Citation for published version


DOI

Link to record in KAR

http://kar.kent.ac.uk/47473/

Document Version

Publisher pdf

Copyright & reuse
Content in the Kent Academic Repository is made available for research purposes. Unless otherwise stated all content is protected by copyright and in the absence of an open licence (eg Creative Commons), permissions for further reuse of content should be sought from the publisher, author or other copyright holder.

Versions of research
The version in the Kent Academic Repository may differ from the final published version. Users are advised to check http://kar.kent.ac.uk for the status of the paper. Users should always cite the published version of record.

Enquiries
For any further enquiries regarding the licence status of this document, please contact:
researchsupport@kent.ac.uk

If you believe this document infringes copyright then please contact the KAR admin team with the take-down information provided at http://kar.kent.ac.uk/contact.html
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

[Bidirectionalization](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 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 given a 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, 2007, 2009; Voigtländer 2009].

Automatic bidirectionalization is one approach to obtaining suitable \(get/put\)-pairs, others are domain-specific languages or 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,

\[
p u t \ ( s \ g e t \ s ) \equiv s .
\]

(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,

\[
g e t \ ( p u t s v') \equiv v'.
\]

(2)

These are the standard consistency conditions [Bancilhon and Spyropoulos 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 ≡ 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 Vogtlander [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 recombinining 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.

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 τ1 and τ2 be types. Let functions get :: τ1 → τ2 and put :: τ1 → τ2 → Maybe τ1 be given. We say that put is consistent for get if:

- For every s :: τ1,
  
  \[ \text{put } s \text{ (get } s) \equiv \text{Just } s. \]

- For every s, s' :: τ1 and v :: τ2, if put s v' :: Just s', then
  
  \[ \text{get } s' \equiv v'. \]

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 Spyraos. The basic idea is that for a function

\[ \text{get} :: \tau_1 \rightarrow \tau_2 \]

one finds a function

\[ \text{compl} :: \tau_1 \rightarrow \tau_3 \]

such that the pairing of the two,

\[ \text{paired} :: \tau_1 \rightarrow (\tau_2, \tau_3) \]

\[ \text{paired } s = (\text{get } s, \text{compl } s) \]

is an injective function. Given an inverse \( \text{inv} :: (\tau_2, \tau_3) \rightarrow \tau_1 \) of paired, one obtains that

\[ \text{put} :: \tau_1 \rightarrow \tau_2 \rightarrow \tau_1 \]

\[ \text{put } s \text{ v'} = \text{inv } (\text{v'}, \text{compl } s) \]

makes equations (1) and (2) true.
In reality, asking for a full inverse \( \text{inv} \) of \( \text{paired} \) is too much. The function \( \text{paired} \) may not even be surjective. So one relaxes \( \text{inv} \) to be a partial function, either implicitly as [Matsuda et al. 2007] do, or explicitly in the type. With

\[
\text{inv} :: (\tau_2, \tau_3) \rightarrow \text{Maybe} \ \tau_1
\]

and the requirements that

- for every \( s :: \tau_1 \),
  \[
  \text{inv} (\text{paired} \ s) \equiv \text{Just} \ s,
  \]
  and
- for every \( s' :: \tau_1, v' :: \tau_2 \), and \( c :: \tau_3 \), if \( \text{inv} (v', c) \equiv \text{Just} \ s' \), then
  \[
  \text{paired} \ s' \equiv (v', c),
  \]
we obtain that

\[
\text{put} :: \tau_1 \rightarrow \tau_2 \rightarrow \text{Maybe} \ \tau_1
\]

\[
\text{put} \ s \ v' = \text{inv} (v', \text{compl} \ s)
\]

is consistent for \( \text{get} \) 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 \( \text{compl} \) from \( \text{get} \) in such a way that \( \text{paired} \) is indeed injective. Then instead of the definition for \( \text{paired} \) above they produce one using a tupling transformation [Pettorossi 1977] that avoids the two independent traversals of \( s \) with \( \text{get} \) and \( \text{compl} \). They syntactically invert \( \text{paired} \) to obtain \( \text{inv} \), and subsequently fuse the computations of \( \text{inv} \) and \( \text{compl} \) in the definition of \( \text{put} \), 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 \( \text{put} \)-function, and also enables us to provide informative error messages if desired.

**Example 1.** Assume our \( \text{get} \)-function is as follows, sieving a list to keep only every second element:

\[
\text{get}_1 :: [\alpha] \rightarrow [\alpha]
\]

\[
\text{get}_1 [\ ] = [ ]
\]

\[
\text{get}_1 [x] = [x]
\]

\[
\text{get}_1 (x : y : zs) = y : (\text{get}_1 \ zs)
\]

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:

\[
\text{data} \ \text{Compl} \ \alpha = \text{C}_1 \mid \text{C}_2 \ \alpha \mid \text{C}_3 \ \alpha \ (\text{Compl} \ \alpha)
\]

\[
\text{compl} :: [\alpha] \rightarrow \text{Compl} \ \alpha
\]

\[
\text{compl} [\ ] = \text{C}_1
\]

\[
\text{compl} [x] = \text{C}_2 \ x
\]

\[
\text{compl} (x : y : zs) = \text{C}_3 \ x \ (\text{compl} \ zs)
\]

