Combining Syntactic and Semantic Bidirectionalization

Janis Voigtländer *
University of Bonn
Römerstraße 164
53117 Bonn, Germany
jv@iai.uni-bonn.de

Zhenjiang Hu
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku
Tokyo 101-8430, Japan
hu@nii.ac.jp

Kazutaka Matsuda
Tohoku University
6-3-09 Aramaki aza Aoba, Aoba-ku
Sendai 980-8579, Japan
kztk@kb.ecei.tohoku.ac.jp

Meng Wang
University of Oxford
Wolfson Building, Parks Road
Oxford OX1 3QD, United Kingdom
meng.wang@comlab.ox.ac.uk

Abstract

Matsuda et al. [2007, ICFP] and Voigtländer [2009, POPL] 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. Being fundamentally different in approach, both techniques have their respective strengths and weaknesses. Here we develop a synthesis of the two techniques to good effect. On the intersection of their applicability domains we achieve more than what a simple union of applying the techniques side by side delivers.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Languages*]: Language Constructs and Features—Data types and structures, Polymorphism; H.2.3 [*Database Management*]: Languages—Data manipulation languages, Query languages

General Terms Design, Languages

Keywords program transformation, view-update problem

1. Introduction

Bidirectionalization is the task to given some function $get :: \tau_1 \to \tau_2$ produce 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. 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. A survey of relevant techniques and open problems has recently appeared [Czarnecki et al. 2009], and 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; Hu et al. 2004; Matsuda et al. 2007, 2009; Voigtländer 2009].

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". (We will briefly introduce both techniques in Section 2.)

Both syntactic and semantic bidirectionalization have their strengths and weaknesses. Syntactic bidirectionalization heavily depends 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 linearity restrictions and absence of intermediate results between 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 at present is unable to deal with view updates that change the shape of a data structure (more on this critical issue below). The syntactic approach, on the other hand, is successful for many such shape-changing updates, and can deal with non-polymorphic functions.

In this paper we develop an approach for combining syntactic and semantic bidirectionalization. The resulting technique inherits the limitations in program coverage from *both* techniques. That is, except for some extensions we will consider later on, only functions that are written in the first-order language, are linear, and treeless in the sense of Wadler [1990], and *moreover* are polymorphic, can be dealt with. What we gain by the combination is improved updatability. Not only do we bring the possibility of shape-changing updates to semantic bidirectionalization, but also will the combined technique be superior to syntactic bidirectionalization on its own in many cases.

To explain what we mean by improved updatability, we have to elaborate on the phrase "in a meaningful way" in the first sentence of this introduction, and on "suitable" at the start of the 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 and Spyratos 1981] known as GetPut and PutGet [Foster et al. 2007].

* The research reported here was performed while this author visited the National Institute of Informatics, Tokyo, under a fellowship by the Japan Society for the Promotion of Science, ID No. PE09076.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICFP'10, September 27–29, 2010, Baltimore, Maryland, USA.
Copyright © 2010 ACM 978-1-60558-794-3/10/09…$10.00

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 updatability. 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. With this measure in hand, one can compare different bidirectionalization methods. Semantic bidirectionalization as proposed by Voigtländer [2009] has the problem that $put\ s\ v'$ can only be defined when $get\ s$ and v' have the same *shape* (length of a list, structure of a tree, ..., and in some situations even with constraints on the equivalence and relative ordering of elements in data structures). Syntactic bidirectionalization as proposed by Matsuda et al. [2007] does not suffer from such a central and common (to all invocations) updatability weakness, but in many cases also rejects updates that one would really like to see accepted. The benefit of our combined technique now is that on the intersection of the classes of programs to which the original syntactic and semantic techniques apply, we can do strictly better in terms of updatability than either technique in isolation. We are never worse than the better of the two in a specific case.

The combination strategy we pursue is 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 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"), updatability is hampered. In our combined approach we divide 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 programs, on which it can yield better results, benefitting both.

Technically, we treat syntactic bidirectionalization as a black box. Or rather, our eventual combined technique does so; for the sake of analyzing examples, we look into the box; but for actually executing the combined technique the syntactic technique could be a completely external component. Semantic bidirectionalization is treated as a glass box; we do look into it, and we refactor it to enable a plugging in of the syntactic technique. Indeed, our dissection of the semantic bidirectionalization technique is an independent contribution of this paper, beyond the specific use case of combining the techniques of Matsuda et al. [2007] and Voigtländer [2009]. In principle, our refactoring allows also other approaches (than that of Matsuda et al.) for obtaining bidirectional transformations on shapes to be plugged into the semantic technique.

Since our purpose here is to focus on the combination of techniques, we concentrate on one specific kind of functions, namely on functions from lists to lists. The original techniques we combine apply to algebraic data types more generally. In particular, Voigtländer [2009, Section 6] employs generic programming techniques to deal with trees and the like. Something similar should be possible here, but we have not worked out the details. Our key ideas can all be explained, and hopefully appreciated, in the setting of lists only, and that explanation is what we seek to do. For the same reason, we do not consider type classes as Voigtländer [2009, Sections 4 and 5] does; again, we think our ideas here could be transferred to those settings, but we refrain from doing so for the sake of focus.

Our presentation will be partly example-driven, partly program-driven as we proceed through the refactoring and discovery process regarding generalization opportunities. We do state lemmas and theorems, but do not give formal proofs. These proofs can all be done similarly to those by Voigtländer [2009], employing free theorems [Wadler 1989]. We will comment in a bit more detail where appropriate.

As a final preparation before diving right in, we slightly revise the consistency conditions (1) and (2). Since our emphasis is on the updatability inherent in a get/put-pair, we make the partiality of put explicit in the type via optionality of the return value. 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
$$get\ s' \equiv v'\,.$$

2. The Original Techniques

We briefly introduce the two techniques we want to combine. Readers content with considering syntactic bidirectionalization as a black box can safely skip the next subsection and directly jump to Section 2.2. The combination approach can still be understood then, but it will be more difficult to appreciate some of the analysis of examples later on.

2.1 Syntactic Bidirectionalization

The technique of Matsuda et al. [2007] builds on the constant-complement approach of Bancilhon and Spyratos [1981]. The basic idea 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 an inverse $inv :: (\tau_2, \tau_3) \to \tau_1$ of $paired$, one obtains that

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

makes equations (1) and (2) true.

In reality, asking for a full inverse inv of $paired$ is too much. The function $paired$ may not even be surjective. So one relaxes inv to be a partial function, either implicitly as Matsuda et al. [2007] do, or explicitly in the type. With

$$inv :: (\tau_2, \tau_3) \to \mathsf{Maybe}\ \tau_1$$

and 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
$$paired\ s' \equiv (v', c)\,,$$

we obtain that

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

is consistent for get in the sense of Definition 1.

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, 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 two examples. A generalization over the above picture is that instead of Maybe we will use an arbitrary monad. This allows for more flexible use of the resulting put-function, and also enables us to provide informative error messages if desired.

Example 1. Assume our get-function is as follows, sieving a list to keep only every second element:

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

This function fulfills the syntactic prerequisites imposed by Matsuda et al. [2007]. They are (necessary[1] and sufficient): that functions must be first-order, must be *linear* (no variable occurs more than once in a single right-hand side), and that there must be no function call with anything else than variables in its arguments.

Given the above, the following complement function is automatically derived:

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

(Matsuda et al. 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.)

The basic ideas for the derivation of $compl$ are that variables dropped when going from left to right in a defining equation of get are collected by $compl$, and that, where necessary, different data constructors (of same arity/type) are used on the right-hand sides of $compl$ to disambiguate between overlapping ranges of right-hand sides of get. (In this specific example, this is not what causes different data constructors to be used. Instead, the simple fact that different arities are required, due to different numbers of dropped variables and recursive calls, leads to different data constructors.)

