An Axiomatic Basis for Bidirectional Programming

HSIANG-SHANG KO, National Institute of Informatics
ZHENJIANG HU, National Institute of Informatics

Among the frameworks of bidirectional transformations proposed for addressing various synchronisation problems, Foster et al.'s asymmetric lenses have influenced the design of a generation of bidirectional programming languages. Most of these languages are highly declarative, and only allow the programmer to specify a consistency relation with limited control over the behaviour of the automatically derived consistency restorer. However, synchronisation problems are diverse and require vastly different consistency restoration strategies, and to cope with the diversity, the programmer must be empowered to fully control and reason about the consistency restoration behaviour of their bidirectional programs. The putback-based approach to bidirectional programming was proposed to address this issue once and for all, and this paper takes the approach one step further by proposing a Hoare-style logic for Ko et al.'s putback-based language BiGUL. With this Hoare-style logic, the BiGUL programmer can precisely characterise the bidirectional behaviour of their programs by reasoning solely in the putback direction. The theory underlying the Hoare-style logic has been formalised and checked in Agda, but this paper presents the Hoare-style logic in a semi-formal way to make it easily understood and usable by the working BiGUL programmer.

ACM Reference format:
Hsiang-Shang Ko and Zhenjiang Hu. 2017. An Axiomatic Basis for Bidirectional Programming. *PACM Progr. Lang.* 1, 1, Article 1 (February 2017), 26 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The need for synchronisation — or *consistency maintenance* — is pervasive in computing. A simple but typical example is synchronisation among documents of different formats, in which case consistency means that the documents have the same content; whenever the content of one document is changed, the other documents should also be updated to restore the consistency. Over the past decade, frameworks of *bidirectional transformations* have been proposed by different communities to address a diverse range of synchronisation problems (Czarnecki et al. 2009). One such framework from the programming language community is Foster et al. (2007)'s *asymmetric lenses*, which are highly influential such that the term *bidirectional programming* has become largely synonymous with lens-based approaches. Asymmetric lenses are designed for synchronising two pieces of data where one side — which is called the *source* — has more information than the other, which is called the *view*. The programmer writes a lens program to describe a forward *get* transformation that computes a consistent view from a source; whenever the source is changed, *get* is rerun to produce a new consistent view. Conversely, the same lens program can also be interpreted as a backward *put* transformation that takes a source and a possibly modified view, and produces an updated source that is consistent with the view and can retain some information of the original source.

By construction, the two transformations denoted by any lens program satisfy two inverse-like *well-behavedness* laws called PutGet and GetPut (Theorem 2.2). Stevens (2010, Section 4.4) provided a revealing perspective to understand these well-behavedness laws: the *get* transformation denoted by a lens can be regarded as defining a consistency relation on the source and view; PutGet then says that the *put* transformation will correctly restore the consistency, i.e., the updated source and the view will satisfy the consistency relation, and GetPut says that the *put* transformation will perform no update if the input source and view are already consistent. From this

2017. 2475-1421/2017/2-ART1 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

perspective, what Foster et al. (2007)'s lenses offer is a highly declarative programming model, in which the programmer only specifies a consistency relation (in terms of a *get* transformation) and obtains a consistency restorer (a *put* transformation) that is guaranteed (by well-behavedness) to respect the consistency relation but is otherwise arbitrary. This is unsatisfactory in practice, since we care not only about consistency but also about how consistency restoration is performed; with lenses it is difficult to understand or control the latter aspect.

To be concrete, let us consider a simple synchronisation problem where the source is a pair of numbers representing the width and height of a rectangle, and the view is a single number, which is consistent with a rectangle exactly when it is equal to the width of the rectangle. With respect to this definition of consistency, there are a variety of consistency restoration strategies: given a rectangle and a view, in addition to replacing the width with the view, which is necessary for restoring the consistency,

- we can always keep the height unchanged — this is a typical "least-change" strategy;
- we can update the height to keep the height-to-width ratio of the rectangle — in general this can be maintaining some kind of internal consistency on the source side;
- we can reset the height to zero if the view is different from the width — although rather drastic, this would be useful when the view side does not know how the source side maintains its internal consistency, and thus simply chooses to invalidate associated data and leave them for the source side to update later;
- we can decide to keep or reset the height depending on whether the difference between the width and the view is small enough — this is a flexible mixture of the above strategy and the "least-change" one;
- we can use the height as a counter that is incremented every time an inconsistency is repaired — this is somewhat bizarre for rectangle width updating, but may well be appropriate for some other interpretations of the problem.

As we can see, even for a simple problem like rectangle width updating, there are already many possible update strategies; this is even more the case in complex, real-world scenarios. All the above update strategies restore the same consistency but have different *retentive* behaviour — the way in which the information of the original source is retained — to meet different requirements. The programmer must be empowered to fully control and reason about the retentive behaviour of their programs to be sure that it is suitable for the intended applications.

Given that there are a myriad possibilities of update strategies, what can be better than having languages for *programming* such strategies, capturing the myriad possibilities once and for all? Following some previous work which took exactly this *putback-based* programming approach (Hu et al. 2014; Pacheco et al. 2014a,b), Ko et al. (2016) proposed a language BiGUL, which is short for the *Bidirectional Generic Update Language*. Like the original lenses, every BiGUL program denotes a well-behaved pair of *put* and *get* transformations; in contrast to the original lenses, BiGUL is designed for expressing *put* transformations, and gives the programmer unprecedented freedom to specify their intended update strategies directly. However, due to the more intricate nature of BiGUL's bidirectional semantics, it is still hard to guarantee that a BiGUL program will exhibit the intended behaviour.

This paper takes the putback-based approach one step further: building on a revised version of BiGUL, we propose a *Hoare-style logic* (Hoare 1969) that empowers the programmer to precisely characterise both the *put* and *get* behaviour of BiGUL programs by reasoning exclusively in the putback direction. For example, the programmer can express the height-keeping and height-resetting strategies as two BiGUL programs *keepHeight* and *resetHeight*, and with our Hoare-style logic, the programmer can prove two Hoare-style triples to make sure that the two programs correctly restore the consistency and have the intended retentive behaviour:

$$\{\,\mathit{True}\,\}\;\; \mathit{keepHeight}\;\; \{\,(w',\,h')\;(\_\;,\,h)\;v \mid w' = v \wedge h' = h\,\}$$
$$\{\,\mathit{True}\,\}\;\; \mathit{resetHeight}\;\; \{\,(w',\,h')\;(w\;,\,h)\;v \mid w' = v \wedge (w = v \Rightarrow h' = h) \wedge (w \neq v \Rightarrow h' = 0)\,\}$$

These two putback triples state that both *keepHeight* and *resetHeight* work on any input pairs of source and view (due to their always-true precondition) and will update the width with the view ($w' = v$), that *keepHeight* will retain the original height ($h' = h$), and that *resetHeight* will retain the height if the original width is equal to the view ($w = v \Rightarrow h' = h$) or reset the height otherwise ($w \neq v \Rightarrow h' = 0$). With a bit more reasoning, the programmer can also prove that the *get* transformations denoted by these two programs work on any input rectangle and extract its width, conforming to the consistency relation ($w' = v$) stated in the above triples.

Here are our contributions in a nutshell: We define Hoare-style *putback triples* for reasoning about the *put* behaviour of BiGUL programs, and show how they can also characterise the *get* behaviour to some extent (Section 3). The putback proof rules provide an axiomatic encapsulation of BiGUL's semantics, and are designed for convenient domain-specific reasoning (Section 4). Uniquely, to adequately characterise the *get* behaviour, our Hoare-style logic also includes *range triples* for estimating the output ranges of BiGUL programs (Section 5). We further propose rules for reasoning about recursive programs (Section 6), and verify a BiGUL implementation of key-based list alignment as a showcase example (Section 7). The presentation will be preceded by a recap of asymmetric lenses (Section 2), and end with some discussion (Section 8) and conclusion (Section 9).

Everything in this paper from theorems to derivation examples has been formalised and checked in Agda version 2.5.2 with standard library version 0.13, but the Agda formalisation is only provided as supplementary material. Instead, this paper will focus on explaining the intuition, and present the Hoare-style logic in a semi-formal way to make it suitable for human reasoning. BiGUL is originally developed in Agda and also ported to Haskell as an embedded language, and this influences the choices of syntax used in this paper: our BiGUL syntax is a hypothetical one abstracted from the Haskell port of BiGUL; the host functional language is total and may be thought of as Agda imperfectly disguised as Haskell — in particular, we will use some standard Haskell types and functions, and allow some general recursion and partiality justifiable in a total setting.

2 A RECAP OF ASYMMETRIC LENSES

We start from a brief recap of some general facts about asymmetric lenses, and state these facts directly in terms of BiGUL. Definition 2.1 and Theorem 2.2 below are provided by Ko et al. (2016) for the original BiGUL, whereas this paper uses a revised version of the language; accordingly, we have revised the internals of the definition and theorem, while keeping their statements the same.

Definition 2.1. A BiGUL program b operating on source type S and view type V is assigned the type $S \hookleftarrow V$, and has two semantics:

$$\begin{aligned} &put\ \ b : S \to V \to Maybe\ S \\ &get\ \ b : S \to \phantom{V \to{}} Maybe\ V \end{aligned}$$

The *put* — or *putback* — semantics is also called the *backward* semantics, and the *get* semantics is also called the *forward* semantics.

As noted by Ko et al., the two semantics in Definition 2.1 are potentially partial computations modelled explicitly as total *Maybe*-computations. That is, *put b* and *get b* may fail to compute a result, in which case they produce *Nothing*; otherwise they return their result wrapped within the *Just* constructor.