(Matsuda et al. work in an untyped language, so they have no need to explicitly introduce the data type \( \text{Compl} \), 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 \( \text{compl} \) are that variables dropped when going from left to right in a defining equation of \( \text{get} \) are collected by \( \text{compl} \), and that, where necessary, different data constructors (of same arity/type) are used on the right-hand sides of \( \text{compl} \) to disambiguate between overlapping ranges of right-hand sides of \( \text{get} \). (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:

\[
\text{paired} :: [\alpha] \rightarrow ([\alpha], \text{Compl} \ \alpha)
\]

\[
\text{paired} [\ ] = ([], \text{C}_1)
\]

\[
\text{paired} [x] = ([], \text{C}_2 \ x)
\]

\[
\text{paired} (x : y : zs) = (y : v, \text{C}_3 \ x \ c)
\]

where \((v, c) = \text{paired} \ zs\)

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

\[
\text{inv} :: \text{Monad} \ \mu \Rightarrow ([\alpha], \text{Compl} \ \alpha) \Rightarrow \mu \ [\alpha]
\]

\[
\text{inv} ([], \text{C}_1) = \text{return} [ ]
\]

\[
\text{inv} ([], \text{C}_2 \ x) = \text{return} [x]
\]

\[
\text{inv} (y : v, \text{C}_3 \ x \ c) = \text{do} \ zs \leftarrow \text{inv} (v, c)
\]

\[
\text{return} (x : y : zs)
\]

\[
\text{inv} = \text{fail} "\text{Update violates complement.}"
\]

Finally,

\[
\text{put} :: \text{Monad} \ \mu \Rightarrow [\alpha] \Rightarrow [\alpha] \Rightarrow \mu \ [\alpha]
\]

\[
\text{put} \ s \ v' = \text{inv} (v', \text{compl} \ s)
\]

can be fused to:

\[
\text{put} :: \text{Monad} \ \mu \Rightarrow [\alpha] \Rightarrow [\alpha] \Rightarrow \mu \ [\alpha]
\]

\[
\text{put} [\ ] = \text{return} [ ]
\]

\[
\text{put} [x] = \text{return} [x]
\]

\[
\text{put} (x : y : zs) (y' : v') = \text{do} \ zs' \leftarrow \text{put} \ zs \ v'
\]

\[
\text{return} (x : y' : zs')
\]

\[
\text{put} = \text{fail} "\text{Update violates complement.}"
\]

Note that for this function, \( \text{put} \ s \ v' \) fails if and only if \( \text{length} \ v' \neq \text{length} \ (\text{get}_1 \ s) \). If it succeeds, it mixes the elements of \( s \) and \( v' \) as in, e.g., from\( \text{Just} \ (\text{put} \ [1 \ldots 6] \ [7 \ldots 9]) = [1, 7, 3, 8, 5, 9] \).

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 \( \text{get} \) to \( \text{compl} \) and \( \text{paired} \). It also performs the syntactic inversion from \( \text{paired} \) to \( \text{compl} \) and \( \text{inv} \), though without the explicit monadic error propagation we have used here. It is not always the case as in the above example that \( \text{inv} \) 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 non-deterministic 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 \( \text{put} \) in terms of \( \text{inv} \) and \( \text{compl} \). 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 determination and fusion steps, because the \( \text{put} \)-function thus obtained typically gives a better picture of the achieved updatability.

---


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:

\[
\begin{align*}
\text{get}_1 & : [\alpha] \to [\alpha] \\
\text{get}_2 & [] = [] \\
\text{get}_2 & [x] = [x] \\
\text{get}_2 & (x : y : zs) = x : (\text{get}’ y zs) \\
\text{get}’ & : \alpha \to [\alpha] \\
\text{get}’ & x [] = [] \\
\text{get}’ & x (y : zs) = x : (\text{get}’ y zs)
\end{align*}
\]

Then the syntactic approach produces the following complement function:

\[
\begin{align*}
data \, \text{Compl} \, \alpha & = C_1 \mid C_2 \alpha \mid C_3 \alpha \\
\text{compl}’ : [\alpha] & \to \text{Compl} \, \alpha \\
\text{compl} & [] = C_2 \\
\text{compl} & [x] = C_2 x \\
\text{compl} & (x : y : zs) = \text{compl}’ y zs \\
\text{compl}’ & : \alpha \to [\alpha] \to \text{Compl} \, \alpha \\
\text{compl}’ & x [] = C_3 x \\
\text{compl}’ & x (y : zs) = \text{compl}’ y zs
\end{align*}
\]

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:

\[
\begin{align*}
\text{put} : \text{Monad} \, \mu & \Rightarrow [\alpha] \to [\alpha] \to \mu [\alpha] \\
\text{put} & [] [] = \text{return} [\alpha] \\
\text{put} & [x] [] = \text{return} [x] \\
\text{put} & (x : y : zs) (x’ : v’) = \text{do} (y’, zs’) \leftarrow \text{put}’ y zs v’ \\
& \quad \quad \quad \text{return} (x’ : y’ : zs’) \\
\text{put} & = \text{fail} "\text{Update violates complement."}
\end{align*}
\]

\[
\begin{align*}
\text{put}’ : \text{Monad} \, \mu & \Rightarrow \alpha \to [\alpha] \to [\alpha] \to \mu (\alpha, [\alpha]) \\
\text{put}’ & y [] [] = \text{return} (y, []) \\
\text{put}’ & y (z : zs) [] = \text{return} (z, zs) \\
\text{put}’ & y zs (x’ : v’) = \text{do} (y’, zs’2) \leftarrow \text{put}’ y zs v’ \\
& \quad \quad \quad \text{return} (x’ : y’ : zs’)
\end{align*}
\]