Tupling gives the following definition for the paired function:

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

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

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

Finally,

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

can be fused to:

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

Note that for this function, $put\ s\ v'$ fails if and only if $length\ v' \neq length\ (get_1\ s)$. If it succeeds, it mixes the elements of s and v' as in, e.g., $fromJust\ (put\ [1\,..\,6]\ [7\,..\,9]) = [1, 7, 3, 8, 5, 9]$.

An implementation of the syntactic bidirectionalization method is available at `http://www.kb.ecei.tohoku.ac.jp/~kztk/bidirectionalization/`. It automatically performs the steps from get to $compl$ and $paired$. It also performs the syntactic inversion from $paired$ to inv, though without the explicit monadic error propagation we have used here. It is not always the case as in the above example that inv can directly be interpreted as a deterministic program. Instead, it can happen that the non-failing equations have overlapping left-hand sides, leading to a nondeterministic program, in which case a backtracking search becomes necessary. Such a backtracking search is what the implementation then does, though in practice it would of course be preferable to directly obtain a deterministic program.[2] Also, the implementation does not at present realize the final fusion step, but instead works with the definition of put in terms of inv and $compl$. Clearly, these "deficiencies" of the implementation only affect the efficiency of the bidirectional transformation, not its correctness/consistency. For the above and the following example, we continue to perform ("by hand") the determinization and fusion steps, because the put-function thus obtained typically gives a better picture of the achieved updatability.

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

[2] An alternative would be to run the syntactically inverted program in a functional logic language [Antoy and Hanus 2010].

Example 2. Assume our get-function is as follows, keeping every element of a list except for the last one:[3]

$$
\begin{array}{l}
get_2 :: [\alpha] \to [\alpha] \\
get_2\ [\,] \qquad\quad\ = [\,] \\
get_2\ [x] \qquad\quad\ = [\,] \\
get_2\ (x : y : zs) = x : (get'\ y\ zs) \\
\\
get' :: \alpha \to [\alpha] \to [\alpha] \\
get'\ x\ [\,] \qquad\ = [\,] \\
get'\ x\ (y : zs) \ = x : (get'\ y\ zs)
\end{array}
$$

Then the syntactic approach produces the following complement function:

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

Note that there are no data constructors around recursive calls. This omission is possible, because no variables are dropped in the respective equations and because automatic range analysis can tell the right-hand sides of those equations never overlap, for any instantiation of variables, with any other right-hand sides of the same function.

Tupling, inversion, and fusion (not spelled out here in detail) ultimately give:

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

The updatability of these functions is that $put\ s\ v'$ succeeds if and only if $length\ v'$ and $length\ (get_2\ s)$ are equal or both greater than zero. For the latter case, the behavior of put is best understood by observing that the definition of put' is semantically equivalent (depending on one of the monad laws) to:

$$
\begin{array}{l}
put' :: \mathsf{Monad}\ \mu \Rightarrow \alpha \to [\alpha] \to [\alpha] \to \mu\ (\alpha, [\alpha]) \\
put'\ y\ zs\ [\,] \qquad\quad = return\ (last\ (y : zs), [\,]) \\
put'\ y\ zs\ (x' : v') = return\ (x', v' +\!\!+ [last\ (y : zs)])
\end{array}
$$

and thus the third defining equation of put is equivalent (again depending on the same monad law) to the following two:

$$
\begin{array}{l}
put\ (x : y : zs)\ (x' : [\,]) \qquad\ = return\ (x' : [last\ (y : zs)]) \\
put\ (x : y : zs)\ (x' : y' : v') = return\ (x' : y' : \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad v' +\!\!+ [last\ (y : zs)])
\end{array}
$$

and thus to:

$$
put\ (x : y : zs)\ (x' : v') = return\ (x' : v' +\!\!+ [last\ (y : zs)])
$$

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

2.2 Semantic Bidirectionalization

As already mentioned, we will develop our combined bidirectionalization technique only for lists, and only for fully polymorphic functions to bidirectionalize. So from now on, let

$$
get :: [\alpha] \to [\alpha]
$$

be fixed but arbitrary (except when discussing concrete examples, of course).

The intuition underlying the method of Voigtländer [2009] is that put can gain information about the get-function by applying it to suitable input. 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 ascending lists over integer values. Say get is $tail$, then every list $[0\,..\,n]$ is mapped to $[1\,..\,n]$, 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 $[0\,..\,n]$ just as well, even to lists over non-integer types, thanks to parametric polymorphism [Reynolds 1983; Strachey 1967].

Let us further consider the $tail$ example as in 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 $n + 1$. Then by applying $tail$ to the same-length list $[0\,..\,n]$, put learns that the original view from which v' was obtained by updating had length n, and also to what element in s each element in that original view corresponded. Being conservative, the current semantic bidirectionalization method will only accept v' if it has retained that length n. For then, we also know directly the associations between elements in v' and positions in the original source. Now, to produce the updated source, we can go over all positions in $[0\,..\,n]$ and fill them with the associated values from v'. For positions for which there is no corresponding value in v', because these positions were omitted when applying $tail$ to $[0\,..\,n]$, we can look up the correct value in s rather than in v'. For the concrete example, this will only concern position 0, for which we naturally take over the head element from s.

The same strategy works also for general get. In short, given s, produce a kind of template $t = [0\,..\,n]$ of the same length, together with an association g between integer values in that template and the corresponding values in s. Then apply get to t and produce a further association h by matching this template view versus the updated proper value view v'. Combine the two associations into a single one h', giving precedence to h whenever an integer template index is found in both h and g. Thus, it is guaranteed that we will only resort to values from the original source s when the corresponding position did not make it into the view, and thus there is no way it could have been affected by the update. Finally, produce an updated source by filling all positions in $[0\,..\,n]$ with their associated values according to h'.

The above strategy is exactly what Voigtländer [2009] implements for the special case $get :: [\alpha] \to [\alpha]$. We recall the corresponding Haskell definitions, reformulating just a bit:

- Instead of presenting a higher-order *bff*-function that turns get into put, we directly give a definition of put that refers to a top-level-defined get.
- We write put in monadic style to provide for more convenient error handling.

We define put as follows, using some functions from module Data.IntMap. Their type signatures, which should provide sufficient documentation, are given in Figure 1. One detail in behavior to mention additionally is that IntMap.$union$ is left-biased for in-

```
fromList            :: [(Int, α)] → IntMap α
fromDistinctAscList :: [(Int, α)] → IntMap α
empty               :: IntMap α
insert              :: Int → α → IntMap α → IntMap α
union               :: IntMap α → IntMap α → IntMap α
lookup              :: Int → IntMap α → Maybe α
```

Figure 1. Functions from module Data.IntMap.

tegers occurring as keys in both input maps. This realizes exactly the "precedence of h over g" alluded to in the informal exposition above.

```
put :: (Monad μ, Eq α) ⇒ [α] → [α] → μ [α]
put s v' =
 do let t = [0 .. length s − 1]
    let g = IntMap.fromDistinctAscList (zip t s)
    h ← assoc (get t) v'
    let h' = IntMap.union h g
    return (map (fromJust ∘ flip IntMap.lookup h') t)

assoc :: (Monad μ, Eq α) ⇒ [Int] → [α] → μ (IntMap α)
assoc []       []       = return IntMap.empty
assoc (i : is) (b : bs) =
 do m ← assoc is bs
    case IntMap.lookup i m of
      Nothing → return (IntMap.insert i b m)
      Just c  → if b == c
                  then return m
                  else fail "Update violates equality."
assoc _        _        = fail "Update changes the length."
```

The following theorem is essentially (up to the different way of expressing partiality of put) what is proved by Voigtländer [2009] in Theorems 1 and 2.

