'$ fails if and only if $\mathit{length}\ (get_1\ s) \neq \mathit{length}\ v'$. Indeed, $\mathit{bff}\ get_1$ (from Example 1 in Section 3) and the above *put* are semantically equivalent (at type $[\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$, for τ that is an instance of Eq).

Running Example 2 (syntactically). The function alluded to in the second running example, keeping every element of a list except for the last one, could have been defined as follows[10]:

$$\begin{aligned}
&get_2 :: [\alpha] \to [\alpha]\\
&get_2\ [\,] &&= [\,]\\
&get_2\ [x] &&= [\,]\\
&get_2\ (x : y : zs) &&= x : (get'\ y\ zs)\\
&get' :: \alpha \to [\alpha] \to [\alpha]\\
&get'\ x\ [\,] &&= [\,]\\
&get'\ x\ (y : zs) &&= x : (get'\ y\ zs)
\end{aligned}$$

[10] A helper function get' is used to prevent a function call with an argument that is not a variable.

Table 1. *Comparing bidirectionalization methods for the get-function from Example 2*

			Semantic	Syntactic	Combined	
s	$get_2\ s$	v'	$bff\ get_2\ s\ v'$	$put\ s\ v'$	$dput_1$ `' '` $s\ v'$	$dput_2$ `' '` $s\ v'$
`""`	`""`	`""`	`Just ""`	`Just ""`	`Just ""`	`Just ""`
`""`	`""`	`"x"`	`Nothing`	`Nothing`	`Just "x "`	`Nothing`
`""`	`""`	`"xy"`	`Nothing`	`Nothing`	`Just "xy "`	`Nothing`
`"a"`	`""`	`""`	`Just "a"`	`Just "a"`	`Just "a"`	`Just "a"`
`"a"`	`""`	`"x"`	`Nothing`	`Nothing`	`Nothing`	`Just "x "`
`"ab"`	`"a"`	`""`	`Nothing`	`Nothing`	`Just ""`	`Just " "`
`"ab"`	`"a"`	`"x"`	`Just "xb"`	`Just "xb"`	`Just "xb"`	`Just "xb"`
`"ab"`	`"a"`	`"xy"`	`Nothing`	`Just "xyb"`	`Just "xy "`	`Just "xy "`
`"abc"`	`"ab"`	`""`	`Nothing`	`Nothing`	`Just ""`	`Just " "`
`"abc"`	`"ab"`	`"x"`	`Nothing`	`Just "xc"`	`Just "x "`	`Just "x "`
`"abc"`	`"ab"`	`"xy"`	`Just "xyc"`	`Just "xyc"`	`Just "xyc"`	`Just "xyc"`
`"abc"`	`"ab"`	`"xyz"`	`Nothing`	`Just "xyzc"`	`Just "xyz "`	`Just "xyz "`

For that function definition, the syntactic approach produces the following complement function:

$$\begin{array}{l}
\textbf{data}\ \mathsf{Compl}\ \alpha = \mathsf{C}_1 \mid \mathsf{C}_2\ \alpha \mid \mathsf{C}_3\ \alpha \\
compl :: [\alpha] \to \mathsf{Compl}\ \alpha \\
compl\ [\,] \qquad\quad = \mathsf{C}_1 \\
compl\ [x] \qquad\; = \mathsf{C}_2\ x \\
compl\ (x : y : zs) = compl'\ y\ zs \\
compl' :: \alpha \to [\alpha] \to \mathsf{Compl}\ \alpha \\
compl'\ x\ [\,] \qquad = \mathsf{C}_3\ x \\
compl'\ x\ (y : zs) = compl'\ y\ zs
\end{array}$$

Tupling, inversion, and fusion (not spelled out here in detail) ultimately give functions *put* and (helper) *put*′ such that $put\ s\ v'$ succeeds if and only if $length\ (get_2\ s)$ and $length\ v'$ are equal or both greater than zero. In contrast, we have seen in Section 3 that the original semantic bidirectionalization technique (again) allows no view updates that change the shape here, i.e., $bff\ get_2\ s\ v'$ is only successful if $length\ (get_2\ s) = length\ v'$. A few representative calls and their results are given in Table 1.

From the examples considered, we see that the syntactic bidirectionalization technique of Matsuda *et al.* (2007) and the original (non-plugged) semantic bidirectionalization technique of Voigtländer (2009) can agree or disagree in terms of updateability. Actually, it seems that for programs that can be handled by both, the syntactic technique on its own is never worse than the original semantic technique on its own. Interestingly, the method of choice for improvement over both, proposed by Voigtländer *et al.* (2010) and recollected in this paper, is to defer the syntactic technique to the role of a plug-in (basically as a black box), with the technique of Voigtländer (2009) in the master role.

Specifically, for functions *get* that are polymorphic and at the same time satisfy the syntactic restrictions imposed by Matsuda *et al.*'s (2007) technique, we can use that technique for deriving *sput* from an explicit syntactic definition for a function *sget* (independent of, but derived from, *get*) that is semantically equivalent to *shapify get*.

Running Example 1 (with syntactic technique as plug-in). We have seen above in the current section and in Section 3 that for the function get_1 in question, both syntactic and (the original version of) semantic bidirectionalization on their own lead to quite limited updateability: both $put\ s\ v'$ and $bff\ get_1\ s\ v'$ only succeed if $length\ (get_1\ s) = length\ v'$.

On the other hand, by combining the two techniques, we can proceed as follows. The *sget* 'corresponding to' get_1, as obtained via a straightforward syntactic transformation from the definition of get_1 given earlier in this section, looks as follows:

```
sget :: Nat → Nat
sget 0          = 0
sget 1          = 0
sget n | n ⩾ 2 = (sget (n − 2)) + 1
```

For it, the syntactic bidirectionalization method of Matsuda *et al.* (2007) produces the following complement function:

```
data SCompl = SC₁ | SC₂
scompl :: Nat → SCompl
scompl 0          = SC₁
scompl 1          = SC₂
scompl n | n ⩾ 2 = scompl (n − 2)
```

Note that the move from $[\alpha]$ to Nat in $get_1 \mapsto sget$ has made the complement function much simpler: no collection of any variables (as was necessary in the definition of *compl*, to make up for the dropping of variables in the definition of get_1), and no constructor around the recursive call. (All this, thanks to explicit optimization effort embedded in Matsuda *et al.*'s (2007) transformation to 'make the complement smaller'.) The advantage is that a simpler/smaller complement function means better updateability of the ultimately obtained *put*-function. Here tupling, inversion, and fusion give:

```
sput :: Nat → Nat → Maybe Nat
sput 0 0                = return 0
sput 1 0                = return 1
sput l_s 0   | l_s ⩾ 2  = sput (l_s − 2) 0
sput l_s l_v′ | l_v′ ⩾ 1 = do l_s′ ← sput l_s (l_v′ − 1)
                             return (l_s′ + 2)
```

which is equivalent to the (desirable) *sput*-function provided for get_1 'by hand' in Section 5.1! So by using $bff_{\text{plug}}\ get_1\ sput$, or $dbff\ get_1\ sput$ ' ', with this *sput* we enjoy good and intuitive updateability without requiring manual intervention.

