Principles and Practice of Bidirectional Programming in BiGUL

Zhenjiang Hu and Hsiang-Shang Ko

National Institute of Informatics, Japan
{hu,hsiang-shang}@nii.ac.jp

Abstract. Putback-based bidirectional programming allows the programmer to write only one backward transformation, from which the unique corresponding forward transformation is derived for free. A key distinguishing feature of putback-based bidirectional programming is full control over the bidirectional behavior, which is important for specifying intended bidirectional transformations without any ambiguity. In this chapter, we will introduce BiGUL, a simple yet powerful putback-based bidirectional programming language, explaining the underlying principles and showing how various kinds of bidirectional application can be developed in BiGUL.

1 Putback-based bidirectional programming

In this chapter, the kind of bidirectional transformations (BXs) we discuss is *aymmetric lenses* [8], which basically consist of a pair of transformations: a *forward* transformation *get* producing a *view* from a *source*, and a *backward*, or *putback*, transformation *put* which takes a source and a possibly modified view, and reflects the modifications on the view to the source, producing an updated source. These two transformations should be *well-behaved* in the sense that they satisfy the following round-tripping laws:

$$put\ s\ (get\ s) = s \qquad \textsc{GetPut}$$
$$get\ (put\ s\ v) = v \qquad \textsc{PutGet}$$

The GetPut property requires that no change to the view should be reflected as no change to the source, while the PutGet property requires that all changes in the view should be completely reflected to the source so that the changed view can be successfully recovered by applying the forward transformation to the updated source.

The purpose of *bidirectional programming* is to develop well-behaved bidirectional transformations to solve various synchronization problems. A straightforward approach to bidirectional programming is to write two unidirectional transformations. Although this ad hoc solution provides full control over both get and putback transformations, and can be realized using standard programming languages, the programmer needs to show that the two transformations satisfy the well-behavedness laws, and a modification to one of the transformations requires

a redefinition of the other transformation as well as a new well-behavedness proof. To ease and enable maintainable bidirectional programming, it is preferable to write just a single program that can denote both transformations.

Lots of work [2, 3, 8, 9, 11, 14, 15] has been devoted to the *get-based* approach, allowing the programmer to write, mainly, the forward transformation *get*, and deriving a suitable putback transformation. While the get-based approach is friendly, a *get* function will typically not be injective, so there may exist many possible *put* functions that can be combined with it to form a valid BX. This ambiguity of *put* is what makes bidirectional programming challenging and unpredictable in practice. For specific domains where declarative approaches suffice, the get-based approach works fine, but when it comes to problems for which it is essential to precisely control *put* behavior, the get-based approach is inherently awkward: while most get-based languages/systems offer some features for programming *put* behavior, the programmer ends up having to break the *get*-based abstraction and figure out the *put* semantics of their *get* programs in excruciating detail to be able to reliably use these features, largely defeating the purpose of these languages/systems.

The main topic of this chapter is the *putback-based* approach to bidirectional programming. In contrast to the get-based approach, it allows the programmer to write a backward transformation *put* and derives a suitable *get* that can be paired with this *put* to form a bidirectional transformation. Interestingly, while *get* usually loses information when mapping from a source to a view, *put* must preserve information when putting back from the view to the source, according to the PutGet property.

Before explaining how to program *put* in practice, let us briefly review the foundations [5–7], showing that "putback" is the essence of bidirectional programming. We start by defining validity of *put* as follows:

Definition 1 (Validity of *put*). *We say that a put function is* valid *if there exists a get function such that both* GetPut *and* PutGet *are satisfied.*

The first interesting fact is that, for a valid *put*, there exists exactly one *get* that can form a BX with it. This is in sharp contrast to get-based bidirectional programming, where many *put*s may be paired with a *get* to form a BX.

Lemma 1 (Uniqueness of *get*). *Given a put function, there exists at most one get function that forms a well-behaved BX.*

The second interesting fact is that it is possible to check the validity of *put* without mentioning *get*. The following are two important properties of *put*.

- The first, which we call *view determination*, says that the equivalence of updated sources produced by a *put* implies equivalence of views that are put back.

$$\forall\ s, s', v, v'.\ put\ s\ v\ =\ put\ s'\ v'\ \Rightarrow\ v\ =\ v' \qquad \textsc{ViewDetermination}$$

  Note that view determination implies that $put\ s$ is injective (with $s = s'$).

– The second, which we call *source stability*, denotes a slightly stronger notion of surjectivity for every source:

$$\forall\ s.\ \exists\ v.\ put\ s\ v\ =\ s \qquad \textsc{SourceStability}$$

These two properties together provide an equivalent characterization of the validity of *put* [5].

Theorem 1. *A put function is valid if and only if it satisfies* ViewDetermination *and* SourceStability.

Practically, there are few languages supporting putback-based bidirectional programming. This is not without reason: as argued by Foster [7], it is more difficult to construct a framework that can directly support putback-based bidirectional programming.

In the rest of this chapter, we will introduce BiGUL [10] (pronounced "beagle"), a simple yet powerful putback-based bidirectional language, which grew out of some prior putback-based languages [12, 13]. BiGUL is implemented as an embedded language in Haskell, and we will assume that the reader is reasonably familiar with Haskell. After briefly explaining how to install BiGUL in Section 2, we will introduce basic BiGUL programming in Section 3, and see a few more examples about lists in Section 4. We will then move on to the underlying principles in Section 5, explaining the design and implementation of BiGUL in detail. Those readers who are more interested in practical applications or want to see more examples first may safely skip Section 5 (which is rather long) and proceed to the last three sections, which will show how various bidirectional applications can be developed, including list alignment in Section 6, relational database updating in Section 7, and parsing and "reflective" printing in Section 8.

2 Preparation: installing BiGUL

BiGUL is implemented as an embedded domain-specific language in Haskell, and this chapter assumes that the readers have some Haskell background. (If not, see `https://wiki.haskell.org/Learning_Haskell` for a list of resources for learning Haskell; for the Haskell environment, it is recommended to install Haskell Platform at `https://www.haskell.org/platform/`.) BiGUL has been released to Hackage, and the latest version (1.0.1 at the time of writing) can be installed using Cabal in the usual way, by executing the following in the command line:

```
$ cabal update
$ cabal install BiGUL
```

If you want to ensure compatibility with this chapter, you can instead install BiGUL-1.0.1 specifically by executing:

```
$ cabal install BiGUL-1.0.1
```

Now you can easily check whether BiGUL is correctly installed. First, create a simple file called `Test.hs` with the following content for importing BiGUL modules.

```
{-# LANGUAGE FlexibleContexts, TemplateHaskell, TypeFamilies #-}
import Generics.BiGUL
import Generics.BiGUL.Interpreter
import Generics.BiGUL.TH
import Generics.BiGUL.Lib
```

Then load it using GHCi.

```
$ ghci Test.hs
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
[1 of 1] Compiling Main             ( Test.hs, interpreted )
Ok, modules loaded: Main.
*Main>
```

If you see the above message, congratulations on your successful installation.

To make it more convenient to play with the BiGUL code in this chapter, the Haskell source files for Section 3 (`Basic.hs`), Section 4 (`List.hs`), Section 6 (`Alignment.hs`), Section 7 (`Brul.hs`), and Section 8 (`BiYacc.hs`) are provided at:

`https://bitbucket.org/prl_tokyo/bigul/src/master/SSBX16/`

They are also available as electronic supplementary material to the online version of this chapter on SpringerLink. There are some dependencies among the files: `List.hs` imports `Basic.hs`, and `Brul.hs` imports `Alignment.hs`. The imported files should be present in the same directory as the files being loaded.

3 A quick tour of BiGUL

Intuitively, we can think of a bidirectional BiGUL program

$$bx :: BiGUL\ s\ v$$

as describing how to manipulate a state consisting of a source component of type s and a view component of type v; the goal is to embed all information in the view to proper places in the source. For each $bx :: BiGUL\ s\ v$, we can run it forwards by calling *get* and backwards by calling *put*:

$$\begin{aligned} &get\ bx :: s \quad\ \ \rightarrow Maybe\ v \\ &put\ bx :: s \rightarrow v \rightarrow Maybe\ s \end{aligned}$$

Here, *get bx* is a function mapping a source to a view, which can possibly fail: it either returns a successfully computed view wrapped in the *Just* constructor of *Maybe*, or signifies failure by producing the *Nothing* constructor. On the other hand, *put bx* accepts an original source and uses a view to update it to get an updated source (and might fail as well).

In BiGUL, it suffices for the programmer to write the *put* behavior (i.e., how to use a view to update the original source to a new source), and the (unique) *get* behavior is obtained for free. The core of BiGUL consists of a small number of primitives and combinators for constructing well-behaved bidirectional transformations, which we introduce below.

3.1 Skip

The first primitive for writing *put* is

$$Skip :: (s \to v) \to BiGUL\ s\ v$$

The put behavior of *Skip f* keeps the source unchanged, provided that the view is computable from the source by f (while in the get direction, the view is fully computed by applying function f to the source). Consider a simple *put* defined by *Skip square* where

$$\begin{aligned} &square \quad :: Num\ a \Rightarrow a \to a \\ &square\ x = x * x \end{aligned}$$

We can test its put behavior as follows:

```
*Basic> put (Skip square) 10 100
Just 10
```

It first checks if the view 100 is the square of the source 10. If that is the case, the original source is returned. But if the view is changed, say to 250, it should produce *Nothing*:

```
*Basic> put (Skip square) 10 250
Nothing
```

To see why *put* produces *Nothing*, we may use *putTrace* instead of *put* to get more information:

```
*Basic> putTrace (Skip square) 10 250
view not determined by the source
```

Each putback transformation in BiGUL is equipped with a unique *get* for doing forward transformation. We can test the *get* behavior as follows:

```
*Basic> get (Skip square) 5
Just 25
```

In prose: doing the forward transformation of *Skip square* on the source 5 gives the view 25. If *get* fails, we can also use *getTrace* to see more information about the failure, analogous to *putTrace*.

As a simple exercise, can you see what the following *skip1* does?

$$\begin{aligned} &skip1\ ::\ BiGUL\ s\ () \\ &skip1 = Skip\ (const\ ()) \end{aligned}$$

3.2 Replace

The second primitive is

$$Replace :: BiGUL\ s\ s$$

which completely replaces the source with the view. For instance,

```
*Basic> put Replace 1 100
Just 100
```

uses the view 100 to replace the source 1 and gets a new source 100.

3.3 Product

If we want to use a view pair (v_1, v_2) to update a source pair (s_1, s_2), we can write *Prod bx1 bx2* or *bx1* ‘*Prod*‘ *bx2*, a product of two putback transformations *bx1* and *bx2*, to use v_1 to update s_1 with *bx1* and v_2 to s_2 with *bx2*.

$$Prod :: BiGUL\ s_1\ v_1 \rightarrow BiGUL\ s_2\ v_2 \rightarrow BiGUL\ (s_1, s_2)\ (v_1, v_2)$$

For instance, we can use *Prod* to combine *Skip* and *Replace* to put a view pair into a source pair.

```
*Basic> put (skip1 `Prod` Replace) (5,1) ((),100)
Just (5,100)
```

Generally, we can use nested *Prod*s to describe a complicated structural mapping:

```
*Basic> put ((skip1 `Prod` Replace) `Prod` Replace) ((5,1),2)
    (((),100),200)
Just ((5,100),200)
```

3.4 Source/view rearrangement

So far, the source and view have been of the same structure. What if we wish to put a view (v_1, v_2) into a source of a different structure, say $((s_0, s_1), s_2)$, to replace s_1 by v_1 and s_2 by v_2? To do that, we need to rearrange the source and view into the same structure, and BiGUL provides a way of rearranging either the source or view through a “simple” λ-expression e:

$$^{\$}(rearrS\ [\![\, e :: s_1 \rightarrow s_2 \,]\!]) :: BiGUL\ s_2\ v\ \ \rightarrow BiGUL\ s_1\ v$$
$$^{\$}(rearrV\ [\![\, e :: v_1 \rightarrow v_2 \,]\!]) :: BiGUL\ s\ \ v_2 \rightarrow BiGUL\ s\ \ v_1$$

The “simple” λ-expression e should be wrapped inside Template Haskell quasi-quotes $[\![\ldots]\!]$ (written as `[| ... |]` in plain text Haskell); it is then processed and expanded by *rearrS* or *rearrV* to “core” BiGUL code, which is spliced (pasted) into the invocation site by Template Haskell, as instructed by $^{\$}(\ldots)$. By “simple” we mean that there should be no wildcards ‘_’ in the argument pattern, and that the body can only contain the argument variables and constructors, and must mention all the argument variables. We will discuss the details later in Section 5.6. Returning to the problem of putting a pair into a triple, we may define the following putback transformation

$$\begin{aligned} putPairOverNPair\ &:: (Show\ s_0, Show\ s_1, Show\ s_2) \\ &\Rightarrow BiGUL\ ((s_0, s_1), s_2)\ (s_1, s_2) \end{aligned}$$

$$
\begin{aligned}
putPairOverNPair \;=\; &\$(rearrV\ [\![\, \lambda(v_1, v_2) \to (((), v_1), v_2)\,]\!])\$ \\
&(skip1\ `Prod`\ Replace)\ `Prod`\ Replace
\end{aligned}
$$

by first rearranging the view (v_1, v_2) to a triple $(((), v_1), v_2)$ with the same structure as the source, and then using $(skip1\ `Prod`\ Replace)\ `Prod`\ Replace$ to put the arranged view $(((), v_1), v_2)$ into the source $((s_0, s_1), s_2)$. The type context $(Show\ s_0, Show\ s_1, Show\ s_2)$ above is required by BiGUL for printing debugging messages. And note that the two '$' signs in the definition off *putPairOverNPair* have different meanings: the first one marks the beginning of a Template Haskell splice, while the second one is the low-precedence application operator.

The mechanism of source/view rearrangement enables us to process algebraic data structures such as lists and trees, by mapping an algebraic structure to the (nested) pair structure. The following example uses the view to replace the first element of a nonempty source list:

$$
\begin{aligned}
&pHead :: Show\ s \Rightarrow BiGUL\ [s]\ s \\
&pHead = \$(rearrS\ [\![\, \lambda(s : ss) \to (s, ss)\,]\!])\$ \\
&\qquad \$(rearrV\ [\![\, \lambda v \to (v, ())\,]\!])\$ \\
&\qquad\quad Replace\ `Prod`\ skip1
\end{aligned}
$$

It rearranges the source (a nonempty list) to a pair with its head element s and its tail ss, and the view v to a pair $(v, ())$, so that we can use v to replace s and $()$ to keep ss.

```
*Basic> put pHead [1,2,3,4] 100
Just [100,2,3,4]
```

What if we wish to define a general putback transformation that uses the view to replace the ith element of the source list? We can define it recursively as follows:

$$
\begin{aligned}
&pNth \;\; :: Show\ s \Rightarrow Int \to BiGUL\ [s]\ s \\
&pNth\ i = \textbf{if}\ i == 0\ \textbf{then}\ pHead \\
&\qquad\qquad \textbf{else}\ \ \$(rearrS\ [\![\, \lambda(x : xs) \to (x, xs)\,]\!])\$ \\
&\qquad\qquad\qquad \$(rearrV\ [\![\, \lambda v \to ((), v)\,]\!])\$ \\
&\qquad\qquad\qquad\quad skip1\ `Prod`\ pNth\ (i - 1)
\end{aligned}
$$

If i is 0, we simply use *pHead* to update the head element of the source with the view. Otherwise, we do the same arrangements on the view and the source as we did for *pHead*, but then keep the head element unchanged and replace the $(i - 1)$th element of the tail of the source by the view.

```
*Basic> put (pNth 3) [1..10] 100
Just [1,2,3,100,5,6,7,8,9,10]
```

As we know, any putback function in BiGUL is equipped with a *get* function. For *pNth*, we can test its *get* behavior as follows; its corresponding *get* function is actually the familiar index function (!!).

```
*Basic> get (pNth 3) [1..10]
Just 4
```

Both *pHead* and *pNth* contain the programming pattern in which both the source and view are rearranged into a product and then further updates are performed on corresponding components. This is a ubiquitous pattern in BiGUL, for which we provide a more compact syntax:

$$^{\$}(update\ ^{\mathbb{P}}[\![\, sourcePattern \,]\!]\ ^{\mathbb{P}}[\![\, viewPattern \,]\!]\ ^{\mathbb{D}}[\![\, updates \,]\!])$$

