Semantic Bidirectionalization Revisited

Meng Wang
Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
wmeng@chalmers.se

Shayan Najd
School of Informatics
University of Edinburgh
Edinburgh, UK
sh.najd@ed.ac.uk

Abstract

A bidirectional transformation is a pair of mappings between source and view data objects, one in each direction. When the view is modified, the source is updated accordingly with respect to some laws. Over the years, a lot of effort has been made to offer better language support for programming such transformations, essentially allowing the programmers to construct one mapping of the pair and have the other automatically generated.

As an alternative to creating specialized new languages, one can try to analyse and transform programs written in general purpose languages, and "bidirectionalize" them. Among others, a technique termed as semantic bidirectionalization [16] stands out in term of user-friendliness. The unidirectional program can be written using arbitrary language constructs, as long as the function is polymorphic and the language constructs respect parametricity. The free theorem that follows from the polymorphic type of the program allows a kind of forensic examination of the transformation, determining its effect without examining its implementation. This is convenient, in the sense that the programmer is not restricted to using a particular syntax; but it does require the transformation to be polymorphic.

In this paper, we revisit the idea of semantic bidirectionalization and reveal the elegant principles behind the current state-of-the-art techniques. Guided by the findings, we derive much simpler implementations that scale easily.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Programming Language*]: Languages Constructs and Features—Data types and structures, Polymorphism; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation

Keywords Bidirectional Transformation, Free Theorem, Haskell, View-Update Problem

1. Introduction

Bidirectionality is a fundamental aspect of computing: transforming data from one format to another, and requiring a transformation in the opposite direction that is in some sense an inverse. The most well-known instance is the *view–update problem* [1] from relational database design: a 'view' represents a kind of virtual database table, computed on the fly from concrete source tables rather than being represented explicitly, and the problem comes when mapping an update of the view back to a 'corresponding' update on the source tables. In the same way, the problem is central to *model transformations*, playing a crucial role in software evolution: having transformed a high-level model into a lower-level implementation, for a variety of reasons one often needs to reverse engineer a revised high-level model from an updated implementation.

By dint of hard effort, one can construct separately the 'forwards transformation' from source to view together with the corresponding 'backwards' transformation. However, this is a significant duplication of work, because the two transformations are closely related; moreover, it is prone to error, because they do really have to correspond in order to avoid bugs; and it introduces a maintenance issue, because changes to one transformation entail matching changes to the other. Therefore, a lot of work has gone into ways to reduce this duplication and the problems it causes; in particular, there has been a recent rise in linguistic (mostly functional) approaches to streamlining bidirectional transformations—this is very much a current problem.

Using terminologies advocated by the *lens* framework [2, 3, 6–8, 10, 11] that traces back to database research: the forwards function is commonly known as get having type $S \to V$, and the backwards one as put having type $S \to V \to S$. The idea is that put, in addition to an updated view, takes the original source as an input, so that get does not have to be bijective to have a backwards semantics. The correctness of the pair of functions is governed by the following *definitional properties* [4, 15].

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

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

Here *consistency* (also known as the PutGet law) roughly corresponds to right-invertibility, basically ensuring that all updates on a view are captured by the updated source, and *acceptability* (also known as the GetPut law) roughly corresponds to left-invertibility, prohibiting changes to the source if no update has been made on the view. Bidirectional transformations satisfying the above two laws are sometimes called *well-behaved* [6]. In addition to these laws, an optional *undoability* property is sometimes introduced:

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

This property states that the result of an update can be undone through the view.

The paradigm of bidirectional programming is about constructing get in a bidirectional language and expecting a corresponding put to be created automatically. Very often, such languages are defined as a collection of combinators, which can be read in two ways: forwards, as ordinary functions from source to view, and back-

PEPM '14, January 20–21, 2014, San Diego, CA, USA.

http://dx.doi.org/10.1145/2543728.2543729

wards, from original source and updated view to updated source. A disadvantage of the combinator-based approach is that transformations have to be encoded in a somewhat inconvenient programming style.

Other than creating special purpose bidirectional languages, an alternative is to mechanically transform existing unidirectional programs to obtain a backwards counterpart, a technique known as *bidirectionalization* [12]. Different flavours of bidirectionalization have been proposed: syntactic [12], semantic [16], and a combination of the two [17, 18]. Syntactic bidirectionalization inspects a get definition written in a somehow restricted syntactic representation and synthesizes a definition for the backwards version. Semantic bidirectionalization on the other hand treats polymorphic get as an opaque semantic object, applying the function independently to a collection of unique identifiers, and the free theorem arising from parametricity states that whatever happens to those identifiers happens in the same way to any other inputs—this information is sufficient to construct the backwards transformation. (We will give more details of the technique in Section 2.) This is convenient, in the sense that the programmer is not confined to a certain (sometime awkward) syntactic representation; but it does require the transformation to be polymorphic.

In the original paper describing semantic bidirectionalization [16], which will be referred as BFF in this paper, the technique developed for fully polymorphic transformations is carefully extended to two instances of constrained polymorphism, namely the Eq and Ord classes in Haskell [14]. The extensions are made in a case-by-case manner, which give rise to a design of having non-compatible infrastructures for individual extensions.

In this paper, we set out to devise a general theory for semantic bidirectionalization (Section 3) that is sufficient to explain the different extensions defined in BFF, and many more that are yet to be defined. This unified theory can be instantiated into general scalable algorithms that work in practise (Section 4). We then discuss extensions and related literature of semantic bidirectionalization (Section 5) before the conclusion (Section 6).

2. The Original Proposal

Following from BFF, we use Haskell with second-rank types. Consider that we are given a polymorphic function $get :: [a] \to [a]$. Parametricity asserts that get can only reorganize, remove and duplicate its input list elements, without inspecting them or creating new ones. Semantic bidirectionalization exploits this fact to conduct a kind of forensic examination of the transformation, determining its effect without examining its implementation. The technique can be implemented as the following higher-order function:

```
bff :: (∀a.[a] → [a]) → (∀a.Eq a ⇒ [a] → [a] → [a])
bff get s v = seq mv (map (lookupL m) is)
  where ms = templ s
        is = map fst ms
        mv = assoc (get is) v
        m  = union mv ms

templ :: [a] → [(Int, a)]
templ as = zip [0..] as

assoc :: Eq a ⇒ [Int] → [a] → [(Int, a)]
assoc is as | length is == length as =
  if and [i =/= j ∨ x == y | (i, x) ← m, (j, y) ← m]
  then m else error "Inconsistent Update!"
  where m = zip is as

union :: [(Int, a)] → [(Int, a)] → [(Int, a)]
union = Data.List.unionBy (λ(i, _) (j, _) → i == j)

lookupL :: [(Int, a)] → Int → a
lookupL ((j, a) : as) i = if i == j then a
                          else lookupL i as
lookupL [] _            =
  error "This shouldn't happen!"
```

The function bff is parameterized on a forwards function, and produces a backwards function that reverses the effect of its function argument. When presented with inputs, the produced backwards function firstly indexes all source elements with identifiers by building a map from indices to values ($templ$). Since get is polymorphic, we can apply it to the indices ($get\ is$) and expect that whatever happens to those indices happens in the same way to any other inputs. Matching the result of $get\ is$ and the updated view v associates the indices with the updated values ($assoc$). If we imagine the indices as locations, $assoc$ effectively reassigns the values in them. Lastly, $union$ merges the original map with the newly created one by given precedence to the latter, so that elements in the source that didn't end up in the view can be accounted for. (The Haskell library function $Data.List.unionBy$ performs set union.) Then the values inside the map can be straightforwardly looked up to construct the updated source.