Theorem 2.2 (well-behavedness). *Any BiGUL program b satisfies the following two* well-behavedness *laws:*

$$\forall s, v, s'.\quad put\ b\ s\ v = Just\ s' \quad\Rightarrow\quad get\ b\ s' = Just\ v \qquad (\textsc{PutGet})$$

$$\forall s, v.\quad get\ b\ s = Just\ v \quad\Rightarrow\quad put\ b\ s\ v = Just\ s \qquad (\textsc{GetPut})$$

As noted by Ko et al., Theorem 2.2 gives a stronger well-behavedness guarantee (Macedo et al. 2013; Pacheco et al. 2014a) than Foster et al. (2007)'s original definition. Even so, this theorem is not as practically useful as it seems because non-well-behavedness is merely swept under the partiality carpet: both *put b* and *get b* perform various checks at runtime to detect non-well-behavedness, and if the programmer does not pay enough attention to well-behavedness requirements, the execution of *put b* or *get b* can unexpectedly fail one of these runtime checks (thereby satisfying PutGet or GetPut vacuously). On the other hand, theoretically we no longer need to worry about well-behavedness and can concentrate on totality, i.e., making sure that BiGUL programs can compute successfully on the inputs that we care about. This is part of the motivation for developing the Hoare-style logic.

Well-behavedness implies the following theorem (originally Foster (2009)'s Lemma 2.2.5), which says that once a putback transformation is specified, the corresponding forward transformation is uniquely determined.

Theorem 2.3 (dominance of putback semantics). *Let l, $r : S \hookleftarrow V$.*

If $\quad put\ l = put\ r \quad$ *then* $\quad get\ l = get\ r \quad .$

This is the motivation behind BiGUL's putback-based design, as it shows that it is theoretically feasible that the BiGUL programmer can think and program solely in the putback direction and still unambiguously specify the forward behaviour. We will see more precisely how this works in practice with the Hoare-style logic.

3 THEORY OF PUTBACK TRIPLES

Programming a putback transformation in BiGUL is comparable to programming with states: the BiGUL programmer is given a source state and a view state, and manipulates the two states with the aim of transferring all information in the view to the source; in the end, the updated source state is returned as the result of the putback transformation. Reasoning about BiGUL programs thus consists of tracking the properties satisfied by the states at each step, and it should not come as a surprise that a Hoare-style logic is well suited for performing this kind of reasoning about states. We will introduce a set of Hoare-style triples for saying when BiGUL programs (as putback transformations) can compute successfully and return results satisfying some specified properties. Before doing so, however, we should first fix our notation of relations (for specifying preconditions and postconditions).

Notation 3.1. Relations on types $A_1, A_2, \ldots, A_n$ are considered inhabitants of the type $\mathcal{P}(A_1 \times A_2 \times \cdots \times A_n)$. When a relation R of this type relates elements $a_1 : A_1, a_2 : A_2, \ldots, a_n : A_n$, we write $R\ a_1\ a_2\ \ldots\ a_n$.

Definition 3.2. A *putback triple* is a BiGUL program $b : S \hookleftarrow V$ surrounded by two *putback assertions*:

$$\{\, R \,\}\ \ b\ \ \{\, R' \,\}$$

where $R : \mathcal{P}(S \times V)$ is the *precondition* (on the original source and the view) and $R' : \mathcal{P}(S \times S \times V)$ is the *postcondition* (on the updated source, the original source, and the view). Valid putback triples are inductively defined by the proof rules in Figure 1 (which will be explained in Section 4).

Instead of requiring preconditions and postconditions to be syntactic entities drawn from a particular logic, we treat them semantically and will freely use whatever relations that are expressible mathematically (or, if that sounds too lax, in Type Theory (Martin-Löf 1984)).

As usual, the proof rules in Figure 1 are designed such that valid putback triples have the intended meaning; this is made precise and proved with respect to BiGUL's semantics (Definition 2.1) as follows.

Theorem 3.3 (soundness of putback triples). *Let $b : S \hookleftarrow V$, $R : \mathcal{P}(S \times V)$, and $R' : \mathcal{P}(S \times S \times V)$.*

If $\quad \{\, R \,\}\ b\ \{\, R' \,\} \quad$ *then* $\quad \forall s, v.\ \ R\ s\ v \ \Rightarrow\ \exists s'.\ \ put\ b\ s\ v = Just\ s' \ \wedge\ R'\ s'\ s\ v \quad .$

In prose: a valid putback triple $\{\, R \,\}\ b\ \{\, R' \,\}$ means that if the original source and the view satisfies the precondition R, then $put\ b$ will successfully produce an updated source satisfying the postcondition R', which can relate the updated source to the original source and the view. The proof of this theorem (which is by induction on the rules in Figure 1 as usual) has been checked in the accompanying Agda formalisation, and is omitted from the presentation due to space restrictions. (The reader is thus asked to trust the soundness of the rules in Figure 1.)

Theorem 2.3 tells us that putback behaviour completely determines forward behaviour. On the other hand, putback triples are about putback behaviour — shouldn't they tell us something about forward behaviour as well? This is indeed the case, as will be shown by Theorem 3.8. Its statement will make use of some important definitions and notational conventions that will also be used throughout the rest of this paper.

Definition 3.4. A *comprehension relation* of type $\mathcal{P}(A_1 \times A_2 \times \cdots \times A_n)$ has the form

$$\langle\, pat_1\ pat_2\ \ldots\ pat_n \mid prop \,\rangle$$

where each pat_i is a pattern for elements of type A_i and $prop$ is a proposition that can refer to the variables in the patterns. The patterns we use in the paper include variables, constructors, and the wildcard pattern ('_'). The relation holds for $a_1 : A_1$, $a_2 : A_2$, ..., $a_n : A_n$ exactly when each a_i matches pat_i and $prop$ holds after substituting the matched components for the corresponding pattern variables.

Notation 3.5. We usually omit the proposition part of a comprehension relation when the proposition is trivially true, keeping only the pattern part. For example, $\langle\, (\_ :: \_) \,\rangle$ holds exactly for non-empty lists, and $\langle\, \_\ \_ \,\rangle$ is the always-true binary relation.

Notation 3.6. The angle brackets delimiting a comprehension relation may be omitted where delimitation is unnecessary, like in an assertion containing only a comprehension relation. For example, $\{\, \langle\, s\ v \mid s = v \,\rangle \,\}$ is abbreviated to $\{\, s\ v \mid s = v \,\}$.

Definition 3.7. The *graph* of a function $f : A \to \mathit{Maybe}\ B$ is a relation $\mathcal{G}f : \mathcal{P}(A \times B)$ which relates $a : A$ and $b : B$ exactly when $f\ a = \mathit{Just}\ b$.

Theorem 3.8 (partial forward consistency). *Let* $b : S \hookleftarrow V$, $R : \mathcal{P}(S \times V)$, *and* $C : \mathcal{P}(S \times V)$.

$$\textit{If}\quad \{\, R \,\}\ \ b\ \ \{\, s'\ \_\ v \mid C\ s'\ v \,\} \quad \textit{then}\quad \mathcal{G}(\mathit{get}\ b) \cap R \subseteq C \quad .$$

Proof. Suppose $\mathit{get}\ b\ s = \mathit{Just}\ v$ and $R\ s\ v$. The latter assumption triggers Theorem 3.3, so we know that $\mathit{put}\ b\ s\ v = \mathit{Just}\ s'$ for some s' and that $C\ s'\ v$ holds. On the other hand, by GetPut, we can turn the first assumption $\mathit{get}\ b\ s = \mathit{Just}\ v$ into $\mathit{put}\ b\ s\ v = \mathit{Just}\ s$. Seeing that $\mathit{put}\ b\ s\ v$ computes to both s' and s, we can deduce $s' = s$, and thus having $C\ s'\ v$ is the same as having $C\ s\ v$. □

That is, if we can prove that a putback transformation establishes consistency C between the updated source and the view, then, roughly speaking, a part of the behaviour of the corresponding forward transformation will be constrained by C. We call Theorem 3.8 *partial* forward consistency for two reasons. The first reason is that Theorem 3.8 does not guarantee that the entire graph of the forward transformation will be contained in C — the containment is guaranteed only for the part of the graph that falls within R. In practice, this makes Theorem 3.8 not very helpful unless R is always true, in which case the entire graph will indeed be contained in C. Even in this case, though, there is still the second reason: Theorem 3.8 says nothing about the totality of the forward transformation, i.e., on which subset of sources the forward transformation can successfully produce results. We will augment Theorem 3.8 to get a practically useful version (Theorem 5.4). But before that, let us look at the concrete putback proof rules and some examples of putback reasoning.