Theorem 1. *Let τ be a type that is an instance of* Eq *in such a way that the definition given for $==$ makes it reflexive, symmetric, and transitive.*

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

Corollary 1. *Let τ be a type that is an instance of* Eq *in a way that the definition given for $==$ agrees with semantic equality. Then $put :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$ is consistent for $get :: [\tau] \to [\tau]$.*

The somewhat complicated definition of $assoc$ and the references to Eq and $==$ in the function definitions and in Theorem and Corollary 1 are due to the fact that get could duplicate some of its input list elements, which requires special handling. As we are anyway going to outlaw such copying (driven by the utilized syntactic bidirectionalization method's inability to deal with non-linear functions), we do not elaborate on this further here. It is discussed in detail by Voigtländer [2009, end of Section 2 and start of Section 3].

Applying semantic bidirectionalization is very easy. We simply put the function definitions of put and $assoc$ side by side with the get-function we want to bidirectionalize.

Example 1 (continued). Just as was the case for syntactic bidirectionalization here, $put\ s\ v'$ fails if and only if $length\ v' \neq length\ (get_1\ s)$. Indeed, the two versions of put are semantically equivalent (at type $[\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$, for τ that is an instance of Eq). Here are a few representative calls and their results:

		syntactic	semantic
s	v'	$put\ s\ v'$	$put\ s\ v'$
"abcd"	"x"	Nothing	Nothing
"abcd"	"xy"	Just "axcy"	Just "axcy"
"abcd"	"xyz"	Nothing	Nothing
"abcde"	"x"	Nothing	Nothing
"abcde"	"xy"	Just "axcye"	Just "axcye"
"abcde"	"xyz"	Nothing	Nothing

Example 2 (continued). While, as we have seen, the put-function obtained via syntactic bidirectionalization succeeds whenever $length\ v'$ and $length\ (get_2\ s)$ are equal or both greater than zero, for the put-function obtained via the semantic technique $put\ s\ v'$ will only be successful if $length\ v' = length\ (get_2\ s)$. Again, a few representative calls and their results:

		syntactic	semantic
s	v'	$put\ s\ v'$	$put\ s\ v'$
""	""	Just ""	Just ""
""	"x"	Nothing	Nothing
"a"	""	Just "a"	Just "a"
"a"	"x"	Nothing	Nothing
"ab"	""	Nothing	Nothing
"ab"	"x"	Just "xb"	Just "xb"
"ab"	"xy"	Just "xyb"	Nothing
"abc"	""	Nothing	Nothing
"abc"	"x"	Just "xc"	Nothing
"abc"	"xy"	Just "xyc"	Just "xyc"
"abc"	"xyz"	Just "xyzc"	Nothing

We see that syntactic and semantic bidirectionalization can agree or disagree in terms of updatability. Our aim is to combine the two into a technique that will represent a significant improvement over both. A reviewer suggested that on the intersection of their applicability domains, the syntactic technique on its own is never worse than the semantic technique on its own. We believe this to be true. So in a sense, we "only" try to improve over the syntactic method. Interestingly, the way forward is to defer that method to the role of a plug-in, with the technique of Voigtländer [2009] in the master role. As preparation, we refactor that latter technique.

3. Refactoring Semantic Bidirectionalization

From now on, assume that for every $n :: \mathsf{Int}$, $get\ [0\,..\,n]$ contains no duplicates. We call this property *semantic linearity*. It will clearly be fulfilled if get's syntactic definition is linear.

3.1 Specialization to Semantically Linear get-Functions

We define

$$put_{\text{linear}} :: \mathsf{Monad}\ \mu \Rightarrow [\alpha] \to [\alpha] \to \mu\ [\alpha]$$

like put (but note the different type), except that the call to $assoc$ is replaced by a call, with the same arguments, to the following function:

```
assoc' :: Monad μ ⇒ [Int] → [α] → μ (IntMap α)
assoc' []       []       = return IntMap.empty
assoc' (i : is) (b : bs) = do m ← assoc' is bs
                              return (IntMap.insert i b m)
assoc' _        _        = fail "Update changes the length."
```

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

Theorem 2. *For every type τ,*

$$put_{\text{linear}} :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is consistent for

$$get :: [\tau] \to [\tau].$$

But semantic linearity gives us more. It rules out one important cause 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* if for every $s, v' :: [\tau]$, if $length\ v' = length\ (get\ s)$, then $put\ s\ v' \equiv \mathsf{Just}\ s'$ for some $s' :: [\tau]$.

Note that the original $put :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$ from Section 2.2 is not in general fixed-shape-friendly for get-functions that are not semantically linear. On the other hand, $put_{\text{linear}} :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$ is not even generally *consistent* for get-functions that are not semantically linear. But since we *have* now restricted get-functions to be semantically linear, we have consistency by the above theorem, and can moreover prove the following one.

Theorem 3. *For every type τ,*

$$put_{\text{linear}} :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is fixed-shape-friendly for get.

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 updatability (which also holds for the put from Section 2.2, of course).

Theorem 4. *For every type τ and $s, v' :: [\tau]$, if $length\ v' \neq length\ (get\ s)$, then $put_{\text{linear}}\ s\ v' :: \mathsf{Maybe}\ [\tau] \equiv \mathsf{Nothing}$.*

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

3.2 Decomposition to Expose the Shape Aspect

We refactor put_{linear} to make the treatment of shapes (list lengths) explicit. To that end, we first define $sput_{\text{naive}}$ as follows:

```
sput_naive :: Monad μ ⇒ Int → Int → μ Int
sput_naive l_s l_v' = if l_v' == length (get [0 .. l_s − 1])
                      then return l_s
                      else fail "Update changes the length."
```

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

```
put_refac :: Monad μ ⇒ [α] → [α] → μ [α]
put_refac s v' =
  do let l_s = length s
     let g = IntMap.fromDistinctAscList (zip [0 .. l_s − 1] s)
     l' ← sput_naive l_s (length v')
     let t = [0 .. l' − 1]
     let h = fromDistinctList (zip (get t) v')
     let h' = IntMap.union h g
     return (map (fromJust ∘ flip IntMap.lookup h') t)

fromDistinctList = IntMap.fromList
```

The refactoring consists of:

- making the check for equal length of $get\ [0\,..\,length\ s-1]$ and v', otherwise performed inside $assoc'$, explicit, and outsourcing it to $sput_{\text{naive}}$, and
- realizing that once this check was successful, the role of $assoc'$ can be taken over by zip and $\mathsf{IntMap}.fromList$.

The following lemma establishes that the refactoring is indeed correct, and thus transports the (good and bad) properties of put_{linear}, namely Theorems 2–4, to put_{refac}.

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

$$put_{\text{linear}}\ s\ v' :: \mathsf{Maybe}\ [\tau] \equiv put_{\text{refac}}\ s\ v' :: \mathsf{Maybe}\ [\tau].$$

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 $sput_{\text{naive}}$, we can change that behavior. For example, it is tempting to change the last line of the above definition of $sput_{\text{naive}}$ to:

```
else return (head [l_s' | l_s' ← [0 ..],
                        l_v' == length (get [0 .. l_s' − 1])])
```

That would correspond to a "brute force" search for an appropriate new source shape. A reviewer pointed out that, thanks to semantic linearity of get, it would be sufficient to start the search for $l_{s'}$ at $l_{v'}$, i.e., that one could replace $[0\,..]$ by $[l_{v'}\,..]$ above, and that further optimizations like memoization might be possible to speed up the search. However, our motivation for discarding the "brute force" approach is not primarily efficiency. We are looking for a more effective approach in the sense that updates should be meaningful to the user. The kind of perfect updatability that could be achieved using pure search (possibly with some limited guidance by the user via heuristics, expressed as reorderings of the candidate list $[l_{v'}\,..]$) could produce quite unintuitive results. As reckoned by the same reviewer, we expect that by replacing $sput_{\text{naive}}$ with a more "intelligent" or "intuition-guided" shape-bidirectionalizer, such as one based on the constant-complement approach, we will get more useful results overall.