The updatability of these functions is that put s v’ succeeds if and only if length v’ and length (get 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:

\[
\begin{align*}
\text{put}’ : \text{Monad} \, \mu & \Rightarrow \alpha \to [\alpha] \to [\alpha] \to \mu (\alpha, [\alpha]) \\
\text{put}’ & y zs [x] = \text{return} (\text{last} (y : zs), []) \\
\text{put}’ & y zs (x’ : v’) = \text{return} (x’, v’ + \text{last} (y : zs))
\end{align*}
\]

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

\[
\begin{align*}
\text{put} & (x : y : zs) (x’ : [\alpha’]) = \text{return} (x’ : \text{last} (y : zs)) \\
\text{put} & (x : y : zs) (x’ : y’ : v’ = \text{return} (x’ : y’ : v’ + \text{last} (y : zs))
\end{align*}
\]

and thus to:

\[
\begin{align*}
\text{put} & (x : y : zs) (x’ : v’) = \text{return} (x’ : v’ + \text{last} (y : zs))
\end{align*}
\]

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

\[
\text{get} : [\alpha] \to [\alpha]
\]

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

The intuition underlying the method of Vogtlander [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 \(\alpha\). This entails that its behavior does not depend on any concrete list elements, but only on positional information. And this positional information can be obtained explicitly by applying get to ascending lists over integer values. Say get is tail, then every list \([0 \ldots n]\) is mapped to \([1 \ldots n]\), 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 \([0 \ldots n]\) just as well, even to lists over non-integer types, thanks to parametric polymorphism [Reynolds1983, Strachey1967].

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 \([0 \ldots n]\), 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 \([0 \ldots n]\) 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 \([0 \ldots n]\), 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 \(t = [0 \ldots n]\) 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 \([0 \ldots n]\) with their associated values according to \(h’\).

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

- Instead of presenting a higher-order hff-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-
tages occurring as keys in both input maps. This realizes exactly the “precedence of $h$ over $g$” alluded to in the informal exposition above.

$$put :: (\text{Monad } \mu, \text{Eq } \alpha) \Rightarrow [\alpha] \to [\alpha] \to \mu [\alpha]$$

$$do\ \text{let } t = [0..\text{length } s - 1]$$

$$let g = \text{IntMap.fromDistinctAscList } (\text{zip } t\ s)$$

$$h \leftarrow \text{assoc } (get\ t)\ v'$$

$$let h' = \text{IntMap.union } h\ g$$

$$\text{return } (\text{map } (\text{fromJust } o \\text{flip } \text{IntMap.lookup } h')\ t)$$

$$\text{assoc } :: (\text{Monad } \mu, \text{Eq } \alpha) \Rightarrow [\text{Int}] \to [\alpha] \to \mu [\text{IntMap } \alpha]$$

$$\text{assoc } (i : is)\ (b : bs) =$$

$$\text{do } m \leftarrow \text{assoc } is\ bs$$

$$\text{case } \text{IntMap.lookup } i\ m\ of$$

$$\text{Nothing } \to \text{return } (\text{IntMap.insert } i\ b\ m)$$

$$\text{Just } b \to \text{if } b == c$$

$$\text{then } \text{return } m$$

$$\text{else fail } \text{"Update violates equality."}$$

$$\text{assoc } . \ . = \text{fail } \text{"Update changes the length."}$$

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 $\tau$ be a type that is an instance of $\text{Eq}$ in such a way that the definition given for $==$ makes it reflexive, symmetric, and transitive.

- For every $s : [\tau],
  \text{put } (s\ get\ s) :\ \text{Maybe } [\tau] \equiv \text{Just } s$.

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

**Corollary 1.** Let $\tau$ be a type that is an instance of $\text{Eq}$ in a way that the definition given for $==$ agrees with semantic equality. Then

$$\text{put } :: [\tau] \rightarrow [\tau] \rightarrow \text{Maybe } [\tau] \equiv \text{Just } s.$$
Theorem 2. For every type $\tau$,
\[ \text{put}_{\text{linear}} :: [\tau] \rightarrow [\tau] \rightarrow \text{Maybe} \ [\tau] \]
is consistent for
\[ \text{get} :: [\tau] \rightarrow [\tau]. \]

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 $\text{put} :: [\tau] \rightarrow [\tau] \rightarrow \text{Maybe} \ [\tau]$ (for some type $\tau$) is fixed-shape-friendly for get if for every $s, v' :: [\tau]$, if $\text{length} \ v' = \text{length} \ (\text{get} \ s)$, then $\text{put} \ s \ v' \equiv \text{Just} \ s'$ for some $s' :: [\tau]$.

Note that the original $\text{put} :: [\tau] \rightarrow [\tau] \rightarrow \text{Maybe} \ [\tau]$ from Section 2.2 is not in general fixed-shape-friendly for get-functions that are not semantically linear. On the other hand, $\text{put}_{\text{linear}} :: [\tau] \rightarrow [\tau] \rightarrow \text{Maybe} \ [\tau]$ 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 $\tau$,
\[ \text{put}_{\text{linear}} :: [\tau] \rightarrow [\tau] \rightarrow \text{Maybe} \ [\tau] \]
is fixed-shape-friendly for get.