A number of checks are in place to ensure that the updates to the view do not cause inconsistency: in $assoc$ it is enforced that the updates to the view shall not change the list length, and if a source element appears more than once in the view, it will make sure the multiple occurrences of the same element are updated consistently (that is why the Eq context is needed). A small technical detail here with Haskell is that function $assoc$ must execute strictly to avoid any errors escaping silently due to laziness, which give rises to the use of seq.

For simplicity, we favour the simple list representation for Maps over more complicated but more efficient variants. It is also worth mentioning that the above technique generalizes almost straightforward to arbitrary tree structures through generic programming as shown in [16]

Key to the practicality of this approach is that it extends to non-full polymorphism. For example, we may want to apply bff to $nub :: Eq\ a \Rightarrow [a] \to [a]$ (a function in Haskell Prelude that removes duplicates from a list) [1]:

```
$ bff nub "abba" "aa"
"aaaa"
```

This behaviour is certainly unacceptable as the result of nub `"aaaa"` is different from `"aa"`, breaking the consistency law. Astute readers may have already realised the problem here. Since we have this newly gained ability of using equality in the forwards function, the free theorems derived from fully polymorphic types are no longer applicable; applying get to a list of unique indices can no longer mimic its effect on the actual source.

In BFF, the problem is confronted head on: a separate bidirectionalization system namely bff_{Eq}

```
bff_Eq :: (∀a.Eq a ⇒ [a] → [a]) →
                    (∀a.Eq a ⇒ [a] → [a] → [a])
```

is introduced with a different indexing strategy and additional validity checks in $assoc$ and $union$ to make sure that the view updates do not conflict with invariants dictated by the template. We explain this technique with the above example. With bff_{Eq}, we no longer simplemindedly index the elements with running numbers. In BFF, a state monad carrying around the elements that have already been encountered and an integer denoting the next available index is used to produce the following.

[1] Of course we have to change the type annotation in bff's definition

```
templ_Eq :: Eq a ⇒ [a] → ([Int], [(Int, a)])
templ_Eq s = case runState (go s) ([], 0)
               of (s', (g, _)) → (s', g)
  where go [] = return []
        go (a : as) = do i ← number_Eq a
                         is ← go as
                         return (i : is)
number_Eq :: Eq a ⇒ a → State ([(Int, a)], Int) Int
number_Eq a =
  do (m, i) ← Control.Monad.State.get
     case Prelude.lookup a (map swap m) of
       Just j  → return j
       Nothing →
         do let m' = (i, a) : m
            Control.Monad.State.put (m', i + 1)
            return i
```

The above code is taken from BFF. We do not explain the code in detail since the functions are not used in this paper; but we give an example to illustrate its behaviour.

```
$ templ_Eq "abba"
([0,1,1,0],[(1,'b'),(0,'a')])
```

There are two parts in the result: the indices sequence and their mappings. The indices now reflect equality among the elements and duplications are removed from the mappings. With this, we regain the power of the free theorem: whatever happens to those indices happens in the same way to the elements. This equality correspondence needs to be respected by view updates; in the above, `"aa"` is an invalid update since it assigns equal values to unequal indices, and will be rejected by the new bidirectionalization system (bff_{Eq}). The bidirectional laws have been reinstated, but in a complicated and non-scalable way. The knowledge that the Eq class is about equality comparisons is hardwired into the definition of $templ_{Eq}$. When we want to move to a different class constraint, Ord as shown in BFF, the above designing process has to be repeated, with greater diligence. The indices now need to reflect the relative ordering of the elements, on top of equality (because Eq is a superclass of Ord).

```
$ templ_Ord "mississippi"
([1,0,3,3,0,3,3,0,2,2,0],
 [(1,'m'),(0,'i'),(3,'s'),(2,'p')])
```

As with $templ_{Eq}$, the definition of $templ_{Ord}$ (which is rather complicated and is omitted from a reproduction here) requires human ingenuity and is critically dependent on the knowledge that Ord class is about inequality comparisons, an effort that cannot be easily reused.

The problem of scalability Type classes are a convenient way of expressing constrained polymorphism. However, they are not general enough for many commonly used polymorphic functions. The principled approach of dealing with (non-fully) polymorphism is through higher-order functions (type classes are desugared into higher-order functions through dictionary translation). In Haskell Prelude (the default library that is imported implicitly), we can find the following functions:

```
takeWhile :: (a → Bool) → [a] → [a]
dropWhile :: (a → Bool) → [a] → [a]
span :: (a → Bool) → [a] → ([a], [a])
break :: (a → Bool) → [a] → ([a], [a])
filter :: (a → Bool) → [a] → [a]
```

As a comparison, there are 3 top-level functions in Prelude that use type class Eq and 2 functions use type class Ord. And for those functions that do use type classes, it is by convention that they are companied by a more general form of "By" functions with user-supplied operations other than the overloaded ones. For example,

```
nubBy :: (a → a → Bool) → [a] → [a]
sortBy :: (a → a → Ordering) → [a] → [a]
```

for nub and $sort$ respectively. Despite being polymorphic, all the above higher-order functions are out of reach from the class-based technique of BFF, because there is no meta-knowledge about the properties of the function arguments (such as Eq is about equality) that we can hinge on to create the specialized indices.

In summary, semantic bidirectionalization based on type classes quickly reaches its limit, not only that it is difficult to extend to new classes, but also that it rules out functions that are not expressed by type classes altogether.

3. Specifying Semantic Bidirectionalization

It has become obvious that indexing in the way of constructing $templ_{Eq}$ quickly becomes over complicated, if not impossible. It is time to go back to the drawing board and review the design. In this section, we start by specifying semantic bidirectionalization, which once done hopefully will enable us to make a more informed decision.

Following from the previous section, we use lists for illustration with the understanding that it applies to algebraic datatypes in general. We write $\overline{o}^n$ to represent n repetitions of os. We abuse the notation to write $\overline{a}^n \to Z$ for $a \to ... \to a \to Z$. We avoid type classes and use explicit higher-order functions $getBy :: \forall a.\ (\overline{a}^n \to Z) \to [a] \to [a]$, where we use lower case for polymorphic type variables and upper case for arbitrary monomorphic types. We call the function of type $\overline{a}^n \to Z$ observer function.

An important semantic aspect of bidirectional systems is the notion of equality that is used to decide the propogation of updates. In this paper, we use structural equality (denoted by $\doteq$) to mean that two values have equal contents, and physical equality (denoted by $\equiv$) to mean that two values are duplicates sharing the same origin. We extended the notations naturally to expressions to represent the equalities between their evaluation results. As expected, physical equality is stronger than structural equality: $x \equiv y \Rightarrow x \doteq y$. We assume that structural equality is implemented by the Haskell function $(==)$ (i.e., $(x == y) \doteq True \Leftrightarrow x \doteq y$), but carefully distinguishes them in presentation.