Running Example 2 (with syntactic technique as plug-in). We have seen further above in the current section and in Section 3 that for the function get_2 in question the updateability achieved by syntactic bidirectionalization is that $put\ s\ v'$ succeeds whenever $length\ (get_2\ s)$ and $length\ v'$ are equal or both greater than zero, while the original semantic technique is only successful if $length\ (get_2\ s) = length\ v'$. Let us analyze how the combination of the two techniques fares.

The move from $[\alpha]$ to Nat yields:

$$
\begin{array}{l}
sget :: \mathsf{Nat} \to \mathsf{Nat} \\
sget\ 0 \qquad\quad = 0 \\
sget\ 1 \qquad\quad = 0 \\
sget\ n \mid n \geqslant 2 = (sget'\ (n-2)) + 1 \\
sget' :: \mathsf{Nat} \to \mathsf{Nat} \\
sget'\ 0 \qquad\quad = 0 \\
sget'\ n \mid n \geqslant 1 = (sget'\ (n-1)) + 1
\end{array}
$$

Note that regarding the helper function get' (from earlier in this section) one argument becomes superfluous. Indeed, when moving from $[\alpha]$ to Nat, there is no role to play anymore for content elements of type α.

The automatic view complement generation of Matsuda *et al.* (2007) yields either of the two functions $scompl_1/scompl_2$ for $sget$ (with **data** $\mathsf{SCompl} = \mathsf{SC}_1 \mid \mathsf{SC}_2 \mid \mathsf{SC}_3$) which differ only in their last defining equation:

$$
\begin{array}{l}
scompl_1 :: \mathsf{Nat} \to \mathsf{SCompl} \\
scompl_1\ 0 \qquad\quad = \mathsf{SC}_1 \\
scompl_1\ 1 \qquad\quad = \mathsf{SC}_2 \\
scompl_1\ n \mid n \geqslant 2 = \mathsf{SC}_1
\end{array}
$$

and

$$
\begin{array}{l}
scompl_2 :: \mathsf{Nat} \to \mathsf{SCompl} \\
scompl_2\ 0 \qquad\quad = \mathsf{SC}_1 \\
scompl_2\ 1 \qquad\quad = \mathsf{SC}_2 \\
scompl_2\ n \mid n \geqslant 2 = \mathsf{SC}_2
\end{array}
$$

while for $sget'$, one obtains the following complement function:

$$
\begin{array}{l}
scompl' :: \mathsf{Nat} \to \mathsf{SCompl} \\
scompl'\ 0 \qquad\quad = \mathsf{SC}_3 \\
scompl'\ n \mid n \geqslant 1 = \mathsf{SC}_3
\end{array}
$$

Tupling, inversion, and fusion ultimately give two choices $sput_1$ and $sput_2$, for $scompl_1$ and $scompl_2$. Let us compare the results of combining syntactic and semantic bidirectionalization, i.e., the now two possible functions $dbff\ get_2\ sput_1$ and $dbff\ get_2\ sput_2$, to the results of either only (the original version of) semantic or only syntactic bidirectionalization, i.e., to $bff\ get_2$ à la Section 3 and to put from the continuation of Example 2 further above in the current section. Table 1 shows a few representative calls and their results, where $dput_1 \equiv dbff\ get_2\ sput_1$ and $dput_2 \equiv dbff\ get_2\ sput_2$. By our coarse measure, comparing the sizes of the applicability domains of *put*-functions, the combined technique is better than either of the two original techniques. However, some skepticism is appropriate regarding results like those for $s =$ `"ab"` and $v' =$ `"xy"` here: all *put*-functions except the one obtained by the purely semantic technique map that (s, v') pair to some s', but the *put*-function obtained using the purely syntactic technique certainly makes the best choice concerning *what* that s' should be. We discuss this aspect further in the next section.[11]

6 Explicit bias

Through the numbering scheme of our 'template sources' via $[1\,..\,n]$ for a concrete source of length n, there is a certain bias that manifests itself when an update changes the length of the view. For example, while it is nice that in the continuation of Example 2 in Section 5.1 (and also in Section 5.2; and similarly for dput_1 in Section 5.3/Table 1) we have

$$\mathit{dbff}\ \mathit{get}_2\ \mathit{sput}\ \text{' '}\ \texttt{""}\ \texttt{"x"} \equiv \mathsf{Just}\ \texttt{"x "}$$

and

$$\mathit{dbff}\ \mathit{get}_2\ \mathit{sput}\ \text{' '}\ \texttt{""}\ \texttt{"xy"} \equiv \mathsf{Just}\ \texttt{"xy "}$$

(in contrast to the completely semantically obtained $\mathit{bff}\ \mathit{get}_2$ and the completely syntactically obtained *put*, which both give Nothing in both cases; cf. Table 1), it is disappointing that

$$\mathit{dbff}\ \mathit{get}_2\ \mathit{sput}\ \text{' '}\ \texttt{"ab"}\ \texttt{"xy"} \equiv \mathsf{Just}\ \texttt{"xy "}$$

(instead of Just "xyb"). The reason for this is simple: The use of $[1\,..\,n]$ and $[1\,..\,n']$ in the definition of $\mathit{bff}_{\text{plug}}$ in Section 4.4 means that when the updated source becomes shorter than the original source, then it is the elements toward the *rear* of the original source that become discarded; while if the updated source becomes longer, then again positions toward the *rear* of the new source will be considered to be 'additional' and thus will be filled with the default value. So there is an implicit assumption that shape-changing updates will always happen in such a way that the corresponding insertions or deletions affect the end of the source list, rather than its front or other elements.