For the proof, we basically just observe that the last defining equation of $\text{assoc}'$ 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 $\text{put}$ from Section 2.2 of course).

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

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

3.2 Decomposition to Expose the Shape Aspect

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

\[
\text{spput}_{\text{naive}} :: \text{Monad} \ \mu \Rightarrow \text{Int} \rightarrow \text{Int} \rightarrow \mu \ \text{Int}
\]

\[
\text{spput}_{\text{naive}} \ l_s \ l_{v'} = \begin{cases} 
\text{length} \ (\text{get} \ [0..l_s - 1]) & \text{if } l_{v'} = \text{length} \ (\text{get} \ s) \\
\text{return} \ l_s & \text{else} \text{ fail } \text{"update changes the length."} 
\end{cases}
\]

Using that function, we then define $\text{spput}_{\text{refac}}$ as follows:

\[
\text{spput}_{\text{refac}} :: \text{Monad} \ \mu \Rightarrow [\alpha] \rightarrow [\alpha] \rightarrow \mu \ [\alpha]
\]

\[
\text{spput}_{\text{refac}} \ s \ v' = \begin{cases} 
\text{do let } l_s = \text{length} \ s \\
\text{let } g = \text{IntMap.fromDistinctAscList} \ (\text{zip} \ [0..l_s - 1] \ s) \\
\text{let } t' = \text{spput}_{\text{naive}} \ l_s \ (\text{length} \ v') \\
\text{let } t = [0..l' - 1] \\
\text{let } h = \text{fromDistinctList} \ (\text{zip} \ (\text{get} \ t) \ v') \\
\text{let } h' = \text{IntMap.union} \ h \ g \\
\text{return} \ (\text{map} \ (\text{fromJust} \ o \ \text{flip} \ \text{IntMap.lookup}) \ h') \ t
\end{cases}
\]

The refactoring consists of:

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

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

Lemma 1. For every type $\tau$ and $s, v' :: [\tau]$, we have
\[ \text{put}_{\text{linear}} \ s \ v' :: \text{Maybe} \ [\tau] \equiv \text{put}_{\text{refac}} \ s \ v' :: \text{Maybe} \ [\tau]. \]

The motivation for our refactoring above is that we make explicit, in $\text{spput}_{\text{naive}}$, 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 $\text{spput}_{\text{naive}}$, we can change that behavior. For example, it is tempting to change the last line of the above definition of $\text{spput}_{\text{naive}}$ to:

\[
\text{else return} \ (\text{head} \ l_{v'} \ | \ l_{v'} \leftarrow [0..] \ l_{v'} :: = \text{length} \ (\text{get} \ [0..l_{v'} - 1]))
\]

That would correspond to a “brute force” search for an appropriate new source shape. A reviewer pointed out that, thanks to semantic linearity of $\text{get}$, it would be sufficient to start the search for $l_{v'}$ 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 $\text{spput}_{\text{naive}}$ 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:

\[
\text{data Nat} = Z \mid S \text{Nat}
\]

\[
\text{toNat} :: \text{Int} \rightarrow \text{Nat}
\]

\[
\text{toNat} \ 0 = Z
\]

\[
\text{toNat} \ n \mid n > 0 = S \ (\text{toNat} \ (n - 1))
\]

\[
\text{fromNat} :: \text{Nat} \rightarrow \text{Int}
\]

\[
\text{fromNat} \ 0 = 0
\]

\[
\text{fromNat} \ (S \ n) = 1 + \text{fromNat} \ n
\]

and then a function $\text{sget}$ as follows:

\[
\text{sget} :: \text{Nat} \rightarrow \text{Nat}
\]

\[
\text{sget} \ l_s = \text{toNat} \ (\text{length} \ (\text{get} \ [0..\text{fromNat} \ l_s - 1]))
\]

The point, later, will be that one can also directly derive a simplified syntactic definition for $\text{sget}$ from a given definition for $\text{get}$. But for the moment, we simply take the above definition.
Next, we assume that some function \(sput\) is given, with the following type:
\[
sput \colon \text{Nat} \to \text{Nat} \to \text{Maybe \text{Nat}},
\]
and that \(sput\) is consistent for \(sget\). Of course,
\[
sput l_s l_{v'} = \text{case } sput_{\text{naive}} \text{ (fromNat } l_s \text{) (fromNat } l_{v'})
\]
of Nothing \to Nothing
\[
\text{Just } l' \to \text{Just (toNat } l')
\]
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 \(put\) as below. There are three differences from \(put_{\text{refac}}\); 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 \(from\text{Just}\) from the last (\(return\text{-}s\)) 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{align*}
\text{put} & \colon \text{Monad } \mu \Rightarrow \alpha \to \alpha \to \mu \text{ [Maybe } \alpha] \\
\text{put} & = \text{do let } l_s = \text{length } s \\
& \quad \quad \text{let } g = \text{IntMap.fromDistinctAscList (zip } [0, \ldots, l_s - 1] s) \\
& \quad \quad \quad \text{\(l' \gets \text{maybe (fail "Could not handle shape change.") \ return \)} (sput \text{ (toNat } l_s) \text{ (toNat (length } v'))) \\
& \quad \quad \text{let } t = [0, \ldots, \text{fromNat } l' - 1] \\
& \quad \quad \text{let } h = \text{fromDistinctAscList (zip } (\text{get } t) v') \\
& \quad \quad \text{let } h' = \text{IntMap.union } h g \\
& \quad \quad \text{return \ (map (\text{flip IntMap.lookup } h') t)}
\end{align*}
\]