The source and view are respectively decomposed using *sourcePattern* and *viewPattern* inside the pattern quasi-quotes $^{\mathbb{P}}[\![\ldots]\!]$ (written as `[p| ... |]` in plain text Haskell), and corresponding elements are updated using the programs provided in the declaration quasi-quote $^{\mathbb{D}}[\![\ldots]\!]$ (`[d| ... |]` in plain text Haskell). For example, we may describe (*skip1* ‘*Prod*‘ *Replace*) ‘*Prod*‘ *Replace* by

$$\begin{aligned}
&testUpdate :: (Show\ a, Show\ b, Show\ c) \Rightarrow BiGUL\ ((a,b),c)\ (((),b),c)\\
&testUpdate = {}^{\$}(update\ ^{\mathbb{P}}[\![\, ((x,y),z) \,]\!]\\
&\qquad\qquad\qquad\qquad\ ^{\mathbb{P}}[\![\, ((x,y),z) \,]\!]\\
&\qquad\qquad\qquad\qquad\ ^{\mathbb{D}}[\![\, x = skip1; y = Replace; z = Replace \,]\!])
\end{aligned}$$

In this concrete example, the three elements of the tuple (in both the source and view) are bound to the variables x, y, and z, and they are sent to the three combinators as arguments in the $^{\mathbb{D}}[\![\ldots]\!]$ part. Note that since *skip1* does nothing on its source but checks if its view is (), we can just match that source element with a wildcard ‘_’ in the source pattern and avoid writing *skip1* in $^{\mathbb{D}}[\![\ldots]\!]$.

$$\begin{aligned}
&testUpdate' :: (Show\ a, Show\ b, Show\ c) \Rightarrow BiGUL\ ((a,b),c)\ (((),b),c)\\
&testUpdate' = {}^{\$}(update\ ^{\mathbb{P}}[\![\, ((_,y),z) \,]\!]\\
&\qquad\qquad\qquad\qquad\ ^{\mathbb{P}}[\![\, (((),y),z) \,]\!]\\
&\qquad\qquad\qquad\qquad\ ^{\mathbb{D}}[\![\, y = Replace; z = Replace \,]\!])
\end{aligned}$$

3.5 Case

The *Case* combinator is for case analysis, and the general structure is as follows:

$$\begin{aligned}
&Case\ [\ {}^{\$}(normal\ [\![\, mainCond :: s \to v \to Bool \,]\!]\ [\![\, exitCond :: s \to Bool \,]\!])\\
&\qquad \Longrightarrow (bx :: BiGUL\ s\ v)\\
&\quad , \cdots\\
&\quad , {}^{\$}(adaptive\ [\![\, mainCond :: s \to v \to Bool \,]\!])\\
&\qquad \Longrightarrow (f :: s \to v \to s)\\
&\quad , \cdots\\
&\quad]\\
&:: BiGUL\ s\ v
\end{aligned}$$

It contains a sequence of cases, each of which is either *normal* or *adaptive*. We try the conditions of these cases in order and decide which branch we go into.

– For a normal case, $^{\$}(normal\ldots)$ takes two predicates, which we call the *main condition* and the *exit condition*. The predicate for the main condition is very general, and we can use any function of type $(s \to v \to Bool)$ to examine the source and view. The predicate for the exit condition checks the source only. If the main and the exit conditions are satisfied, then the BiGUL program after the arrow '$\Longrightarrow$' (written '`==>`' in plain text Haskell and defined in the module `Generics.BiGUL.Lib`) is executed. The exit conditions in different branches are expected to be disjoint for efficient execution of the forward transformation.
– For an adaptive case, if the main condition is satisfied, a function of type $(s \to v \to s)$ is used to produce an adapted source from the current source and view before the whole *Case* is rerun, with the expectation that one of the normal cases will be applicable this time. Note that if adaptation does not lead to a normal case, an error will be reported at runtime. This is to ensure that BiGUL does not stuck in adaptation and fail to terminate.

As a simple example, consider using the view to replace each element in the source list. To do so, we use *Case* to describe a case analysis.

$$
\begin{array}{l}
replaceAll :: (Eq\ s, Show\ s) \Rightarrow BiGUL\ [s]\ s \\
replaceAll = \\
\quad Case\ [^{\$}(normal\ [\![\ \lambda s\ v \to length\ s == 1\ ]\!]\ [\![\ \lambda s \to length\ s == 1\ ]\!]) \\
\qquad \Longrightarrow {}^{\$}(rearrS\ [\![\ \lambda[x] \to x\ ]\!])\ Replace \\
\qquad ,{}^{\$}(normal\ [\![\ \lambda s\ v \to length\ s > 1\ ]\!]\ [\![\ \lambda s \to length\ s > 1\ ]\!]) \\
\qquad \Longrightarrow {}^{\$}(rearrS\ [\![\ \lambda(x : xs) \to (x, xs)\ ]\!])^{\$} \\
\qquad\qquad {}^{\$}(rearrV\ [\![\ \lambda v \to (v, v)\ ]\!])^{\$} \\
\qquad\qquad\quad Replace\ `Prod`\ replaceAll \\
\qquad ,{}^{\$}(adaptive\ [\![\ \lambda s\ v \to length\ s == 0\ ]\!]) \\
\qquad \Longrightarrow \lambda s\ v \to [\bot] \\
\qquad]
\end{array}
$$

It consists of two normal cases and one adaptive case. The first normal case says that if the source is of length 1 (containing a single element), we rearrange the source list by extracting the single element, and replace this element with the view. The second normal case says that if the source has more than 1 element, we rearrange the source list to a pair of its head element and its tail, rearrange the view by duplicating it to a pair, and use one copy of the view to replace the head element, and the other copy to recursively replace each element in the tail of the source. The last adaptive case says that if the source is empty, we adapt the source to a singleton list with the *don't-care* element $\bot$ ('`undefined`' in plain text Haskell), and rerun the whole *Case* executing the first normal case.

```
*Basic> put replaceAll [] 100
Just [100]
*Basic> put replaceAll [1..10] 100
Just [100,100,100,100,100,100,100,100,100,100]
```

Note that in the first running example, the source [ ] is first adapted to [⊥], and the *don't care* element ⊥ is replaced by 100 at the rerun of the whole *Case*.

As another interesting example, we define *emb*, which can safely embed any pair of well-behaved *get* and *put* into BiGUL. It is defined as follows:

```
emb :: Eq v ⇒ (s → v) → (s → v → s) → BiGUL s v
emb g p =
  Case [ $(normal ⟦ λs v → g s == v ⟧ ⟦ λs → True ⟧)
           ⟹ Skip g
       , $(adaptive ⟦ λ_ _ → otherwise ⟧)
           ⟹ p
       ]
```

where, given a pair (g, p) of well-behaved *get* and *put* functions, if the view is the same as that produced by applying g to the source, we make no change on the source with *Skip g* (hinting that the view can be produced using g), otherwise we adapt the source using p to reflect the change on the view to the source. Note that if p and g form a well-behaved bidirectional transformation, in the rerun of the whole *Case* after the adaptation, the first normal case will always be applicable. To see a use of *emb*, we may define the following putback function to update a pair with its sum.

```
pSum2 :: BiGUL (Int, Int) Int
pSum2 = emb g p
  where g (x, y)    = x + y
        p (x, y) v = (v − y, y)
```

While we allow a general function to describe the main condition or the exit condition, it is usually more concise to use patterns to describe these conditions. For instance, we may replace the condition ⟦ λs → *length* s == 1 ⟧ by

```
⟦ λ[x] → True ⟧
```

Here, the meaning of a boolean-valued pattern-matching lambda-expression is redefined as a total function which computes to *False* when an input does not match the pattern; this meaning is different from that of a general pattern-matching lambda-expression, which fails to compute (and throws an exception) when the pattern is not matched. For example, in general the lambda-expression λ[x] → *True* will fail to compute if the first input is not a singleton list; when used in branch construction, however, the lambda-expression will compute to *False* upon encountering an empty list. A unary condition like ⟦ λ[x] → *True* ⟧ where only the pattern part matters can be abbreviated to

```
ℙ⟦ [x] ⟧
```

to further reduce syntactic noise. Finally, to also allow this kind of abbreviation in main conditions, BiGUL provides a special form for the *normal* case where

the main condition is specified as the conjunction of two unary predicates on the source and view respectively:

$$
\begin{aligned}
&{}^{\$}(normalSV\ [\![\, sourceCond :: s \to Bool \,]\!] \\
&\qquad\qquad [\![\, viewCond \quad :: v \to Bool \,]\!] \\
&\qquad\qquad [\![\, exitCond \quad :: s \to Bool \,]\!]) \\
&\quad \Longrightarrow (bx :: BiGUL\ s\ v)
\end{aligned}
$$

and a special form for the *adaptive* case where the main condition is specified as the conjunction of two unary predicates on the source and view respectively:

$$
\begin{aligned}
&{}^{\$}(adaptiveSV\ [\![\, sourceCond :: s \to Bool \,]\!] \\
&\qquad\qquad\quad [\![\, viewCond \quad :: v \to Bool \,]\!]) \\
&\quad \Longrightarrow (f :: s \to v \to s)
\end{aligned}
$$

3.6 View dependency

Sometimes, a view may contain derived values that are computed from other parts of the view, and the view should be consistently changed. For instance, for the view $(x, even\ (x))$, the second component is an indicator showing whether or not the first component is an even number. To capture this, BiGUL provides

$$Dep :: Eq\ v' \Rightarrow (v \to v') \to BiGUL\ a\ v \to BiGUL\ a\ (v, v')$$

to describe this intention. We may, for example, define

$$
\begin{aligned}
&replaceAll2\ ::\ BiGUL\ [Int]\ (Int, Bool) \\
&replaceAll2 = Dep\ even\ replaceAll
\end{aligned}
$$

to replace all elements of the source by the first component of the view, while checking whether the second component is consistent with the first component.

```
*Basic> put replaceAll2 [1..10] (100,True)
Just [100,100,100,100,100,100,100,100,100,100]
*Basic> put replaceAll2 [1..10] (100,False)
Nothing
*Basic> putTrace replaceAll2 [1..10] (100,False)
second view component not determined by the first
```

As seen in the last running of *put*, it reports an error because the view $(100, False)$ is inconsistent: 100 is an even number, so the second component should be *True*.

3.7 Composition

BiGUL programs can be composed sequentially:

$$Compose :: BiGUL\ a\ u \to BiGUL\ u\ b \to BiGUL\ a\ b$$

This combinator is straightforward in the *get* direction: *get* (*Compose* l r) (where $l :: BiGUL\ a\ u$ and $r :: BiGUL\ u\ b$) simply applies *get* l to its input of type a to compute an intermediate value of type u, which is then processed by *get* r to produce the final result of type b. Its *put* direction is more complex: *put* (*Compose* l r) starts with a source $s :: a$ and a view $v :: b$, and the aim is to produce an updated source of type a. The only way to proceed is to use *put* r to put v into some intermediate source m of type u, and to produce this m we are forced to use *get* l on s. We can then update m with v to m' using *put* r, and update a with m' using *put* l. In general, programs involving *Compose* are significantly harder to think about since we have to think in both *put* and *get* directions to figure out precisely what is going on.

As a simple example, consider that we wish to use the view to update the head element of the head element of a list of lists. We can define such a putback function as the following *pHead2* by composing *pHead* with *pHead*.

$$pHead2 :: Show\ a \Rightarrow BiGUL\ [[a]]\ a$$
$$pHead2 = pHead\ `Compose`\ pHead$$

The following is an example to demonstrate this:

```
*Basic> put pHead2 [[1,2],[3,4,5],[]] 100
Just [[100,2],[3,4,5],[]]
```

4 Bidirectional programming on lists

To give some more involved examples, in this section we demonstrate that many list functions can be bidirectionalized using BiGUL. The putback behaviors of these functions are in fact non-trivial, and the reader might want to skip to later sections in which more examples are developed, starting from Section 6.

To show the correspondence with the original list functions, we prefix the original forward function names with *lens*. Note that in our context, the original forward functions can be automatically derived from the new putback transformations by calling *get*.

We shall focus on bidirectionalizing *foldr*, an important higher-order function on lists:

$$foldr :: (a \to b \to b) \to b \to [a] \to b$$
$$foldr\ f\ e\ [\,] \quad\quad = e$$
$$foldr\ f\ e\ (x : xs) = f\ x\ (foldr\ f\ e\ xs)$$

Many interesting functions can be defined in terms of *foldr*:

$$sum \quad = foldr\ (+)\ 0$$
$$map\ f = foldr\ (\lambda a\ r \to f\ a : r)\ [\,]$$

where *sum* sums up all the elements in a list, and *map* f applies f to every element in a list.

We start by developing a putback function for *foldr* in BiGUL:

$$
\begin{aligned}
&\mathit{lensFoldr} \;::\; (\mathit{Show}\ a, \mathit{Show}\ v) \\
&\qquad \Rightarrow \mathit{BiGUL}\ (a, v)\ v \to (v \to \mathit{Bool}) \to \mathit{BiGUL}\ ([a], v)\ v
\end{aligned}
$$

where we hope to define a putback program of type $\mathit{BiGUL}\ ([a], v)\ v$ that is to use the view to update the source, a list together with a value, by recursively applying a simpler putback function of type $\mathit{BiGUL}\ (a, v)\ v$ (until a condition is satisfied or all the list elements have been visited). The program is somewhat tricky, and is probably not easy to understand since *Compose* is involved.

$$
\begin{aligned}
&\mathit{lensFoldr}\ \mathit{bx}\ \mathit{pv} = \\
&\quad \mathit{Case}\ [\,{}^{\$}(\mathit{adaptive}\ [\![\, \lambda(x, y)\ v \to \mathit{pv}\ v \wedge \mathit{length}\ x \not\equiv 0 \,]\!]) \\
&\qquad\quad \Longrightarrow \lambda(x, y)\ v \to ([\,], y) \\
&\qquad , {}^{\$}(\mathit{normal}\ [\![\, \lambda(\mathit{xs}, \_)\ v \to \mathit{null}\ \mathit{xs} \,]\!]\ [\![\, \lambda(\mathit{xs}, \_) \to \mathit{null}\ \mathit{xs} \,]\!]) \\
&\qquad\quad \Longrightarrow {}^{\$}(\mathit{rearrV}\ [\![\, \lambda v \to ((), v) \,]\!])^{\$} \\
&\qquad\qquad\quad {}^{\$}(\mathit{update}\ {}^{\mathbb{P}}[\![\, (\_, v) \,]\!]\ {}^{\mathbb{P}}[\![\, ((), v) \,]\!]\ {}^{\mathbb{D}}[\![\, v = \mathit{Replace} \,]\!]) \\
&\qquad , {}^{\$}(\mathit{normalSV}\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ [\![\, \lambda(\mathit{xs}, \_) \to \mathit{not}\ (\mathit{null}\ \mathit{xs}) \,]\!]) \\
&\qquad\quad \Longrightarrow {}^{\$}(\mathit{rearrS}\ [\![\, \lambda((x : \mathit{xs}), e) \to (x, (\mathit{xs}, e)) \,]\!])^{\$} \\
&\qquad\qquad\quad (\mathit{Replace}\ `\mathit{Prod}`\ \mathit{lensFoldr}\ \mathit{bx}\ \mathit{pv})\ `\mathit{Compose}`\ \mathit{bx} \\
&\qquad]
\end{aligned}
$$

The *lensFoldr* program accepts a putback function bx and a view condition pv, and performs a case analysis to put the view v to the source (xs, e). If the view v satisfies pv but the list xs in the source is not empty, then it adapts the list to be empty. If the list xs in the source is empty, we do nothing but use the view to replace the second component of the source. Otherwise, we rearrange the source from the form of $(x : xs, e)$ to that of $(x, (xs, e))$, and apply *lensFoldr* recursively with a composition with bx. One may understand the composition through the following picture (where $r = \mathit{Replace}\ `\mathit{Prod}`\ \mathit{lensFoldr}\ \mathit{bx}\ \mathit{pv}$).

$$(x, (xs, e)) \overset{r}{\leftrightarrow} (x, e') \overset{bx}{\leftrightarrow} v$$

With *lensFoldr*, we can redefine many list functions from the putback point of view. As the first example, consider *mapAppend*:

$$\mathit{mapAppend}\ f\ (\mathit{xs}, \mathit{ys}) = \mathit{map}\ f\ \mathit{xs} +\!\!+ \mathit{ys}$$