4 BIGUL AND THE PUTBACK PROOF RULES

In this section we will introduce BiGUL's constructs and their putback proof rules shown in Figure 1. Instead of presenting the axiomatic system as if we are still constructing it and justifying its soundness, we will regard the system as already completed and use it as a basis for introducing BiGUL. Except for the consequence rule (the right one in the second row), every rule corresponds to a BiGUL construct and gives an axiomatic encapsulation of its semantics. For each BiGUL construct, we will introduce its type, which is essential for inferring the types of entities in assertions, and explain the corresponding proof rule with the help of the operational meaning of the construct. We will not, however, go into the rationale behind the design of the constructs, for which the

$$\frac{}{\{\,\emptyset\,\}\ \ \mathbf{fail}\ \ \{\,\emptyset\,\}} \qquad \frac{}{\{\,\_\ \_\,\}\ \ \mathbf{replace}\ \ \{\,s'\ \_\ v \mid s' = v\,\}} \qquad \frac{}{\{\,s\ v \mid f\ s = v\,\}\ \ \mathbf{skip}\, f\ \ \{\,s'\ s\ \_ \mid s' = s\,\}}$$

$$\frac{\{\,L\,\}\ \ l\ \ \{\,L'\,\} \qquad \{\,R\,\}\ \ r\ \ \{\,R'\,\}}{\{\,L * R\,\}\ \ l * r\ \ \{\,L' * R'\,\}} \qquad \frac{T\ \subseteq\ R \qquad \{\,R\,\}\ \ b\ \ \{\,R'\,\} \qquad R' \cap \langle\,\_\ s\ v \mid T\ s\ v\,\rangle\ \subseteq\ T'}{\{\,T\,\}\ \ b\ \ \{\,T'\,\}}$$

$$\frac{\{\,s\ \mathit{wpat} \mid R\ s\ \overline{\mathit{wpat}}\,\}\ \ b\ \ \{\,s'\ s\ \mathit{wpat} \mid R'\ s'\ s\ \overline{\mathit{wpat}}\,\}}{\{\,s\ \mathit{vpat} \mid R\ s\ \overline{\mathit{vpat}}\,\}\ \ \mathbf{rearrV}\ \mathit{vpat} \longrightarrow \mathit{wpat} \text{ ↳ } b\ \ \{\,s'\ s\ \mathit{vpat} \mid R'\ s'\ s\ \overline{\mathit{vpat}}\,\}}$$

$$\frac{\{\,\mathit{tpat}\ v \mid R\ \overline{\mathit{tpat}}\ v\,\}\ \ b\ \ \{\,\mathit{tpat}'\ \mathit{tpat}\ v \mid R'\ \overline{\mathit{tpat}'}\ \overline{\mathit{tpat}}\ v\,\}}{\{\,\mathit{spat}\ v \mid R\ \overline{\mathit{spat}}\ v\,\}\ \ \mathbf{rearrS}\ \mathit{spat} \longrightarrow \mathit{tpat} \text{ ↳ } b\ \ \{\,\mathit{spat}'\ \mathit{spat}\ v \mid R'\ \overline{\mathit{spat}'}\ \overline{\mathit{spat}}\ v\,\}}$$

$$\frac{\begin{array}{l} \forall(\mathbf{normal}\ M\ \mathbf{exit}\ E \text{ ↳ } b) \in \mathit{branches}. \\ \quad \{\,R \cap \widehat{M}\,\}\ \ b\ \ \{\,R' \cap \langle\,s'\ \_\ v \mid \widehat{M}\ s'\ v \wedge \widehat{E}\ s'\,\rangle\,\} \\ \forall(\mathbf{adaptive}\ M \text{ ↳ } f) \in \mathit{branches}. \\ \quad \forall s,\ v.\ \ (R \cap \widehat{M})\ s\ v\ \ \Rightarrow \\ \qquad\qquad\qquad (R \cap \mathcal{N})\ (f\ \ s\ v)\ v \\ \qquad\qquad\quad \wedge\ \ \forall s'.\ R'\ s'\ (f\ \ s\ v)\ v \Rightarrow R'\ s'\ s\ v \end{array}}{\{\,R \cap \mathcal{D}\,\}\ \ \mathbf{case}\ \mathit{branches}\ \ \{\,R'\,\}} \qquad \begin{array}{l} \mathbf{where} \\ \quad \mathcal{N} = \bigcup\ [\ \widehat{M} \mid (\mathbf{normal}\ M\ \ \ldots) \in \mathit{branches}\ ] \\ \quad \mathcal{D} = \bigcup\ [\ M \mid (\mathbf{normal}/\mathbf{adaptive}\ M\ \ \ldots) \in \mathit{branches}\ ] \end{array}$$

Fig. 1. Putback proof rules

interested reader is referred to Hu and Ko (2017, Section 5). We emphasise again that the assertions in the proof rules are intended to be semantic rather than syntactic — for example, if the precondition stated in a rule is $\langle\,\_\ \_\,\rangle$, we will regard the rule as directly applicable when the actual precondition is, say, $\langle\,s\ v \mid s = s \wedge v = v\,\rangle$, which is different from $\langle\,\_\ \_\,\rangle$ syntactically but still denotes the always-true binary relation semantically.

4.1 Atomic constructs

BiGUL has three atomic constructs, whose corresponding rules are in the first row of Figure 1.

The **fail** construct has type $S \hookleftarrow V$ for any types S and V. The precondition of the **fail** rule is the empty relation $\emptyset$, saying that no input can make **fail** compute successfully. (On the other hand, the postcondition can be any relation because of the consequence rule, which we will discuss in Section 4.3.)

The **replace** construct has type $S \hookleftarrow S$ for any type S, and replaces the source with the view regardless of what they are; correspondingly, the precondition of the **replace** rule is the always-true relation, and the postcondition states that the updated source s' will be equal to the view v.

The **skip** construct takes a function $f : S \to V$ in the host language as an argument and has type $S \hookleftarrow V$. It ignores the view and leaves the source as it is; correspondingly, the postcondition says that the updated source s' will be equal to the original source s. Unlike **replace**, we cannot **skip** under all circumstances — before throwing the view away, we must ensure that it can be recovered from the source, or otherwise there is no hope to establish PutGet. The precondition thus requires that the view can be computed from the source by f.

4.2 Product

Given two BiGUL programs $l : S \hookleftarrow V$ and $r : T \hookleftarrow W$, we can form the product of the two programs $l * r : (S \times T) \hookleftarrow (V \times W)$. If two putback triples with preconditions L and R have been established for l and r, the precondition of the product program will be

$$L * R\ \ =\ \ \langle\,(s,\ t)\ (v,\ w) \mid L\ s\ v \wedge R\ t\ w\,\rangle$$

The relation-level ‘$*$’ operator can (indeed) be seen as a simple variant of separating conjunction (Reynolds 2002), and can be defined arity-generically to construct a relation on n pairs from an n-ary relation on all the first components and the other on all the second components. The postcondition can then be stated also in terms of this ‘$*$’ operator.

Example 4.1 (parallel replacement). We can now construct simple derivations like the following one for **replace** $*$ **replace**, which we use as an example to explain our derivation format:

$$
\begin{array}{l}
\{\, (\_\,,\,\_)\ (\_\,,\,\_) \,\} \\
\vdots\quad \{\, \_\ \_ \,\} \\
\vdots\quad \mathbf{replace} \\
\vdots\quad \{\, s'\ \_\ v \mid s' = v \,\} \\
*\quad \{\, \_\ \_ \,\} \\
\vdots\quad \mathbf{replace} \\
\vdots\quad \{\, t'\ \_\ w \mid t' = w \,\} \\
\{\, (s', t')\ (\_\,,\,\_)\ (v\,,\,w) \mid s' = v \wedge t' = w \,\}
\end{array}
$$

First note that the syntax tree structure of **replace** $*$ **replace** is reflected in indentation: the top-level node is ‘$*$’, whose two sub-nodes — both being **replace** — are indented to the next level. Then, following the indentation structure, the assertions are added: the assertions about a node are put on the same indentation level as the node, with the precondition and postcondition appearing respectively before and after the node. This format is compact and yet retains the tree structure of the derivation, making it easier to check the correctness of the derivation. (Auxiliary dotted lines are also added to make indentation levels clearer, especially when derivations spread across pages.)

4.3 The consequence rule

The consequence rule we present in Figure 1 may seem unusual:

$$
\frac{T \subseteq R \qquad \{\, R \,\}\ b\ \{\, R' \,\} \qquad R' \cap \langle\, \_\ s\ v \mid T\ s\ v \,\rangle \subseteq T'}{\{\, T \,\}\ b\ \{\, T' \,\}}
$$

but first observe that it is a stronger version of the usual one, so at least there is nothing to lose:

$$
\frac{T \subseteq R \qquad \{\, R \,\}\ b\ \{\, R' \,\} \qquad R' \subseteq T'}{\{\, T \,\}\ b\ \{\, T' \,\}} \tag{1}
$$

To see why we need a stronger consequence rule, consider deriving this triple:

$$
\{\, \_\ (v,\ w) \mid v = w \,\}\ \ \mathbf{replace} * \mathbf{replace}\ \ \{\, (s',\ t')\ \_\ \_ \mid s' = t' \,\}
$$

where there is some entanglement between the first and second components in the precondition and postcondition, so the product rule is not directly applicable. We could try to extend the derivation in Example 4.1 using the usual consequence rule (1):

$$
\begin{array}{l}
\{\, \_\ (v,\ w) \mid v = w \,\} \\
\{\, (\_\,,\,\_)\ (\_\,,\,\_) \,\} \\
\mathbf{replace} * \mathbf{replace} \\
\{\, (s', t')\ (\_\,,\,\_)\ (v\,,\,w) \mid s' = v \wedge t' = w \,\} \\
\quad\vdots \\
\{\, (s',\ t')\ \_\ \_ \mid s' = t' \,\}
\end{array}
$$

Adjacent assertions on the same indentation level indicate an invocation of the consequence rule, with the one above implying the one below. In the first two lines of this derivation, there is one such invocation, which turns $\langle\, \_\ (v\ ,\ w) \mid v = w \,\rangle$ into $\langle\, (\_\ ,\ \_)\ (\_\ ,\ \_) \,\rangle$ so that the product rule can apply. On the other hand, the postcondition that we can establish, i.e., $\langle\, (s',\ t')\ (\_\ ,\ \_)\ (v,\ w) \mid s' = v \land t' = w \,\rangle$, does not imply the postcondition we want to establish, i.e., $\langle\, (s',\ t')\ \_\ \_ \mid s' = t' \,\rangle$. The usual consequence rule (1) can help us to get rid of the entanglement $v = w$, but that entanglement is needed to establish the final implication (by $s' = v = w = t'$). We thus need the stronger consequence rule to be able to carry over whatever we know about the original source and the view from the precondition to the postcondition. When working in our derivation format, the stronger consequence rule allows us to prove an implication between adjacent postconditions using whichever preconditions for the same node (on the same indentation level) as additional premises about the original source and the view.