Rewriting semantic bidirectionalization in our explicit style gives rise to the following:

```
bffBy :: (∀a.(ā^n → Z) → [a] → [a]) →
         (ā^n → Z) → (ī^n → Z) → [a] → [a] → [a]
bffBy getBy f g s v = seq m (map (lookupLBy eqI m) is)
  where ms    = templ s
        is    = map fst ms
        mv    = invfg (assocBy eqA eqI (getBy g is) v)
        m     = invfg (unionBy eqI mv ms)
        invfg = invCk f g
        eqA   = ...
        eqI   = ...


assocBy :: (a → a → Bool) → (i → i → Bool) →
           [i] → [a] → [(i, a)]
assocBy eqA eqI is as | length is == length as =
  if and [¬ (i `eqI` j) ∨ (x `eqA` y) | (i, x) ← m,
                                        (j, y) ← m]
  then m else error "Inconsistent Update!"
  where m = zip is as
```

$$
\begin{aligned}
&unionBy :: (i \to i \to Bool) \to \\
&\qquad [(i,a)] \to [(i,a)] \to [(i,a)] \\
&unionBy\ eqI\ xs\ ys = Data.List.unionBy \\
&\qquad (\lambda(i,\_)\ (j,\_) \to i\ `eqI`\ j)\ xs\ ys \\
&lookupLBy :: (i \to i \to Bool) \to [(i,a)] \to i \to a \\
&lookupLBy\ eqI\ ((j,a) : as)\ i = \mathbf{if}\ i\ `eqI`\ j\ \mathbf{then}\ a \\
&\qquad\qquad \mathbf{else}\ lookupLBy\ eqI\ i\ as \\
&lookupLBy\ \_\ []\ \_ = error\ \texttt{"This should not happen!"}
\end{aligned}
$$

In this version, we have made the observer functions and also the equality operators explicit. Note that we do not parameterize the equality operators but have them locally defined (eqA and eqI), because as we will see from the sequel that they sometimes are derived from the observer functions or from each other, instead of being externally supplied. This difference in treatment also helps to distinguish the different roles of the two pairs of functions: the observer functions are required by forwards functions, whereas the equality operators are used to build the bidirectionalization infrastructure, and different choices of equality allow us to explore different bidirectional semantics. We removed the definition of $templ$ since it depends on the index representation, which is yet to be decided. We also delay a discussion of a guard function $invCk$, which is supposed to enforce consistency between indices and elements.

EXAMPLE 1. *Consider a forwards function using user-defined lessthan operator as the observer function: $get = getBy\ (<_A)$. It is bidirectionalized as*

$$bffBy\ getBy\ (<_A)\ (<_I)$$

□

EXAMPLE 2. *Consider a forwards function using user-defined equality operator as the observer function: $get = getBy\ (==_A)$. It is bidirectionalized as*

$$bffBy\ getBy\ (==_A)\ (==_I)$$

□

We continue to use the term fully polymorphic functions in this higher-order setting to mean polymorphic functions that do not have observer functions as argument.

EXAMPLE 3. *A fully polymorphic forwards function is bidirectionalized as*

$$bffBy\ getBy\ (const\ z)\ (const\ z)$$

where z is an arbitary value that is outputted by the constant function. □

Having the constant function in the place of the observer function conveys the message that nothing can be observed about the elements.

3.1 Map Invariant

With this new setup, we begin with specifying $invCk$ by stating a general property that it enforces.

CONDITION 1 (Map Invariant). *A valid map $m :: [(I,X)]$ must satisfy the following property.*

$$\forall(i_1,x_1)\ ...\ (i_n\ ,x_n) \in m.\ f\ x_1\ ...\ x_n \doteq g\ i_1\ ...\ i_n$$

where $f :: \overline{X}^n \to Z$ and $g :: \overline{I}^n \to Z$ are a pair of observer functions.

□

EXAMPLE 4. *Consider a forwards function using user-defined lessthan operator as the observer function: $get = getBy\ (<_A)$. A valid map $m :: [(I,X)]$ needs to satisfy*

$$\forall(i,x),(j,y) \in m.\ x <_A y \doteq i <_I j$$

□

EXAMPLE 5. *Consider a forwards function using user-defined equality operator as the observer function: $get = getBy\ (==_A)$. A valid map $m :: [(I,X)]$ needs to satisfy*

$$\forall(i,x),(j,y) \in m.\ x ==_A y \doteq i ==_I j$$

□

EXAMPLE 6. *The map invariant for a fully polymorphic forwards function is always satisfied because the observer function $const\ z$ always returns the value z.* □

Map invariant is essentially the precondition for the free theorems, which can be traced back to Wadler's original proposal [19]. The finiteness of the map that the elements and indices are drawn from allows us to perform dynamic checking of the condition. Intuitively, applying f and g to every entry pairs in the map builds two tables, which have header rows and columns occupied by the elements or indices, and cells containing values of type Z – the results of applying f or g to the header values; the dimension of the tables is the arity of f and g, and the size of the tables is determined by the size of map m. We call these tables the *observation tables* because they are basically all the information that can be extracted from the polymorphic list elements by the observer functions, and we require the observation table of elements (represented by $f\ \ x_1\ ...\ x_n$) to coincide with it of indices (represented by $g\ \ i_1\ ...\ i_n$), so that we know that whatever observations made to the indices agrees to them to the elements.

For a given source, the observation tables are fixed by the observer functions, with the function names as the names of the tables. The first step of the bidirectionalization algorithm is to come up with indices that entail a matching observation table as the one of the source, a property captured by imposing the map invariant on the source map ms. In steps that follows, the set of indices (is) contained in ms remain constant, whereas the set of elements changes with the view update. Effectively, the observation table of the indices anchors the changeable table of the elements by the imposition of the map invariant. When the view is updated, the effect is encoded into a map mv. Since the set of indices in $getBy\ g\ is$ is a subset of the original indices in is, its observation table consists of a subset of the rows and columns in the original. The map invariant guarantees that the observation table of the updated view map (mv) matches the one of the indices, and does not cause conflict when combined with the original source table (ms). Let's look at an example illustrating the above process.

EXAMPLE 7. *Given a binary predicate f as the observer function, and a source $[x,y,z]$, which give rise to the following observation table:*

f	x	y	z
x	T	T	F
y	T	T	F
z	F	F	T

We consider forwards function $nubBy\ f$. □

Basically, the above encodes the fact that x and y are considered equivalent by f, but not z. So $nubBy\ f\ [x,y,z] \doteq [x,z]$. Assuming through some magic, we come out with indices $[i,j,k]$ and a function g that produces a matching table.

g	i	j	k
i	T	T	F
j	T	T	F
k	F	F	T

Now suppose we change the z in the view to w, and then union the view map mv with the source map ms to incorporate the updates into the original source. The resulting map m gives rise to the updated tables (with the affected row/column highlighted). Since the index table is not affected by the update, the map invariant dictates that the element table remains constant too.

f	x	y	w
x	T	T	F
y	T	T	F
w	F	F	T

g	i	j	k
i	T	T	F
j	T	T	F
k	F	F	T

This is of course only true if we have $\forall (i,x) \in m.\ f\ x\ w \doteq f\ x\ z \wedge f\ w\ x \doteq f\ z\ x$, which guarantees that the change from z to w is not observable, and thus can be safely accepted.

This strategy of checking map invariant is done by function $invCk$, which is implemented as the following.

$$
\begin{array}{l}
invCk :: (\overline{a}^n \to Z) \to (\overline{I}^n \to Z) \to Map\ a \to Map\ a \\
invCk\ f\ g\ m = \\
\quad \textbf{if}\ and\ [g\ \overline{i}^n == f\ \overline{x}^n \mid \overline{(i,x) \leftarrow m}^n] \\
\quad \textbf{then}\ m\ \textbf{else}\ error\ \texttt{"Invariant Broken!"}
\end{array}
$$

We use a list comprehension to enumerate all combinations of mappings and check whether the observation-table entries are kept consistent.