4. Combining Syntactic and Semantic Bidirectionalization

Our key idea is abstraction: from lists to list lengths (generally, from data structures to their shapes). Since we prefer to work with a more symbolic representation than built-in integers provide, we first define a new data type and conversion functions as follows:

```
data Nat = Z | S Nat

toNat :: Int → Nat
toNat 0         = Z
toNat n | n > 0 = S (toNat (n − 1))

fromNat :: Nat → Int
fromNat Z     = 0
fromNat (S n) = 1 + fromNat n
```

and then a function $sget$ as follows:

```
sget :: Nat → Nat
sget l_s = toNat (length (get [0 .. fromNat l_s − 1]))
```

The point, later, will be that one can also directly derive a simplified syntactic definition for $sget$ from a given definition for get. 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 $sget$. Of course,

$$\begin{aligned} sput\ l_s\ l_{v'} = \mathbf{case}\ & sput_{\text{naive}}\ (fromNat\ l_s)\ (fromNat\ l_{v'}) \\ \mathbf{of}\ & \mathsf{Nothing} \to \mathsf{Nothing} \\ & \mathsf{Just}\ l \quad \to \mathsf{Just}\ (toNat\ l) \end{aligned}$$

is always a valid choice, with any of the versions of $sput_{\text{naive}}$ discussed in Section 3.2, but for many get-functions there will be better alternatives!

We now define put_{comb} as below. There are three differences from put_{refac}: we use Nat instead of Int to call out to $sput$ instead of $sput_{\text{naive}}$, we generate an error message in case $sput$ 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.

$$\begin{aligned} & put_{\text{comb}} :: \mathsf{Monad}\ \mu \Rightarrow [\alpha] \to [\alpha] \to \mu\ [\mathsf{Maybe}\ \alpha] \\ & put_{\text{comb}}\ s\ v' = \\ & \ \mathbf{do}\ \mathbf{let}\ l_s = length\ s \\ & \quad \mathbf{let}\ g = \mathsf{IntMap}.fromDistinctAscList\ (zip\ [0\,..\,l_s - 1]\ s) \\ & \quad l' \leftarrow maybe\ (fail\ \texttt{"Could not handle shape change."}) \\ & \qquad\qquad return \\ & \qquad\qquad (sput\ (toNat\ l_s)\ (toNat\ (length\ v'))) \\ & \quad \mathbf{let}\ t = [0\,..\,fromNat\ l' - 1] \\ & \quad \mathbf{let}\ h = fromDistinctList\ (zip\ (get\ t)\ v') \\ & \quad \mathbf{let}\ h' = \mathsf{IntMap}.union\ h\ g \\ & \quad return\ (map\ (flip\ \mathsf{IntMap}.lookup\ h')\ t) \end{aligned}$$

The proof of the following theorem is very similar to that by Voigtländer [2009] for his Theorems 1 and 2, but of course additionally uses the assumption that $sput$ is consistent for $sget$.

Theorem 5. *Let τ be a type.*

- *For every $s :: [\tau]$,*

  $$put_{\text{comb}}\ s\ (get\ s) :: \mathsf{Maybe\ [Maybe}\ \tau] \equiv \mathsf{Just}\ (map\ \mathsf{Just}\ s)\,.$$

- *For every $s, v' :: [\tau]$ and $s' :: [\mathsf{Maybe}\ \tau]$, if $put_{\text{comb}}\ s\ v' :: \mathsf{Maybe\ [Maybe}\ \tau] \equiv \mathsf{Just}\ s'$, then*

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

The following theorem can also be shown to hold, basically by observing that if $length\ v' = length\ (get\ s)$, then

$$sget\ (toNat\ (length\ s)) \equiv toNat\ (length\ v')\,,$$

and thus, by consistency of $sput$ for $sget$, inside the put_{comb}-definition l' will be successfully assigned the value $toNat\ l_s$, and subsequently every index position from t will lead to a successful lookup in h', because at least g will contain a matching entry.

Theorem 6. *For every type τ and $s, v' :: [\tau]$, if $length\ v' = length\ (get\ s)$, then $put_{\text{comb}}\ s\ v' :: \mathsf{Maybe\ [Maybe}\ \tau] \equiv \mathsf{Just}\ (map\ \mathsf{Just}\ s')$ for some $s' :: [\tau]$.*

As mentioned above, put_{comb} uses an 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. It is usually more convenient to instead use a default value for such positions, so we define a function $dput$ as follows:[4]

$$\begin{aligned} & dput :: \mathsf{Monad}\ \mu \Rightarrow \alpha \to [\alpha] \to [\alpha] \to \mu\ [\alpha] \\ & dput\ d\ s\ v' = \mathbf{do}\ s' \leftarrow put_{\text{comb}}\ s\ v' \\ & \qquad\qquad\qquad return\ (map\ (maybe\ d\ id)\ s') \end{aligned}$$

[4] Concrete examples of using default values appear in the next section.

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

Corollary 2. *For every type τ and $d :: \tau$,*

$$dput\ d :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is consistent for

$$get :: [\tau] \to [\tau]\,.$$

Corollary 3. *For every type τ and $d :: \tau$,*

$$dput\ d :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is fixed-shape-friendly for get. (Moreover, the default value d is not actually used in $dput\ d\ s\ v'$ if $length\ v' = length\ (get\ s)$.)

It is important to note that no general negative statement like Theorem 4 holds for $dput$ (or for put_{comb}). It all depends on the definition of $sput$!

Namely, if from a given get, we make an $sget$, and find a good $sput$ for it, then $dput$ will also be good for get. This is where we can now plug in the work of Matsuda et al. [2007] as a black box. For functions get that are polymorphic and at the same time satisfy the syntactic restrictions imposed by Matsuda et al.'s technique, we can use that technique for deriving $sput$ from $sget$. Voila, done.

5. Analysis of Examples

We detail the execution of the just introduced combination idea on the two examples considered in Section 2. This leads to some general observations about ways in which, and why, the combined approach improves over both its constituent techniques, and also provides motivation for further extensions we will consider in the two subsequent sections.

Example 1 (continued). We have seen in Sections 2.1 and 2.2 that for get_1 both syntactic and semantic bidirectionalization on their own lead to quite limited updatability. Namely, $put\ s\ v'$ only succeeds if $length\ v' = length\ (get_1\ s)$. The same holds for put_{linear} and put_{refac}, of course, as they are only refactorings of the put-function obtained by semantic bidirectionalization.