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

**Theorem 5.** Let \(\tau\) be a type.

- For every \(s :: [\tau]\),
\[
\text{put}_{\text{comb}} s \ (\text{get } s) :: \text{Maybe [Maybe } \tau\text{]} \equiv \text{Just (map Just } s)\text{.}
\]

- For every \(s, v' :: [\tau] \text{ and } s' :: [\text{Maybe } \tau], \text{ if } \text{put}_{\text{comb}} s \ v' :: \text{Maybe [Maybe } \tau\text{]}, \text{ then } \text{get } s' \equiv \text{map Just } v'.
\]

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

\[
\text{put}_{\text{comb}} (\text{toNat } (\text{length } s)) \equiv \text{toNat } (\text{length } v'),
\]

and thus, by consistency of \(sput\) for \(sget\), inside the \(put_{\text{comb}}\) definition \(l'\) will be successfully assigned the value \(\text{toNat } l_s\), 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 \(\tau\) and \(s, v' :: [\tau], \text{ if } \text{length } v' = \text{length } (\text{get } s)\), then \(\text{put}_{\text{comb}} s \ v' :: \text{Maybe [Maybe } \tau\text{]} \equiv \text{Just (map Just } s')\) for some \(s' :: [\tau]\).

As mentioned above, \(\text{put}_{\text{comb}}\) 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 \(\text{dput}\) as follows:

\[
\begin{align*}
\text{dput} & \colon \text{Monad } \mu \Rightarrow \alpha \to [\alpha] \to [\alpha] \to \mu \text{ [Maybe } \alpha] \\
\text{dput} \ d \ s \ v' & = \text{do } s' \gets \text{put}_{\text{comb}} s \ v' \\
& \quad \quad \text{return \ (map (maybe d id) } s')
\end{align*}
\]

\(\text{Concrete examples of using default values appear in the next section.}\)

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

**Corollary 2.** For every type \(\tau\) and \(d :: [\tau]\),
\[
\text{dput } d :: [\tau] \to [\tau] \to \text{Maybe } [\tau]
\]
is consistent for
\[
\text{get} :: [\tau] \to [\tau].
\]

**Corollary 3.** For every type \(\tau\) and \(d :: [\tau]\),
\[
\text{dput } d :: [\tau] \to [\tau] \to \text{Maybe } [\tau]
\]
is fixed-shape-friendly for \(get\). (Moreover, the default value \(d\) is not actually used in \(dput \ d \ s \ v' \text{ if } \text{length } v' = \text{length } (\text{get } s)\).)

It is important to note that no general negative statement like Theorem 4 holds for \(dput\) (or for \(put_{\text{comb}}\)). 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 \(get\), both syntactic and semantic bidirectionalization on their own lead to quite limited updatability. Namely, \(\text{put} \ s \ v'\) only succeeds if \(\text{length } v' = \text{length } (\text{get } s)\). The same holds for \(\text{put}_{\text{linear}}\) and \(\text{put}_{\text{refac}}\), of course, as they are only refactorings of the \(\text{put}\)-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 \(get\), as obtained via a pretty straightforward syntactic transformation, looks as follows:

\[
\begin{align*}
\text{sget} & :: \text{Nat} \to \text{Nat} \\
\text{sget } Z & = Z \\
\text{sget } (S \ Z) & = S (\text{sget } Z) \\
\end{align*}
\]

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

\[
\begin{align*}
\text{data } S\text{Compl} & = S\text{C1 } S\text{C2} \\
\text{scompl} & :: \text{Nat} \to S\text{Compl} \\
\text{scompl } Z & = S\text{C1} \\
\text{scompl } (S \ Z) & = S\text{C2} \\
\text{scompl } (S \ (S \ Z)) & = \text{scompl } Z \\
\end{align*}
\]

Note that the move from \([\alpha]\) to \(\text{Nat}\) in \(get\) to \(sget\) 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 \textit{sput} and \textit{scompl} leads to:
\[
\text{spaired} :: \text{Nat} \rightarrow (\text{Nat}, \text{SCompl}) \\
\text{spared Z} = (Z, SC_1) \\
\text{spared} (S \ Z) = (Z, SC_2) \\
\text{spared} (S \ (S \ zs)) = (S \ v, c) \\
\text{where} \ (v, c) = \text{spaired} \ zs \\
\text{Inversion gives}\textsuperscript{[7]}
\]
\[
\text{sinv} :: \text{Monad} \ \mu \Rightarrow (\text{Nat}, \text{SCompl}) \rightarrow \mu \text{Nat} \\
\text{sinv} (Z, SC_1) = \text{return} \ Z \\
\text{sinv} (Z, SC_2) = \text{return} (S \ Z) \\
\text{sinv} (S \ v, c) = \text{do} \ zs \leftarrow \text{sinv} (v, c) \\
\text{return} (S \ (S \ zs)) \\
\]
and finally,
\[
\text{sput} :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Maybe} \text{Nat} \\
\text{sput s v'} = \text{sinv} (v', \text{scompl s}) \\
\]