This framework based on observer functions is general: it models the separate cases discussed in BFF uniformly. More importantly, we observe that little restriction on indices other than their resemblance to the elements is actually needed, suggesting that we can employ a trivial indexing strategy of copying the elements: $templ$ duplicates the source with one copy as constant indices and the other as updatable elements, and the observer function for the indices is the same as the one for the elements.[2]

$$
\begin{array}{l}
templ :: [a] \to [(a,a)] \\
templ\ xs = zip\ xs\ xs
\end{array}
$$

And consequently, the pair of observer functions and the pair of equality operations on indices and elements coincide, and the definition of $bffBy$ can be simplified to the following.

$$
\begin{array}{lll}
bffBy\ getBy\ f\ s\ v = seq\ m\ (map\ (lookupLBy\ eqI\ m)\ is) \\
\quad \textbf{where}\ ms & = templ\ s \\
\qquad is & = map\ fst\ ms \\
\qquad mv & = invfg\ (assocBy\ eqA\ eqI\ (getBy\ f\ is)\ v) \\
\qquad m & = invfg\ (unionBy\ eqI\ mv\ ms) \\
\qquad invfg & = invCk\ f\ f \\
\qquad eqA & = ... \\
\qquad eqI & = eqA
\end{array}
$$

3.2 Observable Equivalence

Now the only missing part is a notion of equality for comparison between elements and indices. The equality is needed in $unionBy$ to remove duplicates, and the same equality is used in looking up from the map in constructing the updated source. Moreover, in $assocBy$ we need equalities for both indices and elements to guarantee consistent updates to duplicated values.

This choice of equality decides the precise semantics of the backwards transformations, which will be the topic of Section 4. In this section, we aim for maximum generality and use an observation-based equivalence, which is weaker than equality and is compatible with the observation-based map invariant.

DEFINITION 2 (Observable Equivalence (Unary)). *Let $f :: X \to Z$ be an observer function, and let $d :: [X]$ be a subset of f's domain. We say two values $x \in d$ and $y \in d$ are observable equivalent with respect to f under element set d denoted by $x \sim_{f;d} y$ if*

$$f\ x \doteq f\ y$$

□

DEFINITION 3 (Observable Equivalence (Binary)). *Let $f :: X \to Z$ be an observer function, and let $d :: [X]$ be a subset of f's domain. We say two values $x \in d$ and $y \in d$ are observable equivalent with respect to f under element set d denoted by $x \sim_{f;d} y$ if*

$$\forall z \in d.\ f\ z\ x \doteq f\ z\ y \wedge f\ x\ z \doteq f\ y\ z$$

□

The above definition generalises to arbitrary arities in a straightforward way. We sometimes omit the domain d, and even the observer function, when they are clear from the context, or irrelevant. Observable equivalence is by definition a weaker notion of equality and is reflexive, symmetric, and transitive.

EXAMPLE 8. *Equality is a special instance of observable equivalence*

$$\forall d.x \sim_{(==);d} y \Rightarrow x \doteq y$$

□

For a given observer function, observable equivalences are ordered according to the element sets they are under.

COROLLARY 4. *Given two element sets $c \subseteq d$, we have*

$$\forall x, y.\ x \sim_{f;d} y \Rightarrow x \sim_{f;c} y$$

for all f. □

From the definition of map invariant, we can easily derive the following correspondence of observable equivalences.

COROLLARY 5. *Let $f :: \overline{X}^n \to Z$, and $g :: \overline{I}^n \to Z$ be observer functions, and let $m :: [(I,X)]$ be a finite map satisfying the map invariant. We have*

$$\forall (i,x),(j,y) \in m.\ x \sim_{f;d} y \Leftrightarrow i \sim_{g;di} j$$

where $(di, d) = unzip\ m$. □

We can now look back at Example 7 above and see observable equivalence at work. The missing definition of eqA can be filled in with a definition of observable equivalence for binary observer functions.

$$
\begin{array}{l}
eqA\ a\ b = and\ [(f\ x\ a == f\ x\ b) \wedge (f\ a\ x == f\ b\ x) \mid \\
\qquad\qquad\qquad\qquad x \leftarrow s +\!\!+ v]
\end{array}
$$

When we try to union mv which is $[(i,x),(k,w)]$ and ms which is $[(i,x),(j,y),(k,z)]$, we have $i \sim_{g;di} j$ where $di = \{i,j,k\}$. As a result, the map m now has only two entries, instead of three, $[(i,x),(k,w)]$. When looked up, the map produces the updated source $[x,x,w]$. Some may worry that the result is different from

[2] Readers familiar with BFF may notice that we do not model exactly the same semantics as the original for fully polymorphic forwards functions. We will address this issue in Section 4.

the seemingly only correct one $[x, y, w]$. But in this observation-based setting, x and y are not differentiable. (We will come back to this point later in Section 4.) This is the reason that the absence of the entry (j, y) is unimportant; the above corollary guarantees that if $i \sim_{g;di} j$, observation-wise it does not matter whether they are mapped to x or y, or even any other values that are equivalent to the two.

As another example, let's consider a variant of Example 7. If we update the view $[x, z]$ to $[x, y]$, then the union of mv which is $[(i, x), (k, y)]$ and ms which is $[(i, x), (j, y), (k, z)]$ produces m which is $[(i, x), (k, y)]$. This will be rejected by the map invariant check because $f\ x\ y$ is not equal to $g\ i\ k$. Indeed, if we do not reject this update, $map\ (lookup\ m)\ is$ would result $[x, x, y]$ and $get\ [x, x, y]$ would be $[x]$, which breaks the consistency law.

We are now ready to state the bidirectional laws in this new setting.

THEOREM 6 (Consistency). *Let $get = getBy\ f$ and $put = bffBy\ getBy\ f$. Assuming successful executions, we have*

$$get\ (put\ s\ v) \sim_{f;s \# v} v$$

PROOF SKECH.

$$\begin{array}{ll} & get\ (put\ s\ v) \\ \doteq & \{\text{ definition of } bffBy \} \\ & get\ (map\ (lookupLBy\ eqI\ m)\ is) \\ \doteq & \{\text{ Corollary 5 and free theorem}\} \\ & map\ (lookupLBy\ eqI\ m)\ (get\ is) \\ \doteq & \{\text{ all indices in } get\ is \text{ is in } mv \text{ and } mv \text{ is a prefix of } m\} \\ & map\ (lookupLBy\ eqI\ mv)\ (get\ is) \\ \sim & \{\text{ Corollary 5}\} \\ & v \end{array}$$

□

The proof is based on the following free theorem, and Corollary 5 together with the definition of $lookupLBy$ serve as its precondition. Also from Corollary 5, we know that for entries in mv, equal indices always map to equal elements in the last step above.

LEMMA 7. *Let $getBy :: \forall a.(\overline{a}^n \to Z) \to [a] \to [a]$, let $f :: \overline{A}^n \to Z$ and $g :: \overline{I}^n \to Z$ be observer functions for elements and indices respectively. Let $h :: I \to A$ and $is :: [I]$. We have*

$$\begin{array}{l} \forall \overline{i}^n \in is.g\ i_1\ i_2\ ...\ i_n \doteq f\ (h\ i_1)\ (h\ i_2)\ ...\ (h\ i_n\ ) \Rightarrow \\ \qquad getBy\ f\ (map\ h\ is) \doteq map\ h\ (getBy\ g\ is) \end{array}$$

THEOREM 8 (Acceptability). *Let $get = getBy\ f$ and $put = bffBy\ getBy\ f$. Assuming successful executions, we have*

$$put\ s\ (get\ s) \sim_{f;s \# v} s$$

PROOF SKECH.

$$\begin{array}{ll} & put\ s\ (get\ s) \\ \doteq & \{\text{ definition of } bffBy\} \\ & map\ (lookupLBy\ eqI\ m)\ is \\ \doteq & \{\text{ definition of } m\} \\ & map\ (lookupLBy\ eqI\ (unionBy\ eqI\ mv\ ms))\ is \\ \sim & \{\text{ no update is made in } v\} \\ & map\ (lookupLBy\ eqI\ ms)\ is \\ \sim & \{\text{ definition of } templ \text{ and } lookupLBy\} \\ & s \end{array}$$