On the other hand, for the combination of the two techniques, we can proceed as follows. The $sget$ corresponding to get_1, as obtained via a pretty straightforward syntactic transformation, looks as follows:

$$\begin{aligned} & sget :: \mathsf{Nat} \to \mathsf{Nat} \\ & sget\ \mathsf{Z} \qquad\quad = \mathsf{Z} \\ & sget\ (\mathsf{S}\ \mathsf{Z}) \qquad = \mathsf{Z} \\ & sget\ (\mathsf{S}\ (\mathsf{S}\ zs)) = \mathsf{S}\ (sget\ zs) \end{aligned}$$

For it, the syntactic bidirectionalization method of Matsuda et al. [2007] produces the following complement function:

$$\begin{aligned} & \mathbf{data}\ \mathsf{SCompl} = \mathsf{SC}_1 \mid \mathsf{SC}_2 \\ & scompl :: \mathsf{Nat} \to \mathsf{SCompl} \\ & scompl\ \mathsf{Z} \qquad\quad = \mathsf{SC}_1 \\ & scompl\ (\mathsf{S}\ \mathsf{Z}) \qquad = \mathsf{SC}_2 \\ & scompl\ (\mathsf{S}\ (\mathsf{S}\ zs)) = scompl\ zs \end{aligned}$$

Note that the move from $[\alpha]$ to Nat in $get_1 \mapsto sget$ has obviated the need to collect any dropped variables in the complement function. As a consequence, with the help of range analysis, no data constructor is necessary around the recursive call. (That is a crucial optimization embedded in Matsuda et al.'s transformation.) For the two non-recursive equations, different data constructors are needed, because the ranges of the original right-hand sides overlap.

Tupling of $sget$ and $scompl$ leads to:

$$\begin{array}{ll} spaired :: \mathsf{Nat} \to (\mathsf{Nat}, \mathsf{SCompl}) & \\ spaired\ \mathsf{Z} & = (\mathsf{Z}\ \ , \mathsf{SC}_1) \\ spaired\ (\mathsf{S}\ \mathsf{Z}) & = (\mathsf{Z}\ \ , \mathsf{SC}_2) \\ spaired\ (\mathsf{S}\ (\mathsf{S}\ zs)) & = (\mathsf{S}\ v, c) \\ \quad \mathbf{where}\ (v, c) = spaired\ zs & \end{array}$$

Inversion gives:[5]

$$\begin{array}{ll} sinv :: \mathsf{Monad}\ \mu \Rightarrow (\mathsf{Nat}, \mathsf{SCompl}) \to \mu\ \mathsf{Nat} & \\ sinv\ (\mathsf{Z}\ \ , \mathsf{SC}_1) & = return\ \mathsf{Z} \\ sinv\ (\mathsf{Z}\ \ , \mathsf{SC}_2) & = return\ (\mathsf{S}\ \mathsf{Z}) \\ sinv\ (\mathsf{S}\ v, c) & = \mathbf{do}\ zs \leftarrow sinv\ (v, c) \\ & \qquad\quad return\ (\mathsf{S}\ (\mathsf{S}\ zs)) \end{array}$$

and finally,

$$\begin{array}{l} sput :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat} \\ sput\ s\ v' = sinv\ (v', scompl\ s) \end{array}$$

can be fused to:

$$\begin{array}{lll} sput :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat} & & \\ sput\ \mathsf{Z} & \mathsf{Z} & = return\ \mathsf{Z} \\ sput\ (\mathsf{S}\ \mathsf{Z}) & \mathsf{Z} & = return\ (\mathsf{S}\ \mathsf{Z}) \\ sput\ (\mathsf{S}\ (\mathsf{S}\ zs)) & \mathsf{Z} & = sput\ zs\ \mathsf{Z} \\ sput\ s & (\mathsf{S}\ v') & = \mathbf{do}\ zs \leftarrow sput\ s\ v' \\ & & \qquad\quad return\ (\mathsf{S}\ (\mathsf{S}\ zs)) \end{array}$$

The benefit of the combination of syntactic and semantic bidirectionalization can be observed by comparing $dput$ as obtained from the above $sput$-function to the function put from Example 1 in Section 2.1 (which we have seen is equivalent to put, put_{linear}, and put_{refac} as obtained via semantic bidirectionalization). Here are a few representative calls and their results:

s	v'	$put\ s\ v'$	$dput$ `' '` $s\ v'$
`"abcd"`	`"x"`	Nothing	Just `"ax"`
`"abcd"`	`"xy"`	Just `"axcy"`	Just `"axcy"`
`"abcd"`	`"xyz"`	Nothing	Just `"axcy z"`
`"abcd"`	`"xyzv"`	Nothing	Just `"axcy z v"`
`"abcde"`	`"x"`	Nothing	Just `"axc"`
`"abcde"`	`"xy"`	Just `"axcye"`	Just `"axcye"`
`"abcde"`	`"xyz"`	Nothing	Just `"axcyez "`
`"abcde"`	`"xyzv"`	Nothing	Just `"axcyez v "`

Note that when $length\ v' \neq length\ (get_1\ s)$, $dput$ `' '` $s\ v'$ extends, making use of the default value, or shrinks the source list by a number of elements that is a multiple of two (to preserve the remainder modulo two, as fixed via $scompl$). All updates can be successfully handled, in contrast to all the versions of put we have considered for this example before!

As a "lesson" from the above example, we could formulate:

> The move from $[\alpha]$ to Nat can make the get-function considerably simpler. In particular, no data values have to be kept. Here, this has even led (thanks to range analysis) to one constructor in the complement creation becoming superfluous completely, which resulted in perfect updatability.

Example 2 (continued). We have seen in Sections 2.1 and 2.2 that for get_2/get' the updatability achieved by syntactic bidirectionalization is that $put\ s\ v'$ succeeds whenever $length\ v'$ and $length\ (get_2\ s)$ are equal or both greater than zero, while the semantic technique is only successful if $length\ v' = length\ (get_2\ s)$. Let us analyze how the combined technique fares.

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

$$\begin{array}{ll} sget :: \mathsf{Nat} \to \mathsf{Nat} & \\ sget\ \mathsf{Z} & = \mathsf{Z} \\ sget\ (\mathsf{S}\ \mathsf{Z}) & = \mathsf{Z} \\ sget\ (\mathsf{S}\ (\mathsf{S}\ zs)) & = \mathsf{S}\ (sget'\ zs) \end{array}$$

$$\begin{array}{ll} sget' :: \mathsf{Nat} \to \mathsf{Nat} & \\ sget'\ \mathsf{Z} & = \mathsf{Z} \\ sget'\ (\mathsf{S}\ zs) & = \mathsf{S}\ (sget'\ zs) \end{array}$$

Note that regarding the helper function get' 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 two functions $scompl_1/scompl_2$ for $sget$ (with $\mathbf{data}\ \mathsf{SCompl} = \mathsf{SC}_1 \mid \mathsf{SC}_2 \mid \mathsf{SC}_3$) which differ only in their last defining equation:

$$\begin{array}{ll} scompl_? :: \mathsf{Nat} \to \mathsf{SCompl} & \\ scompl_?\ \mathsf{Z} & = \mathsf{SC}_1 \\ scompl_?\ (\mathsf{S}\ \mathsf{Z}) & = \mathsf{SC}_2 \\ scompl_?\ (\mathsf{S}\ (\mathsf{S}\ zs)) & = \mathsf{SC}_? \end{array}$$

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

$$\begin{array}{ll} scompl' :: \mathsf{Nat} \to \mathsf{SCompl} & \\ scompl'\ \mathsf{Z} & = \mathsf{SC}_3 \\ scompl'\ (\mathsf{S}\ zs) & = \mathsf{SC}_3 \end{array}$$

Note that injectivity analysis (of $sget'$) has enabled the omission of recursive calls, and the use of a constant function for $scompl'$. Due to range analysis, we have a choice between SC_1 and SC_2 in the equation $scompl_?\ (\mathsf{S}\ (\mathsf{S}\ zs)) = \ldots$.