We can define its putback function as follows.

$$
\begin{aligned}
&\mathit{lensMapAppend} :: (\mathit{Show}\ a, \mathit{Show}\ b) \Rightarrow \mathit{BiGUL}\ a\ b \to \mathit{BiGUL}\ ([a], [b])\ [b] \\
&\mathit{lensMapAppend}\ \mathit{pf} = \mathit{lensFoldr}\ \mathit{bx}\ \mathit{null} \\
&\quad \textbf{where}\ \mathit{bx} = {}^{\$}(\mathit{rearrV}\ [\![\, \lambda(v : \mathit{vs}) \to (v, \mathit{vs}) \,]\!])^{\$} \\
&\qquad\qquad\qquad \mathit{pf}\ `\mathit{Prod}`\ \mathit{Replace}
\end{aligned}
$$

Here bx has the type of $\mathit{BiGUL}\ (a, [b])\ [b]$ and is defined on pf that has the type of $\mathit{BiGUL}\ a\ b$.

```
*List> put (lensMapAppend dec1) ([0..10],[]) [100..110]
Just ([99,100,101,102,103,104,105,106,107,108,109],[])
*List> get (lensMapAppend dec1) ([1..10],[])
Just [2,3,4,5,6,7,8,9,10,11]
```

Note that, for testing, we embed into our framework the bijective functions for increasing and decreasing a number by 1.

$$
\begin{aligned}
&\mathit{dec1} :: (\mathit{Eq}\ a, \mathit{Num}\ a) \Rightarrow \mathit{BiGUL}\ a\ a \\
&\mathit{dec1} = \mathit{emb}\ g\ p \\
&\quad \textbf{where}\ g\ s\ \ = s + 1 \\
&\qquad\qquad\ p\ s\ v = v - 1
\end{aligned}
$$

For a second example, consider the function $\mathit{sum}\ (\mathit{xs}, e)$, which is to sum up all elements of the list xs starting from the seed e. If the sum is changed, there are many ways to reflect this change to the input (xs, e). The following describes one way in BiGUL:

$$
\begin{aligned}
&\mathit{lensSum} :: \mathit{BiGUL}\ ([\mathit{Int}], \mathit{Int})\ \mathit{Int} \\
&\mathit{lensSum} = \mathit{lensFoldr}\ \mathit{pSum2}\ (\mathit{const}\ \mathit{False})
\end{aligned}
$$

which will reflect the change difference on the view to the head element of xs if xs is not empty, or to the seed e otherwise. We may choose other ways, say to reflect the change difference on the view only to the seed by defining

$$
\begin{aligned}
&\mathit{lensSum}' :: \mathit{BiGUL}\ ([\mathit{Int}], \mathit{Int})\ \mathit{Int} \\
&\mathit{lensSum}' = \mathit{lensFoldr}\ (^{\$}(\mathit{rearrS}\ [\![\, \lambda(x, y) \to (y, x)\,]\!])\ \mathit{pSum2})\ (\mathit{const}\ \mathit{False})
\end{aligned}
$$

Note that although $\mathit{get}\ \mathit{lensSum}\ ([1,2,3], 0) = \mathit{get}\ \mathit{lensSum}'\ ([1,2,3], 0) = \mathit{Just}\ 6$, their putback behaviors are different:

$$
\begin{aligned}
&\mathit{put}\ \mathit{lensSum}\ \ ([1,2,3], 0)\ 16 = \mathit{Just}\ ([11,2,3], 0) \\
&\mathit{put}\ \mathit{lensSum}'\ ([1,2,3], 0)\ 16 = \mathit{Just}\ ([1,2,3], 10)
\end{aligned}
$$

It is worth noting that our definition of *lensFoldr* is just one putback function for *foldr*, and there are many others. This reflects the fact that one *foldr* can have many *put*s, each describing one updating strategy.

5 BiGUL's bidirectionality

We have been writing *put* programs, usually having a corresponding *get* in mind but not explicitly describing it, and yet BiGUL is capable of finding the right *get* behaviour as if it could read our mind. How? We will see that, when writing a BiGUL program, we are always simultaneously describing both a *put* function and a *get* function, which are guaranteed to be a well-behaved pair. And the "mind-reading" ability is far from magic: It is the consequence of the fact that well-behavedness directly implies that *get* is uniquely determined by *put*, which

is the main motivation for taking a putback-based approach. In this section, we will first review the theory, this time explicitly taking *partiality* into account, and then we will dive into BiGUL's internals to get a taste of putback-based design.

This is a fairly long section, but it is not a prerequisite for subsequent sections; readers who wish to see more examples first or are more interested in practical BiGUL applications can safely skip this section and proceed to Section 6.

5.1 Lenses, well-behavedness, and the fundamental theorem

Formally, we call a well-behaved pair of *put* and *get* a *lens*:

Definition 2 (lens). *A* lens *between a source type s and a view type v consists of two functions:*

$$\begin{aligned} &put :: s \to v \to Maybe\ s \\ &get :: s \quad\ \to Maybe\ v \end{aligned}$$

satisfying two well-behavedness laws:

$$put\ s\ v = Just\ s' \quad \Rightarrow \quad get\ s' = Just\ v \qquad \textsc{PutGet}$$

$$get\ s = Just\ v \quad \Rightarrow \quad put\ s\ v = Just\ s \qquad \textsc{GetPut}$$

In the original formulation [8], a lens refers to just a pair of functions having the right types, and one needs to explicitly say "well-behaved lens" to mean a well-behaved pair; we will, however, discuss well-behaved lenses only, so we build well-behavedness into our definition of lenses by default. Note that this definition models partial transformations explicitly as *Maybe*-valued functions: *put* and *get* are *total* functions that can nevertheless produce *Nothing* to indicate failure. From now on, this definition replaces the one in Section 1, where only total lenses were discussed. Also note that these well-behavedness laws are actually easy to satisfy vacuously, by making the transformations produce *Nothing* all (or most of) the time. One important task of the BiGUL programmer is thus to meet certain side conditions for guaranteeing the totality of their BiGUL programs. These side conditions will be introduced below along with the relevant BiGUL constructs.

From this revised definition of well-behavedness, we can immediately prove a reformulation of Lemma 1:

Theorem 2 (uniqueness of *get*). *Given two lenses whose put components are equal, their get components are also equal.*

Proof. Let l and r be two lenses; denote their put/get components as $put\ l/get\ l$ and $put\ r/get\ r$ respectively, and assume that $put\ l = put\ r$. Then for any

s and v,

$$
\begin{array}{ll}
 & \mathit{get}\ l\ s = \mathit{Just}\ v \\
\Leftrightarrow & \{\,\text{well-behavedness of } l\,\} \\
 & \mathit{put}\ l\ s\ v = \mathit{Just}\ s \\
\Leftrightarrow & \{\,\mathit{put}\ l = \mathit{put}\ r\,\} \\
 & \mathit{put}\ r\ s\ v = \mathit{Just}\ s \\
\Leftrightarrow & \{\,\text{well-behavedness of } r\,\} \\
 & \mathit{get}\ r\ s = \mathit{Just}\ v
\end{array}
$$

(This also entails that $\mathit{get}\ l\ s = \mathit{Nothing}$ if and only if $\mathit{get}\ r\ s = \mathit{Nothing}$.) □

This might be called the "fundamental theorem" of putback-based bidirectional programming, as the theorem guarantees that the BiGUL programmer is in full control of the bidirectional behaviour — programming the *put* behavior is sufficient to determine the *get* behaviour. Also, to the language designer, the theorem gives a kind of reassurance that, once the *put* behaviour of a construct is determined, there is no need to worry about which *get* behaviour should be adopted — there is at most one possibility. This is in contrast to *get*-based design, in which there are usually more than one viable *put* semantics that can be assigned to a *get*-based construct, and the designer needs to justify the choice or provide several versions.

For the rest of this section, we will look at several constructs of BiGUL in detail to get a taste of putback-based design. Each BiGUL construct is conceived, at the design stage, as a lens (like *Skip* and *Replace*) or a lens *combinator* (like *Case*), which constructs a more complex lens from simpler ones. The *put* and *get* components of these lenses usually have to be developed together, but for each lens we will employ a more "*put*-oriented" design process: We start from an intended *put* behaviour, and then add restrictions so that we can find a corresponding *get*. This does not guarantee that the lenses we arrive at will have a "strong *put* flavour" — that is, some of the lenses will be as (or even more) suitable for *get*-based programming as for putback-based programming. But we will also see that some other lenses are more naturally understood in terms of their *put* behaviour.

5.2 Replacement

The simplest lens is probably *Replace*, which replaces the entire source with the view:

$$\mathit{put}\ \mathit{Replace}\ s\ v = \mathit{Just}\ v$$

Is there a *get* semantics that can be paired with this *put*? Yes, quite obviously — in fact, PutGet directly gives us the definition of *get Replace*:

$$\mathit{get}\ \mathit{Replace}\ v = \mathit{Just}\ v$$

We still need to verify GetPut, which can be easily checked to be true.

5.3 Skipping

Coming up next is *Skip*, whose natural behaviour is

$$put\ Skip\ s\ v = Just\ s$$

Considering PutGet, though, we immediately see that this behaviour is too liberal: If the view is simply thrown away, how can *get Skip* possibly recover it? One way out is to require that the view is trivial enough such that it can be thrown away and still be recovered, by setting the view type of *Skip* to the unit type (). Then it is easy for *get Skip* to recover the view, for which there is only one choice:

$$get\ Skip\ s = Just\ ()$$

This is the approach adopted prior to BiGUL 1.0.

More generally, we can establish well-behavedness as long as *get Skip* has only one view choice for each source, regardless of what the view type is. The existence of this "unique choice" is witnessed by a function $f :: s \to v$, which we add as an additional argument to *Skip*. The *get* direction is then

$$get\ (Skip\ f)\ s = Just\ (f\ s)$$

From the *put* direction, we may think of this function f as specifying a consistency relation, saying that the view information is completely included in the source (since you can compute the view from the source) and can be safely discarded. *Skip f* can be used if and only if the source and view are consistent in that sense, and this is the side condition about *Skip* that the BiGUL programmers need to be aware of if they want their programs using *Skip* to be total. We thus arrive at:

$$put\ (Skip\ f)\ s\ v = \textbf{if}\ v == f\ s\ \textbf{then}\ return\ s\ \textbf{else}\ Nothing$$

This pair of *put* and *get* can be verified to be well-behaved. *Skip f*, which features in BiGUL 1.0, is one lens which turns out to be more easily understood from the *get* direction — it bidirectionalizes any *get* function whose codomain has decidable equality, albeit trivially. We recover the first version of *Skip* as a special case by setting f to *const* ().

5.4 Product

For a simplest example of a lens combinator, we look at *Prod*. Both the source and view types should be pairs; *Prod* accepts two lenses, say l and r, and applies them respectively to the left and right components:

$$\begin{aligned} put\ (l\ `Prod`\ r)\ (sl, sr)\ (vl, vr) = \textbf{do}\ & sl' \leftarrow put\ l\ \ sl\ \ vl \\ & sr' \leftarrow put\ r\ sr\ vr \\ & return\ (sl', sr') \end{aligned}$$

The *get* direction is unsurprising:

$$\begin{aligned} get\ (l\ `Prod`\ r)\ (sl, sr) = \mathbf{do}\ & vl \leftarrow get\ l\ \ sl \\ & vr \leftarrow get\ r\ sr \\ & return\ (vl, vr) \end{aligned}$$

Having constructed *put* and *get* from l and r, we also expect that their well-behavedness is a consequence of the well-behavedness of l and r. While this may look obvious, we take this opportunity to show how a well-behavedness proof for a lens combinator can be carried out formally and in detail. To prove PutGet, for example, we should prove that the assumption

$$put\ (l\ `Prod`\ r)\ (sl, sr)\ (vl, vr) = Just\ (sl', sr') \tag{1}$$

implies the conclusion

$$get\ (l\ `Prod`\ r)\ (sl', sr') = Just\ (vl, vr) \tag{2}$$

Both equations say that a somewhat complicated monadic *Maybe*-program computes successfully to some value. It may seem that we need some messy case analysis, but what we know about *Maybe*-programs tells us that such a program computes successfully if and only if every step of the program does, and this helps us to split both (1) and (2) into simpler equations. Formally, we have this lemma:

Lemma 2. *Let* $mx :: Maybe\ a$ *and* $f :: a \to Maybe\ b$*. Then, for all* $y :: b$*,*

$$mx \gg\!= f = Just\ y$$

if and only if

$$mx = Just\ x \quad and \quad f\ x = Just\ y \qquad for\ some\ x :: a$$

Proof. Case analysis on mx. □

This lemma can be nicely applied to *Maybe*-programs written in the **do**-notation, transforming such programs into *predicates* saying that a program computes to some given value. To do it more formally: Define a translation $\mathcal{S}$ from **do**-blocks of type $Maybe\ a$ to predicates on a by

$$\begin{aligned} \mathcal{S}\ (\mathbf{do}\ \{x \leftarrow mx; B\})\ y &= (\exists x.\ mx = Just\ x \wedge \mathcal{S}\ (\mathbf{do}\ B)\ y) \\ \mathcal{S}\ (\mathbf{do}\ \{my\})\ y &= (my = Just\ y) \end{aligned}$$

Then we can extend Lemma 2 to the following:

Lemma 3. *The proposition*

$$\mathcal{S}\ (\mathbf{do}\ B)\ y$$

is true if and only if

$$\mathbf{do}\ B = Just\ y$$

Proof. By induction on the list structure of B, using Lemma 2 repeatedly. □

For example, applying $\mathcal{S}$ to $put\ (l\ `Prod`\ r)\ (sl, sr)\ (vl, vr)$ yields

$$\begin{aligned}\lambda(sl'', sr'').\ &\exists sl'.\ put\ l\ sl\ vl = Just\ sl' \wedge \\ &\exists sr'.\ put\ r\ sr\ vr = Just\ sr' \wedge \\ &return\ (sl', sr') = Just\ (sl'', sr'')\end{aligned}$$

where the last equation is equivalent to $sl' = sl'' \wedge sr' = sr''$ (since $return = Just$ for the *Maybe* monad). Applying Lemma 3 and doing some simplification, (1) is equivalent to

$$put\ l\ sl\ vl = Just\ sl' \quad \wedge \quad put\ r\ sr\ vr = Just\ sr'$$

Similarly, (2) can be shown to be equivalent to

$$get\ l\ sl' = Just\ vl \quad \wedge \quad get\ r\ sr' = Just\ vr$$

The entailment is then just PutGet for l and r.

5.5 Case analysis

This is a representative combinator in BiGUL, and arguably the most complex one. For simplicity, let us consider a two-branch variant of *Case*. A branch is a condition and a body; since in *put* we manipulate both a source and a view, the conditions in general can be binary predicates on both the source and view. We thus define the type of branches as:

$$\textbf{type}\ CaseBranch\ s\ v = (s \to v \to Bool, BiGUL\ s\ v)$$

and consider the following variant of *Case*:

$$Case :: CaseBranch\ s\ v \to CaseBranch\ s\ v \to BiGUL\ s\ v$$

The straightforward behaviour is

$$\begin{aligned}put\ (Case\ (pl, l)\ (pr, r))\ s\ v = \ &\textbf{if}\ \ pl\ s\ v\ \textbf{then}\ put\ l\ s\ v \\ &\textbf{else if}\ pr\ s\ v\ \textbf{then}\ put\ r\ s\ v \\ &\textbf{else}\ Nothing\end{aligned}$$

That is, depending on which condition is satisfied (with pl having higher priority), we execute either $put\ l$ or $put\ r$, or fail the computation if neither of the conditions is satisfied. Now, again, we ask the question: Can we find a *get* behaviour to pair with this *put*?

Ruling out branch switching for PutGet. An important working assumption here is that we want lens combinators to be *compositional*: When we looked at *Prod*, for example, we defined its *put* and *get* in terms of those of the smaller lenses, and derived the overall well-behavedness from that of the smaller lenses. For *Case*, this implies that, when establishing well-behavedness, we want a *get* following a *put* (or a *put* following a *get*) to use the same branch taken by the *put* (or the *get*), so we can make use of PutGet (or GetPut) of the branch. The current *put* behaviour of *Case* does not leave any clue in the updated source about which branch is used to produce it, though, so it is impossible for *get* to always choose the correct branch.