□

THEOREM 9 (Undoability). *Let $get = getBy\ f$ and $put = bffBy\ getBy\ f$. Assuming successful executions, we have*

$$put\ (put\ s\ v')\ (get\ s) \sim_{f;s} s$$

PROOF SKECH.
As a shorthand, we write $x \bigcup y$ for $unionBy\ eqI\ x\ y$.
Let

$$\begin{array}{l} ms_1 = templ\ s \\ is_1 = map\ fst\ ms_1 \\ mv_1 = invfg\ (assocBy\ eqA\ eqI\ (getBy\ f\ is_1)\ v') \\ m_1 = invfg\ (mv_1 \bigcup ms_1) \end{array}$$

and let

$$\begin{array}{l} s' = put\ s\ v' \\ v = get\ s \\ ms_2 = templ\ s' \\ is_2 = map\ fst\ ms_2 \\ mv_2 = invfg\ (assocBy\ eqA\ eqI\ (getBy\ f\ is_2)\ v) \\ m_2 = invfg\ (mv_2 \bigcup ms_2) \end{array}$$

Given the map invariant, we know that the observation tables for is_1 and is_2 coincide: the table entries match each other even though the header values may be different. Thus, no matter whether is_1 or is_2 is used as the indices in a backwards execution, the results will not be affected.

Let $ms21$ be a version of ms_1 with the indices replaced by their counter parts in is_2:

$$ms21 = zip\ is_2\ (map\ snd\ ms_1)$$

Similarly,

$$mv21 = zip\ (getBy\ f\ is_2)\ (map\ snd\ mv_1)$$

We then prove the theorem with the following derivation.

$$\begin{array}{ll} & map\ (lookupLBy\ eqI\ m_2)\ is_2 \\ \sim_{f;s' \# v} & \{\text{ definition of } m_2\} \\ & map\ (lookupLBy\ eqI\ (mv_2 \bigcup ms_2))\ is_2 \\ \sim_{f;s' \# v} & \{\text{ definition of } put \text{ and } templ\} \\ & map\ (lookupLBy\ eqI\ (mv_2 \bigcup (mv21 \bigcup ms21)))\ is_2 \\ \doteq & \{\ map\ fst\ mv_2 \doteq map\ fst\ mv21\ \} \\ & map\ (lookupLBy\ eqI\ (mv_2 \bigcup ms21))\ is_2 \\ \sim_{f;s \# v} & \{\text{ Acceptability: } put\ s\ (get\ s) \sim_{f;s \# v} s\} \\ & s \end{array}$$

Since $v \subseteq s \subseteq s' +\!\!+ v$, we conclude the proof. □

A key step above is that the new update mv_2 completely overwrites the previous update $mv21$.

So far we have completed a specification of semantic bidirectionalization. We demonstrate that the various seemingly ad hoc decisions made in BFF are by no means arbitrary. There exists a neat underlying theory of semantic bidirectionalization that explains the handcrafted instances, and guides further development. On the other hand, it is also obvious that the specification developed here is not as intuitive and deterministic as desired. When the observable functions are discerning, such as $(==)$, observable equivalent works perfectly, but less so otherwise, for example the fully polymorphic case.

In the sequel, we will look at instantiating observable equivalent with two commonly-used notions of equality, namely structural equality and physical equality, and how the choices will affect the indexing technique.

4. Equality-Based Bidirectional Transformation

As concluded in the previous section, using observable equivalence in $unionBy$, $assocBy$ and $lookupBy$ is less intuitive. The bidirectional laws hold up only to observable equivalence now, not to the

more common notions of equality. In this section, we will look at ways to improve this situation.

Conceptually, in the algorithm the need for some equality operator (let it be observable equivalence or others) comes from the construction of the updated map by $unionBy$; we need to be able to merge separate entries in the map that have a common index into one, pointing to the most up-to-date element with duplications handled consistently ($assocBy$), and the following lookup straightforwardly makes use of the same equality. Since the correctness of the bidirectional system is guaranteed by observation-based map invariant, the change from observable equivalence to a different notion of equality results in a two-track system: observation-based map invariant checking, and equality-based map union and looking up. Such a two-track system requires some careful plumbing.

Before we start, it is worth emphasising that the discussion in this section is about different semantics of bidirectional transformation when different equality is used in $unionBy$, $assocBy$ and $lookupBy$. It is not related to the different kinds of observer functions that are used to form the forwards functions.

4.1 Structural Equality

Roughly speaking, our notion of structural equality ($\doteq$) means a kind of equality describing two objects with the same contents. Commonly, such equality is implemented in Haskell by `==`, where two values are equal if they have the same structure and the matching components are equal.

DEFINITION 10 (Structure Equality). *For all x and y,*

$$x \doteq y \Leftrightarrow (x == y \doteq True)$$

□

We assume that observer functions are functional, contrasting to relational, with respect to $\doteq$.

ASSUMPTION 11 (Functional). *For all observer function f,*

$$x \doteq y \Rightarrow f\ x \doteq f\ y$$

From the definition of observable equivalence, we derive the following corollary.

COROLLARY 12. *For all observer function f,*

$$x \doteq y \Rightarrow x \sim_f y$$

□

On the other hand, we do not expect the other direction of the implication to hold, which means that the map invariant is no longer a sufficient condition for the bidirectional laws up to equality. Let's look at a forwards function that filters elements of a list.

EXAMPLE 9. *We consider a backwards execution*

$$bffBy\ filter\ (1<)\ [0,2,3]\ [4,5]$$

□

The above execution produces a view map $[(2,4),(3,5)]$. For the particular observer function, we have $2 \sim_f 3$ and $4 \sim_f 5$, and the above map passes the map invariant check. Since first entry $[(2,4)]$ shadows the second, when looked up it produces an updated source $[0,4,4]$. This result is certainly lawful with respect to observation because $filter\ (1<)\ [0,4,4] \sim_f [4,5]$, but not to equality because $filter\ (1<)\ [0,4,4]$ is not equal to $[4,5]$.

This problem arises because in $unionBy$ and $lookupBy$, we merge entries with equal indices, but those indices map to observable equivalent but not necessarily equal elements. If we want to have the laws up to equality, we need a stronger condition that guarantees the merged entries are not differentiable by equality comparison either.

CONDITION 13. *Given a map m, we require*

$$\forall (i,x),(j,y) \in m.\ i \doteq j \Rightarrow x \doteq y$$

□

We impose this condition in addition to the map invariant in our bidirectionalization algorithm. Note that different from the map invariant, the condition does not include the other direction of the implication because it is used to safeguard the merging of map entries, which only operates on the indices, not to serve as the precondition of the free theorem, which demands two-way correspondence of the indices and elements.

$$\begin{aligned}
&bff_S\ getBy\ f\ s\ v = seq\ m\ (map\ (lookupLBy\ eqI\ m)\ is)\\
&\quad \textbf{where}\ ms = templ\ s\\
&\qquad\qquad is = map\ fst\ ms\\
&\qquad\qquad mv = invfg\ (assocBy\ eqA\ eqI\ (getBy\ f\ is)\ v)\\
&\qquad\qquad m = invfg\ (unionBy\ eqI\ mv\ ms)\\
&\qquad\qquad invfg = invCk_S\ f\ f\\
&\qquad\qquad eqA = (==)\\
&\qquad\qquad eqI = eqA\\
&invCk_S :: (\overline{a}^n \to Z) \to (\overline{I}^n \to Z) \to Map\ a \to Map\ a\\
&invCk_S\ f\ g\ m =\\
&\quad \textbf{if}\ and\ [g\ \overline{i}^n == f\ \overline{x}^n \mid \overline{(i,x) \leftarrow m}^n] \wedge\\
&\qquad and\ [i =/= j \vee x == y \mid (i,x) \leftarrow m, (j,y) \leftarrow m]\\
&\quad \textbf{then}\ m\ \textbf{else}\ error\ \texttt{"Invariant Broken!"}
\end{aligned}$$

With the right condition in place, we are able to state a version of the bidirectional laws up to structural equality.