There is an easy remedy for the observed phenomenon. If we simply replace the lines

$$\textbf{let}\ t = [1\,..\,n]$$

and

$$\textbf{let}\ t = [1\,..\,n']$$

in the definition of $\mathit{bff}_{\text{plug}}$ by

$$\textbf{let}\ t = \mathit{reverse}\ [1\,..\,n]$$

and

$$\textbf{let}\ t = \mathit{reverse}\ [1\,..\,n']$$

respectively, then Theorems 5 and 6 – and thus Corollaries 5 and 6 – continue to hold, but instead of a rear update (insertion/deletion) bias, there is now a front update bias.

[11] Orthogonally, there would be more to say, and is said by Voigtländer *et al.* (2010), about updateability solely in terms of applicability domains. In particular, Section 7 of that paper contains examples showing how by involving additional syntactic transformations (Giesl, 2000; Giesl *et al.*, 2007) one can extend applicability further. Also note that, by virtue of the plug-in approach, we will directly profit from further (independent) improvements of the syntactic bidirectionalization technique itself.

For example, the table in the continuation of Example 2 in Section 5.1 (the interesting subset thereof; all other entries remain unchanged) now becomes:

s	$get_2\ s$	v'	$dbff\ get_2\ sput$ ' ' $s\ v'$
`""`	`""`	`"x"`	`Just "x  "`
`""`	`""`	`"xy"`	`Just "xy  "`
`"a"`	`""`	`"x"`	`Just "xa"`
`"ab"`	`"a"`	`""`	`Just "b"`
`"ab"`	`"a"`	`"xy"`	`Just "xyb"`
`"abc"`	`"ab"`	`""`	`Just "c"`
`"abc"`	`"ab"`	`"x"`	`Just "xc"`
`"abc"`	`"ab"`	`"xyz"`	`Just "xyzc"`

The entries that have changed are shaded above. Only where no (last element) value from the original source list is available do we still use a default value in the updated source. One could argue that in this specific case all the changes are for the better, but in general it is desirable to be able to influence what bias is used.

Making the bias explicit, and thus putting it under the potential control of the user, is easily possible by defining a further variation of bff_{plug}:

```
type Bias = Nat → [Nat]
bff_bias :: Monad μ ⇒ (∀α.[α] → [α]) → (Nat → Nat → Maybe Nat)
                     → Bias → (∀α.[α] → [α] → μ [Maybe α])
bff_bias get sput bias s v′ = do let n = length s
                                 let t = bias n
                                 let g = NatMap.fromDistinctList (zip t s)
                                 let g′ = foldr NatMap.delete g (get t)
                                 n′ ← case sput n (length v′) of
                                        Nothing → fail "..."
                                        Just n′  → return n′
                                 let t = bias n′
                                 let h = NatMap.fromDistinctList (zip (get t) v′)
                                 let h′ = NatMap.union h g′
                                 return (map (flip NatMap.lookup h′) t)
```

as well as:

```
bdbff :: Monad μ ⇒ (∀α.[α] → [α]) → (Nat → Nat → Maybe Nat)
                 → Bias → (∀α.α → [α] → [α] → μ [α])
bdbff get sput bias d s v′ = do s′ ← bff_bias get sput bias s v′
                                return (map (λcase {Nothing → d; Just y → y}) s′)
```

The only formal requirement that we impose on a proper $bias$ :: Bias, ensuring that analogues of Theorems 5 and 6 and of Corollaries 5 and 6 continue to hold, is that for every n :: Nat, $bias\ n$ should return a list that is a permutation of $[1\,..\,n]$. Then we in particular obtain the following two corollaries.

Corollary 7. Let $get :: [\alpha] \to [\alpha]$ be semantically affine. Let *sput* be consistent for *shapify get*. Let $bias :: \mathsf{Bias}$ be proper (in the way just described). For every type τ and $d :: \tau$,

$$bdbff\ get\ sput\ bias\ d :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is consistent for $get :: [\tau] \to [\tau]$.

Corollary 8. Let $get :: [\alpha] \to [\alpha]$ be semantically affine. Let *sput* be consistent for *shapify get*. Let $bias :: \mathsf{Bias}$ be proper. For every type τ and $d :: \tau$,

$$bdbff\ get\ sput\ bias\ d :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is fixed-shape-friendly for $get :: [\tau] \to [\tau]$. (Moreover, the default value d is not actually used in $bdbff\ get\ sput\ bias\ d\ s\ v'$ if $length\ (get\ s) = length\ v'$.)

Some good examples for *bias* are:

$rear :: \mathsf{Bias}$
$rear\ n = [1\,..\,n]$
$front :: \mathsf{Bias}$
$front\ n = reverse\ [1\,..\,n]$
$middle :: \mathsf{Bias}$
$middle\ n = [1,3\,..\,n] +\!\!+ (reverse\ [2,4\,..\,n])$
$borders :: \mathsf{Bias}$
$borders\ n = (reverse\ [1,3\,..\,n]) +\!\!+ [2,4\,..\,n]$

Some examples for the *get*-function from Example 1 (with *sput* as given in Section 5.1 and automatically obtained in Section 5.3), illustrating the effects of different bias strategies, are given in Table 2, where $bdput\ \equiv\ bdbff\ get_1\ sput$.

The beneficial effects, still for the case of the *get*-function from Example 1, might become even more apparent when also looking at cases where the data values in the source and view lists are not disjoint, as in Table 3. The simple hints about which bias to apply when reflecting specific updated views back to the source level are quite effective.