One solution, which does not require changing the syntax of *Case*, is to check that the ranges of the branches are *disjoint*. In general, for a lens, the range of a *put* can be shown to coincide with the domain of the corresponding *get*. So the *get* behaviour of *Case* can simply try to execute both branches on the input source, and there will be at most one branch that computes successfully. We can put (expensive) disjointness checks into *put* such that if *put* succeeds, the subsequent *get* will have at most one branch to choose:

$$
\begin{array}{l}
put\ (Case\ (pl, l)\ (pr, r))\ s\ v = \\
\quad \textbf{if} \quad\ pl\ s\ v\ \textbf{then do}\ s' \leftarrow put\ l\ s\ v \\
\qquad\qquad\qquad\qquad\quad maybe\ (return\ s')\ (const\ Nothing)\ (get\ r\ s') \\
\quad \textbf{else if}\ pr\ s\ v\ \textbf{then do}\ s' \leftarrow put\ r\ s\ v \\
\qquad\qquad\qquad\qquad\quad maybe\ (return\ s')\ (const\ Nothing)\ (get\ l\ s') \\
\quad \textbf{else}\ Nothing
\end{array}
$$

The *maybe* function is from Haskell's prelude and has type $b \to (a \to b) \to Maybe\ a \to b$; depending on whether the third, *Maybe*-typed, argument is *Nothing* or a *Just*-value, the result is either the first argument or the second argument applied to the value wrapped inside *Just*. In the first branch of the code above, if $put\ l\ s\ v$ successfully produces an updated source s', we will ensure that $get\ r\ s'$ does not succeed: If $get\ r\ s'$ is *Nothing* as we want, we will $return\ s'$; otherwise we emit *Nothing*.

If *get* favours the first branch, meaning that it declares success as soon as the first branch succeeds (without requiring that the second branch fails),

$$
get\ (Case\ (pl, l)\ (pr, r))\ s = maybe\ (get\ r\ s)\ return\ (get\ l\ s)
$$

then we can also omit the check in *put*'s first branch:

$$
\begin{array}{l}
put\ (Case\ (pl, l)\ (pr, r))\ s\ v = \\
\quad \textbf{if} \quad\ pl\ s\ v\ \textbf{then}\ put\ l\ s\ v \\
\quad \textbf{else if}\ pr\ s\ v\ \textbf{then do}\ s' \leftarrow put\ r\ s\ v \\
\qquad\qquad\qquad\qquad\quad maybe\ (return\ s')\ (const\ Nothing)\ (get\ l\ s') \\
\quad \textbf{else}\ Nothing
\end{array}
$$

Ruling out branch switching for GetPut. The GetPut direction, on the other hand, still does not avoid branch switching — the outcome of *get* does not say anything about which of *pl* and *pr* will be satisfied in the subsequent *put*. So we add some checks to *get* such that *get*'s success will tell us which branch will be chosen by *put*:

```
get (Case (pl, l) (pr, r)) s = maybe (do v ← getBranch (pr, r) s
                                         if pl s v then Nothing
                                                   else return v)
                                     return
                                     (getBranch (pl, l) s)
getBranch (p, b) s = do v ← get b s
                        if p s v then return v
                                 else Nothing
```

The definition of *put* should also be revised to use *getBranch* for the disjointness check. This fixes GetPut, but breaks PutGet! Since *put* does not guarantee that the *updated* (not the original) source and the view satisfy the condition of the branch executed, even though *get* will be able to choose the correct branch, the subsequent, newly added check is not guaranteed to succeed. We thus also need to add similar checks to *put*:

```
put (Case (pl, l) (pr, r)) s v =
  if      pl s v then do s' ← put l s v
                         if pl s' v then return s'
                                    else Nothing
  else if pr s v then do s' ← put r s v
                         if pr s' v then maybe (return s')
                                              (const Nothing)
                                              (getBranch (pl, l) s')
                                    else Nothing
  else Nothing
```

Now this pair of *put* and *get* can be verified to be well-behaved.

Improving the efficiency of *get*. The efficiency of the current *get* does not look very good, especially when, in general, more than two branches are allowed, and *get* has to try to execute each branch, possibly with a high cost, until it reaches a successful one; also, inefficient *get* affects the efficiency of *put*, since this calls *get* to check range disjointness. An idea is to ask the programmer to make a rough "prediction" of the range of each branch: We enrich *CaseBranch* with a third component, which is a source predicate:

```
type CaseBranch s v = (s → v → Bool, BiGUL s v, s → Bool)
```

This new predicate is supposed to be satisfied by the updated source; we again add checks to *put* to ensure this:

```
put (Case (pl, l, ql) (pr, r, qr)) s v =
  if       pl s v then do s' ← put l s v
                          if pl s' v ∧ ql s' then return s'
                                             else Nothing
  else if pr s v then do s' ← put r s v
                          if pr s' v ∧ qr s'
                            then maybe (return s')
                                       (const Nothing)
                                       (getBranch (pl, l, ql) s')
                            else Nothing
  else Nothing
```

Let us call *pl* and *pr* the *main conditions*, and *ql* and *qr* the *exit conditions*. The exit condition, in general, over-approximates the range of a branch. Well-behavedness tells us that the range of *put* is exactly the domain of the corresponding *get*. Thus, in the *get* direction, every source in the domain of a branch satisfies the exit condition. Contrapositively, if a source does not satisfy the exit condition, then *get* for that branch will necessarily fail, and we do not need to try to execute the branch at all. This leads to the following revised definition of *getBranch*:

```
getBranch (pl, l, ql) s = if ql s then do v ← get l s
                                          if pl s v then return v
                                                    else Nothing
                                  else Nothing
```

If we do not care about efficiency, we can simply use *const True* as exit conditions, and the behaviour will be exactly the same as the previous version. But if we supply disjoint exit conditions, then *get* will try at most one branch. Incidentally (but actually no less importantly), making exit conditions explicit also encourages the programmer to think about range disjointness, which is essential to guaranteeing the totality of *Case*.

Adaptation. We have seen that, to make *Case* total, one thing we need to ensure is that the main condition of a branch should be satisfied again after the update. In practice, the main condition is usually closely related to the consistency relation, and we will only be able to deal with sources and views that are already more or less consistent; this is a rather severe restriction. As we have seen in Section 3.5, the solution is to introduce a different kind of branch called *adaptive branches*, which can deal with sources and views that are too inconsistent by adapting the source to establish enough consistency such that a normal branch becomes applicable. Again, for simplicity, we consider only a variant of *Case* which has just one adaptive branch at the end:

```
type CaseAdaptiveBranch s v = (s → v → Bool, s → v → s)
Case :: CaseBranch s v → CaseBranch s v →
        CaseAdaptiveBranch s v → BiGUL s v
```

The execution structure of *put* becomes slightly more complicated, as the whole thing has to be run again after adaptation; to ensure termination, we require that the second run does not match an adaptive branch again. This is realized in BiGUL in continuation-passing style:

```
put (Case bl br ba) s v =
  putWithAdaptation bl br ba s v (λsa →
    putWithAdaptation bl br ba sa v (const Nothing))
putWithAdaptation ::
  CaseBranch s v → CaseBranch s v → CaseAdaptiveBranch s v →
  s → v → (s → Maybe s) → Maybe s
putWithAdaptation (pl, l, ql) (pr, r, qr) (pa, f) s v cont =
  if      pl s v then do s' ← put l s v
                         if pl s' v ∧ ql s' then return s'
                                            else Nothing
  else if pr s v then do s' ← put r s v
                         if pr s' v ∧ qr s'
                           then maybe (return s')
                                      (const Nothing)
                                      (getBranch (pl, l, ql) s')
                           else Nothing
  else if pa s v then cont (f s v)
  else Nothing
```

Major work is now moved into a separate function *putWithAdaptation*, which takes an extra *cont* argument of type $s \to \mathit{Maybe}\ s$. This extra argument is a continuation that takes over after the body of an adaptive branch is executed, and is invoked with the adapted source. The requirement of not doing adaptation twice is met by setting *putWithAdaptation* itself as a continuation, and this inner *putWithAdaptation* takes the continuation that always fails.

What about *get*? It turns out that *get* can simply ignore the adaptive branch! If you have doubt about this "choice", just invoke the fundamental theorem (Theorem 2): The *put* behaviour is exactly what we want, and we can verify that the pair of *put* and *get* is well-behaved, so we are reassured that our "choice" is "correct", simply because there is no other choice of *get*.

To sum up, we have arrived at a simpler variant of *Case* which nevertheless has all the features of the multi-branch *Case* in BiGUL. We have inserted various dynamic checks into the *put* semantics, and the BiGUL programmer needs to be aware of these constraints to make execution of *Case* succeed: For each normal branch, (i) the main condition should be satisfied after the update, (ii) the main conditions of the branches before this one should not be satisfied after the update, and (iii) the exit condition should be satisfied by the updated source. Also the ranges of all the normal branches should be disjoint; the programmer is encouraged to write disjoint exit conditions, which imply disjointness of the ranges, and improve the efficiency of *get*. Finally, for each adaptive branch, the adapted source and the view should match the main condition of a normal branch.

5.6 Rearrangement

Source and view rearrangements are also among the more complex constructs of BiGUL. Their complexity lies in the strongly and generically typed treatment of pattern matching, though, rather than their bidirectional behavior. (We are referring to "pattern matching" in functional programming, where a pattern matching checks whether a value has a specific shape and decomposes it into components. For example, matching a list with a pattern $x{:}y{:}xs$ checks whether the list has two or more elements, and then binds x to the first element, y to the second one, and xs to the rest of the list.) The two kinds of rearrangement are similar, and we will discuss view rearrangement only. We will start by formalizing pattern matching as a bidirectional operation — in fact an isomorphism. Based on pattern matching, evaluation and inverse evaluation of rearranging λ-expressions can be defined, again forming an isomorphism. The semantics of a view rearrangement is then the composition of this latter isomorphism with the lens obtained by interpreting the inner BiGUL program.

Strongly typed pattern matching, bidirectionally. Pattern matching is inherently a bidirectional operation: In one direction, we break something into a collection of its components at the variable positions of a pattern. This collection can be considered as indexed by the variable positions, and acting like an *environment* for expression evaluation. Indeed, conversely, if we have a pattern and a corresponding environment, we can treat the pattern as an expression and evaluate it in the environment. These two directions are inverse to each other, i.e., they form a (partial) isomorphism. For the language designer, it may be slightly tedious to establish such isomorphisms, but for the programmer, pattern matching and evaluation are arguably the most natural way to decompose and rearrange things. Previous bidirectional languages usually provide theoretically simpler combinators for decomposition and rearrangement, but they are hard to use in practice. BiGUL's native support of pattern matching, on the other hand, turns out to be one important contributing factor in its usability.

BiGUL's patterns are strongly typed: The programmer has to declare a target type for a pattern, and the pattern is guaranteed, through typechecking, to make sense for that target type. This can be achieved by defining the datatype of patterns as a generalised algebraic datatype:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Pat}\ a\ \textbf{where} \\
\quad \begin{array}{lll}
\mathit{PVar} & :: \mathit{Eq}\ a \Rightarrow & \mathit{Pat}\ a \\
\mathit{PConst} & :: \mathit{Eq}\ a \Rightarrow a \rightarrow & \mathit{Pat}\ a \\
\mathit{PProd} & :: \mathit{Pat}\ a \rightarrow \mathit{Pat}\ b \rightarrow & \mathit{Pat}\ (a, b) \\
\mathit{PLeft} & :: \mathit{Pat}\ a \rightarrow & \mathit{Pat}\ (\mathit{Either}\ a\ b) \\
\mathit{PRight} & :: \mathit{Pat}\ b \rightarrow & \mathit{Pat}\ (\mathit{Either}\ a\ b) \\
\mathit{PIn} & :: \mathit{InOut}\ a \Rightarrow \mathit{Pat}\ (F\ a) \rightarrow & \mathit{Pat}\ a
\end{array}
\end{array}
$$

A pattern can be a (nameless) variable, a constant, a product, a *Left* or *Right* injection (for the *Either* type), or a generic constructor, and its target type

is given as the index in its type. For example, the pattern $PLeft\ (PConst\ ())$ has type $Pat\ (Either\ ()\ b)$, and can only be used to match those values of type $Either\ ()\ b$ (and matching succeeds only for the value $Left\ ()$). The $InOut$ typeclass contains the types that are isomorphic to (and therefore interconvertible with) a sum-of-products representation. The isomorphism is witnessed by

$$inn :: InOut\ a \Rightarrow F\ a\ a \to a \qquad \text{and} \qquad out :: InOut\ a \Rightarrow a \to F\ a\ a$$

which will be used to define pattern matching and evaluation. For example, $[a]$ is an instance of $InOut$, and $F\ [a]$, an isomorphic sum-of-products representation of $[a]$, is $Either\ ()\ (a, [a])$. The two functions witnessing the isomorphism for lists are defined by

$$\begin{aligned} inn\ (Left\ \ ())\ \ &= [\,] \\ inn\ (Right\ (x, xs)) &= x : xs \\ out\ [\,]\ \ &= Left\ \ () \\ out\ (x : xs) &= Right\ (x, xs) \end{aligned}$$

How do we define pattern matching? As we mentioned above, the result of matching a value against a pattern is an environment indexed by the variable positions of the pattern. For example, matching a list against the cons pattern

$$PIn\ (PRight\ (PProd\ PVar\ PVar)) \tag{3}$$

should produce an environment containing its head and tail. Here we want a safe (but not necessarily efficient) representation of the environment type, in the sense that the indices into the environment should be exactly the variable positions of the pattern, and we want that to be enforced statically by typechecking. In other words, this environment type depends on the pattern, and a way to compute this type is to encode it as a second index of the Pat datatype:

$$\begin{aligned} &\textbf{data}\ Pat\ a\ env\ \textbf{where} \\ &\quad PVar\ \ \ :: Eq\ a \Rightarrow && Pat\ a\ (Var\ a) \\ &\quad PConst :: Eq\ a \Rightarrow a \to && Pat\ a\ () \\ &\quad PProd\ \ :: Pat\ a\ a' \to Pat\ b\ b'\ b'' \to && Pat\ (a, b)\ (a', b') \\ &\quad PLeft\ \ :: Pat\ a\ a' \to && Pat\ (Either\ a\ b)\ a' \\ &\quad PRight :: Pat\ b\ b' \to && Pat\ (Either\ a\ b)\ b' \\ &\quad PIn\ \ \ :: InOut\ a \Rightarrow Pat\ (F\ a)\ b \to && Pat\ a\ b \end{aligned}$$

Notice that an environment type is just a product of Var types — for example, the environment type computed for the cons pattern (3) is

$$(Var\ a, Var\ [a]) \tag{4}$$

We will discuss Var later, which is simply defined by

$$\textbf{newtype}\ Var\ a = Var\ a$$

Now we can define the (strongly typed) pattern matching operation:

```
deconstruct :: Pat a env → a → Maybe env
deconstruct PVar          x        = return (Var x)
deconstruct (PConst c)    x        = if c == x then return () else Nothing
deconstruct (l `PProd` r) (x, y)   = liftM2 (,) (deconstruct l x)
                                                (deconstruct r y)
deconstruct (PLeft p)     (Left x) = deconstruct p x
deconstruct (PLeft _)     _        = Nothing
deconstruct (PRight p)    (Right x) = deconstruct p x
deconstruct (PRight _)    _        = Nothing
deconstruct (PIn p)       x        = deconstruct p (out x)
```

and its inverse (which is total):

```
construct :: Pat a env → env → a
construct PVar          (Var x)      = x
construct (PConst c)    _            = c
construct (l `PProd` r) (envl, envr) = (construct l envl, construct r envr)
construct (PLeft p)     env          = Left  (construct p env)
construct (PRight p)    env          = Right (construct p env)
construct (PIn p)       env          = inn (construct p env)
```

Precisely speaking, we have

$$deconstruct\ p\ x = Just\ e \quad \Leftrightarrow \quad construct\ p\ e = x$$

for all $p :: Pat\ a\ env$, $x :: a$, and $e :: env$, establishing a (half-) partial isomorphism between env and a.