THEOREM 14. *Let $get = getBy\ f$ and $put = bff_S\ getBy\ f$. Assuming successful executions, we have*

Consistency $get\ (put\ s\ v) \doteq v$
Acceptability $put\ s\ (get\ s) \doteq s$
Undoability $put\ (put\ s\ v')\ (get\ s) \doteq s$

□

Compare to the ones up to observable equivalence, the above laws are obviously more natural. Moreover, similar to the map invariant, the additional Condition 13 does not demand anything from the indices more than their correspondence to the elements. So still, we can copy the elements to construct indices and the same observer function for the elements can be used for the indices.

At a glance, we seem to have obtained the best of both worlds: a simple indexing algorithm and strong bidirectional laws. Actually, the compromise is in an aspect of bidirectionalization that is implicit in the laws, namely the "definedness" of the backwards function, which specifies how much the range of the forwards function is covered by the domain of its backwards counterpart. With the additional condition, we inevitably reject more view updates, sometimes arguably unnecessarily. Let's revisit Example 9, but with a different source.

EXAMPLE 10. *We consider a backwards execution*

$$bff_S\ filter\ (1<)\ [0,2,2]\ [4,5]$$

□

The above update will be rejected due to Condition 13: the view map mv which is $[(2,4),(2,5)]$ does not satisfy the condition. But after a second thought, this is not strictly necessary. The two 2s in the source do not come from the same origin, and the updating of them to different values are consistent considering the origins. Our system of indexing elements with themselves certainly will not make this distinction, as structural-equal values are considered identical.

This behaviour is certainly defendable: in many situations, we do want to treat equal values as identical, for example the name of an employee is better kept consistent across different tables in a company database. Yet, at the same time, it is wrong to consider it as the only option. Our investigation so far on one hand discovered that the use of elements as indices has granted us an unmatched generality, on the other hand proven that the use of a different representation of indices in BFF is not completely unnecessary, since it supports a more discerning notion of equality concerning the origins of the elements.

4.2 Physical Equality

We have seen in the previous subsection that the migration from an observation-based system to an equality-based system is modular: given a notion of equality satisfying Assumption 11, we can simply add Condition 13 on top of the map invariant. In this section, we would like to consider physical equality ($\equiv$), which is a kind of equality describing two values with the same origin in the source. As an example the two 2s from Example 10 are no longer equal due to their different origins.

We encoded physical equality by indexing the source elements with a running sequence of integers so that different integers represent different origins. The new definition of $templ$ is straightforward.

$$templ_P :: [a] \to Map\ a$$
$$templ_P\ as = zip\ [0\,..]\ as$$

Physical equality among elements is encoded by the structural equality of the integer indices. So it only makes sense to discuss physical equality between indexed elements.

DEFINITION 15 (Physical Equality). *For all map m,*

$$\forall (i,x),(j,y) \in m.\ i \doteq j \Leftrightarrow x \equiv y$$

Note that this shift to physical equality is only to alter the "definedness" property of the backwards functions, and does not affect the expressiveness of $bffBy$ (in the sense of different instantiations of the observer functions) and the bidirectional laws: we still want the bidirectional laws to hold up to structural equality between elements. Consequently the validity check is the same as it is for structural equality: the map invariant does not change as always, and Condition 13 (reproduced below)

$$\forall (i,x),(j,y) \in m.\ i \doteq j \Rightarrow x \doteq y$$

trivially holds provided $x \equiv y \Rightarrow x \doteq y$. Effectively, the new indexing mechanism $templ_P$ guarantees that map entries with structural equal indices always point to structurally equal elements.

The tricky part here is to come up with an appropriate observer function for the indices. (We can no longer simply use the same observer function for the elements, and we certainly do not want to go back to the old way of BFF, which tunes the indices to suit a given function.) Again, the finiteness of the input domain is essential here: the semantics of the observer function is fully specified by the observation table introduced in Section 3. Let's revisit Example 7 with integers as indices. Given a source and an observer function f on elements, table f is fixed, and table g for indices is simply a copy.

f	x	y	z
x	T	T	F
y	T	T	F
z	F	F	T

g	0	1	2
0	T	T	F
1	T	T	F
2	F	F	T

We also know that during the entire execution of the backwards function, table g never changes. As a result, we can derive a definition of g from this table.

$$\begin{aligned}
bff_P\ getBy\ f\ s\ v &= seq\ m\ (map\ (lookupLBy\ eqI\ m)\ is)\\
\textbf{where}\ ms &= templ_P\ s\\
is &= map\ fst\ ms\\
mv &= \\
&\quad invfg\ (assocBy\ eqA\ eqI\ (getBy\ f\ is)\ v)\\
m &= invfg\ (unionBy\ eqI\ mv\ ms)\\
invfg &= invCk\ f\ g\\
eqA &= (==)\\
eqI &= (==)\\
g\ i_1\ ...\ i_n &= f\ (lookupLBy\ (==)\ ms\ i_1)\ ...\\
&\quad\ \ (lookupLBy\ (==)\ ms\ i_n\ )
\end{aligned}$$

In the above, we make use of the knowledge that the entries in table g and table f are identical, so we can simply apply f to the elements that correspond to the input indices in the source map ms. Both equalities for indices and elements are simply structural equality of respective types.

So we have moved to a different notion of equality, but end up with exactly the same set of bidirectional laws. The point is that there are noticeable semantic differences between the two systems in their handling of structurally equal but physically unequal elements, as we will see next.

4.3 Comparing the Effects of Different Equalities

We have discussed three options for semantic bidirectionalization, based on observable equivalence, structural equality and physical equality respectively. These three equalities form a total order: physical equality implies structural equality and structural equality implies observable equivalence. Yet, this order between equalities does not translate directly to an order of "definedness". On one hand, the system based on physical equality is potentially more "defined" than the one based on structural equality because the former does not require Condition 13 (i.e., the condition trivially holds). The system based on observable equivalence is even more "defined" as the checks in $assocBy$ are more likely to pass if observable equivalence is used. On the other hand, the use of physical equality in $unionBy$ merges less entries than the cases of structural equality and observable equivalence, resulting in a less aggressive "propagation" of the updates, which increases the chances of violating the map invariant. We explore the different scenarios with examples.

EXAMPLE 11. *Consider function sieve that removes every second elements from a list as the forwards function.*

$$sieve :: [a] \to [a]$$
$$sieve\ (x : y : zs) = y : sieve\ zs$$
$$sieve\ \_ = [\,]$$

s	$get\ s$	v	$put\ s\ v$ Obs.	$put\ s\ v$ Struct.	$put\ s\ v$ Phy.
`"ababa"`	`"bb"`	`"bb"`	`"bbbbb"`	`"ababa"`	`"ababa"`
`"ababa"`	`"bb"`	`"Bb"`	`"BBBBB"`	$*$	`"aBaba"`
`"abba"`	`"ba"`	`"Ba"`	`"BBBBB"`	`"aBBa"`	`"aBba"`

□

As we can see from the second row above, the second `'b'` in the view cannot be changed in the structural-equality setting as it is supposed to remain equal to the other `'b'`, whereas the two `'b'`s are considered unequal in the physical-equality setting. From the

third row, we can see that the system based on structural equality propagates the updates to the other `'b'` in the source that does not appear in the view. For the observation-based system, since the forwards function is fully polymorphic, which has a trivial observer function, the result is pretty arbitrary.