The stronger consequence rule also simplifies other rules. For example, if we only had the usual consequence rule (1), we would have to enrich the **skip** rule to something like:

$$\frac{}{\{\, s\ v \mid f\ s = v \land R\ s\ v \,\}\ \ \mathbf{skip}\ f\ \ \{\, s'\ s\ \_ \mid s' = s \land R\ s'\ v \,\}}$$

saying additionally that whatever is true for the original source and the view in the precondition is also true for the updated source and the view in the postcondition. This enriched **skip** rule can be derived from the simpler one given in Figure 1 and the stronger consequence rule.

4.4 Rearrangement

A guiding intuition for BiGUL programming is to manipulate the source and view to make their shapes match, which is achieved mainly with the *rearrangement* operations. To provide a more concrete motivation: We have seen that the product combinator (Section 4.2) allows us to synchronise source and view tuples of arbitrary size, provided that their structures are the same. When this is not the case, in BiGUL we can use a simple class of pattern-matching λ-expressions to rearrange the source and/or the view to make them match structurally and so ready for further synchronisation. For example, the height-keeping strategy we proposed for the rectangle width updating problem (Section 1) can be expressed in BiGUL as:

```
keepHeight : (ℕ , ℕ) ↩ ℕ
keepHeight = rearrV v → (v, ())
                 ⋮  ⋮  replace
                 ⋮  ∗  skip const ()
```

Initially, the view is a single number, whereas the source is a pair. To make their structures match, we use the view rearrangement operation **rearrV** $v \to (v,\ ())$ to apply the λ-expression $\lambda\ v \to (v,\ ())$ to the view and make the result the new view. Inside the **rearrV**, the source and view are both pairs, so we can use **replace** $*$ **skip** *const* () to update the width and keep the height as it is. Below we will mainly discuss view rearrangement; source rearrangement is largely analogous, and will only be briefly discussed towards the end of this subsection.

View rearrangement. The general form of a view rearrangement is **rearrV** *vpat* $\to$ *wpat* ↳ $b : S \hookleftarrow V$, where *vpat* is a pattern for the original view type V, *wpat* is a "pattern" for a new view type W, and the inner program b has type $S \hookleftarrow W$. (The symbol '↳' indicates that b is syntactically a sub-node of '**rearrV** *vpat* $\to$ *wpat*'.) Strictly speaking, *wpat* is not a pattern but an expression, which can be built using variables in *vpat* and constructors. (Apart from the fact that *wpat* looks similar to a pattern, we will explain why it is beneficial to think of *wpat* as a pattern shortly.) Wildcards are not allowed in *vpat*, and all variables in *vpat* must appear in *wpat* and can appear multiple times — these synctactic restrictions ensure that the λ-expression is invertible, or, more intuitively speaking, does not lose information.

The view rearrangement rule. Intuitively, a rearrangement only massages the state into an alternate shape suitable for further processing, rather than applying an arbitrary and distorting transformation. This intuition significantly influences the design of the rearrangement rules, which reflect that rearrangement is essentially just a "change of perspective". If we are rearranging the view from *vpat* to *wpat*, it must mean that the view matches *vpat* right before the rearrangement, and we should be able to state properties satisfied by the view at that point in terms of its components. The precondition for **rearrV** is thus a comprehension relation:

$$\langle\, s\ \mathit{vpat} \mid R\ s\ \overline{\mathit{vpat}}\,\rangle$$

which requires that the view matches *vpat* and that any properties about the view are stated in terms of the variables in *vpat*, which we denote by $\overline{\mathit{vpat}}$. After the rearrangement, due to the invertibility restrictions, the new view will retain all the components of the original view; the components may be shuffled around and duplicated, but whatever we knew about the components will remain true. The precondition for the inner program b is thus:

$$\langle\, s\ \mathit{wpat} \mid R\ s\ \overline{\mathit{wpat}}\,\rangle$$

This inner precondition asserts that the new view matches *wpat* and that whatever holds for $\overline{\mathit{vpat}}$ in the outer precondition also holds here for $\overline{\mathit{wpat}}$, which is the same as $\overline{\mathit{vpat}}$ because of the invertibility restrictions. Being able to state this inner precondition is the reason that we think of *wpat* also as a pattern, even though that means in general we have to allow non-linear patterns, where multiple occurrences of the same variable indicate implicitly that values at those positions should be equal. When using the rule in actual derivations, the two preconditions will differ only in their view patterns and have the same proposition part. Therefore, the view rearrangement rule only changes the shape we expect the view to take, not the properties we know about the content of the view. This change-of-perspective interpretation works for the postconditions as well.

Example 4.2 (rectangle width updating — keeping the height). We can now verify the *keepHeight* program as follows, where the precondition (assertion 1) is always true and the postcondition (assertion 2) says that the consistency will be established ($w' = v$) and the height will be retained ($h' = h$):

```
$\{\ \_\ \_\ \}_1$
$\mathbf{rearrV}\ v \rightarrow (v, ())$
    $\{\ \_\ (\_\ ,\ ())\ \}_3$
        $\{\ \_\ \_\ \}$
        $\mathbf{replace}$
        $\{\ w'\ \_\ v \mid w' = v\ \}$
    *   $\{\ \_\ ()\ \}$
        $\{\ h\ v \mid \mathit{const}\ ()\ h = v\ \}$
        $\mathbf{skip}\ \mathit{const}\ ()$
        $\{\ h'\ h\ \_ \mid h' = h\ \}$
        $\{\ h'\ h\ () \mid h' = h\ \}$
    $\{\ (w', h')\ (\_\ , h)\ (v, ()) \mid w' = v \land h' = h\ \}_4$
$\{\ (w', h')\ (\_\ , h)\ v \mid w' = v \land h' = h\ \}_2$
```

Note that when constructing the derivation inwards from the initial precondition and postcondition, it is effortless to push them inside the **rearrV** and turn them into assertions 3 and 4 just by changing the view pattern to a pair pattern, as instructed by the **rearrV**.

Source rearrangement and the corresponding rule. Analogous to view rearrangement, the general form of a source rearrangement is **rearrS** $\mathit{spat} \rightarrow \mathit{tpat} \mathbin{\Lsh} b : S \hookleftarrow V$, where *spat* is a pattern for the original source type S, *tpat* is a pattern for the new source type T, and the inner program b has type $T \hookleftarrow V$. The same syntactic restrictions for invertibility apply to *spat* and *tpat*. Operationally, the source is transformed using $\lambda\ spat \rightarrow tpat$, and b is executed on the new source and the view; after that, the updated source must match *tpat*, and will be transformed back to the shape of *spat* (as if evaluating $\lambda\ spat \rightarrow tpat$ backwards). Dual to the view rearrangement rule, the source rearrangement rule also reflects a change of perspective by varying the source patterns. Notably, the postcondition for b

$$\langle\ tpat'\ tpat\ v \mid R'\ \overline{tpat'}\ \overline{tpat}\ v\ \rangle$$

says explicitly that the updated source produced by b should match *tpat′* (which is just *tpat* with its variables freshly renamed, to avoid name clashes with those variables in the other occurrence of *tpat*), which is a requirement often overlooked by novice BiGUL programmers.

Formalisation. The rearrangement rules we have presented rely essentially on pattern-matching comprehension relations, but the actual Agda formalisation takes a different approach and avoids modelling pattern-matching comprehension relations entirely. The formalisation is sketched here for the curious (and perhaps sceptical) reader. We build on Ko et al. (2016)'s formalisation of pattern matching, which makes use of heavily indexed types to guarantee various kinds of safety. To avoid introducing the heavy Agda notation, below we will give a simplified account, hiding all parameters and indices. Suppose that we are rearranging the view from *vpat* for type V to *wpat* for type W. The actual formalisation first defines a type *PatResult*: an inhabitant of *PatResult* is the result of matching a value of type V with *vpat*, and can be thought of as a mapping from every variable in *vpat* to the component matching the variable. Two relations are then defined: $Match : \mathcal{P}(V \times PatResult)$ relates $v : V$ and $r : PatResult$ exactly when v matches *vpat* and r is the result of the matching, and $Eval : \mathcal{P}(PatResult \times W)$ relates $r : PatResult$ and $w : W$ exactly when w is the result of evaluating *wpat* (as an expression) using r as the environment. With these two relations, we state the precondition for the whole **rearrV** program as $\langle\ s\ v \mid \exists r.\ Match\ v\ r \land R\ s\ r\ \rangle$, and the precondition for the inner program as $\langle\ s\ w \mid \exists r.\ Eval\ r\ w \land R\ s\ r\ \rangle$, making no use of pattern-matching comprehension relations.

4.5 Case analysis

More sophisticated programs require case analysis, for which BiGUL provides a powerful and intricate **case** construct. For a simple example, the height-resetting strategy for the rectangle width updating problem (Section 1) can be expressed as:

```
resetHeight : (ℕ , ℕ) ↩ ℕ
resetHeight = case
                ⋮ normal (w, _) v | w = v exit _
                ⋮ ⋮ skip fst
                ⋮ adaptive _ _
                ⋮ ⋮ λ _ v → (v, 0)
```

Roughly speaking, this program checks whether the width of the source is equal to the view, and skips if that is the case; otherwise, it creates a new rectangle whose width is the view and whose height is zero.

Syntax of **case**. In general, a case analysis in BiGUL has the form **case** *branches* : $S \hookleftarrow V$ where *branches* is an arbitrary number of **normal** or **adaptive** branches. For normal branches, the general form is **normal** M **exit** E ↳ b where $M : \mathcal{P}(S \times V)$ is called the *main condition*, $E : \mathcal{P}(S)$ is called the *exit condition*, and $b : S \hookleftarrow V$ is the branch body. For adaptive branches, the general form is **adaptive** M ↳ f where $M : \mathcal{P}(S \times V)$ is again the main condition, and $f : S \rightarrow V \rightarrow S$ is a function in the host language. The syntactic conventions described by Notation 3.5 and Notation 3.6 are also adopted for comprehension relations used as main or exit conditions. We make M and E comprehension relations only to simplify the presentation — in reality, where programs should be executable,

M and E should be "comprehension expressions" that compute to boolean values instead of propositions, but that would mess up the assertions where we would have to write propositions like $M\ s\ v = \mathit{true}$ instead of just $M\ s\ v$. We are careful not to sacrifice computability, though: the comprehension relations we use as conditions in this paper can all be straightforwardly converted to "comprehension expressions" in real programs.