λ-expressions for rearrangement and their evaluation. Now consider view rearrangement, which evaluates a "simple" pattern-matching λ-expression on the view and continues execution with the transformed view. The body of the λ-expression refers to the variables appearing in the pattern. How do we represent such references? We have seen that an environment type is a product, i.e., a binary tree; to refer to a component in an environment, we can use a *path* that goes from the root to a sub-tree. In BiGUL, these paths are called *directions*:

```
data Direction env a where
  DVar   ::                    Direction (Var a) a
  DLeft  :: Direction a t → Direction (a, b) t
  DRight :: Direction b t → Direction (a, b) t
```

The type of a direction is indexed by the environment type it points into and the component type it points to. Note that the type of *DVar* is specified to work with only environment types marked with *Var*; this is for ensuring that a direction goes all the way down to an actual component at a variable position of the pattern, rather than stopping half-way and pointing to a sub-tree which include more than one component. For example, for the environment type (4) for the cons

pattern, only two directions are valid, namely *DLeft DVar* and *DRight DVar*, whereas *DVar* alone would point to the entire environment instead of one of the variable positions, and is ruled out by typechecking (in the sense that it is impossible for *DVar* to have type $Direction\ (Var\ a, Var\ [a])\ b$ for any b). It is easy to extract a component from an environment following a direction:

```
retrieve :: Direction env a → env → a
retrieve DVar        (Var x) = x
retrieve (DLeft d)   (x, _)  = retrieve d x
retrieve (DRight d)  (_, y)  = retrieve d y
```

Now we can define *expressions*, which are similar to patterns but include directions rather than variables, to represent the body of rearranging λ-expressions:

```
data Expr env a where
  EDir   :: Direction env a → Expr env a
  EConst :: (Eq a) ⇒ a →                  Expr env a
  EProd  :: Expr env a → Expr env b →     Expr env (a, b)
  ELeft  :: Expr env a →                  Expr env (Either a b)
  ERight :: Expr env b →                  Expr env (Either a b)
  EIn    :: (InOut a) ⇒ Expr env (F a) → Expr env a
```

For example, the rearranging λ-expression

$$\lambda(x : xs) \to (x, xs) \tag{5}$$

is represented by the cons pattern (3) and the pair expression

```
EProd (EDir (DLeft DVar)) (EDir (DRight DVar))
```
(6)

Evaluating an expression under an environment is similar to inverse pattern matching:

```
eval :: Expr env a → env → a
eval (EDir d)      env = retrieve d env
eval (EConst c)    env = c
eval (l `EProd` r) env = (eval l env, eval r env)
eval (ELeft e)     env = Left  (eval e env)
eval (ERight e)    env = Right (eval e env)
eval (EIn e)       env = inn (eval e env)
```

The type of *RearrV* is then:

```
RearrV :: Pat v env → Expr env v' → BiGUL s v' → BiGUL s v
```

Note that in the type of *RearrV*, the types of the pattern and expression share the same environment type index, ensuring that the directions in the expression can only refer to the variable positions in the pattern. And the *put* behaviour of *RearrV* is simply:

```
put (RearrV p e b) s v = do env ← deconstruct p v
                            put b s (eval e env)
```

Inverse evaluation of rearranging λ-expressions. For the *get* direction, after executing the inner BiGUL program to obtain an intermediate view, we should reverse the roles of the pattern and body in the rearranging λ-expression $\lambda p \to e$, using e as a (possibly non-linear) pattern to match the intermediate view, and computing the final view by evaluating p. For example, the *put* direction of view rearrangement with the λ-expression (5) turns a view list into a pair, on which the inner program operates; in the *get* direction, the inner program will extract from the source an intermediate view pair, which should be converted back to a list by the inverse λ-expression $\lambda(x, xs) \to (x : xs)$. In more detail, given an intermediate view pair (x, xs), we match it with the pair expression (6), and see that x is associated with the direction *DLeft DVar* and xs with *DRight DVar*. From such associations we can reconstruct an environment of type (4) with x and xs in the right places, and then we can evaluate the cons pattern (3) in this reconstructed environment, arriving at the final view $x : xs$.

In general, the intermediate view will be decomposed according to the body expression, and eventually each of its components will be paired with a direction indicating which variable position the component should go into in the reconstructed environment. To do the reconstruction, we can prepare a "container" which is similar to an environment except that the variable positions are initially empty. For each pair of a component and a direction, we try to put that component into the place in the container pointed to by the direction; if two components are put into the same position (indicating that the λ-expression uses a variable more than once), then they must be equal. In the end, we check that all places in the container are filled, and then use it as an environment to evaluate the pattern. Again, to compute the type of containers from a pattern, we add a third index to *Pat*:

```
data Pat a env con where
  PVar   :: Eq a ⇒ Pat a (Var a) (Maybe a)
  PConst :: Eq a ⇒ a →                          Pat a () ()
  PProd  :: Pat a a' a'' → Pat b b' b'' →       Pat (a, b) (a', b') (a'', b'')
  PLeft  :: Pat a a' a'' →                      Pat (Either a b) a' a''
  PRight :: Pat b b' b'' →                      Pat (Either a b) b' b''
  PIn    :: InOut a ⇒ Pat (F a) b c →           Pat a b c
```

A container type is just like an environment type except that the variable positions give rise to *Maybe* instead of *Var*. For the cons example, the computed container type is

$$(Maybe\ a, Maybe\ [a]) \tag{7}$$

The first step — matching a value with an expression — can then be implemented as:

```
uneval :: Pat a env con → Expr env b → b → con → Maybe con
uneval p (EDir d)    x        con = unevalD p d x con
uneval p (EConst c)  x        con = if c == x then return con
```

```
                                                    else  Nothing
uneval p (EProd l r) (x, y)      con = uneval p l x con >>= uneval p r y
uneval p (ELeft e)   (Left x)    con = uneval p e x con
uneval p (ELeft _)   x           con = Nothing
uneval p (ERight e)  (Right x)   con = uneval p e x con
uneval p (ERight _)  x           con = Nothing
uneval p (EIn e)     x           con = uneval p e (out x) con

unevalD :: Pat a env con → Direction env b → b → con → Maybe con
unevalD PVar         DVar        x (Just y)      = if x == y
                                                   then return (Just x)
                                                   else  Nothing
unevalD PVar         DVar        x Nothing       = return (Just x)
unevalD (PConst c)   _           x con           = return con
unevalD (l `PProd` r) (DLeft  d) x (conl, conr) = liftM (, conr)
                                                     (unevalD l d x conl)
unevalD (l `PProd` r) (DRight d) x (conl, conr) = liftM (conl, )
                                                     (unevalD r d x conr)
unevalD (PLeft p)    d           x con           = unevalD p d x con
unevalD (PRight p)   d           x con           = unevalD p d x con
unevalD (PIn p)      d           x con           = unevalD p d x con
```

This function *uneval* initially takes an empty container, which is generated by:

```
emptyContainer :: Pat v env con → con
emptyContainer PVar          = Nothing
emptyContainer (PConst c)    = ()
emptyContainer (l `PProd` r) = (emptyContainer l, emptyContainer r)
emptyContainer (PLeft p)     = emptyContainer p
emptyContainer (PRight p)    = emptyContainer p
emptyContainer (PIn p)       = emptyContainer p
```

And then we can try to convert a container to an environment, checking whether the container is full in the process:

```
fromContainerV :: Pat v env con → con → Maybe env
fromContainerV PVar          Nothing      = Nothing
fromContainerV PVar          (Just v)     = return (Var v)
fromContainerV (PConst c)    con          = return ()
fromContainerV (l `PProd` r) (conl, conr) = liftM2 (,)
                                              (fromContainerV l conl)
                                              (fromContainerV r conr)
fromContainerV (PLeft p)     con          = fromContainerV pat con
fromContainerV (PRight p)    con          = fromContainerV pat con
fromContainerV (PIn p)       con          = fromContainerV pat con
```

We can let out a sigh of relief once we successfully get hold of an environment, since the last step — inverse pattern matching — is total. To sum up:

$$
\begin{aligned}
get\ (RearrV\ p\ e\ b)\ s = \mathbf{do}\ v' &\leftarrow get\ b\ s \\
con &\leftarrow uneval\ p\ e\ v'\ (emptyContainer\ p) \\
env &\leftarrow fromContainerV\ p\ con \\
&return\ (construct\ p\ env)
\end{aligned}
$$

To be concrete, let us go through the steps of inverse rearranging in the cons example. Starting with an intermediate view (x, xs) and an empty container $(Nothing, Nothing)$ of type (7), *uneval* will invoke *unevalD* twice, the first time updating the container to $(Just\ x, Nothing)$ and the second time to $(Just\ x, Just\ xs)$. The resulting container is full, and thus $fromContainerV$ will successfully turn it into an environment ($Var\ x$, $Var\ xs$) of type (4), in which we evaluate the cons pattern (3) and obtain $x : xs$.

Conceptually, this is just reversing pattern matching and expression evaluation. To actually prove the well-behavedness, though, we need to reason about stateful computation (which is what *uneval* essentially is), which involves coming up with suitable invariants and proving that they are maintained throughout the computation.

It is interesting to mention that there would be a catch if we designed this combinator from the *get* direction: It is tempting to think that, since a rearranging λ-expression gives rise to a partial isomorphism, which can be lifted to a lens, we can simply compose the lens lifted from the isomorphism with the inner lens to give a lens semantics to $RearrV$. This would result in a redundant computation of an intermediate source which is immediately discarded, and now the success of the whole computation would unnecessarily depend on that of the intermediate source. To eliminate the redundant computation, we would need to use a special composition which composes a lens directly with an isomorphism on the right. Such a need would be hard to notice since the *get* behaviour of the two compositions are the same; that is, we really have to think in terms of *put* to see that the special composition is needed.

5.7 Summary

In one (long) section, we have examined the internals of BiGUL. After seeing the definition of (well-behaved) lenses that takes partiality explicitly into account, we have gone through the development of most of BiGUL's constructs and justified their well-behavedness — in the case of *Prod*, we have even seen a more formal and detailed well-behavedness proof. The *Case* construct is the most interesting one in terms of its design for achieving bidirectionality, while the rearrangement operations showcase more advanced datatype-generic programming techniques in Haskell for guaranteeing type safety. We will now shift our focus back to BiGUL programming, this time looking at some larger examples.

6 Position-, key-, and delta-based list alignment

In the next three sections, we will talk about some applications in BiGUL, starting with the list alignment problem. List alignment is one of the tasks that frequently

show up when developing bidirectional applications. When the source and view are both lists, and the *get* direction (i.e., the consistency relation) is a *map*, how do we put an updated view — the updates on which might involve insertions, deletions, in-place modifications, and reordering — into the source? This topic has be treated by Barbosa et al.'s matching lenses [1], which are special-purpose lenses into which several fixed alignment strategies are hard-coded. Below we will see how a number of alignment strategies can be programmed with BiGUL's general-purpose constructs, instead of having to extend the language with special-purpose alignment constructs.

Throughout the section, we use a concrete example to introduce three variations of list alignment. Suppose that we represent a payroll database as a list. (This is a slightly inadequate setting for explaining list alignment, because entries in a database are usually unordered. But let us assume that order matters.) Each entry is a triple — more precisely, a pair whose second component is again a pair — consisting of an identification number ("id" henceforth), a name, and a salary number:

```
type Source = (Id, (Name, Salary))
type Id     = Int
type Name   = String
type Salary = Int
```

For example, here is a sample payroll database:

```
employees :: [Source]
employees = [ (0, ("Zhenjiang", 1000))
            , (1, ("Josh"      , 400 ))
            , (2, ("Jeremy"    , 2000))]
```

Suppose that the human resource department is in charge of hiring or sacking employees but does not handle salary numbers, so the entries of the database are presented to them only as pairs of ids and names:

```
type View = (Id, Name)
```

For example, *employees* is presented to them as

```
[(0, "Zhenjiang"), (1, "Josh"), (2, "Jeremy")]
```

on which they can make modifications. It is easy to write a BiGUL program to synchronize the source and view elements:

```
bx :: BiGUL Source View
bx = $(rearrV [| λ(id, name) → (id, (name, ())) |])$
       Replace `Prod` (Replace `Prod` Skip (const ()))
```

The problem is then how the correspondences between sources and views in the two lists can be determined, so that *bx* can be applied to the right pairs.

6.1 Position-based alignment

As a first exercise, we consider the simplest strategy, which matches source and view elements by their positions in the lists. If the source list has more elements than the view list, the extra elements at the tail are simply dropped; if the source list has fewer elements, then new source elements have to be created, which we can specify as a function:

$$
\begin{aligned}
&cr :: \mathit{View} \to \mathit{Source} \\
&cr\ (i, n) = (i, (n, 0))
\end{aligned}
$$

The salary is set to zero, which could be taken care of by, say, the accounting department later. We will use bx and cr as the element synchronizer and creator respectively for our payroll database throughout this section, but our alignment programs will not be restricted to the payroll database setting — we will develop our alignment programs generically, setting the source and view types as polymorphic type parameters (s and v below) and also the element synchronizer and element creator as parameters (b and c below), so the alignment programs can be widely applicable. Here is how we implement position-based alignment, which is fairly standard:

$$
\begin{aligned}
&\mathit{posAlign} :: (\mathit{Show}\ s, \mathit{Show}\ v) \Rightarrow \mathit{BiGUL}\ s\ v \to (v \to s) \to \mathit{BiGUL}\ [s]\ [v] \\
&\mathit{posAlign}\ b\ c = \mathit{Case} \\
&\quad [\ ^{\$}(\mathit{normalSV}\ ^{\mathbb{P}}[\![\,[\,]\,]\!]\ ^{\mathbb{P}}[\![\,[\,]\,]\!]\ ^{\mathbb{P}}[\![\,[\,]\,]\!]) \\
&\qquad \Longrightarrow\ ^{\$}(\mathit{update}\ ^{\mathbb{P}}[\![\,[\,]\,]\!]\ ^{\mathbb{P}}[\![\,[\,]\,]\!]\ ^{\mathbb{D}}[\![\,]\!]) \\
&\quad ,\ ^{\$}(\mathit{normalSV}\ ^{\mathbb{P}}[\![\,\_ : \_\,]\!]\ ^{\mathbb{P}}[\![\,\_ : \_\,]\!]\ ^{\mathbb{P}}[\![\,\_ : \_\,]\!]) \\
&\qquad \Longrightarrow\ ^{\$}(\mathit{update}\ ^{\mathbb{P}}[\![\,x : xs\,]\!]\ ^{\mathbb{P}}[\![\,x : xs\,]\!]\ ^{\mathbb{D}}[\![\,x = b; xs = \mathit{posAlign}\ b\ c\,]\!]) \\
&\quad ,\ ^{\$}(\mathit{adaptiveSV}\ ^{\mathbb{P}}[\![\,\_ : \_\,]\!]\ ^{\mathbb{P}}[\![\,[\,]\,]\!]) \\
&\qquad \Longrightarrow \lambda\_\ \_ \to [\,] \\
&\quad ,\ ^{\$}(\mathit{adaptiveSV}\ ^{\mathbb{P}}[\![\,[\,]\,]\!]\ ^{\mathbb{P}}[\![\,\_ : \_\,]\!]) \\
&\qquad \Longrightarrow \lambda\_\ (v : \_) \to [\,c\ v\,] \\
&\quad]
\end{aligned}
$$

The normal branches deal with the situations where both lists are empty or non-empty, and the adaptive branches remove or create elements when the lengths of the two lists differ.

The *get* direction of *posAlign* does exactly what we want it to do:

```
*Alignment> get (posAlign bx cr) employees
Just [(0,"Zhenjiang"),(1,"Josh"),(2,"Jeremy")]
```

It should be quite obvious, though, that the *put* direction is not so useful for our purpose. If we sack Josh:

$$
\begin{aligned}
&\mathit{updatedEmployees0} :: [\,\mathit{View}\,] \\
&\mathit{updatedEmployees0} = [(0, \texttt{"Zhenjiang"}), (2, \texttt{"Jeremy"})]
\end{aligned}
$$

then the database will be updated to:

```
*Alignment> put (posAlign bx cr) employees updatedEmployees0
Just [(0,("Zhenjiang",1000)),(2,("Jeremy",400))]
```

where Jeremy inadvertently gets Josh's original salary. Even if we do not remove any employee, we may still want to reorder them:

$$\begin{aligned} &\mathit{updatedEmployees1} :: [\mathit{View}] \\ &\mathit{updatedEmployees1} = [(2, \texttt{"Jeremy"}), (0, \texttt{"Zhenjiang"}), (1, \texttt{"Josh"})] \end{aligned}$$

and now everyone gets the wrong salary:

```
*Alignment> put (posAlign bx cr) employees updatedEmployees1
Just [(2,("Jeremy",1000)),(0,("Zhenjiang",400)),(1,("Josh",2000)
    )]
```

This first exercise shows that the alignment problem is inherently one that should be solved from the *put* direction. It is easy to implement the *get* direction correctly, but what matters is the *put* behavior.