Tupling, inversion, and fusion (again not spelled out here in detail) ultimately give:

$$\begin{array}{lll} sput_1 :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat} & & \\ sput_1\ \mathsf{Z} & \mathsf{Z} & = return\ \mathsf{Z} \\ sput_1\ (\mathsf{S}\ \mathsf{Z}) & \mathsf{Z} & = return\ (\mathsf{S}\ \mathsf{Z}) \\ sput_1\ (\mathsf{S}\ (\mathsf{S}\ zs)) & \mathsf{Z} & = return\ \mathsf{Z} \\ sput_1\ \mathsf{Z} & (\mathsf{S}\ v') & = return\ (\mathsf{S}\ (\mathsf{S}\ v')) \\ sput_1\ (\mathsf{S}\ (\mathsf{S}\ zs)) & (\mathsf{S}\ v') & = return\ (\mathsf{S}\ (\mathsf{S}\ v')) \\ sput_1\ _ & _ & = fail\ \texttt{"...."} \end{array}$$

for $scompl_1$, and a variant in which the third and fourth equation become:

$$\begin{array}{lll} sput_2\ (\mathsf{S}\ (\mathsf{S}\ zs)) & \mathsf{Z} & = return\ (\mathsf{S}\ \mathsf{Z}) \\ sput_2\ (\mathsf{S}\ \mathsf{Z}) & (\mathsf{S}\ v') & = return\ (\mathsf{S}\ (\mathsf{S}\ v')) \end{array}$$

for $scompl_2$.

Let us compare the results of combining syntactic and semantic bidirectionalization, i.e. the now two possible $dput$-functions, to the results of either only syntactic or only semantic bidirectionalization, i.e. to put from Example 2 in Section 2.1 and to $put_{\text{linear}} \equiv put_{\text{refac}}$ from Section 3. We call the $dput$-function obtained from $sput_1$ above, $dput_1$, the other one, obtained from $sput_2$, we call $dput_2$. Figure 2 shows a few representative calls and their results.

As a lesson from this example, we could formulate:

> The move from $[\alpha]$ to Nat can lead to injectivity, and hence to considerably simpler (even constant) complement functions. This clearly benefits updatability.

[5] Note that there is no need for a fall-back function equation $sinv\ _ = fail$ `"Update violates complement."`, because in fact the pattern-match is exhaustive. This eventually means that all updates/cases can be dealt with!

		syntactic	semantic	combined	
s	v'	$put\ s\ v'$	$put_{\text{linear}}\ s\ v'$	$dput_1$ ' ' $s\ v'$	$dput_2$ ' ' $s\ v'$
""	""	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"	""	Nothing	Nothing	Just ""	Just "a"
"ab"	"x"	Just "xb"	Just "xb"	Just "xb"	Just "xb"
"ab"	"xy"	Just "xyb"	Nothing	Just "xy "	Just "xy "
"abc"	""	Nothing	Nothing	Just ""	Just "a"
"abc"	"x"	Just "xc"	Nothing	Just "xb"	Just "xb"
"abc"	"xy"	Just "xyc"	Just "xyc"	Just "xyc"	Just "xyc"
"abc"	"xyz"	Just "xyzc"	Nothing	Just "xyz "	Just "xyz "

Figure 2. Comparing different bidirectionalization methods for the get-function from Example 2.

6. Explicit Bias

Through the numbering scheme of our "template sources" via $[0\,..\,l-1]$ for a concrete source of length l, there is a certain bias that manifests itself when an update changes the length of the view. For example, while it is nice that for Example 2, as just seen, we have

$$dput_1\ \text{' '}\ \text{""}\ \texttt{"x"} \equiv \mathsf{Just}\ \texttt{"x "}$$

and

$$dput_1\ \text{' '}\ \text{""}\ \texttt{"xy"} \equiv \mathsf{Just}\ \texttt{"xy "}$$

(in contrast to the completely syntactically obtained put and the completely semantically obtained put_{linear}, which both give Nothing in both cases), it is maybe a bit disappointing that

$$dput_1\ \text{' '}\ \texttt{"ab"}\ \texttt{"xy"} \equiv \mathsf{Just}\ \texttt{"xy "}$$

(instead of Just "xyb"). The reason for this is simple: the use of $[0\,..\,l_s-1]$ and $[0\,..\,fromNat\ \ l'-1]$ in the definition of put_{comb} means that when the updated source becomes shorter than the original source, then it's the elements towards the *rear* of the original source that become discarded; while if the updated source becomes longer, then again positions towards 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

$$\mathbf{let}\ g = \mathsf{IntMap}.\mathit{fromDistinctAscList}\ (\mathit{zip}\ [0\,..\,l_s-1]\ s)$$

and

$$\mathbf{let}\ t = [0\,..\,\mathit{fromNat}\ l'-1]$$

in the definition of put_{comb} by

$$\mathbf{let}\ g = \mathit{fromDistinctList}\ (\mathit{zip}\ (\mathit{reverse}\ [0\,..\,l_s-1])\ s)$$

and

$$\mathbf{let}\ t = \mathit{reverse}\ [0\,..\,\mathit{fromNat}\ l'-1]$$

respectively, then Theorems 5 and 6, and thus Corollaries 2 and 3, continue to hold, but instead of a rear update (insertion/deletion) bias, there is now a front update bias.

For example, Figure 2 (the interesting subset thereof; all other entries remain unchanged) now becomes:

s	v'	$put\ s\ v'$	$dput_1$ ' ' $s\ v'$	$dput_2$ ' ' $s\ v'$
""	"x"	Nothing	Just "x "	Nothing
""	"xy"	Nothing	Just "xy "	Nothing
"a"	"x"	Nothing	Nothing	Just "xa"
"ab"	""	Nothing	Just ""	Just "b"
"ab"	"xy"	Just "xyb"	Just "xyb"	Just "xyb"
"abc"	""	Nothing	Just ""	Just "c"
"abc"	"x"	Just "xc"	Just "xc"	Just "xc"
"abc"	"xyz"	Just "xyzc"	Just "xyzc"	Just "xyzc"

The entries that have changed are shaded above. 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 put_{comb}:[6]

```
type Bias = Int → [Int]
put_bias :: Monad μ ⇒ Bias → [α] → [α] → μ [Maybe α]
put_bias bias s v' =
  do let l_s = length s
     let g = fromDistinctList (zip (bias l_s) s)
     l' ← maybe (fail "...")
                return
                (sput (toNat l_s) (toNat (length v')))
     let t = bias (fromNat l')
     let h = fromDistinctList (zip (get t) v')
     let h' = IntMap.union h g
     return (map (flip IntMap.lookup h') t)
```

as well as:

```
bdput :: Monad μ ⇒ Bias → α → [α] → [α] → μ [α]
bdput bias d s v' = do s' ← put_bias bias s v'
                       return (map (maybe d id) s')
```

The only formal requirement imposed on a proper $bias$ :: Bias, to ensure that analogues of Theorems 5 and 6 and of Corollaries 2 and 3 continue to hold, is that for every $n \geqslant 0$, $bias\ \ n$ should return a list of length exactly n and with no duplicate elements. Then, we in particular obtain the following two corollaries.

[6] No change whatsoever is necessary to $sput$!

Corollary 4. *Let* $bias$ :: Bias *be proper (in the way just described). For every type* τ *and* $d :: \tau$*,*

$$bdput\ bias\ d :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is consistent for

$$get :: [\tau] \to [\tau].$$

Corollary 5. *Let* $bias$ :: Bias *be proper. For every type* τ *and* $d :: \tau$*,*

$$bdput\ bias\ d :: [\tau] \to [\tau] \to \mathsf{Maybe}\ [\tau]$$

is fixed-shape-friendly for get*. (Moreover, the default value* d *is not actually used in* $bdput\ bias\ d\ s\ v'$ *if* $length\ v' = length\ (get\ s)$*.)*

For $bdput$ to behave well in practice, it makes sense to (at least) additionally impose that whenever $n < m$, the elements of the list $bias\ n$ should form a subset of the elements of $bias\ m$. Some good examples are:

$$\begin{aligned}
&rear :: \mathsf{Bias}\\
&rear\ l = [0\,..\,l-1]\\
&front :: \mathsf{Bias}\\
&front\ l = reverse\ [0\,..\,l-1]\\
&middle :: \mathsf{Bias}\\
&middle\ l = [1,3\,..\,l] +\!\!+ (reverse\ [2,4\,..\,l])\\
&borders :: \mathsf{Bias}\\
&borders\ l = (reverse\ [1,3\,..\,l]) +\!\!+ [2,4\,..\,l]
\end{aligned}$$

Some examples for the get-function from Example 1 (with $sput$ as given for this example in Section 5), illustrating the effects of different bias strategies, are given in Figure 3 (on the next page).