The **case** *rule.* Operationally, the execution of a **case** finds the first branch whose main condition is satisfied by the source and view, and enters that branch. Suppose that the precondition and postcondition we want to verify for the entire **case** is R and R' respectively. The **case** rule in Figure 1 says that the precondition should be restricted to $R \cap \mathcal{D}$ where $\mathcal{D}$ is the union of all the main conditions, so that the precondition is strong enough to guarantee that some branch will be entered. (Of course, the consequence rule allows us to use just R as the precondition if $R \subseteq \mathcal{D}$.) The rest of the job is to verify each branch.

Normal branches. If a normal branch **normal** M **exit** E ↳ b is entered, its body b is executed; the **case** rule thus requires us to verify the behaviour of b by deriving the following triple:

$$\{\, R \cap \widehat{M} \,\}\ \ b\ \ \{\, R' \cap \langle\, s' \mathbin{\_} v \mid \widehat{M}\ s'\ v \wedge \widehat{E}\ s' \,\rangle \,\}$$

The precondition is strengthened with the main condition since we know that the source and view must satisfy the main condition if the branch is entered. However, the precise condition satisfied is not M — since the branches are tried in order and a branch is entered only when the main conditions of all the previous branches are not satisfied, we should regard the actual main condition of a branch as M intersected with the negations of the main conditions of all the previous branches. We denote this actual main condition by $\widehat{M}$, and the precondition for b is strengthened to $R \cap \widehat{M}$. (The programmer can avoid distinguishing actual and "superficial" main conditions by writing programs where the main conditions are all disjoint, in which case $\widehat{M} = M$.) As for the postcondition, in addition to R', we also require (i) that the updated source and the view satisfy the actual main condition $\widehat{M}$ and (ii) that the updated source satisfy the actual exit condition $\widehat{E}$, which is E intersected with the negations of the exit conditions of all the previous normal branches. Requirement (i) is for guaranteeing well-behavedness, while requirement (ii) makes exit conditions comparable with "assertions" (not to be confused with Hoare-style assertions) in reversible conditionals (Yokoyama and Glück 2007).

Adaptive branches. Requirement (i) above for normal branches turns out to be very restrictive, making normal branches only capable of dealing with "almost consistent" cases in practice. However, we often need branches whose main condition describes a particular kind of inconsistency and whose purpose is to repair that inconsistency — that is, their main conditions are supposed to be broken after updating, and this is against the nature of normal branches. Instead, for repairing inconsistency, we use adaptive branches, which are comparable with Foster et al. (2007)'s "fixup functions". When entered, an adaptive branch **adaptive** M ↳ f applies f to the source and view to produce an adapted source; this adapted source then takes the place of the original source, and the whole **case** is rerun. Naturally, requirements have to be imposed on f, as stated in the **case** rule:

$$\begin{array}{lll} \forall s, v.\quad (R \cap \widehat{M})\ s\ v & \Rightarrow & (R \cap \mathcal{N})\ (f\ s\ v)\ v \\ & \wedge & \forall s'.\ \ R'\ s'\ (f\ s\ v)\ v \ \Rightarrow\ R'\ s'\ s\ v \end{array}$$

Like normal branches, we know that R and $\widehat{M}$ hold for the original source s and the view v, and that has to be strong enough to make s and v satisfy two requirements:

- First, the adapted source $f\ s\ v$ and the view v can make the whole **case** rerun successfully. That is, they should satisfy the precondition R for the entire **case** and also $\mathcal{N}$, which denotes the union of the actual main conditions of the **normal** branches — this is for ensuring that the rerunning of the **case** will go into a normal branch and terminate there, instead of revisiting adaptive branches indefinitely.

- Second, the rerunning of the **case** establishes the postcondition for the updated source, the adapted source, and the view, but ultimately we want the postcondition established not for the adapted source but the original source. Therefore, whichever updated source s' is produced by the rerunning, the postcondition $R'\ s'\ (f\ s\ v)\ v$ established by the reunning has to be sufficient for the ultimate postcondition $R'\ s'\ s\ v$.

In practice, the first requirement leads us to write adaptive behaviour that performs enough inconsistency-repairing so as to be able to go back into normal branches, while the second requirement discourages us from radically changing the source during adaptation so that it is possible to derive properties about the original source from properties about the adapted source. (We will see how these two guidelines are applied in a more illustrative scenario in Section 7.)

Representing the **case** *rule in our derivation format.* Before we see some derivation examples, we should think about how the **case** rule — in particular, the two requirements for adaptive branches — are to be incorporated into our derivation format. Observe that the two requirements can be rewritten as relational inclusions:

$$\begin{array}{ll} & \forall s, v.\ \ (R \cap \widehat{M})\ s\ v \ \Rightarrow\ (R \cap \mathcal{N})\ (f\ s\ v)\ v \\ \equiv & R \cap \widehat{M}\ \subseteq\ \langle\, s\ v \mid (R \cap \mathcal{N})\ (f\ s\ v)\ v\,\rangle \\ & \forall s, v.\ \ (R \cap \widehat{M})\ s\ v \ \Rightarrow\ \forall s'.\ \ R'\ s'\ (f\ s\ v)\ v \ \Rightarrow\ R'\ s'\ s\ v \\ \equiv & \langle\, s'\ s\ v \mid R'\ s'\ (f\ s\ v)\ v\,\rangle \cap \langle\, \_\ s\ v \mid (R \cap \widehat{M})\ s\ v\,\rangle\ \subseteq\ R' \end{array}$$

which match the forms of the two inclusions in the consequence rule. Indeed, if f were a BiGUL operation (symbolising the rerunning of the **case**) such that $\{\,\mathcal{F}\,\}\ f\ \{\,\mathcal{F}'\,\}$, where the precondition

$$\mathcal{F}\ \ =\ \langle\, s\ v \mid (R \cap \mathcal{N})\ (f\ s\ v)\ v\,\rangle$$

states that (before the rerunning) the adapted source $f\ s\ v$ and the view v should be guaranteed to satisfy $R \cap \mathcal{N}$, and the postcondition

$$\mathcal{F}'\ =\ \langle\, s'\ s\ v \mid R'\ s'\ (f\ s\ v)\ v\,\rangle$$

states that (after the rerunning) R' is established for the updated source s', the adapted source $f\ s\ v$, and the view v, then we could invoke the consequence rule:

$$\frac{R \cap \widehat{M}\ \subseteq\ \mathcal{F} \qquad \{\,\mathcal{F}\,\}\ f\ \{\,\mathcal{F}'\,\} \qquad \mathcal{F}' \cap \langle\, \_\ s\ v \mid (R \cap \widehat{M})\ s\ v\,\rangle\ \subseteq\ R'}{\{\,R \cap \widehat{M}\,\}\ f\ \{\,R'\,\}} \tag{2}$$

We therefore propose the following derivation format for adaptive branches:

$$\begin{array}{l} \textbf{adaptive}\ M \\ \quad \{\,R \cap \widehat{M}\,\} \\ \quad\ \ \vdots \\ \quad \{\,\mathcal{F}\,\} \\ \quad f \\ \quad \{\,\mathcal{F}'\,\} \\ \quad\ \ \vdots \\ \quad \{\,R'\,\} \end{array}$$

The way to think about this format is that eventually we want to establish $\{\,R \cap \widehat{M}\,\}\ f\ \{\,R'\,\}$, but the actual precondition and postcondition of f are respectively $\mathcal{F}$ and $\mathcal{F}'$ instead, and hence we should invoke the consequence rule (2) and prove the two inclusions. Let us emphasise that assertions in adaptive branches do not really constitute triples, but are merely an organisation of the proof obligations for adaptive branches such that we can work with the proof obligations in the same way as we work with real triples.

Example 4.3 (rectangle width updating — resetting the height). Now we can verify the *resetHeight* program as follows, where the precondition is always true and the postcondition says that the consistency will be established and the height will be retained or reset depending on whether the view is consistent or not:

```
$\{\ \_\ \_\ \}$
$\{\ \langle\ \_\ \_\ \rangle \cap (\langle\ (w,\ \_)\ v \mid w = v\ \rangle \cup \langle\ \_\ \_\ \rangle)\ \}_1$
**case**
    **normal** $(w,\ \_)\ v \mid w = v$ **exit** $\_$
        $\{\ \langle\ \_\ \_\ \rangle \cap \langle\ (w,\ \_)\ v \mid w = v\ \rangle\ \}$
        $\{\ (w,\ h)\ v \mid w = v\ \}$
        $\{\ (w,\ h)\ v \mid \mathit{fst}\ (w,\ h) = v\ \}$
        **skip** $\mathit{fst}$
        $\{\ (w',\ h')\ (w,\ h)\ \_ \mid (w',\ h') = (w,\ h)\ \}$
        $\{\ (w',\ h')\ (w,\ h)\ v \mid w' = v \wedge (w = v \Rightarrow h' = h) \wedge (w \neq v \Rightarrow h' = 0)\ \}_3$
        $\{\ (w',\ h')\ (w,\ h)\ v \mid w' = v \wedge (w = v \Rightarrow h' = h) \wedge (w \neq v \Rightarrow h' = 0) \wedge w' = v \wedge \langle\ \_\ \rangle\ (w',\ h')\ \}_2$
    **adaptive** $\_\ \_$
        $\{\ \langle\ \_\ \_\ \rangle \cap (\langle\ \_\ \_\ \rangle \cap \neg\ \langle\ (w,\ \_)\ v \mid w = v\ \rangle)\ \}$
        $\{\ (w,\ \_)\ v \mid w \neq v\ \}_8$
        $\{\ \_\ v \mid \langle\ \_\ \_\ \rangle\ (v,\ 0)\ v \wedge \langle\ (w\ ,\ \_)\ v \mid w = v\ \rangle\ (v,\ 0)\ v\ \}$
        $\lambda\ \_\ v \rightarrow (v,\ 0)$
        $\{\ s'\ \_\ v \mid \langle\ (w',\ h')\ (w,\ h)\ v \mid w' = v \wedge (w = v \Rightarrow h' = h) \wedge (w \neq v \Rightarrow h' = 0)\ \rangle\ s'\ (v,\ 0)\ v\ \}_4$
        $\{\ (w',\ h')\ \_\ v \mid w' = v \wedge h' = 0\ \}_6$
        $\{\ (w',\ h')\ (w,\ h)\ v \mid w' = v \wedge (w = v \Rightarrow h' = h) \wedge (w \neq v \Rightarrow h' = 0)\ \}_7$
$\{\ (w',\ h')\ (w,\ h)\ v \mid w' = v \wedge (w = v \Rightarrow h' = h) \wedge (w \neq v \Rightarrow h' = 0)\ \}_5$
```