7 Going generic

Up to here, we have only considered the case of lists by applying bidirectionalization to functions $get :: [\alpha] \to [\alpha]$. On the other hand, both the syntactic bidirectionalization technique of Matsuda *et al.* (2007) and the original (non-plugged) semantic bidirectionalization technique of Voigtländer (2009) are already able to work on other data structures than lists. In this section we catch up to such a more generic setting by showing how bff_{plug} can be made suitably polymorphic over the type constructors on the input and output sides of *get*-functions. We do this in a similar way as the reformulation by Foster *et al.* (2012, Section 5.4) of the generic version of the original semantic bidirectionalization technique, namely by using explicit separations of data structures into their shape and content aspects, in the spirit of the *shape calculus* (Jay, 1995) and *container representations* (Abbott *et al.*, 2003). Specifically, we start by introducing an abstraction for types that can hold shapes of members of other types, in the sense in which Nat has served as the type of shapes for

Table 2. *Comparing bias strategies for our combined technique on the get-function from Example 1*

s	*get*$_1$ *s*	*v*′	*bdput rear* ’ ’ *s v*′	*bdput front* ’ ’ *s v*′	*bdput middle* ’ ’ *s v*′	*bdput borders* ’ ’ *s v*′
`"abcd"`	`"bd"`	`"x"`	Just `"ax"`	Just `"cx"`	Just `"ax"`	Just `" x"`
`"abcd"`	`"bd"`	`"xyz"`	Just `"axcy z"`	Just `" xaycz"`	Just `"ax ycz"`	Just `" x y z"`
`"abcd"`	`"bd"`	`"xyzv"`	Just `"axcy z v"`	Just `" x yazcv"`	Just `"ax y zcv"`	Just `" xaycz v"`
`"abcde"`	`"bd"`	`"x"`	Just `"axc"`	Just `"cxe"`	Just `"axe"`	Just `" x "`
`"abcde"`	`"bd"`	`"xyz"`	Just `"axcyez "`	Just `" xaycze"`	Just `"axcy ze"`	Just `" x y z "`
`"abcde"`	`"bd"`	`"xyzv"`	Just `"axcyez v "`	Just `" x yazcve"`	Just `"axcy z ve"`	Just `" xayczev "`
`"abcde"`	`"bd"`	`"xyzvw"`	Just `"axcyez v w "`	Just `" x y zavcwe"`	Just `"axcy z v we"`	Just `" x y z v w "`

https://doi.org/10.1017/S0956796813000130 Published online by Cambridge University Press

Table 3. *More update bias examples for* get_1 *from Example 1*

bias	s	$get_1\ s$	v'	*bdput bias* ’ ’ $s\ v'$
rear	`"abcd"`	`"bd"`	`"x"`	Just `"ax"`
rear	`"abcde"`	`"bd"`	`"x"`	Just `"axc"`
front	`"abcd"`	`"bd"`	`"x"`	Just `"cx"`
front	`"abcde"`	`"bd"`	`"x"`	Just `"cxe"`
middle	`"abcd"`	`"bd"`	`"x"`	Just `"ax"`
middle	`"abcde"`	`"bd"`	`"x"`	Just `"axe"`
borders	`"abcd"`	`"bd"`	`"x"`	Just `" x"`
borders	`"abcde"`	`"bd"`	`"x"`	Just `" x "`
rear	`"abcd"`	`"bd"`	`"bdx"`	Just `"abcd x"`
rear	`"abcd"`	`"bd"`	`"bdxy"`	Just `"abcd x y"`
rear	`"abcde"`	`"bd"`	`"bdx"`	Just `"abcdex "`
rear	`"abcde"`	`"bd"`	`"bdxy"`	Just `"abcdex y "`
front	`"abcd"`	`"bd"`	`"xbd"`	Just `" xabcd"`
front	`"abcd"`	`"bd"`	`"xybd"`	Just `" x yabcd"`
front	`"abcde"`	`"bd"`	`"xbd"`	Just `" xabcde"`
front	`"abcde"`	`"bd"`	`"xybd"`	Just `" x yabcde"`
middle	`"abcd"`	`"bd"`	`"bxd"`	Just `"ab xcd"`
middle	`"abcd"`	`"bd"`	`"bxyd"`	Just `"ab x ycd"`
middle	`"abcde"`	`"bd"`	`"bxd"`	Just `"abcx de"`
middle	`"abcde"`	`"bd"`	`"bxyd"`	Just `"abcx y de"`
borders	`"abcd"`	`"bd"`	`"xbdy"`	Just `" xabcd y"`
borders	`"abcde"`	`"bd"`	`"xbdy"`	Just `" xabcdey "`
borders	`"abcde"`	`"bd"`	`"xybdzv"`	Just `" x yabcdez v "`

lists. The way we set up this here is as a type class whose only operation is one that tells us how many element positions are associated with a given shape:

class ShapeT σ **where**
  arity :: $\sigma \to$ Nat

For example, for the shape type Nat the number of positions is the given natural number itself:

instance ShapeT Nat **where**
  arity $n = n$

Given a shape type, we can express that some data structure type is shaped accordingly, by providing functions for separating a data structure *into* shape and content and for reassembling a data structure *from* shape and content. The interface is as follows[12]:

class ShapeT $\sigma \Rightarrow$ Shaped $\sigma\ \kappa \mid \kappa \to \sigma$ **where**
  shape   :: $\kappa\ \alpha \to \sigma$
  content :: $\kappa\ \alpha \to [\alpha]$
  fill   :: $(\sigma, [\alpha]) \to \kappa\ \alpha$

[12] The *functional dependency* annotation ‘$\mid \kappa \to \sigma$’ serves to resolve ambiguities and reduce the need for type annotations. It imposes the constraint that the data structure type always uniquely determines the underlying shape type so that in particular the output type of any application *shape* x already follows from the type of x.

and we expect some natural laws to hold. The notion of position numbers must be consistent with how many content elements are actually extracted from a data structure of a given shape: $arity\ (shape\ x) = length\ (content\ x)$. The separation of a data structure into shape and content must be faithful, i.e., reassembly must be possible: $fill\ (shape\ x, content\ x)\ \equiv\ x$. Moreover, if a data structure is put together from some shape and some content, then each of the two aspects must be respected: $shape\ (fill\ (sh,l))\ \equiv\ sh$ and $content\ (fill\ (sh,l))\ \equiv\ l$. Of course, the latter two properties can only be expected if sh and l actually fit together, i.e., if $arity\ sh = length\ l$. Indeed, we will only ever apply $fill$ to such (sh,l)-combinations.

To describe that lists are shaped over Nat, in precisely the way used in this paper so far, we can simply express that the shape of a list is its length, its content is itself, and reassembly is equally straightforward:

```
instance Shaped Nat [] where
  shape l = length l
  content l = l
  fill (n,l) | (n == length l) = l
```

It is easy to see that the laws mentioned above all hold for this instance.