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 Figure 4. (When interpreting the results, note that both get_1 `"abcd"` and get_1 `"abcde"` equal `"bd"`.) The simple hints about which bias to apply when reflecting specific updated views back to the source level are quite effective. In practice, which bias to choose could be determined on a case-by-case basis, with decisions being made based on a form of *diff* between the original view and the updated view, or based on information about performed editing operations, or even something more clever. The possibilities are open, since we have exposed the bias strategy explicitly.

7. Extending Applicability

It turns out that the separation of shape and content, through the resultant move from $[\alpha]$ to Nat in the task posed to the syntactic bidirectionalization subsystem, and with the help of some known syntactic program transformations, leads to applicability (and good results) of the combined technique in new situations otherwise out of reach. We illustrate this with two examples.

Example 3. Assume our get-function is as follows, reversing a list:

$$\begin{aligned}
&get_3 :: [\alpha] \to [\alpha]\\
&get_3\ [\,] \qquad = [\,]\\
&get_3\ (x : xs) = get'\ xs\ [x]\\
&get' :: [\alpha] \to [\alpha] \to [\alpha]\\
&get'\ [\,] \qquad ys = ys\\
&get'\ (x : xs)\ ys = get'\ xs\ (x : ys)
\end{aligned}$$

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

Figure 4. More update bias examples for get_1 from Example 1.

Due to the accumulating parameter of get', the technique of Matsuda et al. [2007] cannot be applied. The technique of Voigtländer [2009] can be applied, but fails to permit any shape-changing updates:

s	v'	$put\ s\ v'$
`"abc"`	`"x"`	Nothing
`"abc"`	`"xy"`	Nothing
`"abc"`	`"xyz"`	Just `"zyx"`
`"abc"`	`"xyzv"`	Nothing

Let us try the combined technique. The move from $[\alpha]$ to Nat yields:

$$\begin{aligned}
&sget :: \mathsf{Nat} \to \mathsf{Nat}\\
&sget\ \mathsf{Z} \qquad = \mathsf{Z}\\
&sget\ (\mathsf{S}\ xs) = sget'\ xs\ (\mathsf{S}\ \mathsf{Z})\\
&sget' :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\
&sget'\ \mathsf{Z} \qquad ys = ys\\
&sget'\ (\mathsf{S}\ xs)\ ys = sget'\ xs\ (\mathsf{S}\ ys)
\end{aligned}$$

Still, an accumulating parameter is used, preventing direct application of the technique of Matsuda et al. to this new subproblem. However, it is now possible to apply a semantics-preserving program transformation of Giesl [2000] to transform $sget'$ as follows:

$$\begin{aligned}
&sget' :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\
&sget'\ \mathsf{Z} \qquad ys = ys\\
&sget'\ (\mathsf{S}\ xs)\ ys = \mathsf{S}\ (sget'\ xs\ ys)
\end{aligned}$$

and to subsequently propagate the constant element $(\mathsf{S}\ \mathsf{Z})$ from $sget$ to the now never-changed second parameter of $sget'$, finally yielding:

$$\begin{aligned}
&sget :: \mathsf{Nat} \to \mathsf{Nat}\\
&sget\ \mathsf{Z} \qquad = \mathsf{Z}\\
&sget\ (\mathsf{S}\ xs) = sget'\ xs\\
&sget' :: \mathsf{Nat} \to \mathsf{Nat}\\
&sget'\ \mathsf{Z} \qquad = \mathsf{S}\ \mathsf{Z}\\
&sget'\ (\mathsf{S}\ xs) = \mathsf{S}\ (sget'\ xs)
\end{aligned}$$

s	v'	*bdput rear* ' ' s v'	*bdput front* ' ' s v'	*bdput middle* ' ' s v'	*bdput borders* ' ' s v'
"abcd"	"x"	Just "ax"	Just "cx"	Just "ax"	Just "bx"
"abcd"	"xyz"	Just "axcy z"	Just " xaycz"	Just "ax ycz"	Just " xbydz"
"abcd"	"xyzv"	Just "axcy z v"	Just " x yazcv"	Just "ax y zcv"	Just " xaycz v"
"abcde"	"x"	Just "axc"	Just "cxe"	Just "axe"	Just "bxd"
"abcde"	"xyz"	Just "axcyez "	Just " xaycze"	Just "axcy ze"	Just " xbydz "
"abcde"	"xyzv"	Just "axcyez v "	Just " x yazcve"	Just "axcy z ve"	Just " xayczev "
"abcde"	"xyzvw"	Just "axcyez v w "	Just " x y zavcwe"	Just "axcy z v we"	Just " x ybzdv w "

Figure 3. Comparing different bias strategies for our combined technique on the *get*-function from Example 1.

Now not only has the technique of Matsuda et al. [2007] become applicable, but their injectivity analysis even detects both the above functions to be injective, which leads to the use of constant functions for *scompl* and *scompl'*. Tupling, inversion, and fusion then give an *sput*-function that is equivalent to:

$$\begin{aligned}&\mathit{sput} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Maybe\ Nat}\\ &\mathit{sput}\ s\ v' = \mathit{return}\ v'\end{aligned}$$

which leads to perfect updatability for the combined technique (no matter what kind of bias from the previous section is used):

s	v'	*dput* ' ' s v'
"abc"	"x"	Just "x"
"abc"	"xy"	Just "yx"
"abc"	"xyz"	Just "zyx"
"abc"	"xyzv"	Just "vzyx"

While reversing a list may appear a bit toy, in particular as it does not omit any information when going from the source to the view, so that the bidirectionalization task essentially becomes one of "only" inversion, the important point here is that through the move from $[\alpha]$ to Nat the *get*-function becomes simpler, in general, so that additional benefit can be gained by exploiting readily available syntactic techniques.[7] We further demonstrate this with another example (and another syntactic phenomenon).