In this derivation, some assertions (like assertion 1) are given so that it is easier to compare the derivation with the generic **case** rule, but in practice we can often skip these assertions and see that the **case** rule is indeed applicable. For example, we tend to omit assertion 1 in practice since we can see that the adaptive branch is a catch-all branch; we even tend to omit assertion 2 since we can just check whether assertion 3 covers the extra conditions about the updated source, namely $w' = v \wedge \langle\ \_\ \rangle\ (w',\ h')$.

What happens in the adaptive branch is worth tracing. After the rerunning of the **case** produces an updated source s', assertion 4 states that the postcondition (assertion 5) holds for s', the adapted source $(v,\ 0)$, and the view v. We can then deduce that the updated width w' is v and, by substituting $(v,\ 0)$ for $(w,\ h)$ in the "retentive conjunct" $w = v \Rightarrow h' = h$ in assertion 4, that the updated height h' is zero, arriving at assertion 6. Having $h' = 0$ is sufficient for establishing the "resetting conjunct" $w \neq v \Rightarrow h' = 0$ in assertion 7, whose "retentive conjunct" $w = v \Rightarrow h' = h$ is vacuous due to assertion 8.

Example 4.4 (embedding pairs of transformations into BiGUL). The *resetHeight* program in fact exhibits a general programming pattern: a **case** with a normal branch accepting consistent states and leaving the source as it is, and an adaptive branch recovering from inconsistency. We can abstract this pattern to the following *emb* program, which takes a pair of (total) forward and backward transformations and embeds them into BiGUL:

```
$\mathit{emb} : \mathit{Eq}\ V \Rightarrow (S \rightarrow V) \rightarrow (S \rightarrow V \rightarrow S) \rightarrow (S \hookleftarrow V)$
$\mathit{emb}\ g\ p =$ **case**
                **normal** $s\ v \mid g\ s = v$ **exit** $\_$
                    **skip** $g$
```

```
    adaptive _ _
      p
```

The '*Eq V*' constraint in the type of *emb* indicates that we need decidable equality on V for the program to be executable. It is easy to see that $resetHeight = emb\ fst\ (\lambda\ \_\ v \to (v,\ 0))$. What we proved for *resetHeight* in Example 4.3 can be generalised to the following triple for *emb g p*, where g is used to define consistency:

$$\{\ \_\ \_\ \}\ \ emb\ g\ p\ \ \{\ s'\ s\ v \mid g\ s' = v \land (g\ s = v \Rightarrow s' = s) \land (g\ s \neq v \Rightarrow s' = p\ s\ v)\ \}$$

Interestingly, to prove this triple, we only require g and p to satisfy PutGet ($\forall s,\ v.\ g\ (p\ s\ v) = v$, which is Theorem 2.2's PutGet specialised for total functions); GetPut of *emb g p* arises from the logic of *emb* itself and does not depend on g and p. Indeed, in the case of *resetHeight*, the pair of transformations being embedded only satisfies PutGet.

On the other hand, if we have both PutGet and GetPut ($\forall s.\ p\ s\ (g\ s) = s$), then we can derive a stronger triple saying that the putback behaviour of *emb g p* completely coincides with p:

```
{ _ _ }
case
  normal s v | g s = v exit _
    { s v | g s = v }
    skip g
    { s′ s _ | s′ = s }
    { s′ s _ | s′ = p s v ∧ g s′ = v }₁
  adaptive _ _
    { s v | g s ≠ v }
    { s v | g (p s v) = v }₂
    p
    { s′ s v | s′ = p (p s v) v }
    { s′ s v | s′ = p s v }₃
{ s′ s v | s′ = p s v }
```

GetPut is used for assertion 1, and PutGet is used for assertion 2. Assertion 3 requires the PutTwice property:

$$\forall s,\ v.\ \ p\ (p\ s\ v)\ v = p\ s\ v$$

which is known to follow from PutGet and GetPut (see, e.g., Fischer et al. 2015a, Section 3).

5 RANGE TRIPLES AND TOTAL FORWARD CONSISTENCY

We have seen in Example 4.4 that any pair of transformations satisfying only PutGet can be embedded into BiGUL and become well-behaved by using the *emb* program to wrap extra logic around the transformations to achieve GetPut, although the putback behaviour is not exactly the same as the backward transformation being embedded. For example, *resetHeight* in Example 4.3 can be seen as *emb fst reset* where $reset = \lambda\ \_\ v \to (v,\ 0)$. There is, however, a way to embed *reset* precisely, which is to express it directly in terms of BiGUL's constructs:

```
alwaysResetHeight : (ℕ , ℕ) ⟵ ℕ
alwaysResetHeight = rearrV v → (v, 0)
                        replace
                      * replace
```

We can easily prove that the putback behaviour of *alwaysResetHeight* coincides with *reset* by deriving the putback triple $\{\, \_ \ \_ \,\}$ *alwaysResetHeight* $\{\, s' \ \_ \ v \mid s' = (v,\ 0) \,\}$; then, by Theorem 2.2, the *put* and *get* semantics of *alwaysResetHeight* must form a well-behaved pair. Let us compare *alwaysResetHeight* and *emb fst reset*:

- With *emb fst reset*, apart from providing *reset*, we also need to provide a forward transformation to pair with *reset* and prove PutGet for them. In the end we obtain well-behavedness but not precise embedding of *reset*'s behaviour.
- With *alwaysResetHeight*, we simply express *reset* in BiGUL, and obtain the corresponding forward transformation, well-behavedness, and precise embedding.

It might seem that *alwaysResetHeight* is clearly the better solution, but that is only because we have been looking in the putback direction exclusively. In the forward direction, Theorem 3.8 tells us that

$$\mathcal{G}(get\ (emb\ fst\ reset)) \ \subseteq\ \langle\, (w,\ \_)\ v \mid w = v \,\rangle \quad \text{and} \quad \mathcal{G}(get\ alwaysResetHeight) \ \subseteq\ \langle\, (w,\ 0)\ v \mid w = v \,\rangle$$

so both *get* (*emb fst reset*) and *get alwaysResetHeight* necessarily behave like *fst*. But that is not the whole story — in fact, *get* (*emb fst reset*) is more versatile than *get alwaysResetHeight*, because the former can compute successfully on all inputs, whereas the latter can only compute successfully on inputs whose second component is zero, and is usually not what one wants in practice.

The moral of the above story is that we still lack the machinery for understanding forward behaviour, so that we could not see that *alwaysResetHeight* was actually less versatile. What helped us to arrive at the right conclusion is the ability to estimate the *domain* of a forward transformation, i.e., the subset of sources on which the forward transformation can compute successfully. To make such estimates for BiGUL programs, which are understood as putback transformations, the key insight is that, for a well-behaved pair of *put* and *get*, the domain of *get* coincides with the *range* of *put*, i.e., the subset of sources that can be produced by *put*. (To see the coincidence, observe that PutGet (as stated in Theorem 2.2) can be read roughly as "if s' is produced by *put b*, i.e., s' is in the range of *put b*, then *get b* will compute successfully on s', i.e., s' will be in the domain of *get b*", and GetPut says the converse.) The problem with *alwaysResetHeight* is now clear: it can only produce the sources whose second component is zero, so *get alwaysResetHeight* can compute successfully only on those sources. In contrast, *emb fst reset* is capable of producing all possible pairs. Our way ahead is to develop machinery for making such range estimates reliably, and that machinery is a second set of Hoare-style triples.

5.1 Theory of range triples

Definition 5.1. A *range triple* is a BiGUL program $b : S \hookleftarrow V$ surrounded by two *range assertions*:

$$\{\!\{\, R \,\}\!\}\ \ b\ \ \{\!\{\, P' \,\}\!\}$$

where $R : \mathcal{P}(S \times V)$ is the precondition or *input range* (on the original source and the view) and $P' : \mathcal{P}S$ is the postcondition or *output range* (on the updated source). Valid range triples are inductively defined by the proof rules in Figure 2 (which will be explained in Section 5.2).

While the intended interpretation of a range triple for b is about the range of *put b*, we actually need a (slightly) stronger interpretation about *get b* (to make Theorem 5.4 work), as stated by the following soundness theorem, whose conclusion is, in a sense, dual to that of Theorem 3.3.

Theorem 5.2 (soundness of range triples). *Let* $b : S \hookleftarrow V$, $R : \mathcal{P}(S \times V)$, *and* $P' : \mathcal{P}S$.