6.2 Key-based alignment

A more reasonable strategy is to match source and view elements by some *key* value. In our example, we can use the id as the key. Key-based alignment might seem much more complex than position-based alignment, but, in fact, we can just revise *posAlign* to get a BiGUL program for key-based alignment!

First of all, we need to somehow obtain the keys. In our example, on both the source and view we can use *fst* to extract the key value. In general, we can further parametrize the alignment program with key extraction functions $ks :: s \to k$ and $kv :: v \to k$ for some type k of key values:

$$\begin{aligned} &\mathit{keyAlign} :: (\mathit{Show}\ s, \mathit{Show}\ v, \mathit{Eq}\ k) \\ &\qquad \Rightarrow (s \to k) \to (v \to k) \to \mathit{BiGUL}\ s\ v \to (v \to s) \to \mathit{BiGUL}\ [s]\ [v] \end{aligned}$$

The first normal branch of *posAlign* still works perfectly. As for the second normal branch, we should revise the main condition to also require that the head elements of the two lists have the same key value:

$$\lambda(s : ss)\ (v : vs) \to ks\ s == kv\ v$$

The first adaptive branch, again, works well. The second adaptive branch, on the other hand, is no longer applicable: since the main condition of the second normal branch has been tightened, it is no longer the case that this adaptive branch will receive only empty source lists. In fact, whether the source list is empty or not is irrelevant here — what matters now is whether the key of the first view is in the source list. If it is, then we bring the (first) source element with the same key value to the head position, and the second normal branch can take over; otherwise, we create a new source element. This gives us key-based alignment:

```
keyAlign :: forall s v k. (Show s, Show v, Eq k)
         ⇒ (s → k) → (v → k) → BiGUL s v → (v → s) → BiGUL [s] [v]
keyAlign ks kv b c = Case
  [ $(normalSV ℙ⟦ [] ⟧ ℙ⟦ [] ⟧ ℙ⟦ [] ⟧)
      ⟹ $(update ℙ⟦ [] ⟧ ℙ⟦ [] ⟧ ℙ⟦ [] ⟧ 𝔻⟦ ⟧)
  , $(normal ⟦ λ(s : ss) (v : vs) → ks s == kv v ⟧ ℙ⟦ _ : _ ⟧)
      ⟹ $(update ℙ⟦ x : xs ⟧ ℙ⟦ x : xs ⟧ 𝔻⟦ x = b; xs = keyAlign ks kv b c ⟧)
  , $(adaptiveSV ℙ⟦ _ : _ ⟧ ℙ⟦ [] ⟧)
      ⟹ λ_ _ → []
  , $(adaptive ⟦ λss (v : vs) → kv v ∈ map ks ss ⟧)
      ⟹ λss (v : _) → uncurry (:) (extract (kv v) ss)
  , $(adaptiveSV ℙ⟦ _ ⟧ ℙ⟦ _ : _ ⟧)
      ⟹ λss (v : _) → c v : ss
  ]
  where
    extract :: k → [s] → (s, [s])
    extract k (x : xs) | ks x == k   = (x, xs)
                       | otherwise = let (y, ys) = extract k xs
                                     in  (y, x : ys)
```

Note that the program does not assume that keys are unique — if there are n view elements having the same key, then the first n source elements with that key will be retained and synchronised with those view elements in order. This strategy is a somewhat arbitrary choice, but can be changed by, for example, using a different *extract*. (On the other hand, in practice it is probably wiser to enforce uniqueness of keys, so that we can be sure which source element will be used to match a view element, and do not need to rely on the choices made by the implementation.)

Back to our payroll database example. The *get* direction behaves the same:

```
*Alignment> get (keyAlign fst fst bx cr) employees
Just [(0,"Zhenjiang"),(1,"Josh"),(2,"Jeremy")]
```

Unlike position-based alignment, view element deletion can now be reflected correctly:

```
*Alignment> put (keyAlign fst fst bx cr) employees
    updatedEmployees0
Just [(0,("Zhenjiang",1000)),(2,("Jeremy",2000))]
```

And reordering as well:

```
*Alignment> put (keyAlign fst fst bx cr) employees
    updatedEmployees1
Just [(2,("Jeremy",2000)),(0,("Zhenjiang",1000)),(1,("Josh",400)
    )]
```

So it seems that key-based alignment is just what we need. Indeed, key-based alignment usually works well, but there is an important assumption: the key

values should not be changed. If, for example, we decide to assign a different id to Josh:

$$updatedEmployees2 :: [View]$$
$$updatedEmployees2 = [(0, \texttt{"Zhenjiang"}), (100, \texttt{"Josh"}), (1, \texttt{"Jeremy"})]$$

Then the effect is the same as sacking Josh and then hiring him again, and his salary is thus reset:

```
*Alignment> put (keyAlign fst fst bx cr) employees
    updatedEmployees2
Just [(0,("Zhenjiang",1000)),(100,("Josh",0)),(1,("Jeremy",400))
    ]
```

The problem is that we cannot distinguish modification from deletion and insertion pairs. To be able to have such distinction, we need the notion of *deltas* [4], which allows us to explicitly represent and keep track of the correspondences between source and view elements.

6.3 Delta-based alignment

A (horizontal) *delta* between a source list and a view list is a list of pairs of corresponding positions:

$$\textbf{type } Delta = [(Int, Int)]$$

For example, the delta we have in mind between the source list *employees* and the view list *updatedEmployees2* is $[(0,0),(1,1),(2,2)]$, which, in particular, associates the source and view entries for Josh since $(1,1)$ is included, instead of $[(0,0),(2,2)]$, which indicates that Josh's source entry does not correspond to any view entry and should be deleted, and that Josh's view entry does not correspond to any source entry and is thus new. Deltas can easily represent reordering as well. For example, we would supply the delta between *employees* and *updatedEmployees1* as $[(0,1),(1,2),(2,0)]$, associating the 0th element in the source — namely the one for Zhenjiang — with the 1st element in the view, and so on. Comparing this treatment with the key-based one, we might say that keys are "poor man's correspondences", which are not as explicit and unambiguous as *Delta*. A *Delta* between source and view lists directly describes the accurate correspondences between them, whereas with keys the correspondences can only be inferred, sometimes inaccurately.

So the input now includes not only source and view lists but also a delta between them. Recall key-based alignment: what it does overall is to bring the first matching source element to the front for each view element, so the source list is updated throughout execution, with the links between the source and view elements gradually and implicitly restored. If we are doing something similar with delta-based alignment, then when the source list is updated, the delta should also be updated to reflect the restored consistency. This suggests that the delta

should be paired with the source list, so that it can be updated. The type we use for the delta-based alignment program is thus:

```
deltaAlign :: (Show s, Show v)
           ⇒ BiGUL s v → (v → s) → BiGUL ([s], Delta) [v]
```

Here we take a simpler approach to implementing *deltaAlign*, analyzing the problem into just two cases: The delta can tell us either that the source and view elements are all in correspondence, in which case a simple position-based alignment suffices, or that we need to do some rearrangement of the source elements, which can be done by adaptation. In BiGUL:

```
idDelta     :: [s] → Delta
idDelta ss  = [(i, i) | i ← [0 .. length ss]]
deltaAlign :: (Show s, Show v)
           ⇒ BiGUL s v → (v → s) → BiGUL ([s], Delta) [v]
deltaAlign b c = Case
  [$(normal ⟦ λ(ss, d) vs → length ss == length vs ∧ d == idDelta ss ⟧
            ℙ⟦ _ ⟧)
    ⟹ $(rearrV ⟦ λvs → (vs, ()) ⟧)$posAlign b c `Prod` Skip (const ())
  ,$(adaptive ⟦ λ_ _ → otherwise ⟧)
    ⟹ λ(ss, d) vs →
         let d'  = map swap d
             ss' = [maybe (c v) (ss!!) (lookup j d') | (v, j) ← zip vs [0 ..]]
         in (ss', idDelta ss')
  ]
```

The source and view lists are in full correspondence if and only if they have the same length and the delta associates all their elements positionally. This full positional delta can be computed by *idDelta*. When this is the case, it suffices to call *posAlign* to carry out element-wise synchronization, since no rearrangement is required. Otherwise, we enter the adaptive branch, which constructs a new source list in full correspondence with the view list, drawing elements from the original source list or creating new ones as the delta dictates. The new source list is in full correspondence with the view list, so the delta we pair with it is the one computed by *idDelta*.

Only when performing *put* does a delta make sense. When performing *get*, however, we still need to supply a delta since it is part of the source; but there is a natural choice, namely *idDelta*. So we define:

```
putDeltaAlign :: (Show s, Show v)
              ⇒ BiGUL s v → (v → s) → [s] → Delta → [v] → Maybe [s]
putDeltaAlign b c ss d vs = fmap fst (put (deltaAlign b c) (ss, d) vs)
getDeltaAlign :: (Show s, Show v)
              ⇒ BiGUL s v → (v → s) → [s] → Maybe [v]
getDeltaAlign b c ss = get (deltaAlign b c) (ss, idDelta ss)
```

It is easy to prove that, given the same b and c, these two functions do form a lens. The key observation is that the delta produced by *put* (*deltaAlign b c*) is necessarily the one computed by *idDelta*, so, for example, in PUTGET, throwing away the delta in the *put* direction is fine because it can be recomputed by *idDelta*, and the *get* direction can resume from exactly the same source pair.

Back to our example. We can now update Josh's id without resetting his salary by providing a full delta indicating that there are only in-place updates:

```
*Alignment> putDeltaAlign bx cr employees [(0,0), (1,1), (2,2)]
    updatedEmployees2
Just [(0,("Zhenjiang",1000)),(100,("Josh",400)),(1,("Jeremy
    ",2000))]
```

Besides obvious modifications like reordering, we can also do some fairly subtle modifications now: If we actually sack Josh and replace him with a new Josh (inheriting the original Josh's id) whose salary should be reset (to be re-considered by the accounting department), we can say so by providing a partial delta:

```
*Alignment> putDeltaAlign bx cr employees [(0,0), (2,2)] =<<
    getDeltaAlign bx cr employees
Just [(0,("Zhenjiang",1000)),(1,("Josh",0)),(2,("Jeremy",2000))]
```

One alignment to rule them all. Where do deltas come from? In general, we may provide a special view editor which monitors how the view is modified and produces a suitable delta. But in more specialized scenarios, deltas can simply be computed by, for example, comparing the source and view. We can formalize this delta computation as:

$$\textbf{type}\ \mathit{DeltaStrategy}\ s\ v = [s] \to [v] \to \mathit{Delta}$$

and further parametrize *putDeltaAlign*:

$$\begin{aligned}&\mathit{putDeltaAlignS}\ ::\ (\mathit{Show}\ s, \mathit{Show}\ v) \Rightarrow \mathit{DeltaStrategy}\ s\ v\\ &\qquad\qquad \to \mathit{BiGUL}\ s\ v \to (v \to s) \to [s] \to [v] \to \mathit{Maybe}\ [s]\\ &\mathit{putDeltaAlignS}\ \mathit{dst}\ b\ c\ \mathit{ss}\ \mathit{vs} = \mathit{putDeltaAlign}\ b\ c\ \mathit{ss}\ (\mathit{dst}\ \mathit{ss}\ \mathit{vs})\ \mathit{vs}\end{aligned}$$

Position-based and key-based alignment can then be seen as special cases of delta-based alignment using specific delta-computing strategies. For position-based alignment, we simply compute the identity delta:

$$\begin{aligned}&\mathit{byPosition} :: \mathit{DeltaStrategy}\ s\ v\\ &\mathit{byPosition}\ \mathit{ss}\ \_ = \mathit{idDelta}\ \mathit{ss}\end{aligned}$$

And for key-based alignment, we compute a delta associating source and view elements with the same key:

$$\begin{aligned}&\mathit{byKey} :: \mathit{Eq}\ k \Rightarrow (s \to k) \to (v \to k) \to \mathit{DeltaStrategy}\ s\ v\\ &\mathit{byKey}\ \mathit{ks}\ \mathit{kv}\ \mathit{ss}\ \mathit{vs} =\end{aligned}$$

```
  let sis = zip ss [0..]
  in  catMaybes [ fmap (λ(_, i) → (i, j)) (find (λ(s, _) → ks s == kv v) sis)
                | (v, j) ← zip vs [0..]]
```

We can check that these strategies indeed give us position-based and key-based alignment:

```
*Alignment> putDeltaAlignS byPosition bx cr employees
    updatedEmployees0
Just [(0,("Zhenjiang",1000)),(2,("Jeremy",400))]
*Alignment> putDeltaAlignS (byKey fst fst) bx cr employees
    updatedEmployees1
Just [(2,("Jeremy",2000)),(0,("Zhenjiang",1000)),(1,("Josh",400)
    )]
```

7 Bidirectionalizing relational queries with BiGUL

In work on relational databases, the view-update problem is about how to translate update operations on the view table to corresponding update operations on the source table properly. Relational lenses [3] try to solve this problem by providing a list of combinators that let the user write get functions (queries) with specified updated policies for put functions (updates); however this can only provide limited control of update policies. To resolve this problem, we define a new library BRUL [16], where two *putback*-based combinators (operators) are designed to specify update policies, from which forward queries (selection, projection, join) can be automatically derived.

- *align* is to update a source list with a view list by aligning part of source elements filtered by a predicate with view elements according to a matching criteria between source element and view element;
- *unjoin* is to decompose a join view to update two sources.

In this section, we will focus on *align*. As will be seen in Section 7.3, it can describe more flexible update strategies (related to selection/projection queries) than relational lenses, while the well-behavedness is guaranteed for free.

7.1 Relational database representation

A relational table (RT) is denoted by a list of records (where the order does not really matter), and each record ($Record$) is denoted by a list of attributes of type $RType$, which could be an integer, a string, a floating point number, or a double-precision floating point number.

```
type RT     = [Record]
type Record = [RType]
data RType  = RInt Int
```

Track	Date	Rating	Album	Quantity
Lullaby	1989	3	Galore	2
Lullaby	1989	3	Show	3
Lovesong	1989	5	Galore	2
Lovesong	1989	5	Paris	4
Trust	1992	4	Wish	5

Fig. 1. Source table

```
  | RString String
  | RFloat Float
  | RDouble Double
deriving (Show, Eq, Ord)
```

To allow pattern matching on the newly defined algebraic data type *RType* in BiGUL, we need to add the following declaration.

```
deriveBiGULGeneric ''RType
```

Consider the table in Figure 1 that stores five music track records, and each record contains its Track name, release Date, Rating, Album, and the Quantity of this Album. We can represent it as follows, where all the records have the same structure.

```
s = [[RString "Lullaby" , RInt 1989, RInt 3, RString "Galore", RInt 1]
    ,[RString "Lullaby" , RInt 1989, RInt 3, RString "Show"  , RInt 3]
    ,[RString "Lovesong", RInt 1989, RInt 5, RString "Galore", RInt 1]
    ,[RString "Lovesong", RInt 1989, RInt 5, RString "Paris" , RInt 4]
    ,[RString "Trust"   , RInt 1992, RInt 4, RString "Wish"  , RInt 5]
    ]
```

7.2 Relation Alignment

The alignment of two relational tables, which is related by a selection/projection query, is similar to the key-based list alignment in Section 6. The difference is that we need to consider filtering on (i.e., selection of) the source records based on a condition.