EXAMPLE 12. *Consider function* $dropWhile :: (a \to Bool) \to [a] \to [a]$ *that removes elements from a list until a particular predicate cease to hold, and* $dropWhile$ `(<'c')` *as the forwards function.*

s	$get\ s$	v	$put\ s\ v$		
			Obs.	Struct.	Phy.
`"abca"`	`"ca"`	`"ca"`	`"aaca"`	`"abca"`	`"abca"`
`"abca"`	`"ca"`	`"cb"`	`"bbcb"`	`"bbcb"`	`"abcb"`
`"abca"`	`"ca"`	`"cd"`	$*$	$*$	$*$

□

Note that the application of $dropWhile$ to the predicate `(<'c')` instantiates the type variable to $Char$. However, this is not an issue because $dropWhile$ is defined as a polymorphic function. The observation-based system is the most defined among all: when it fails (because of the map invariant), the other two fail too. In this case where the observer function is more discerning, the observation-based system more deterministically produces results that are similar to those of the other two, because the particular instance of observable equivalence is closer to equality.

EXAMPLE 13. *Consider function* $nubBy :: (a \to a \to Bool) \to [a] \to [a]$ *that removes duplicates from a list according to a given equality predicate, and* $nubBy$ $(==)$ *as the forwards function.*

s	$get\ s$	v	$put\ s\ v$		
			Obs.	Struct.	Phy.
`"aa"`	`"a"`	`"a"`	`"aa"`	`"aa"`	`"aa"`
`"aa"`	`"a"`	`"b"`	`"bb"`	`"bb"`	$*$

□

This is another example with a discerning observer function. But since not all the source elements appear in the view, the difference between different equalities used in $unionBy$ becomes evident. As we can see from the second row above, with structural equality $unionBy$ merges $mv = [(a, b)]$ and $ms = [(a, a), (a, a)]$ into $[(a, b)]$ which passes the map invariant check, whereas with physical equality $unionBy$ merges $mv = [(0, b)]$ and $ms = [(0, a), (1, a)]$ into $[(0, b), (1, a)]$ which will be rejected by the map invariant, because the indices 0 and 1 are equivalent whereas the corresponding elements b and a are not.

In conclusion, in terms of "definedness" the structural-equality-based system performs better for discerning observer functions when the map invariant is more dominant, while the physical-equality-based system is better in other cases when Condition 13 is responsible for most rejections. In BFF, the three separate systems are split into two categories using different equalities: the fully-polymorphic case is based on physical equality, whereas the Eq and Ord cases are based on structural equality. The result we obtained here allows us to confirm that the seemingly ad hoc decision made in BFF is actually thoughtful. More importantly, in contrast to BFF, our system can be simply deployed to handle any polymorphic functions of the right type without the need of any extension or adaptation.

5. Discussion and Related Work

In this section, we look at some features of semantic bidirectionalization and discuss how our proposal interact with them.

5.1 Combining with Syntactic Bidirectionalization

In Section 4.3, we have looked at "definedness" as an important property of bidirectional transformations; and semantic bidirectionalization by itself only permits data updates (in the sense of changing the element values), but not shape updates (in the sense of changing the list structure). It is known that this limitation can be lifted by incorporating semantic bidirectionalization with other techniques. For example as shown in [17], on the intersection of the application domain, the combination of syntactic and semantic bidirectionalization outperforms either. Our framework of semantic bidirectionalization preserves this nice property.

Roughly the intuition behind the combined approach [17] is the recognition that the semantic approach to bidirectionalization is very good at handling changes in elements but not shapes, whereas the syntactic approach [12] permits shape updates, and performs better if the elements are out of the way. Thus, the hope is that a combined approach dedicating shape and element updates to the syntactic and semantic approaches respectively will achieve better results. We implement this idea in our framework as the following.

$$
\begin{aligned}
&\mathit{bffBy} :: (\forall a.(\overline{a}^n \to Z) \to [a] \to [a]) \to (\forall a.(\overline{a}^n \to Z) \to \\
&\qquad [a] \to [a] \to [a]) \\
&\mathit{bffBy}\ \mathit{getBy}\ f\ s\ v = \mathit{map}\ (\mathit{lookup}\ m)\ \mathit{is} \\
&\quad \textbf{where}\ \mathit{ms} = \mathit{templ}\ s \\
&\qquad\qquad \mathit{is} = \mathit{map}\ \mathit{fst}\ \mathit{ms} \\
&\qquad\qquad \mathit{sh} = \mathit{sput}\ (\mathit{map}\ \mathit{unit}\ s)\ (\mathit{map}\ \mathit{unit}\ v) \\
&\qquad\qquad \mathit{is}' = [0\,..] \\
&\qquad\qquad \mathit{mv} = \mathit{assoc}\ (\mathit{getBy}\ g\ \mathit{is}')\ v \\
&\qquad\qquad m = \mathit{unionBy}\ f\ \mathit{mv}\ \mathit{ms} \\
&\qquad\qquad g\ = g'\ f\ \mathit{ms} \\
&\mathit{unit} :: a \to () \\
&\mathit{unit}\ \_ = ()
\end{aligned}
$$

Instead of associating $getBy\ g\ is$ with v, which fails if the shape of v is different from $getBy\ f\ s$, we employ the backwards function $sput$ (stands for shape put) from the syntactic approach. Of course, $sput$ may fail at certain inputs, but when it succeeds we obtain a new source shape that matches the update view. We then perform the backwards computation with the new source shape. The key for establishing the correctness of this algorithm is that the elements appearing in the view are correctly indexed in mv, while the rest of s can be more or less arbitrarily positioned since they do not influence the view anyway. This is certainly only true for fully polymorphic forwards functions, which is the only case that is supported in [17] (for this reason we have chosen physical equality in the above code). Though it is clear from the above code fragment that our framework offers good support for such a combined approach, we are not able to go beyond its current applicability. Since sh is only a shape skeleton, it is not clear how a non-trivial observation table for it, which the map invariant is based on, can be built.

5.2 Performance

The bidirectional systems based on the lens framework usually has supra-linear runtime performance. Conceptually, without additional information describing an update, we must reverse-engineer the update later by performing some kind of difference analysis on the original and updated views. Practically, the backwards function typically traverse the original source and the updated view side-by-side so that the two steps of identifying and incorporating changes

are fused into one. Still, even a small change to the view implies a complete re-traversal.

In this paper, we have favoured presentation clarity than performance in the code. It is clear that there are some standard optimizations that we can deploy. For example, some of the steps can be fused to reduce the number of traversals, and a more efficient Map representation than plain list will dramatically reduce the complexity of $lookupBy$ and $unionBy$. For the physical-equality-based system, we can have composite indices, which include both an integer part and an element part, so that no lookup is required in the definition of g'. Different from BFF, our proposal has the additional work of constructing and checking against the observation table, which has a complexity as a polynomial of the source size, depending on the arity of the observer function.

As updates are typically far smaller than the data, the saving gained through improving the processing speed is likely to be dwarfed by the saving through reducing the data that is actually processed. As a result, incremental computation as an optimization of the backwards function has attracted much interest recently [5, 11, 20]. In our proposal, the cost of checking the observation table can be greatly reduced by only checking the columns and rows of the updated elements. Together with the fact that the observation table only need to be created once for arbitrary number of backwards computation, and our simple indexing algorithm is more efficient, we expect our proposal to have complexity similar to BFF.