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

<table>
<thead>
<tr>
<th>s</th>
<th>v'</th>
<th>put s v'</th>
<th>\text{dput} s v'</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abcd&quot;</td>
<td>&quot;x&quot;</td>
<td>Nothing</td>
<td>Just &quot;ax&quot;</td>
</tr>
<tr>
<td>&quot;abcd&quot;</td>
<td>&quot;xy&quot;</td>
<td>Just &quot;axy&quot;</td>
<td>Just &quot;axy&quot;</td>
</tr>
<tr>
<td>&quot;abcd&quot;</td>
<td>&quot;xyz&quot;</td>
<td>Nothing</td>
<td>Just &quot;axyz&quot;</td>
</tr>
<tr>
<td>&quot;abcde&quot;</td>
<td>&quot;xyzv&quot;</td>
<td>Nothing</td>
<td>Just &quot;axyz v&quot;</td>
</tr>
<tr>
<td>&quot;abcde&quot;</td>
<td>&quot;x&quot;</td>
<td>Nothing</td>
<td>Just &quot;axcye&quot;</td>
</tr>
<tr>
<td>&quot;abcde&quot;</td>
<td>&quot;xy&quot;</td>
<td>Just &quot;axcye&quot;</td>
<td>Just &quot;axcye&quot;</td>
</tr>
<tr>
<td>&quot;abcde&quot;</td>
<td>&quot;xyzv&quot;</td>
<td>Nothing</td>
<td>Just &quot;axcyez&quot;</td>
</tr>
<tr>
<td>&quot;abcde&quot;</td>
<td>&quot;xyzv&quot;</td>
<td>Nothing</td>
<td>Just &quot;axcyez v&quot;</td>
</tr>
</tbody>
</table>

Note that when \text{length} v' \neq \text{length} (\text{get}_s s), \text{dput} s v' \text{ 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 \textit{scompl}). All updates can be successfully handled, in contrast to all versions of \textit{put} we have considered for this example before!

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

The move from \([\alpha]\) to \text{Nat} can make the \textit{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\textsuperscript{2}(continued). We have seen in Sections 2.1 and 2.2 that for \textit{get}/\textit{get}' the updatability achieved by syntactic bidirectionalization is that \textit{put} s v' succeeds whenever \text{length} v' and \text{length} (\text{get}_s s) are equal or both greater than zero, while the semantic technique is only successful if \text{length} v' = \text{length} (\text{get}_s s). Let us analyze how the combined technique fares.

The move from \([\alpha]\) to \text{Nat} yields:
\[
\text{get} :: \text{Nat} \rightarrow \text{Nat} \\
\text{get Z} = Z \\
\text{get} (S \ Z) = Z \\
\text{get} (S \ (S \ zs)) = S \ (\text{get}' \ zs) \\
\text{get}' :: \text{Nat} \rightarrow \text{Nat} \\
\text{get}' Z = Z \\
\text{get}' (S \ Z) = S \ (\text{get}' \ zs) \\
\]

Note that regarding the helper function \textit{get}' one argument becomes superfluous. Indeed, when moving from \([\alpha]\) to \text{Nat}, there is no role to play anymore for content elements of type \(\alpha\).

The automatic view complement generation of Matsuda et al. \textsuperscript{[2007]} yields either of two functions \textit{scompl}_1/\textit{scompl}_2 for \textit{get} (with data \text{SCompl} = SC_1 \mid SC_2 \mid SC_3) which differ only in their last defining equation:
\[
\text{scompl}_1 :: \text{Nat} \rightarrow \text{SCompl} \\
\text{scompl}_1 Z = SC_1 \\
\text{scompl}_1 (S \ Z) = SC_2 \\
\text{scompl}_1 (S \ (S \ zs)) = SC_3 \\
\]
while for \textit{get}', one obtains the following complement function:
\[
\text{scompl}_2 :: \text{Nat} \rightarrow \text{SCompl} \\
\text{scompl}_2 Z = SC_3 \\
\text{scompl}_2 (S \ zs) = SC_3 \\
\]

Note that injectivity analysis (of \textit{get}') has enabled the omission of recursive calls, and the use of a constant function for \textit{scompl}_2. Due to range analysis, we have a choice between \textit{SC}_1 and \textit{SC}_2 in the equation \textit{scompl}_1 (S \ (S \ zs)) = \ldots. Tupling, inversion, and fusion (again not spelled out here in detail) ultimately give:
\[
\text{sput}_1 :: \text{Nat} \rightarrow \text{Nat} \\
\text{sput}_1 Z = Z \\
\text{sput}_1 (S \ Z) = \text{return} (S \ Z) \\
\text{sput}_1 (S \ (S \ zs)) = \text{return} Z \\
\text{sput}_1 Z (S \ v') = \text{return} (S \ (S \ v')) \\
\text{sput}_1 Z (S \ (S \ zs)) (S \ v') = \text{return} (S \ (S \ v')) \\
\text{sput}_1 _ = _ = \text{return} "..." \\
\]
for \textit{scompl}_1, and a variant in which the third and fourth equation become:
\[
\text{sput}_2 (S \ (S \ zs)) Z = \text{return} (S \ Z) \\
\text{sput}_2 (S \ (S \ zs)) (S \ v') = \text{return} (S \ (S \ v')) \\
\]
for \textit{scompl}_2.