For the sake of an example for another data structure type than lists, consider the following data type definition:

```
data Tree α = Leaf α | Branch (Tree α) (Tree α)
```

One possibility for expressing the shape of a tree is to use a simpler tree with all content elements erased, or rather overwritten with a trivial element. Using the unit type (), whose only value is also denoted by (), the following ShapeT instance makes available this notion of shape tree along with the appropriate *arity*-function:

```
instance ShapeT (Tree ()) where
  arity (Leaf ())       = 1
  arity (Branch t1 t2) = arity t1 + arity t2
```

The other operations are also relatively straightforward recursive traversals:

```
instance Shaped (Tree ()) Tree where
  shape (Leaf _)        = Leaf ()
  shape (Branch t1 t2) = Branch (shape t1) (shape t2)
  content (Leaf a)        = [a]
  content (Branch t1 t2) = content t1 ++ content t2
  fill (s,l) = case go s l of (t,[]) → t
    where go (Leaf ())        l = (Leaf (head l), tail l)
          go (Branch s1 s2) l = (Branch t1 t2, l'')
            where (t1,l')  = go s1 l
                  (t2,l'') = go s2 l'
```

One can again establish that the required laws all hold. More generally, it is possible to provide suitable instances of this kind (i.e., κ () as shape type for some κ) for a whole range of traversable data structures in a generic fashion (Gibbons & Oliveira, 2009).

Given the generic setup, we can now make our contribution from the previous sections data-type-polymorphic. Let us start with the *shapify*-function. It previously had the type $\mathit{shapify} :: (\forall\alpha.[\alpha] \to [\alpha]) \to (\mathsf{Nat} \to \mathsf{Nat})$ and definition $\mathit{shapify}\ \mathit{get}\ n = \mathit{length}\ (\mathit{get}\ [1\,..\,n])$. Now we can provide a generic version:

$$\begin{aligned}&\mathit{shapify} :: (\mathsf{Shaped}\ \sigma\ \kappa, \mathsf{Shaped}\ \sigma'\ \kappa') \Rightarrow (\forall\alpha.\kappa\ \alpha \to \kappa'\ \alpha) \to (\sigma \to \sigma')\\ &\mathit{shapify}\ \mathit{get}\ \mathit{sh} = \mathit{shape}\ (\mathit{get}\ (\mathit{fill}\ (\mathit{sh}, [1\,..\,(\mathit{arity}\ \mathit{sh})])))\end{aligned}$$

in which lists on the input and output sides of *get* have been replaced by some shaped types (type constructors κ and κ') and as a result, instead of a function from Nat to Nat, we obtain a function between the corresponding shape types (σ and σ'). For example, if *get* has type $\mathsf{Tree}\ \alpha \to [\alpha]$, then *shapify get* has type $\mathsf{Tree}\ () \to \mathsf{Nat}$. The implementation of the generic *shapify*-function follows the same idea as the list-specific one, namely to apply *get* to a template structure built according to the given original shape (and with irrelevant content), and to extract the resulting shape at the end. But instead of doing so in an *ad hoc* fashion by direct list construction and consumption, only the interface provided by ShapeT and Shaped is used. The same principle guides the implementation of a data-type-polymorphic version of $\mathit{bff}_{\mathrm{plug}}$:

```
bff_plug :: (Monad μ, Shaped σ κ, Functor κ, Shaped σ' κ') ⇒
            (∀α.κ α → κ' α) → (σ → σ' → Maybe σ)
             → (∀α.κ α → κ' α → μ (κ (Maybe α)))
bff_plug get sput s v' = do let sh = shape s
                            let n = arity sh
                            let t = fill (sh, [1..n])
                            let g = NatMap.fromDistinctAscList (zip [1..n] (content s))
                            let g' = foldr NatMap.delete g (content (get t))
                            sh' ← case sput sh (shape v') of
                                    Nothing → fail "Could not handle shape change."
                                    Just sh'  → return sh'
                            let t = fill (sh', [1..(arity sh')])
                            let h = NatMap.fromDistinctList (zip (content (get t)) (content v'))
                            let h' = NatMap.union h g'
                            return (fmap (flip NatMap.lookup h') t)
```

Instead of constructing a template $[1\,..\,n]$ from a list, we abstract a more general data structure to its shape, construct a template list from that, use it to 'redecorate' the original data structure, and work from there, using the list of content items separately when constructing g. 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 we apply similar adaptations for constructing h later on), and instead of applying *sput* to the lengths of lists, applying it to the *shapes* of s and v'. We create the second t, shadowing the first one, from the new shape (rather than just new list length), and in the end traverse it with *fmap* instead of the list-specific *map*-function. The latter explains why the type signature for $\mathit{bff}_{\mathrm{plug}}$ demands a suitable Functor instance for κ. The same is true in the case of *dbff*, which now has type

$$\begin{aligned}\mathit{dbff} :: (&\mathsf{Monad}\ \mu, \mathsf{Shaped}\ \sigma\ \kappa, \mathsf{Functor}\ \kappa, \mathsf{Shaped}\ \sigma'\ \kappa') \Rightarrow\\ &(\forall\alpha.\kappa\ \alpha \to \kappa'\ \alpha) \to (\sigma \to \sigma' \to \mathsf{Maybe}\ \sigma)\\ &\to (\forall\alpha.\alpha \to \kappa\ \alpha \to \kappa'\ \alpha \to \mu\ (\kappa\ \alpha))\end{aligned}$$

and definition exactly as before, just with *map* replaced by *fmap*.

It is easy to see (by comparing definitions) that the generic functions bff_{plug}, *shapify*, and *dbff* reduce to their list-specific versions given earlier when accordingly applied, since with the ShapeT/Shaped-instances set up for the list case, essentially $arity \equiv id$, $shape \equiv length$, $content \equiv id$, and $fill \equiv snd$. Of course, the real worth is that now we can apply the functions at other types than lists as well. For example, we can use bff_{plug} or *dbff* for a *get* that operates on Trees, and call out to an *sput* derived using the technique of Matsuda *et al.* (2007) from *sget* semantically equivalent to *shapify get* (and thus simpler than *get* itself). The two examples we will instead be looking at here are a bit more mundane, but serve well to illustrate some interesting aspects regarding the generic setup we have established now.

Example 3. Assume we want to bidirectionalize the function $length :: [\alpha] \to \mathsf{Nat}$. It is a bit of an extreme case, since the output side has no content elements, instead only a monomorphic value. Nevertheless, the task should in principle be doable, and there are also some reasonable expectations what the bidirectionalized function might do (probably cutting off a source list at some point if the updated view is a natural number smaller than the original source list length, and extending the list appropriately in the opposite case).