Let us see how to extend *keyAlign* (in Section 6) to implement the new align *pAlign* that can deal with filtering of source elements. We extend *keyAlign* with two new arguments; one is the predicate p for filtering source elements, and the other is the function h for hiding/concealing source elements if their corresponding elements are removed from the view. As seen below, *pAlign* has a similar case structure as that of *keyAlign*, except that we refine the third case of *keyAlign* into two cases (the third and the fourth cases of *pAlign*): the third case says that if the view v is empty but the first record in the source satisfies p, we should hide this record using h, and the fourth case says that if the first

record of the source does not satisfy p, we simply ignore it and continue with the remaining records.

$$
\begin{array}{l}
pAlign :: forall\ s\ v\ k.\ (Show\ s, Show\ v, Eq\ k) \\
\qquad \Rightarrow (s \to Bool) \quad \text{-- predicate} \\
\qquad \to (s \to k) \to (v \to k) \to BiGUL\ s\ v \to (v \to s) \\
\qquad \to (s \to Maybe\ s) \quad \text{-- conceal function} \\
\qquad \to BiGUL\ [s]\ [v] \\
pAlign\ p\ ks\ kv\ b\ c\ h = Case \\
\quad [\$(normalSV\ ^{\mathbb{P}}[\![\,[\,]\,]\!]\ ^{\mathbb{P}}[\![\,[\,]\,]\!]\ ^{\mathbb{P}}[\![\,[\,]\,]\!]) \\
\qquad \Longrightarrow \$(update\ ^{\mathbb{P}}[\![\,[\,]\,]\!]\ ^{\mathbb{P}}[\![\,[\,]\,]\!]\ ^{\mathbb{D}}[\![\,]\!]) \\
\quad , \$(normal\ [\![\,\lambda(s:ss)\ (v:vs) \to p\ s \wedge ks\ s == kv\ v\,]\!]\ [\![\,\lambda(s:ss) \to p\ s\,]\!]) \\
\qquad \Longrightarrow \$(update\ ^{\mathbb{P}}[\![\,x:xs\,]\!]\ ^{\mathbb{P}}[\![\,x:xs\,]\!]\ ^{\mathbb{D}}[\![\,x = b; xs = pAlign\ p\ ks\ kv\ b\ c\ h\,]\!]) \\
\quad , \$(adaptive\ [\![\,\lambda(s:ss)\ v \to p\ s \wedge null\ v\,]\!]) \\
\qquad \Longrightarrow \lambda(s:ss)\ v \to maybe\ [\,]\ (:[\,])\ (h\ s) +\!\!+ ss \\
\quad , \$(normal\ [\![\,\lambda(s:ss)\ v \to not\ (p\ s)\,]\!]\ [\![\,\lambda(s:ss) \to not\ (p\ s)\,]\!]) \\
\qquad \Longrightarrow \$(update\ ^{\mathbb{P}}[\![\,\_:xs\,]\!]\ ^{\mathbb{P}}[\![\,xs\,]\!]\ ^{\mathbb{D}}[\![\,xs = pAlign\ p\ ks\ kv\ b\ c\ h\,]\!]) \\
\quad , \$(adaptive\ [\![\,\lambda ss\ (v:vs) \to kv\ v \in map\ ks\ (filter\ p\ ss)\,]\!]) \\
\qquad \Longrightarrow \lambda ss\ (v{:}\ \_) \to uncurry\ (:)\ (extract\ (kv\ v)\ ss) \\
\quad , \$(adaptiveSV\ ^{\mathbb{P}}[\![\,\_\,]\!]\ ^{\mathbb{P}}[\![\,\_{:}\ \_\,]\!]) \\
\qquad \Longrightarrow \lambda ss\ (v{:}\ \_) \to filterCheck\ p\ (c\ v) : ss \\
\quad] \\
\quad \textbf{where} \\
\qquad extract :: k \to [s] \to (s, [s]) \\
\qquad extract\ k\ (x:xs) \mid p\ x \wedge ks\ x == k = (x, xs) \\
\qquad\qquad\qquad\quad \mid otherwise \qquad\quad = \textbf{let}\ (y, ys) = extract\ k\ xs \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \textbf{in}\ \ (y, x:ys) \\
\qquad filterCheck\ p\ v \mid p\ v \qquad\quad = v \\
\qquad\qquad\qquad\quad \mid otherwise = error\ \texttt{"error in filter checking"}
\end{array}
$$

To test, recall the example in Section 6. Consider the following use of *pAlign*, denoting that the view is selected from those records from the source whose salary is greater than 1000, and that if a view record is removed, the corresponding record in the source will be removed (and thus hidden).

$$
\begin{array}{l}
pSelProj = pAlign\ (\lambda(k,(n,s)) \to s > 1000)\ fst\ fst\ bx\ cr'\ (const\ Nothing) \\
\quad \textbf{where}\ cr'\ (k,n) = (k,(n,2000))
\end{array}
$$

We have:

```
*Brul> get pSelProj employees
Just [(2,"Jeremy")]
*Brul> put pSelProj employees updatedEmployees0
Just [(0,("Zhenjiang",1000)),(1,("Josh",400)),(0,("Zhenjiang
    ",2000)),(2,("Jeremy",2000))]
```

7.3 Describing update policies in selection/projection

With *pAlign*, we can describe various update policies for the selection/projection queries. To be concrete, consider the following selection/projection query:

select *Track*, *Rating*, *Album*, *Quantity* **as** v
from s
where $Quantity > 2$

which extracts the track, rating, album and quality information from those music tracks in the source s whose quantity is greater than 2. Let us see how to write a single BiGUL program so that its *get* does the above query and its *put* describes a specific update policy.

The first BiGUL program is *u0* below.

```
u0    :: RType → BiGUL [Record] [Record]
u0 d = pAlign
         (λr → (r !! 4) > RInt 2)
         (λs → (s !! 0, s !! 3))
         (λv → (v !! 0, v !! 2))
         $(update ℙ⟦(t : _ : r : a : q : [])⟧
                  ℙ⟦(t : r : a : q : [])⟧
                  𝔻⟦t = Replace; r = Replace; a = Replace; q = Replace⟧)
         (λ(t : r : a : q : []) → (t : d : r : a : q : []))
         (const Nothing)
```

It tries to match the source records whose *Quantity* is greater than 2 with the view records by the key (*Track*, *Album*). There are three cases:

- A source record is matched with a view record: we first use a rearrangement function to rearrange the view from a four-element list $[t, r, a, q]$ to a five-element list $[t, \_, r, a, q]$ with the second element matched against a widecard. This rearrangement function reshapes the view to match the shape of the source. Then, the element in the source is *Replace*d by the corresponding element in the view.
- A view record that has no matching source record: a new source record is created with a default value d filled into the Date.
- A source record that has no matching view record: we simply delete this record by returning *Nothing*.

Now if we wish to hide the source record by setting its *Quantity* to 0 rather than deleting it if it has no matching view record, we could simply change the last line of *u0* and get *u1* as follows.

```
u1    :: RType → BiGUL [Record] [Record]
u1 d = pAlign
         (λr → (r !! 4) > RInt 2)
```

$$
\begin{array}{l}
(\lambda s \to (s \mathbin{!!} 0, s \mathbin{!!} 3)) \\
(\lambda v \to (v \mathbin{!!} 0, v \mathbin{!!} 2)) \\
{}^{\$}(\mathit{update}\ {}^{\mathbb{P}}[\![\, (t : \_ : r : a : q : [\,]) \,]\!] \\
\qquad\qquad {}^{\mathbb{P}}[\![\, (t : r : a : q : [\,]) \,]\!] \\
\qquad\qquad {}^{\mathbb{D}}[\![\, t = \mathit{Replace}; r = \mathit{Replace}; a = \mathit{Replace}; q = \mathit{Replace} \,]\!]) \\
(\lambda(t : r : a : q : [\,]) \to (t : d : r : a : q : [\,])) \\
(\lambda(t : d : r : a : \_ : [\,]) \to \mathit{Just}\ (t : d : r : a : \mathit{RInt}\ 0 : [\,]))
\end{array}
$$

To test, let us see some concrete running examples of using *u0*. Recall *s* defined in Section 7.1. We can confirm that *get* performs the query given at the start of this subsection.

```
*Brul> get (u0 (RInt 2000)) s
Just [[RString "Lullaby",RInt 3,RString "Show",RInt 3],[RString "
    Lovesong",RInt 5,RString "Paris",RInt 4],[RString "Trust",RInt 4,
    RString "Wish",RInt 5]]
```

Now suppose that we change the above result (view) to the following by raising the rating of *Lullaby* from 3 to 4, raising the quality of *lovesong* from 4 to 7, and deleting *Trust*:

$$
\begin{array}{l}
v = [[\mathit{RString}\ \texttt{"Lullaby"}, \mathit{RInt}\ 4, \mathit{RString}\ \texttt{"Show"}, \mathit{RInt}\ 3] \\
\quad , [\mathit{RString}\ \texttt{"Lovesong"}, \mathit{RInt}\ 5, \mathit{RString}\ \texttt{"Paris"}, \mathit{RInt}\ 7] \\
\quad]
\end{array}
$$

We can reflect these changes to the source by performing *put* with *u0*.

```
*Brul> put (u0 (RInt 2000)) s v
Just [[RString "Lullaby",RInt 1989,RInt 3,RString "Galore",RInt 1],[
    RString "Lullaby",RInt 1989,RInt 4,RString "Show",RInt 3],[RString
    "Lovesong",RInt 1989,RInt 5,RString "Galore",RInt 1],[RString "
    Lovesong",RInt 1989,RInt 5,RString "Paris",RInt 7]]
```

In the updated source, the changes of rating and quality are correctly reflected, and the music track *Trust* is removed. Note that we may reflect the changes to the source by performing *put* with *u1*, another update strategy, and we will keep the music track *Trust* while setting its quality to be 0.

```
*Brul> put (u1 (RInt 2000)) s v
Just [[RString "Lullaby",RInt 1989,RInt 3,RString "Galore",RInt 1],[
    RString "Lullaby",RInt 1989,RInt 4,RString "Show",RInt 3],[RString
    "Lovesong",RInt 1989,RInt 5,RString "Galore",RInt 1],[RString "
    Lovesong",RInt 1989,RInt 5,RString "Paris",RInt 7],[RString "Trust
    ",RInt 1992,RInt 4,RString "Wish",RInt 0]]
```

8 Parsing and reflective printing

When we mention the *front-end* of a compiler, we usually think of a *parser* that turns *concrete syntax*, which is designed to be programmer-friendly and provides

convenient syntactic sugar, into *abstract syntax*, which is concise, structured, and easily manipulable by the compiler back-end. There is another direction, though, in which a *printer* turns abstract syntax back into concrete syntax. This is useful, for example, for reporting the result of compiler optimizations done on abstract syntax to the programmer, who knows only concrete syntax. In this case, though, we would want to print the optimized program in a form that is as close to the original program as possible, so the programmer can spot what has changed — and not changed — correctly and more easily. This is where the notion of *reflective printing* comes in: By taking both the original concrete program and the optimized abstract program as input, we can try to retain the look of the original program as much as possible. Below we will use a simplified arithmetic expression language to explain how reflective printing can be implemented in BiGUL.

8.1 Well-behavedness

It is probably obvious that the idea of reflective printing comes from *put* transformations; parsing, then, is the *get* direction. Before we proceed to implement parsing and reflective printing in BiGUL, a natural question to ask is: is well-behavedness meaningful in the context of parsing of reflective printing? The answer is yes, especially for PutGet: An abstract syntax tree (AST) may be thought of as a concise and canonical representation of a concrete program, so it would be strange if a concrete program printed from an AST could not be parsed back to the same AST. GetPut, on the other hand, is in fact not strong enough for our purpose, as it only says that, when an AST is unmodified, printing it reflectively to the original program does not change anything, whereas we would have liked to also say that "small" changes to the AST lead to only "small" changes to the concrete program. That is, we would like reflective printing to conform to some sort of least-change principle, a topic which is still unsettled and actively investigated by the BX community. It is at least a good start to have GetPut, though. We thus conclude that BiGUL is indeed a suitable language for implementing reflective printers and corresponding parsers.

8.2 Additive expressions

Here we use a minimal example which is simple and yet can demonstrate what reflective printing is capable of. Consider the following abstract syntax of arithmetic expressions consisting of integer constants, addition, and subtraction:

```
data Arith = Num Int
           | Add Arith Arith
           | Sub Arith Arith
         deriving Show
```

This is a nice representation for the compiler, but we cannot expect the programmer to write something like "*Sub* (*Num* 1) (*Add* (*Num* 2) (*Num* 3))", and should

provide a concrete syntax so that they can write "$1 - (2 + 3)$". Such a concrete syntax is usually defined in terms of a BNF grammar:

```
Exp    → Exp '+' Factor
       | Exp '-' Factor
       | Factor
Factor → Int
       | '-' Factor
       | '(' Exp ')'
```

The two-level structure of *Exp* and *Factor* ensures that plus and minus associate to the left by default; to change association, we should use parentheses. And, to spice up the problem a little, we allow minus to be used also as a negative sign, as specified by the second production rule for *Factor*. BiGUL deals with structured data only, so we should represent a string generated using this grammar as a concrete syntax tree of the following type:

```
data Exp = Plus Exp Factor
         | Minus Exp Factor
         | EF Factor
         | ENull
data Factor = Lit Int
            | Neg Factor
            | Paren Exp
            | FNull
```

Again, we need to provide one *deriveBiGULGeneric* statement for each of the above datatypes to allow BiGUL to operate on them:

```
deriveBiGULGeneric ''Arith
deriveBiGULGeneric ''Exp
deriveBiGULGeneric ''Factor
```

Apart from the *Null* constructors, which are inserted to represent incomplete trees that can occur during reflective printing, these two datatypes are in direct correspondence with the grammar, so it is easy to recover the string from a concrete syntax tree:

```
instance Show Exp where
  show (Plus  e f) = show e ++ "+" ++ show f
  show (Minus e f) = show e ++ "-" ++ show f
  show (EF      f) = show f
  show  ENull      = "."
instance Show Factor where
  show (Lit   n) = show n
  show (Neg   f) = "-" ++ show f
```

$$
\begin{aligned}
&show\ (Paren\ \ e) = \texttt{"("} +\!\!+ show\ e +\!\!+ \texttt{")"} \\
&show\ \ FNull\ \ \ \ = \texttt{"."}
\end{aligned}
$$

Conversely, using modern parser technologies like Haskell's `parsec` parser combinator library, we can easily implement a "concrete parser" that turns a string into a concrete syntax tree:

$$parseExp :: String \to Exp$$

The rest of the job is then to write a BiGUL program between *Exp* and *Arith*.

8.3 Reflective printing in BiGUL

The program is basically a case analysis: For example, when the concrete side is a plus and the abstract side is an addition, they match, and we can go into their sub-trees recursively. For the concrete side, the right sub-tree is of type *Factor* instead of *Exp*, so in fact we will write two (mutually recursive) programs:

$$
\begin{aligned}
&pExpArith\ \ \ \ :: BiGUL\ Exp\ Arith \\
&pExpArith\ \ \ \ = Case\ \bot \\
&pFactorArith :: BiGUL\ Factor\ Arith \\
&pFactorArith = Case\ \bot
\end{aligned}
$$

The branch for plus and addition can then be written as:

$$^{\$}(update\ ^{\mathbb{P}}[\![\ Plus\ l\ r\ ]\!]\ ^{\mathbb{P}}[\![\ Add\ l\ r\ ]\!]\ ^{\mathbb{D}}[\![\ l = pExpArith; r = pFactorArith\ ]\!])$$

Following the same line of thought, we can fill in other branches to relate all abstract constructors with concrete production rules:

$$
\begin{aligned}
&pExpArith :: BiGUL\ Exp\ Arith \\
&pExpArith = Case \\
&\quad [\,^{\$}(normalSV\ ^{\mathbb{P}}[\![\ Plus\ \_\ \_\ ]\!]\ ^{\mathbb{P}}[\![\ Add\ \_\ \_\ ]\!]\ ^{\mathbb{P}}[\![\ Plus\ \_\ \_\ ]\!]) \\
&\qquad \Longrightarrow\ ^{\$}(update\ ^{\mathbb{P}}[\![\ Plus\ l\ r\ ]\!]\ ^{\mathbb{P}}[\![\ Add\ l\ r\ ]\!] \\
&\qquad\qquad\qquad\ ^{\mathbb{D}}[\![\ l = pExpArith; r = pFactorArith\ ]\!]) \\
&\quad ,\,^{\$}(normalSV\ ^{\mathbb{P}}[\![\ Minus\ \_\ \_\ ]\!]\ ^{\mathbb{P}}[\![\ Sub\ \_\ \_\ ]\!]\ ^{\mathbb{P}}[\![\ Minus\ \_\ \_\ ]\!]) \\
&\qquad \Longrightarrow\ ^{\$}(update\ ^{\mathbb{P}}[\![\ Minus\ l\ r\ ]\!]\ ^{\mathbb{P}}[\![\ Sub\ l\ r\ ]\!] \\
&\qquad\qquad\qquad\ ^{\mathbb{D}}[\![\ l = pExpArith; r = pFactorArith\ ]\!]) \\
&\quad ,\,^{\$}(normalSV\ ^{\mathbb{P}}[\![\ EF\ \_\ ]\!]\ ^{\mathbb{P}}[\![\ \_\ ]\!]\ ^{\mathbb{P}}[\![\ EF\ \_\ ]\!]) \\
&\qquad \Longrightarrow\ ^{\$}(update\ ^{\mathbb{P}}[\![\ EF\ t\ ]\!]\ ^{\mathbb{P}}[\![\ t\ ]\!] \\
&\qquad\qquad\qquad\ ^{\mathbb{D}}[\![\ t = pFactorArith\ ]\!]) \\
&\quad] \\
&pFactorArith :: BiGUL\ Factor\ Arith \\
&pFactorArith = Case \\
&\quad [\,^{\$}(normalSV\ ^{\mathbb{P}}[\![\ Lit\ \_\ ]\!]\ ^{\mathbb{P}}[\![\ Num\ \_\ ]\!]\ ^{\mathbb{P}}[\![\ Lit\ \_\ ]\!]) \\
&\qquad \Longrightarrow\ ^{\$}(update\ ^{\mathbb{P}}[\![\ Lit\ i\ ]\!]\ ^{\mathbb{P}}[\![\ Num\ i\ ]\!]\ ^{\mathbb{D}}[\![\ i = Replace\ ]\!])
\end{aligned}
$$

```
  , $(normalSV ℙ⟦ Neg _ ⟧ ℙ⟦ Sub (Num 0) _ ⟧ ℙ⟦ Neg _ ⟧)
      ⟹ $(update ℙ⟦ Neg t ⟧ ℙ⟦ Sub (Num 0) t ⟧ 𝔻⟦ t = pFactorArith ⟧)
  , $(normalSV ℙ⟦ Paren _ ⟧ ℙ⟦ _ ⟧ ℙ⟦ Paren _ ⟧)
      ⟹ $(update ℙ⟦ Paren t ⟧ ℙ⟦ t ⟧ 𝔻⟦ t = pExpArith ⟧)
  ]
```