If $\{\!\{\, R \,\}\!\}\ \ b\ \ \{\!\{\, P' \,\}\!\}$ *then* $\forall s.\ \ P'\ s \ \Rightarrow\ \exists v.\ \ get\ b\ s = \mathit{Just}\ v \ \wedge\ R\ s\ v \quad .$

We can recover the intended interpretation of range triples by showing that the conclusion of Theorem 5.2 is equivalent to a statement primarily about *put*, as stated in the following lemma.

Lemma 5.3. *The conclusion of Theorem 5.2 is equivalent to:*

$$(\forall s'.\ \ P'\ s' \ \Rightarrow\ \exists s, v.\ \ R\ s\ v\ \wedge\ \mathit{put}\ b\ s\ v = \mathit{Just}\ s') \quad \wedge \quad \mathcal{G}(\mathit{get}\ b) \cap \langle\, s\ \_ \mid P'\ s \,\rangle\ \subseteq\ R$$

The left conjunct in Lemma 5.3 is the primary way in which we think about the range triples: if $\{\!\{\, R \,\}\!\}\ b\ \{\!\{\, P' \,\}\!\}$ can be derived, then the range of updated sources produced by applying *put b* to those inputs satisfying R will be at least P'. The right conjunct in Lemma 5.3 says that there is an unintended "side effect" when we think about these triples in the putback direction: the input range considered will be forced to include those related by *get* with its domain restricted to P'. So, for example, we will not be able to deduce $\{\!\{\, m\ n \mid m = n + 1 \,\}\!\}$ **replace** $\{\!\{\, \_ \,\}\!\}$ even though the left conjunct in Lemma 5.3 is true for this pair of R and P'. This "side effect" normally does not prevent us from deriving range triples, though, since preconditions are normally larger than consistency relations, which in turn contain the graphs of *get* transformations.

Back in Section 3, where we only had putback triples, Theorem 3.8 only enabled us to understand the forward behaviour of BiGUL programs to a limited extent. Now supplemented with range triples, we can prove a stronger and satisfactory result.

Theorem 5.4 (total forward consistency). *Let* $b : S \hookleftarrow V$, $R : \mathcal{P}(S \times V)$, $C : \mathcal{P}(S \times V)$, *and* $P' : \mathcal{P}S$.

$$\mathit{If}\quad \{\, R \,\}\ \ b\ \ \{\, s'\ \_\ v \mid C\ s'\ v \,\}\quad \mathit{and}\quad \{\!\{\, R \,\}\!\}\ \ b\ \ \{\!\{\, P' \,\}\!\}\quad \mathit{then}\quad \forall s.\ \ P'\ s\ \Rightarrow\ \exists v.\ \ \mathit{get}\ b\ s = \mathit{Just}\ v\ \wedge\ C\ s\ v \quad .$$

Proof. Suppose that P' holds for a source s. By the range triple and Theorem 5.2, *get b s* will compute successfully to some view v such that $R\ s\ v$ holds, making s and v fall into $\mathcal{G}(\mathit{get}\ b) \cap R$. The putback triple and Theorem 3.8 can then take over and establish $C\ s\ v$ as required. □

Theorem 5.4 tells us that, by supplementing a putback triple for b with a range triple with the same precondition, we can know on which subset of sources *get b* will compute successfully and that the behaviour of *get b* will conform to the consistency relation stated in the putback triple.

In summary, now we have enough machinery to tell us all we want to know about the bidirectional behaviour of a BiGUL program: By deriving a putback triple $\{\, R \,\}\ b\ \{\, s'\ s\ v \mid C\ s'\ v \wedge R'\ s'\ s\ v \,\}$ to reason about the behaviour of b, we know that *put b* will compute successfully on R, establish consistency C, and have retentive behaviour R'. Then, by additionally deriving a range triple $\{\!\{\, R \,\}\!\}\ b\ \{\!\{\, P' \,\}\!\}$ to estimate the range of b, we know that *get b* will compute successfully on P' and conform to the same consistency relation C established by *put b*. Notably, as we will see next, derivations of range triples are usually significantly easier than derivations of putback triples, so in practice there is usually not much more work to do than deriving putback triples.

5.2 The range proof rules

The range proof rules are shown in Figure 2. The most interesting rule is probably the consequence rule (the right one in the second row), whose direction is just the opposite of the putback consequence rule given in Figure 1. An explanation is that the ultimate interpretation of range triples, as stated by Theorem 5.2, is about forward behaviour, and the output/input range in a range triple in fact serves the role of precondition/postcondition for the forward transformation. But, interestingly, the consequence rule can also be understood in the putback direction: If $\{\!\{\, R \,\}\!\}\ b\ \{\!\{\, P' \,\}\!\}$ has been established, meaning that the inputs in R can induce everything in P' through the execution of b, then a larger input range T can still induce everything in P', or indeed everything in any output range Q' smaller than P'. In fact, T is not necessarily larger than R: if what we eventually target is a smaller output range, then we will be allowed to also shrink the input range. The consequence rule makes it clear what kind of shrinking is valid: we are allowed to use Q' to constrain the sources in the input range. Since the direction of the range consequence rule is the opposite of the putback one, when deriving a range triple using the consequence rule in our derivation format, logical implications go upwards, and we can use the postconditions

$$\frac{}{\{\!\{\,\emptyset\,\}\!\}\ \ \mathbf{fail}\ \ \{\!\{\,\emptyset\,\}\!\}} \qquad \frac{}{\{\!\{\, s\ v \mid s = v \,\}\!\}\ \ \mathbf{replace}\ \ \{\!\{\,\_\,\}\!\}} \qquad \frac{}{\{\!\{\, s\ v \mid f\ s = v \,\}\!\}\ \ \mathbf{skip}\ f\ \ \{\!\{\,\_\,\}\!\}}$$

$$\frac{\{\!\{\, L \,\}\!\}\ \ l\ \ \{\!\{\, P' \,\}\!\} \qquad \{\!\{\, R \,\}\!\}\ \ r\ \ \{\!\{\, Q' \,\}\!\}}{\{\!\{\, L * R \,\}\!\}\ \ l * r\ \ \{\!\{\, P' * Q' \,\}\!\}} \qquad \frac{R \cap \langle\, s\ \_ \mid Q'\ s \,\rangle \ \subseteq\ T \qquad \{\!\{\, R \,\}\!\}\ \ b\ \ \{\!\{\, P' \,\}\!\} \qquad Q' \ \subseteq\ P'}{\{\!\{\, T \,\}\!\}\ \ b\ \ \{\!\{\, Q' \,\}\!\}}$$

$$\frac{\{\!\{\, s\ \mathit{wpat} \mid R\ s\ \overline{\mathit{wpat}} \,\}\!\}\ \ b\ \ \{\!\{\, P' \,\}\!\}}{\{\!\{\, s\ \mathit{vpat} \mid R\ s\ \overline{\mathit{vpat}} \,\}\!\}\ \ \mathbf{rearrV}\ \mathit{vpat} \longrightarrow \mathit{wpat} ↳ b\ \ \{\!\{\, P' \,\}\!\}}$$

$$\frac{\{\!\{\, \mathit{tpat}\ v \mid R\ \overline{\mathit{tpat}}\ v \,\}\!\}\ \ b\ \ \{\!\{\, \mathit{tpat} \mid P'\ \overline{\mathit{tpat}} \,\}\!\}}{\{\!\{\, \mathit{spat}\ v \mid R\ \overline{\mathit{spat}}\ v \,\}\!\}\ \ \mathbf{rearrS}\ \mathit{spat} \longrightarrow \mathit{tpat} ↳ b\ \ \{\!\{\, \mathit{spat} \mid P'\ \overline{\mathit{spat}} \,\}\!\}}$$

$$\frac{\forall n = (\mathbf{normal}\ M\ \mathbf{exit}\ E ↳ b) \in \mathit{branches}.\quad \{\!\{\, R \cap \widehat{M} \,\}\!\}\ \ b\ \ \{\!\{\, P'_n \,\}\!\}}{\{\!\{\, R \,\}\!\}\ \ \mathbf{case}\ \mathit{branches}\ \ \{\!\{\, \mathcal{P}' \,\}\!\}} \quad \begin{array}{l} \mathbf{where} \\ \mathcal{P}' = \bigcup\ [\ P'_n \cap \widehat{E} \mid n = (\mathbf{normal}\ M\ \mathbf{exit}\ E ↳ b) \in \mathit{branches}\ ] \end{array}$$

Fig. 2. Range proof rules

for a node as additional premises when proving implications between preconditions for the same node, opposite to what we do in putback derivations.

Other rules should be largely intuitive. The **fail** rule has the empty predicate as its output range since **fail** can never produce anything (and its input range can be any relation because of the consequence rule). The **replace** rule says that **replace** can produce everything as long as the input range is large enough — because of the "side effect" explained below Lemma 5.3, the input range has to be large enough to include the graph of **replace**'s forward semantics, which is the identity transformation. The **skip** rule has the same precondition as its putback counterpart and says that **skip** can produce everything. The product and rearrangement rules are analogous to their putback counterparts. For **case**, only normal branches matter since execution of **case** always ends in a normal branch. We estimate an output range P'_n for the body of every normal branch n, and the estimated output range for that branch is P'_n intersected with the actual exit condition, since only outputs satisfying the actual exit condition can be produced. The estimated output range for the entire **case** is then the union of the estimated output ranges for all the normal branches.