A small technical problem is that the type $[\alpha] \to \mathsf{Nat}$ cannot be directly seen as $\kappa\ \alpha \to \kappa'\ \alpha$ for some instances of κ and κ', precisely because Nat is just a monomorphic type, not a polymorphic type constructor applied to α. This is easily overcome, though, using an auxiliary type constructor definition:

$$\textbf{newtype}\ \mathsf{Const}\ \alpha\ \beta = \mathsf{Const}\ \alpha$$

and then actually bidirectionalizing the following function:

```
$get_3 :: [\alpha] \to \mathsf{Const\ Nat}\ \alpha$
$get_3 = \mathsf{Const} \circ length$
```

Now the question is what shape type to introduce for the type constructor Const Nat, i.e., which σ to choose so that one can give reasonable definitions of functions $arity :: \sigma \to \mathsf{Nat}$, $shape :: \mathsf{Const\ Nat}\ \alpha \to \sigma$, $content :: \mathsf{Const\ Nat}\ \alpha \to [\alpha]$, and $fill :: (\sigma, [\alpha]) \to \mathsf{Const\ Nat}\ \alpha$. It makes sense to consider σ to be Nat itself, although we should be careful not to reuse the interpretation of Nat as shapes for lists from earlier on. After all, the situation is quite different here, as we can see by focusing on *content*. No value of a type Const Nat α can ever contain any α-values, so *content* should never return a non-empty list here. Accordingly, *arity* should always return 0, in contrast to $arity \equiv id :: \mathsf{Nat} \to \mathsf{Nat}$ from the earlier instance. On the other hand, we cannot simply use a trivial type for σ, since the natural number stored in a value of a type Const Nat α must not be lost completely when separating shape and content. In fact, conceptually it should be preserved in the notion of shape. So the appropriate notion of shape here *is* a natural number, but it should be represented using a different (though isomorphic) type than Nat itself. One convenient way to proceed is to use the unit type () again, as follows:

```
**instance** ShapeT (Const Nat ()) **where**
  $arity\ (\mathsf{Const}\ n) = 0$
**instance** Shaped (Const Nat ()) (Const Nat) **where**
  $shape\ (\mathsf{Const}\ n) = \mathsf{Const}\ n$
```

$$\begin{aligned}&\mathit{content}\ (\mathsf{Const}\ n) = [\,]\\&\mathit{fill}\ (\mathsf{Const}\ n, [\,]) = \mathsf{Const}\ n\end{aligned}$$

It is easy to see that the required laws all hold.

What now about $\mathit{sget} \equiv \mathit{shapify}\ \mathit{get}_3 :: \mathsf{Nat} \rightarrow \mathsf{Const\ Nat}\ ()$? Since get_3 maps a list to its length, and the shape of a list is also its length, and the shape of a $\mathsf{Const\ Nat}\ \alpha$ is simply the embedded natural number rewrapped with Const, we have that *sget* is simply Const as well, in particular it is injective and surjective! This makes the task of bidirectionalizing *sget* very simple since for forward functions that are injective and surjective, there is always only one appropriate backward function, namely the one which ignores the original source and applies the inverse of the forward function to the updated view. Here this means:

$$\begin{aligned}&\mathit{sput} :: \mathsf{Nat} \rightarrow \mathsf{Const\ Nat}\ () \rightarrow \mathsf{Maybe\ Nat}\\&\mathit{sput}\ \_\ (\mathsf{Const}\ n) = \mathsf{Just}\ n\end{aligned}$$

Had we invested into a generic version of the search approach from Section 5.2 as well, we could even have avoided the manual provision of *sput* here, instead obtaining the same function simply as $\mathit{sput}_{\text{search}}\ \mathit{sget}$.[13] One way or the other, we obtain

$$\begin{aligned}&\mathit{put}_3 :: \mathsf{Monad}\ \mu \Rightarrow [\alpha] \rightarrow \mathsf{Const\ Nat}\ \alpha \rightarrow \mu\ [\mathsf{Maybe}\ \alpha]\\&\mathit{put}_3 = \mathit{bff}_{\text{plug}}\ \mathit{get}_3\ \mathit{sput}\end{aligned}$$

that behaves exactly as desired. In fact, if we use a default value and encapsulate the Const-wrapping for convenience:

$$\begin{aligned}&\mathit{put}' :: \mathsf{Monad}\ \mu \Rightarrow \alpha \rightarrow [\alpha] \rightarrow \mathsf{Nat} \rightarrow \mu\ [\alpha]\\&\mathit{put}'\ d\ s\ v' = \mathit{dbff}\ \mathit{get}_3\ \mathit{sput}\ d\ s\ (\mathsf{Const}\ v')\end{aligned}$$

we obtain results like these:

s	$\mathit{get}_3\ s$	v'	put' `' '` $s\ v'$
`"abc"`	Const 3	2	Just `"ab"`
`"abc"`	Const 3	4	Just `"abc "`

Example 4. A well-known 'challenge problem' for bidirectionalization approaches is a function that takes a list of pairs and applies a projection to each pair, e.g., the function $\mathit{map\ snd} :: [(\alpha,\beta)] \rightarrow [\beta]$ in Haskell. If a list of (α,β)-pairs is mapped to just the β-components, and the resulting list of βs is shortened or extended, what should happen concerning the (superfluous or missing) αs? These shape changes are neither successfully handled by the syntactic bidirectionalization technique of Matsuda *et al.* (2007) nor by the non-plugged semantic bidirectionalization technique of Voigtländer (2009). Let us see how our new approach fares. (We will solve the shape-updatability, but not the more challenging aspect of potential realignment of αs and βs.)

At first glance, it might seem as if there is not much data-type-genericity to this example, since it is all about lists. However, the type $[(\alpha,\beta)]$ is actually more interesting.

[13] Since *sget* is injective and surjective, a data-type-generic version of $\mathit{sput}_{\text{search}}$ working by enumeration of possible inputs would be guaranteed to succeed and end up with *sput* behaving exactly like the manually given one here.