Example 4. Assume our *get*-function is as follows, returning the first half of a list:

$$\begin{aligned}&\mathit{get}_4 :: [\alpha] \to [\alpha]\\ &\mathit{get}_4\ [\,] \qquad = [\,]\\ &\mathit{get}_4\ (x : xs) = x : (\mathit{get'}\ xs\ xs)\\ &\mathit{get'} :: [\alpha] \to [\alpha] \to [\alpha]\\ &\mathit{get'}\ xs \quad [\,] \qquad = [\,]\\ &\mathit{get'}\ xs \quad [y] \qquad = [\,]\\ &\mathit{get'}\ (x : xs)\ (y : z : zs) = x : (\mathit{get'}\ xs\ zs)\end{aligned}$$

Since the function definition of get_4 is not syntactically linear, the technique of Matsuda et al. [2007] is not applicable. The technique of Voigtländer [2009] can be applied, and since get_4 is indeed *semantically* linear, even with the strong guarantees from Section 3.1. Of course, shape-changing updates will fail:

s	v'	$\mathit{put}_{\text{linear}}\ s\ v'$
"abc"	"x"	Nothing
"abc"	"xyz"	Nothing

For the combined technique, we again first move from $[\alpha]$ to Nat:

$$\begin{aligned}&\mathit{sget} :: \mathsf{Nat} \to \mathsf{Nat}\\ &\mathit{sget}\ \mathsf{Z} \qquad = \mathsf{Z}\\ &\mathit{sget}\ (\mathsf{S}\ xs) = \mathsf{S}\ (\mathit{sget'}\ xs\ xs)\\ &\mathit{sget'} :: \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}\\ &\mathit{sget'}\ xs \quad \mathsf{Z} \qquad = \mathsf{Z}\\ &\mathit{sget'}\ xs \quad (\mathsf{S}\ \mathsf{Z}) \qquad = \mathsf{Z}\\ &\mathit{sget'}\ (\mathsf{S}\ xs)\ (\mathsf{S}\ (\mathsf{S}\ zs)) = \mathsf{S}\ (\mathit{sget'}\ xs\ zs)\end{aligned}$$

Some straightforward syntactic analysis now shows that, in particular when called with two equal arguments, *sget'* never really needs its first argument (in contrast to the situation with *get'*, where the first argument plays a crucial role for supplying the output list elements). So we can simplify to:

$$\begin{aligned}&\mathit{sget} :: \mathsf{Nat} \to \mathsf{Nat}\\ &\mathit{sget}\ \mathsf{Z} \qquad = \mathsf{Z}\\ &\mathit{sget}\ (\mathsf{S}\ xs) = \mathsf{S}\ (\mathit{sget'}\ xs)\\ &\mathit{sget'} :: \mathsf{Nat} \to \mathsf{Nat}\\ &\mathit{sget'}\ \mathsf{Z} \qquad = \mathsf{Z}\\ &\mathit{sget'}\ (\mathsf{S}\ \mathsf{Z}) \qquad = \mathsf{Z}\\ &\mathit{sget'}\ (\mathsf{S}\ (\mathsf{S}\ zs)) = \mathsf{S}\ (\mathit{sget'}\ zs)\end{aligned}$$

Now *this* is a program to which the technique of Matsuda et al. [2007] can be applied. Doing so, and combining the result with the semantic technique of Voigtländer as described at the end of Section 4, gives very good updatability. An update only fails if either the source or the updated view is empty while the other is not. Of the different kinds of update bias available from Section 6, *middle* and *borders* are particularly appropriate (not surprisingly, on reflection, given the nature of the *get*-function under consideration here):

s	v'	*bdput middle* ...	*bdput borders* ...
""	""	Just ""	Just ""
"abc"	"x"	Just "x"	Just "x"
"abc"	"xy"	Just "xyc"	Just "xyc"
"abc"	"xyz"	Just "xyz c"	Just "xyzc "
"abcd"	"xy"	Just "xycd"	Just "xycd"
"abcd"	"xyzv"	Just "xyzv cd"	Just "xyzvcd "
"abcdefgh"	"xy"	Just "xygh"	Just "xyef"

8. Conclusion

We have developed an approach for combining the bidirectionalization methods of Matsuda et al. [2007] and Voigtländer [2009]. By separating shape from content, we exploit the respective strengths of the two previous methods maximally. The key insight is that when we simplify the problem of explicit bidirectionalization by posing it only on the shape level (going from *get* to *sget*), the existing syntactic technique can give far better results than for the general problem. The existing semantic technique does the rest.

The improvements achieved on the syntactic level (all caused by the fact that no data values have to be kept) can be classified as 1) making the complement smaller, 2) introducing injectivity,

[7] It is also possible to remove the accumulating parameter from the original, list-based *get'*-function in Example 3 using techniques of Giesl [2000] and Giesl et al. [2007], but the resulting program will still not be amenable to the method of Matsuda et al. [2007]. The move from $[\alpha]$ to Nat is really essential to be successful here.

3) enabling additional transformations that may bring programs into the required form in the first place, and 4) permitting non-linear programs to be made linear. We have seen representative examples for all four phenomena (Examples 1–4, in this order), all in the case of lists. We expect to observe the same, even amplified, when considering functions on other data types.

The move from $[\alpha]$ to Nat might appear somewhat ad-hoc, and very specific to lists. However, actually a very general principle is at work here. We could have equivalently replaced $[\alpha]$ by $[()]$, for the unit type $()$.[8] That is indeed a generic way to characterize the shape data type corresponding to a polymorphic data type: replace the polymorphic component α by $()$. It is also a good way to think about implementing the $get \mapsto sget$ step. A prototype of such an implementation (for the special case of lists) exists and has been packaged with the earlier implementation of the syntactic bidirectionalization method as well as with the relevant functions from Sections 4 and 6 of this paper, so that it is really possible to apply our combined bidirectionalization method automatically. The system is available at `http://www.kb.ecei.tohoku.ac.jp/~kztk/b18n-combined/`.

Using the observation about the general principle above, it should be clear that the abstraction/combination ideas in this paper can be applied similarly to other data types than lists. Dealing with type class polymorphism as Voigtländer [2009] does would be a bit more challenging, because a more refined notion of "shape" is needed then. Also, finding good pragmatic bias strategies as in Section 6 would be more complicated (but also interesting) in the case of non-lists.

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. 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, the latter is required to hold whenever $put\ s\ v'$ is defined, and the former if additionally $put\ (put\ s\ v')\ v''$ is indeed defined. The technique of Matsuda et al. [2007] satisfies these two laws, by virtue of being based on the constant-complement approach of Bancilhon and Spyratos [1981]. Although not explicitly proved by Voigtländer [2009], his technique also satisfies these two additional laws. In fact, it can be reformulated via the constant-complement approach as well.[9] So the question is natural whether our combined technique can also be so based, and satisfies PutPut and undoability as well. The answer is No, as invocations like $dput$ `' '` `"abcd"` `"x"` $\equiv$ Just `"ax"` $\equiv$ $dput$ `' '` `"abyd"` `"x"` for Example 1 show. Clearly, there is no way that $dput$ `' '` `"ax"` `"bd"` is both Just `"abcd"` and Just `"abyd"` as undoability would demand; instead: $dput$ `' '` `"ax"` `"bd"` $\equiv$ Just `"ab d"`. (PutPut fails for a similar reason.) Is that bad news? We would argue that 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 [Foster et al. 2007; Gottlob et al. 1988; Keller 1987].

[8] Clearly, disregarding partial values like $\perp$, Nat and $[()]$ are isomorphic.

[9] No formal reference is available for this observation, but slides of a recent talk at the Workshop on Bidirectional Transformation in Architecture-Based Component Composition (`http://www.iai.uni-bonn.de/~jv/bt_in_abc2010-slides.pdf`).

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions.

References

S. Antoy and M. Hanus. Functional logic programming. *Communications of the ACM*, 53(4):74–85, 2010.

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

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

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

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

J.N. Foster, M.B. Greenwald, J.T. Moore, B.C. Pierce, and A. Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Transactions on Programming Languages and Systems*, 29(3):17, 2007.

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

J. Giesl. Context-moving transformations for function verification. In *Logic-Based Program Synthesis and Transformation 1999, Selected Papers*, volume 1817 of *LNCS*, pages 293–312. Springer-Verlag, 2000.

J. Giesl, A. Kühnemann, and J. Voigtländer. Deaccumulation techniques for improving provability. *Journal of Logic and Algebraic Programming*, 71 (2):79–113, 2007.

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

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

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

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

K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalizing programs with duplication through complementary function derivation. *Computer Software*, 26(2):56–75, 2009.

A. Pettorossi. Transformation of programs and use of tupling strategy. In *Informatica, Proceedings*, pages 1–6, 1977.

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

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

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

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

P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.