Let us compare the results of combining syntactic and semantic bidirectionalization, i.e. the now two possible \textit{dput}-functions, to the results of either only syntactic or only semantic bidirectionalization, i.e. to \textit{put} from Example \textsuperscript{2} in Section 2.1 and to \textit{putlinear} \(\equiv\) \textit{putrefac} from Section 3. We call the \textit{dput}-function obtained from \textit{sput}_1 above, \textit{dput}_1, the other one, obtained from \textit{sput}_2, we call \textit{dput}_2. Figure \textsuperscript{2} shows a few representative calls and their results.

As a lesson from this example, we could formulate:

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

\textsuperscript{7}Note that there is no need for a fall-back function equation \textit{sinv} _ = \text{fail} "Update violates complement.", because in fact the pattern-match is exhaustive. This eventually means that all updates/cases can be dealt with!
6. Explicit Bias

Through the numbering scheme of our “template sources” via \([0 \ldots l - 1]\) 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
\[
dput_1 \to "x" \equiv \text{Just } "x"
\]
and
\[
dput_1 \to "xy" \equiv \text{Just } "xy"
\]
(in contrast to the completely syntactically obtained \(\text{put}\) and the completely semantically obtained \(\text{putlinear}\), which both give Nothing in both cases), it may be a bit disappointing that
\[
dput_1 \to "ab" "xy" \equiv \text{Just } "xy"
\]
(instead of Just "xyb"). The reason for this is simple: the use of \([0 \ldots l - 1]\) and \([0 \ldots \text{fromNat } l' - 1]\) in the definition of \(\text{putcomb}\) 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 behavior. If we simply replace the lines
\[
\text{let } g = \text{IntMap.fromDistinctAscList } (\text{zip } [0 \ldots l_s - 1] \ s)\]
and
\[
\text{let } t = [0 \ldots \text{fromNat } l' - 1]\]
in the definition of \(\text{putcomb}\) by
\[
\text{let } g = \text{fromDistinctList } (\text{zip } (\text{reverse } [0 \ldots l_s - 1]) \ s)\]
and
\[
\text{let } t = \text{reverse } [0 \ldots \text{fromNat } l' - 1]\]
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:

\[
\begin{array}{c|c|c|c|c}
\text{put } s \ s' & \text{putlinear } s \ s' & \text{dput } s_1 \ s \ s' & \text{dput}_2 \ s \ s' \\
\hline
"" "" & Just "" & Just "" & Just "" & Just "" \\
"" "x" & Nothing & Nothing & Just "x" & Nothing \\
"" "xy" & Nothing & Nothing & Just "xy" & Nothing \\
"a" "" & Just "a" & Just "a" & Just "a" & Just "a" \\
"a" "x" & Nothing & Nothing & Just "x" & Nothing \\
"ab" "" & Nothing & Nothing & Just "ab" & Nothing \\
"ab" "x" & Just "xb" & Just "xb" & Just "xb" & Just "xb" \\
"ab" "xy" & Just "xyb" & Nothing & Just "xy" & Just "xy" \\
"abc" "" & Nothing & Nothing & Just "abc" & Nothing \\
"abc" "x" & Just "xc" & Nothing & Just "xc" & Nothing \\
"abc" "xy" & Just "xyb" & Nothing & Just "xy" & Nothing \\
"abc" "xyz" & Just "xyzc" & Nothing & Just "xyz" & Nothing \\
\end{array}
\]

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 \(\text{putcomb}\):
Corollary 4. Let bias :: Bias be proper (in the way just described). For every type \( \tau \) and \( d :: \tau \),
\[
\text{bdput bias } d :: [\tau] \rightarrow [\tau] \rightarrow \text{Maybe } [\tau]
\]
is consistent for
\[
\text{get :: } [\tau] \rightarrow [\tau].
\]

Corollary 5. Let bias :: Bias be proper. For every type \( \tau \) and \( d :: \tau \),
\[
\text{bdput bias } d :: [\tau] \rightarrow [\tau] \rightarrow \text{Maybe } [\tau]
\]
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 \( (\text{get } s) \).)

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:

- \( \text{rear} :: \text{Bias} \)
  \( \text{rear } l = [0 \ldots l - 1] \)
- \( \text{front} :: \text{Bias} \)
  \( \text{front } l = \text{reverse } [0 \ldots l - 1] \)
- \( \text{middle} :: \text{Bias} \)
  \( \text{middle } l = [1, 3 \ldots ] + (\text{reverse } [2, 4 \ldots ]) \)
- \( \text{borders} :: \text{Bias} \)
  \( \text{borders } l = (\text{reverse } [1, 3 \ldots ]) + [2, 4 \ldots ] \)

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 3 (When interpreting the results, note that both get \( s \) "abcd" and get \( t \), "ababcd" 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:

\[
\begin{align*}
\text{get} & :: [\alpha] \rightarrow [\alpha]  \\
\text{get} & [\ ] = [ ]  \\
\text{get} & (x : xs) = \text{get} \ ' (xs) \ [x]  \\
\text{get}' & :: [\alpha] \rightarrow [\alpha] \rightarrow [\alpha]  \\
\text{get}' & [\ ] \ ys = ys  \\
\text{get}' & (x : xs) \ ys = \text{get} \ ' (xs) \ (x : ys)
\end{align*}
\]