This covers only "normal" cases though, namely when the source and view are "the same" except for parentheses and literals. What about the cases where the source and view have mismatched shapes? For these cases, we need adaptation. Corresponding to each branch we have already written, we add an adaptive branch which looks at the shape of the view only, throws away a mismatched source, and creates an incomplete one whose shape matches that of the view; the source will be completely created through recursive processing. For example, corresponding to the plus/addition branch, we write:

```
$(adaptiveSV ℙ⟦ _ ⟧ ℙ⟦ Add _ _ ⟧)
  ⟹ λ_ _ → Plus ENull FNull
```

The full programs are:

```
pExpArith :: BiGUL Exp Arith
pExpArith = Case
  [ $(normalSV ℙ⟦ Plus _ _ ⟧ ℙ⟦ Add _ _ ⟧ ℙ⟦ Plus _ _ ⟧)
      ⟹ $(update ℙ⟦ Plus l r ⟧ ℙ⟦ Add l r ⟧
                 𝔻⟦ l = pExpArith; r = pFactorArith ⟧)
  , $(normalSV ℙ⟦ Minus _ _ ⟧ ℙ⟦ Sub _ _ ⟧ ℙ⟦ Minus _ _ ⟧)
      ⟹ $(update ℙ⟦ Minus l r ⟧ ℙ⟦ Sub l r ⟧
                 𝔻⟦ l = pExpArith; r = pFactorArith ⟧)
  , $(normalSV ℙ⟦ EF _ ⟧ ℙ⟦ _ ⟧ ℙ⟦ EF _ ⟧)
      ⟹ $(update ℙ⟦ EF t ⟧ ℙ⟦ t ⟧
                 𝔻⟦ t = pFactorArith ⟧)
  , $(adaptiveSV ℙ⟦ _ ⟧ ℙ⟦ Add _ _ ⟧)
      ⟹ λ_ _ → Plus ENull FNull
  , $(adaptiveSV ℙ⟦ _ ⟧ ℙ⟦ Sub _ _ ⟧)
      ⟹ λ_ _ → Minus ENull FNull
  , $(adaptiveSV ℙ⟦ _ ⟧ ℙ⟦ _ ⟧)
      ⟹ λ_ _ → EF FNull
  ]
pFactorArith :: BiGUL Factor Arith
pFactorArith = Case
  [ $(normalSV ℙ⟦ Lit _ ⟧ ℙ⟦ Num _ ⟧ ℙ⟦ Lit _ ⟧)
      ⟹ $(update ℙ⟦ Lit i ⟧ ℙ⟦ Num i ⟧ 𝔻⟦ i = Replace ⟧)
  , $(normalSV ℙ⟦ Neg _ ⟧ ℙ⟦ Sub (Num 0) _ ⟧ ℙ⟦ Neg _ ⟧)
      ⟹ $(update ℙ⟦ Neg t ⟧ ℙ⟦ Sub (Num 0) t ⟧ 𝔻⟦ t = pFactorArith ⟧)
```

$$
\begin{array}{l}
\quad , {}^{\$}(normalSV\ {}^{\mathbb{P}}[\![\, Paren\ \_ \,]\!]\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ {}^{\mathbb{P}}[\![\, Paren\ \_ \,]\!]) \\
\qquad \Longrightarrow {}^{\$}(update\ {}^{\mathbb{P}}[\![\, Paren\ t \,]\!]\ {}^{\mathbb{P}}[\![\, t \,]\!]\ {}^{\mathbb{D}}[\![\, t = pExpArith \,]\!]) \\
\quad , {}^{\$}(adaptiveSV\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ {}^{\mathbb{P}}[\![\, Num\ \_ \,]\!]) \\
\qquad \Longrightarrow \lambda \_\ \_ \rightarrow Lit\ 0 \\
\quad , {}^{\$}(adaptiveSV\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ {}^{\mathbb{P}}[\![\, Sub\ (Num\ 0)\ \_ \,]\!]) \\
\qquad \Longrightarrow \lambda \_\ \_ \rightarrow Neg\ FNull \\
\quad , {}^{\$}(adaptiveSV\ {}^{\mathbb{P}}[\![\, \_ \,]\!]\ {}^{\mathbb{P}}[\![\, \_ \,]\!]) \\
\qquad \Longrightarrow \lambda \_\ \_ \rightarrow Paren\ ENull \\
\quad]
\end{array}
$$

8.4 Reflecting optimizations and evaluation sequences

The BiGUL programs, being bidirectional, can be executed in the *put* direction as a reflective printer, or in the *get* direction as a parser. Let us look at parsing first. For example:

```
*BiYacc> get pExpArith (parseExp "(-(3+0))")
Just (Sub (Num 0) (Add (Num 3) (Num 0)))
```

Note that a unary minus is regarded as syntactic sugar, and is desugared into a subtraction whose left operand is zero. Also note that parentheses are turned into correct structure of the abstract syntax tree, and nothing more — excessive parentheses are cleanly discarded.

For reflective printing, as we mentioned, one application is reporting what compiler optimizations do. We can optimize the sub-expression $3 + 0$ by getting rid of the superfluous $+ 0$, for example, and the reflective printer will be able to retain the excessive parentheses:

```
*BiYacc> put pExpArith (parseExp "(-(3+0))") (Sub (Num 0) (Num
    3))
Just (-(3))
```

Notice also that the unary minus is preserved. If the original concrete expression uses a binary minus instead, it will be preserved as well:

```
*BiYacc> put pExpArith (parseExp "(0-(3+0))") (Sub (Num 0) (Num
    3))
Just (0-(3))
```

In the above example, the pair of parentheses around 3 is also preserved. This is more a coincidence, though — if we change *Sub* to *Add*, for example, the pair of parentheses will not be preserved:

```
*BiYacc> put pExpArith (parseExp "(0-(3+0))") (Add (Num 0) (Num
    3))
Just (0+3)
```

This behavior is indeed what we described with our BiGUL program: the concrete binary minus does not match the abstract *Add*, so the whole concrete expression `0-(3+0)` inside the outermost pair of parentheses is discarded, and a new concrete expression `0+3` is generated by adaptation. This behavior does not give us "least change", however: the pair of parentheses around 3 could have been kept. This is one example showing that, while GetPut (no view change implies no source change) is guaranteed by BiGUL, least-change behavior (small view change implies small source change) is another matter completely, and requires extra care and effort to achieve.

Another thing we can do is reflecting the steps in an evaluation sequence of an abstract syntax tree to concrete syntax. For example, starting from:

```
*BiYacc> get pExpArith (parseExp "1+(2+3)")
Just (Add (Num 1) (Add (Num 2) (Num 3)))
```

it takes two steps to evaluate this expression:

```
*BiYacc> put pExpArith (parseExp "1+(2+3)") (Add (Num 1) (Num 5)
    )
Just 1+(5)
*BiYacc> put pExpArith (parseExp "1+(5)") (Num 6)
Just 6
```

This means that if we have an evaluator on the abstract syntax, we will automatically get an evaluator on the concrete syntax!

A reflective printer can also be used as an ordinary printer by setting the original source to an empty one. For example:

```
*BiYacc> put pExpArith ENull (Sub (Num 0) (Add (Num 1) (Num 1)))
Just 0-(1+1)
```

Note that the subtraction is reflected as a binary minus instead of a unary one, despite that the left operand is zero. This behavior is easily customizable: By adding an adaptive branch before the one dealing generically with *Sub* in *pExpArith*:

$$^{\$}(adaptiveSV\ ^{\mathbb{P}}[\![\ \_\ ]\!]\ ^{\mathbb{P}}[\![\ Sub\ (Num\ 0)\ \_\ ]\!])$$
$$\Longrightarrow \lambda\_\ \_ \to EF\ FNull$$

the above abstract syntax tree can be printed as:

```
*BiYacc> put pExpArith ENull (Sub (Num 0) (Add (Num 1) (Num 1)))
Just 0-(1+1)
```

8.5 A domain-specific language

As a final remark, the above programs may look long, but at the core of them are merely the correspondences between concrete production rules and abstract constructors. We can design a domain-specific language (DSL) that expresses

such correspondences concisely, and then expand programs in this DSL into BiGUL. In fact, we have already done so, and the DSL is called *BiYacc*. For example, all the programs we have written can be generated from the following eight-line BiYacc program:

```
Arith +> Exp
Add l r +> (l +> Exp) '+' (r +> Factor);
Sub l r +> (l +> Exp) '-' (r +> Factor);
f       +> (f +> Factor);

Arith +> Factor
Num n         +> (n +> Int);
Sub (Num 0) r +> '-' (r +> Factor);
f             +> '(' (f +> Exp) ')';
```

See our SLE 2016 paper [17] for more interesting experiments about reflective printing, done on a more realistic imperative language.

9 Conclusion

We have given an introduction to BiGUL programming, explained the underlying design of its putback-based language constructs, and presented a number of applications. BiGUL in its current form is merely one step toward a versatile bidirectional programming language, though. We conclude this chapter by laying out some future directions.

While BiGUL is designed to ensure that programmers can freely describe whatever consistency restoration strategies they have in mind and guarantees that the described strategies are well-behaved, well-behavedness guarantees may be trivial if a described strategy is not actually well-behaved and consequently fails some dynamic checks at runtime. Working with the current BiGUL can thus involve a lot of tedious testing to see if those dynamic checks can go through; also, since we must keep the dynamic checks in place to ensure well-behavedness, at runtime they can incur serious performance overheads. We need ways to precisely characterize the behavior of the dynamic checks, so that it is possible to know that they are redundant and can be safely skipped during execution.

Also we have observed that, as consistency relations or consistency restoration strategies become more complex, BiGUL programs can quickly become awkward to write and hard to read. It is also not that easy to develop reusable libraries because BiGUL programs are not easily composable. (The only general composition operator, namely the classical lens composition, behaves obscurely in the putback direction and is difficult to understand in practice. A discussion of this problem is offered by, e.g., Diskin et al. [4, Section 2.2].) We need to design new language constructs that improve composability of BiGUL programs, discover programming patterns and architectures, and eventually build reusable libraries to facilitate program development.

Apart from language-specific issues, there are also challenges faced by the functional programming approach to BXs in general. For one, graphs have always

been a kind of data structure that is hard to deal with in functional programming, but the application domains of other BX sub-communities usually require the ability to work with graphs; this is an area where BiGUL and other functional programming–based languages/tools need to catch up. More fundamentally, while programmable from one single direction, asymmetric lenses are a less expressive BX formalism, and we probably should not restrict the future version of BiGUL and new bidirectional languages to the framework of asymmetric lenses. We should recognize that the essence of BiGUL is its full programmability of bidirectional behavior, not the framework of asymmetric lenses which it currently supports, and we should strive to bring this programmability into other existing BX formalisms, or, if that is difficult, come up with new formalisms that are designed with such programmability in mind.

Acknowledgements

We would like to thank James Cheney, Jeremy Gibbons, and, in particular, Anthony Anjorin for their meticulous and helpful comments on this paper. This work is supported by JSPS Grant-in-Aid for Scientific Research (A) No. 25240009 of Japan, the Nation Basic Research Program (973 Program) of China under Grant No. 2015CB352201, and the National Natural Science Foundation of China under Grant No. 61620106007.

References

1. Barbosa, D.M.J., Cretin, J., Foster, J.N., Greenberg, M., Pierce, B.C.: Matching lenses: alignment and view update. In: International Conference on Functional Programming. pp. 193–204. ACM (2010), https://doi.org/10.1145/1863543.1863572
2. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: Symposium on Principles of Programming Languages. pp. 407–419. ACM (2008), https://doi.org/10.1145/1328438.1328487
3. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: Symposium on Principles of Database Systems. pp. 338–347. ACM (2006), https://doi.org/10.1145/1142351.1142399
4. Diskin, Z., Xiong, Y., Czarnecki, K.: From state- to delta-based bidirectional model transformations: the asymmetric case. Journal of Object Technology 10, 6:1–25 (2011), https://doi.org/10.5381/jot.2011.10.1.a6
5. Fischer, S., Hu, Z., Pacheco, H.: A clear picture of lens laws. In: International Conference on Mathematics of Program Construction. pp. 215–223. Lecture Notes in Computer Science, Springer (2015), https://doi.org/10.1007/978-3-319-19797-5
6. Fischer, S., Hu, Z., Pacheco, H.: The essence of bidirectional programming. SCIENCE CHINA Information Sciences 58(5), 1–21 (2015), https://doi.org/10.1007/s11432-015-5316-8
7. Foster, J.: Bidirectional Programming Languages. Ph.D. thesis, University of Pennsylvania (December 2009)

8. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: a linguistic approach to the view-update problem. ACM Transactions on Programming Languages and Systems 29(3), 17 (2007), https://doi.org/10.1145/1232420.1232424
9. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Matsuda, K., Nakano, K.: Bidirectionalizing graph transformations. In: International Conference on Functional Programming. pp. 205–216. ACM (2010), https://doi.org/10.1145/1932681.1863573
10. Ko, H.S., Zan, T., Hu, Z.: BiGUL: a formally verified core language for putback-based bidirectional programming. In: Workshop on Partial Evaluation and Program Manipulation. pp. 61–72. ACM (2016), https://doi.org/10.1145/2847538.2847544
11. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: International Conference on Functional Programming. pp. 47–58. ACM (2007), https://doi.org/10.1145/1291220.1291162
12. Pacheco, H., Hu, Z., Fischer, S.: Monadic combinators for "putback" style bidirectional programming. In: Workshop on Partial Evaluation and Program Manipulation. pp. 39–50. ACM (2014), https://doi.org/10.1145/2543728.2543737
13. Pacheco, H., Zan, T., Hu, Z.: BiFluX: a bidirectional functional update language for XML. In: International Symposium on Principles and Practice of Declarative Programming. pp. 147–158 (2014), https://doi.org/10.1145/2643135.2643141
14. Voigtländer, J.: Bidirectionalization for free! In: Symposium on Principles of Programming Languages. pp. 165–176. ACM (2009), https://doi.org/10.1145/1480881.1480904
15. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: International Conference on Automated Software Engineering. pp. 164–173. ACM (2007), https://doi.org/10.1145/1321631.1321657
16. Zan, T., Liu, L., Ko, H.S., Hu, Z.: Brul: a putback-based bidirectional transformation library for updatable views. In: International Workshop on Bidirectional Transformations. pp. 77–89. CEUR-WS (2016), http://ceur-ws.org/Vol-1571/paper_3.pdf
17. Zhu, Z., Zhang, Y., Ko, H.S., Martins, P., Saraiva, J., Hu, Z.: Parsing and reflective printing, bidirectionally. In: International Conference on Software Language Engineering. pp. 2–14. ACM (2016), https://doi.org/10.1145/2997364.2997369