Additionally semantic bidirectionalization fits particularly well with the incremental computation technique proposed in [20]. The key insight of the proposal in [20] is that if one is able to identify the source segments that are affected by an update, we can simply try to update the segments instead of the whole source which, under the right condition, will dramatically reduce the work required. Assume that we know the segments of the view that are updated; the corresponding source segments can be identified by indexing and tracing the source elements that appear in the view. This is a setup very similar to the one of semantic bidirectionalization, and works best with polymorphic forwards functions. We expect an integration of the two to be particularly promising.

5.3 Constant Complement

It is well known that injective functions are naturally bidirectional. Thus, the core of bidirectionalizing a uni-directional program is to make it injective. One can assume that the source s can be factored into the view v and its (fairly orthogonal) complement c in such a way that the pair (v, c) uniquely determines s. When v is updated, the complement c stays constant; if the updated result (v', c) remains in the range of the forwards transformation, then the straightforward reconstruction serves as a backwards transformation. The constant nature of the complement gives rise to the name constant complement approach.

In any case, a trivial complement is the original source itself, where the pair (v, s) to v requiring changes to s will violate the constant nature of complement s, and the resulting backwards execution will inevitably fail. Note that in this case it is not that the updated view is not producible from any source, but the updated view is not producible from any source with the given complement. Thus, the challenge is to come up with a "small" complement, which is less likely to cause a conflict and admitting more updates. This is the reason that the combination of syntactic and semantic bidirectionalization techniques as discussed earlier on in Section 5.1 outperforms the syntactic approach used alone: the elements appearing in the complements often prevent the complements from being "reduced".

Semantic bidirectionalization makes use of constant complement in an implicit way [9]. For example, for the fully polymorphic case, the shape of the source together with the elements that is dropped by the forwards execution serves as the complement. Our result in this paper shows a more principled approach with the observation table as the constant complement, which applies to all polymorphic functions. This finding is likely to offer a uniform framework for systematic comparison and integration of different bidirectionalization techniques.

5.4 Semantic Bidirectionalization for Monomorphic Functions

In this paper and in BFF, semantic bidirectionalization is about polymorphic forwards functions. This limitation stems from the fact that it is the types, not any other information about the forwards functions, that guarantees correctness. And consequently, we enjoyed the convenience of not having to touch the actual source code of the functions that is bidirectionalized.

If it is the case that the source code is available, and one were ready to get his or her hands dirty to instrument the code, a similar technique as the one proposed in this paper can be used to bidirectionalize certain monomorphic forwards functions [13]. The main idea is that the instrumented code may produce a record of observations made to element values at run-time, and the observation table is constructed only for those observation that actually happened, not for all that are potentially possible. Assuming correct code instrumentation (which is partially guaranteed by the design of the system [13]), correct bidirectionalization can be achieved. As said, the obvious downside is that the forwards functions have to be reimplemented for instrumentation.

6. Conclusion

We have revisited semantic bidirectionalization, and performed an in-depth analysis of this approach to bidirectional programming. To our pleasant surprise, our investigation has shown that the original technique of BFF can be understood in a much more principled way. This new understanding in turn allows us to derive new implementations of semantic bidirectionalization, which are simpler, and yet more general than the original. As a result, our proposal extends the applicability of semantic bidirectionalization to its full potential, without compromising its strength.

Acknowledgments

We would like to thank Kazutaka Matsuda, Janis Voigtländer and members of Functional Programming group at Chalmers for their helpful comments on a preliminary version of the paper. This work has been partly supported by the Swedish Foundation for Strategic Research (SSF), project RAWFP.

References

[1] Francois Bancilhon and Nicolas Spyratos. Update semantics of relational views. *ACM Transactions on Database Systems*, 6(4):557–575, 1981.

[2] Davi M.J. Barbosa, Julien Cretin, J. Nathan Foster, Michael Greenberg, and Benjamin C. Pierce. Matching lenses: alignment and view update. In *International Conference on Functional Programming (ICFP)*, pages 193–204, New York, NY, USA, 2010. ACM.

[3] Aaron Bohannon, J. Nathan Foster, Benjamin C. Pierce, Alexandre Pilkiewicz, and Alan Schmitt. Boomerang: Resourceful lenses for string data. In *Principles of Programming Languages*, pages 407–419, New York, NY, USA, January 2008. ACM.

[4] Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. Bidirectional transformations: A cross-discipline perspective. In *Theory and Practice of Model Transformations*, pages 260–283, Berlin, Heidelberg, 2009. Springer-Verlag.

[5] Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. From state- to delta-based bidirectional model transformations. In *Theory and Practice of Model Transformations*, ICMT'10, pages 61–76, Berlin, Heidelberg, 2010. Springer-Verlag.

[6] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. *ACM Transactions on Programming Languages and Systems*, 29(3), May 2007. Preliminary version in POPL '05.

[7] J. Nathan Foster, Benjamin C. Pierce, and Steve Zdancewic. Updatable security views. In *Computer Security Foundations*, pages 60–74, Washington, DC, USA, 2009. IEEE Computer Society.

[8] J. Nathan Foster, Alexandre Pilkiewicz, and Benjamin C. Pierce. Quotient lenses. In *International Conference on Functional Programming (ICFP)*, pages 383–396, New York, NY, USA, 2008. ACM.

[9] Nate Foster, Kazutaka Matsuda, and Janis Voigtländer. Three complementary approaches to bidirectional programming. In Jeremy Gibbons, editor, *Generic and Indexed Programming*, volume 7470 of *Lecture Notes in Computer Science*, pages 1–46. Springer Berlin Heidelberg, 2012.

[10] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Symmetric lenses. In *Principles of Programming Languages (POPL)*, POPL '11, pages 371–384, New York, NY, USA, 2011. ACM.

[11] Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Edit lenses. In *Principles of Programming Languages (POPL)*, POPL '12, pages 495–508, New York, NY, USA, 2012. ACM.

[12] Kazutaka Matsuda, Zhenjiang Hu, Keisuke Nakano, Makoto Hamana, and Masato Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *International Conference on Functional Programming (ICFP)*, pages 47–58, New York, NY, USA, 2007. ACM.

[13] Kazutaka Matsuda and Meng Wang. Bidirectionalization for free with runtime recording: or, a light-weight approach to the view-update problem. In *Proceedings of the 15th Symposium on Principles and Practice of Declarative Programming*, PPDP '13, pages 297–308, New York, NY, USA, 2013. ACM.

[14] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[15] Perdita Stevens. Bidirectional model transformations in QVT: semantic issues and open questions. *Software and Systems Modeling*, 9:7–20, 2010.

[16] Janis Voigtländer. Bidirectionalization for free! (Pearl). In *Principles of Programming Languages (POPL)*, pages 165–176, New York, NY, USA, 2009. ACM.

[17] Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda, and Meng Wang. Combining syntactic and semantic bidirectionalization. In *International Conference on Functional Programming (ICFP)*, pages 181–192, New York, NY, USA, 2010. ACM.

[18] Janis Voigtländer, Zhenjiang Hu, Kazutaka Matsuda, and Meng Wang. Enhancing semantic bidirectionalization via shape bidirectionalizer plug-ins. *Journal of Functional Programming*, 23:515–551, 9 2013.

[19] Philip Wadler. Theorems for free! In *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM.

[20] Meng Wang, Jeremy Gibbons, and Nicolas Wu. Incremental updates for efficient bidirectional transformations. In *International Conference on Functional Programming (ICFP)*, ICFP '11, pages 392–403, New York, NY, USA, 2011. ACM.