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

\[
\begin{align*}
\text{put} & s \ v' :  \\
"abcd" & \rightarrow \ "x" \rightarrow \text{Nothing}  \\
"abcd" & \rightarrow \ "xyz" \rightarrow \text{Nothing}  \\
"abcd" & \rightarrow \ "xyzv" \rightarrow \text{Nothing}
\end{align*}
\]

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

\[
\begin{align*}
\text{sget} & :: \text{Nat} \rightarrow \text{Nat}  \\
\text{sget} & Z = Z  \\
\text{sget} & (S \ xs) = \text{sget'} \ xs \ (S \ Z)  \\
\text{sget'} & :: \text{Nat} \rightarrow \text{Nat}  \\
\text{sget'} & Z \ ys = \text{ys}  \\
\text{sget'} & (S \ xs) \ ys = \text{sget'} \ xs \ (S \ ys)
\end{align*}
\]

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:

\[
\begin{align*}
\text{sget'} & :: \text{Nat} \rightarrow \text{Nat}  \\
\text{sget'} & Z \ ys = \text{ys}  \\
\text{sget'} & (S \ xs) \ ys = S \ (\text{sget'} \ xs \ ys)
\end{align*}
\]

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

\[
\begin{align*}
\text{sget} & :: \text{Nat} \rightarrow \text{Nat}  \\
\text{sget} & Z = Z  \\
\text{sget} & (S \ xs) = \text{sget'} \ xs  \\
\text{sget'} & :: \text{Nat} \rightarrow \text{Nat}  \\
\text{sget'} & Z = S \ Z  \\
\text{sget'} & (S \ xs) = S \ (\text{sget'} \ xs)
\end{align*}
\]
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:

\[
\text{sput} :: \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Maybe Nat}
\]

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

\[
\begin{array}{ccc}
  s & v' & \text{bdput} \rightarrow \text{s v}' \\
  "abcd" & "x" & \text{Just } "ax" \\
  "abcd" & "xyz" & \text{Just } "axcy z" \\
  "abcd" & "xyzyv" & \text{Just } "axcy z v v w" \\
  "abcd" & "x" & \text{Just } "ax" \\
  "abcd" & "xyz" & \text{Just } "axcy ez" \\
  "abcd" & "xyzyv" & \text{Just } "axcy ez v w" \\
  "abcd" & "xyzyv" & \text{Just } "x y zavcwe" \\
\end{array}
\]

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 itself). So we can simplify to:

\[
\begin{array}{ll}
  \text{sget} :: \text{Nat} \rightarrow \text{Nat} \\
  \text{sget} Z = Z \\
  \text{sget} (S x s) = S (\text{sget} x s) \\
  \text{sget′} :: \text{Nat} \rightarrow \text{Nat} \\
  \text{sget′} Z = Z \\
  \text{sget′} (S Z) = Z \\
  \text{sget′} (S (S Z)) = S (\text{sget} x s) \\
\end{array}
\]

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 itself). So we can simplify to:

\[
\begin{array}{ll}
  \text{sget} :: \text{Nat} \rightarrow \text{Nat} \\
  \text{sget} Z = Z \\
  \text{sget} (S x s) = S (\text{sget} x s) \\
  \text{sget′} :: \text{Nat} \rightarrow \text{Nat} \\
  \text{sget′} Z = Z \\
  \text{sget′} (S Z) = Z \\
  \text{sget′} (S (S Z)) = S (\text{sget} x s) \\
\end{array}
\]

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 3.1, 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 3.1, middle and borders are particularly appropriate (not surprisingly, on reflection, given the nature of the get-function under consideration here):

\[
\begin{array}{ll}
  s & v' \\
  "abc" & "z" \\
  "abc" & "xy" \\
  "abc" & "xyz" \\
  "abc" & "xyzv" \\
\end{array}
\]

Of course, shape-changing updates will fail:

\[
\begin{array}{ll}
  s & v' \\
  "abc" & "x" \\
  "abc" & "xyz" \\
  "abc" & "xyzv" \\
\end{array}
\]

\[
\begin{array}{ll}
  \text{bdput} \rightarrow \text{s v}' \\
  \text{Nothing} \\
  \text{Nothing} \\
\end{array}
\]

For the combined technique, we again first move from [α] to Nat:

\[\text{bdput middle} \rightarrow \text{s v}' \]

\[\text{bdput borders} \rightarrow \text{s v}' \]

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,
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 \([[1]]\), for the unit type \([1]\). That is indeed a generic way to characterize the shape data type corresponding to a polymorphic data type: replace the polymorphic component \(\alpha\) by \(()\). It is also a good way to think about implementing the get \(\rightarrow\) set 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. 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 \(dp\ put\) fail to "\(ab\ cd\" "xy\" \(\equiv\) Just "ax\" \(\equiv\) dp\ put\ "ab\yd\" "x"" for Example 1 show. Clearly, there is no way that \(dp\ put\) "\(ab\ cd\" "xy\"\) is both Just "\(ab\ cd\" and Just "\(ab\yd\" as undoability would demand; instead: \(dp\ put\) "\(ab\ cd\" \(\equiv\) 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.

Acknowledgments

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

References