In particular, when abstracting it to a shape type, we *cannot* simply use a list length as notion of shape. Instead, when abstracting away the αs, any reasonable notion of shape must incorporate the βs, and *vice versa*. To be able to express abstraction from just one of the two element types, we again need a bit of wrapping via an extra type constructor, like with Const in Example 3. We define:

$$\textbf{newtype}\ \mathsf{PairList}\ \alpha\ \beta = \mathsf{PairList}\ [(\alpha,\beta)]$$

and then consider:

$$\begin{aligned}&get_4 :: \mathsf{PairList}\ \alpha\ \beta \to [\beta]\\ &get_4\ (\mathsf{PairList}\ l) = map\ snd\ l\end{aligned}$$

We need a shape type for the type constructor PairList α (since this will be κ here, while κ' will be $[\,]$). As motivated above, we need to preserve all αs, and indeed, the following are very natural definitions (and satisfy all required laws):

$$\begin{aligned}&\textbf{instance}\ \mathsf{ShapeT}\ [\alpha]\ \textbf{where}\\ &\quad arity = length\\ &\textbf{instance}\ \mathsf{Shaped}\ [\alpha]\ (\mathsf{PairList}\ \alpha)\ \textbf{where}\\ &\quad shape\ (\mathsf{PairList}\ l) = map\ fst\ l\\ &\quad content\ (\mathsf{PairList}\ l) = map\ snd\ l\\ &\quad fill\ (as,bs)\mid(length\ as == length\ bs) = \mathsf{PairList}\ (zip\ as\ bs)\end{aligned}$$

Due to the 'Functor $\kappa \Rightarrow$' constraint in the type of bff_{plug} ($dbff$), we will also need a Functor instance for PairList α further below, and provide it as follows:

$$\begin{aligned}&\textbf{instance}\ \mathsf{Functor}\ (\mathsf{PairList}\ \alpha)\ \textbf{where}\\ &\quad fmap\ f\ (\mathsf{PairList}\ l) = \mathsf{PairList}\ (map\ (\lambda(a,b) \to (a,f\ b))\ l)\end{aligned}$$

Now let us take a look at $sget \equiv shapify\ get_4$. Conveniently, it has type $[\alpha] \to \mathsf{Nat}$, which is exactly the type dealt with in Example 3. In fact, more than that, $shapify\ get_4$ is semantically equivalent to *length* as used in Example 3. That is, we can use $put'\ d$ (for some d) from Example 3 as *sput* here, and obtain[14]:

$$\begin{aligned}&put_4 :: \mathsf{Monad}\ \mu \Rightarrow \alpha \to \mathsf{PairList}\ \alpha\ \beta \to [\beta] \to \mu\ (\mathsf{PairList}\ \alpha\ \beta)\\ &put_4\ d = dbff\ get_4\ (put'\ d)\ \bot\end{aligned}$$

Another way to put this is that:

$$put_4\ d = dbff\ get_4\ (\lambda s \to dbff\ (\mathsf{Const} \circ shapify\ get_4)\ sput\ d\ s \circ \mathsf{Const})\ \bot$$

where *sput* is the one from Example 3. Given our observation that in Example 3 the *sput* could actually have been obtained as $sput_{\mathrm{search}}\ (shapify\ get_3)$, we could even write the above as:

$$\begin{aligned}put_4\ d = \ &\textbf{let}\ get_3 = \mathsf{Const} \circ shapify\ get_4\\ &\textbf{in}\ dbff\ get_4\ (\lambda s \to dbff\ get_3\ (sput_{\mathrm{search}}\ (shapify\ get_3))\ d\ s \circ \mathsf{Const})\ \bot\end{aligned}$$

[14] It so happens that the third argument of *dbff* will never be used here, so we choose $\bot$ as that default value.

Note that this working depends only on the type of get_4. In particular, it does not depend on the observation from above that *shapify* get_4 is semantically equivalent to *length*. The shape abstraction from get_4 to $get_3 \equiv \mathsf{Const} \circ \mathit{shapify}\ get_4$ and then double use of *dbff*, on the outer level for get_4 and on the inner level for get_3 defined in terms of get_4, would have been possible for other get_4-functions as well. In fact, besides the strategies from Sections 5.1–5.3, we could now add a fourth one: ‘5.4 Shape-bidirectionalization by bootstrapping’, which uses $\mathit{bff}_{\mathrm{plug}}$ (or *dbff*) *itself* as a plug-in. Of course, all this is only possible since we have provided a data-type-generic account of pluggable bidirectionalization. For otherwise, we would not have been able to use *dbff* both for a function and its shape-abstracted version.

Given the specific get_4-function above, we obtain results like these:

s	get_4 (PairList s)	v'	put_4 `' '` (PairList s) v'
zip `"ab"` $[1,2]$	$[1,2]$	$[3]$	Just (PairList $[($`'a'`$,3)]$)
zip `"ab"` $[1,2]$	$[1,2]$	$[3,4,5]$	Just (PairList $[($`'a'`$,3),($`'b'`$,4),($`' '`$,5)]$)

Finding good pragmatic bias strategies, as in Section 6, for the case of non-lists is a possible topic for future work.

8 Conclusion

We have shown how to refactor the semantic bidirectionalization technique of Voigtländer (2009) in such a way that other techniques can be used as plug-ins. The key idea is to separate shape from content, thus simplifying the problem of explicit bidirectionalization by posing it only on the shape level (going from *get* to $sget \equiv \mathit{shapify}\ get$). That way, for example, the existing syntactic bidirectionalization technique of Matsuda *et al.* (2007) can give far better results (in combination) than for the general problem (on its own). We have also developed a data-type-generic account. An interesting development is that we have moved automatic bidirectionalization toward more customizability by users/programmers, both in terms of choosing plug-ins and in terms of providing explicit bias. That brings the techniques closer in spirit to the domain-specific language approaches in the tradition of Foster *et al.* (2007).

Finally, a few more words about formal properties of *get*/*put*-pairs are in order. We have taken laws GetPut (1) and PutGet (2), in the form of Definition 1, as consistency conditions. So in the terminology of Foster *et al.* (2012), we have considered partial well-behaved lenses. The literature also knows PutPut:

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

which as one interesting consequence together with GetPut implies undoability:

$$put\ (put\ s\ v')\ (get\ s) \equiv s\ .$$

Or, for partial *put*, both are required to hold whenever $put\ s\ v'$ is defined. The technique of Matsuda *et al.* (2007) satisfies these two laws (thus producing partial very well-behaved lenses) by virtue of being based on the constant-complement approach of Bancilhon & Spyratos (1981). Although not explicitly proved by Voigtländer (2009), the same is true for his technique. In fact, it can be reformulated via the constant-complement approach as well (Foster *et al.*, 2012). So the question is natural whether semantic bidirectionalization with plug-ins can also be so based, and satisfies PutPut and undoability as well. The answer is No, as invocations like *dput* `' '` `"abcd"` `"x"` ≡ `Just "ax"` ≡ *dput* `' '` `"abyd"` `"x"` for Example 1 show, where *dput* = *dbff* get_1 *sput* (cf. the continuation of this example in Section 5.1). Clearly, there is no way that *dput* `' '` `"ax"` `"bd"` is both `Just "abcd"` and `Just "abyd"` as undoability would demand; instead: *dput* `' '` `"ax"` `"bd"` ≡ `Just "ab d"`. (PutPut fails for a similar reason.) Is that a bad news? We think not: *any* method that successfully deals with insertion and deletion updates for a function like the get_1 under consideration here will have to give up PutPut and undoability. Indeed, these two properties are often considered undesirable, precisely because they significantly limit the transformations one can hope to deal with (Keller, 1987; Gottlob *et al.*, 1988; Foster *et al.*, 2007).

Acknowledgments

We thank the anonymous reviewers of the earlier related paper (Voigtländer *et al.*, 2010) and of the present paper for their insightful comments and suggestions. Part of this work is supported by JSPS KAKENHI grant numbers 22800003 and 24700020.

References

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

Bancilhon, F. & Spyratos, N. (1981) Update semantics of relational views. *ACM Trans. Database Syst.* **6**(3), 557–575.

Bohannon, A., Foster, J. N., Pierce, B. C., Pilkiewicz, A. & Schmitt, A. (2008) Boomerang: Resourceful lenses for string data. In *Proceedings of Principles of Programming Languages*. ACM Press, pp. 407–419.

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

Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A. & Terwilliger, J. F. (2009) Bidirectional transformations: A cross-discipline perspective. In *Proceedings of the International Conference on Model Transformation*, LNCS, vol. 5563. Springer-Verlag, pp. 260–283.

Foster, J. N., Greenwald, M. B., Moore, J. T., Pierce, B. C. & Schmitt, A. (2007) Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.* **29**(3), 17.

Foster, J. N., Matsuda, K. & Voigtländer, J. (2012) Three complementary approaches to bidirectional programming. In *Revised Lectures of Spring School on Generic and Indexed Programming 2010*, LNCS, vol. 7470. Springer-Verlag, pp. 1–46.

Foster, J. N., Pilkiewicz, A. & Pierce, B. C. (2008) Quotient lenses. In *Proceedings of the International Conference on Functional Programming*. ACM Press, pp. 383–395.

Gibbons, J. & Oliveira, B. C. d. S. (2009) The essence of the iterator pattern. *J. Funct. Program.* **19**(3–4), 377–402.

Giesl, J. (2000) Context-moving transformations for function verification. In *Selected Papers on Logic-Based Program Synthesis and Transformation 1999*, LNCS, vol. 1817. Springer-Verlag, pp. 293–312.

Giesl, J., Kühnemann, A. & Voigtländer, J. (2007) Deaccumulation techniques for improving provability. *J. Logic Algebr. Program.* **71**(2), 79–113.

Gottlob, G., Paolini, P. & Zicari, R. (1988) Properties and update semantics of consistent views. *ACM Trans. Database Syst.* **13**(4), 486–524.

Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K. & Nakano, K. (2010) Bidirectionalizing graph transformations. In *Proceedings of the International Conference on Functional Programming*. ACM Press, pp. 205–216.

Hu, Z., Mu, S.-C. & Takeichi, M. (2008) A programmable editor for developing structured documents based on bidirectional transformations. *Higher-Order Symb. Comput.* **21**(1–2), 89–118.

Jay, C. B. (1995) A semantics for shape. *Sci. Comput. Program.* **25**(2–3), 251–283.

Keller, A. M. (1987) Comments on Bancilhon and Spyratos' 'Update semantics and relational views'. *ACM Trans. Database Syst.* **12**(3), 521–523.

Matsuda, K., Hu, Z., Nakano, K., Hamana, M. & Takeichi, M. (2007) Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proceedings of the International Conference on Functional Programming*. ACM Press, pp. 47–58.

Matsuda, K., Hu, Z., Nakano, K., Hamana, M. & Takeichi, M. (2009) Bidirectionalizing programs with duplication through complementary function derivation. *Comput. Softw.* **26**(2), 56–75 (In Japanese).

Matsuda, K. & Wang, M. (2013) Bidirectionalization for free with runtime recording. In *Proceedings of Principles and Practice of Declarative Programming*. ACM Press, pp. 297–308.

Pacheco, H. & Cunha, A. (2010) Generic point-free lenses. In *Proceedings of Mathematics of Program Construction,* LNCS, vol. 6120. Springer-Verlag, pp. 331–352.

Pacheco, H., Cunha, A. & Hu, Z. (2012) Delta lenses over inductive types. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.* **49**. Available at: http://journal.ub.tu-berlin.de/eceasst/issue/view/59

Pettorossi, A. (1977) Transformation of programs and use of tupling strategy. In *Proceedings of Informatica 77, Bled, Yugoslavia*. pp. 1–6.

Peyton Jones, S. L. (ed). (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge, UK: Cambridge University Press.

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

Strachey, C. (2000) *Fundamental Concepts in Programming Languages*. Lecture notes for a course at the International Summer School in Computer Programming, 1967. Reprint appeared in *Higher-Order Symb. Comput.* **13**(1–2), 11–49.

Voigtländer, J. (2009) Bidirectionalization for free! In *Proceedings of Principles of Programming Languages*. ACM Press, pp. 165–176.

Voigtländer, J. (2012) Ideas for connecting inductive program synthesis and bidirectionalization. In *Proceedings of Partial Evaluation and Program Manipulation*. ACM Press, pp. 39–42.

Voigtländer, J., Hu, Z., Matsuda, K. & Wang, M. (2010) Combining syntactic and semantic bidirectionalization. In *Proceedings of the International Conference on Functional Programming*. ACM Press, pp. 181–192.

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

Wadler, P. (1990) Deforestation: Transforming programs to eliminate trees. *Theor. Comput. Sci.* **73**(2), 231–248.

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

Wang, M., Gibbons, J., Matsuda, K. & Hu, Z. (2013) Refactoring pattern matching. *Sci. Comput. Program.* **78**(11), 2216–2242.

Wang, M., Gibbons, J. & Wu, N. (2011) Incremental updates for efficient bidirectional transformations. In *Proceedings of the International Conference on Functional Programming*. ACM Press, pp. 392–403.