Example 5.5 (embedding pairs of transformations in BiGUL). As we mentioned, derivations of range triples can be very straightforward in simple cases. In the case of *emb*, for example, we can effortlessly prove that it produces everything:

```
{{ _ _ }}
case
  normal s v | g s = v exit _
    {{ s v | g s = v }}
    skip g
    {{ _ }}
  adaptive _ _
    p
{{ _ }}
```

Example 5.6 (rectangle width updating — always resetting the height). Verifying the range of *alwaysResetHeight* is a more interesting example, where we will see how a non-trivial output range can be derived with the help of the consequence rule:

```
⦃ _ _ ⦄
rearrV v → (v, 0)
    ⦃ _ (_ , 0) ⦄
        ⦃ _ _ ⦄
        ⦃ w v | w = v ⦄
        replace
        ⦃ _ ⦄
    *   $⦃ \_\ 0 ⦄_3$
        $⦃ h\ v \mid h = v ⦄_1$
        replace
        $⦃ \_ ⦄_2$
        $⦃ 0 ⦄_4$
    ⦃ (_ , 0) ⦄
⦃ (_ , 0) ⦄
```

The interesting part is the second **replace**, for which we first establish the input and output ranges as assertions 1 and 2 according to the **replace** rule. However, because of the outer **rearrV**, the views in the actual input range for the second **replace** are restricted to zero, as stated by assertion 3, which does not contain assertion 1. We therefore need to shrink our estimate of the output range of **replace** to just zero (assertion 4) to allow us to also restrict the views in the input range to zero. Logically, assertions 1 and 4 together indeed imply assertion 3, adhering to the consequence rule.

6 RECURSION

BiGUL is designed to work datatype-generically (Gibbons 2007) with inductive data structures. A lot of recursive programs processing inductive data have been written using the Haskell port of BiGUL, and our Hoare-style logic would not be useful at all if we could not reason about such recursive BiGUL programs. For example, let us take a peek at the key-based list alignment program *keyAlign* in Figure 3, which we will verify in Section 7. All we need to care about in regard to *keyAlign* now is its recursive structure. Both the source and view are lists, and in the second branch of the **case**, they are both non-empty. Inside the branch, each of them is rearranged into a pair of its head and tail, and we recursively invoke the program to process the tails. Intuitively, we know that this program terminates for any input because the size of the initial source and view is strictly larger than the size of the source and view at the point of the recursive invocation. We will need to incorporate this size-based termination argument into our proof rules for recursive programs, and show that the rules are sound.

There is difficulty dealing with recursive programs in the Agda formalisation underlying this paper, though. Observe that the program structure of *keyAlign* is infinite, which is fine in Haskell; in the Agda formalisation, however, BiGUL programs are modelled inductively and are necessarily finite. One possible solution is to redefine BiGUL programs coinductively and bring in the partiality monad (Capretta 2005) to explicitly model non-termination in Haskell, but this means abandoning most (if not all) of the previous formalisation effort. Another possible solution is to stay with inductive BiGUL programs and introduce a "terminating fixed-point" which can decide, for every input, how many times the body of a fixed-point should be expanded, thereby circumventing the modelling of infinite program structures. This approach will be relevant in a constructive setting, but we anticipate that there will be extra constructivity requirements that are not relevant for the Haskell port of BiGUL, in which most (recursive) BiGUL programs are written. Since what we aim at in this paper is not thorough formalisation but a semi-formal reasoning framework for the working BiGUL programmer, we will develop just enough theory to justify our rules for reasoning about recursive programs, and refrain from delving into coinductiveness or constructive termination.

Let $b : S \hookleftarrow V$ be a recursive program of the form $b = f\ b$ where $f : (S \hookleftarrow V) \rightarrow (S \hookleftarrow V)$ is the usual non-recursive function defining the body of b, and suppose that we want to verify that b can successfully turn any input satisfying a precondition R into an output satisfying a postcondition R'. We cannot hope to establish $\{\, R \,\}\ b\ \{\, R' \,\}$ (although we will abuse this notation in Section 7) since putback triples are defined for finite programs, whereas b is infinite. Instead, when we say in this paper that we are verifying b, what we precisely mean is verifying the behaviour of all *finite expansions* of its body f, where finite expansions are defined by:

$$\begin{array}{l}
\mathit{expand} : \mathbb{N} \rightarrow ((S \hookleftarrow V) \rightarrow (S \hookleftarrow V)) \rightarrow (S \hookleftarrow V) \\
\mathit{expand}\ \mathrm{zero}\qquad f = \mathbf{fail} \\
\mathit{expand}\ (\mathrm{suc}\ n)\ f = f\ (\mathit{expand}\ n\ f)
\end{array}$$

The basic idea is to prove something about f like:

$$\forall \mathit{rec} : S \hookleftarrow V.\quad \{\, R \,\}\ \ \mathit{rec}\ \ \{\, R' \,\} \quad\Rightarrow\quad \{\, R \,\}\ \ f\ \mathit{rec}\ \ \{\, R' \,\} \tag{3}$$

which can then be iterated to produce $\{\, R \,\}\ \mathit{expand}\ n\ f\ \{\, R' \,\}$ for any n — the base case is $\{\, R \,\}\ \mathbf{fail}\ \{\, R' \,\}$, which implies $\{\, R \,\}\ f\ \mathbf{fail}\ \{\, R' \,\}$, and then $\{\, R \,\}\ f\ (f\ \mathbf{fail})\ \{\, R' \,\}$, etc. This idea cannot be directly valid, however, since in general only a subset of R can be successfully processed by a finite expansion of f (whose execution can be thought of as executing b but allowing recursive invocations only to a certain depth). We thus introduce a function of type $S \rightarrow V \rightarrow \mathbb{N}$ for measuring the size of the source and view in assertions, and include size restrictions in the preconditions for the finite expansions, leading to the following theorem.

Theorem 6.1 (finite expansion of putback triples). *Let* $f : (S \hookleftarrow V) \rightarrow (S \hookleftarrow V)$, $R : \mathcal{P}(S \times V)$, $R' : \mathcal{P}(S \times S \times V)$, *and* $\mathit{measure} : S \rightarrow V \rightarrow \mathbb{N}$. *If*

$$\begin{array}{ll}
\forall n, \mathit{rec}. & (\forall m.\ \{\, R \cap \langle\, s\ v \mid \mathit{measure}\ s\ v = m \wedge m < n \,\rangle \,\}\ \ \mathit{rec}\ \ \{\, R' \,\}) \\
& \quad\Rightarrow\quad \{\, R \cap \langle\, s\ v \mid \mathit{measure}\ s\ v = n \,\rangle \,\}\ \ f\ \mathit{rec}\ \ \{\, R' \,\}
\end{array} \tag{PutbackRecursion}$$

then:

$$\forall l, n.\ \ n \leqslant l\ \Rightarrow\ \ \{\, R \cap \langle\, s\ v \mid \mathit{measure}\ s\ v = n \,\rangle \,\}\ \ \mathit{expand}\ (\mathrm{suc}\ l)\ f\ \ \{\, R' \,\}$$

PutbackRecursion will be the proof rule we use for reasoning about the putback behaviour of recursive programs. It combines the basic proof idea (3) with the sized-based termination argument given in the beginning of this section: in the precondition for $f\ \mathit{rec}$, the size of the input source and view is bound to a ghost variable n, and we can make recursive invocations wherever the size m of the current source and view is strictly less than n. The soundness of PutbackRecursion is justified by Theorem 6.1, whose conclusion implies that any input in R can be successfully turned into an output in R' as long as f is expanded enough times. We again omit the proof of the theorem (which is by nested induction on l and n) from the presentation.

Analogously, we have a RangeRecursion rule for estimating the output ranges of recursive programs.

Theorem 6.2 (finite expansion of range triples). *Let* $f : (S \hookleftarrow V) \rightarrow (S \hookleftarrow V)$, $R : \mathcal{P}(S \times V)$, $P' : \mathbb{N} \rightarrow \mathcal{P} S$, *and* $\mathit{measure} : S \rightarrow V \rightarrow \mathbb{N}$. *If*

$$\begin{array}{ll}
\forall n, \mathit{rec}. & (\forall m.\ \{\!\{\, R \cap \langle\, s\ v \mid \mathit{measure}\ s\ v = m \,\rangle \,\}\!\}\ \ \mathit{rec}\ \ \{\!\{\, P'\ \ m \cap \langle\, \_ \mid m < n \,\rangle \,\}\!\}) \\
& \quad\Rightarrow\quad \{\!\{\, R \cap \langle\, s\ v \mid \mathit{measure}\ s\ v = n \,\rangle \,\}\!\}\ \ f\ \mathit{rec}\ \ \{\!\{\, P'\ \ n \,\}\!\}
\end{array} \tag{RangeRecursion}$$

```
keyAlign : Eq K ⇒ (S → K) → (V → K) → (S ↩ V) → (V → S) → ([S] ↩ [V])
keyAlign ks kv b c =
  case
    normal [] [] exit []
      rearrV [] → ()
        skip const ()
    normal (s :: _) (v :: _) | ks s = kv v exit (_ :: _)
      rearrS (s :: ss) → (s, ss)
        rearrV (v :: vs) → (v, vs)
              b
            * keyAlign ks kv b c
    adaptive (_ :: _) []
      λ _ _ → []
    adaptive ss (v :: _) | kv v ∈ map ks ss
      λ ss (v :: _) → extract ks kv v ss
    adaptive _ (_ :: _)
      λ ss (v :: _) → c v :: ss
  where
    extract : Eq K ⇒ (S → K) → (V → K) → V → [S] → [S]
    extract ks kv v (s :: ss) = if ks s = kv v then s :: ss
                                              else let (s′ :: ss′) = extract ks kv v ss
                                                   in  s′ :: s :: ss′
```

Fig. 3. Key-based list alignment in BiGUL

then:

$$\forall l, n.\ \ n \leqslant l \ \Rightarrow\ \ \{\!\{\, R \cap \langle\, s\ v \mid \mathit{measure}\ s\ v = n \,\rangle \,\}\!\}\ \ \mathit{expand}\ (\mathrm{suc}\ l)\ f\ \ \{\!\{\, P'\ n \,\}\!\}$$

The RangeRecursion rule instructs us to derive an output range $P'\ n$ that can depend on the ghost variable n bound to the size of the input source and view in the precondition. (For example, the range triple we will derive for *keyAlign* is $\{\!\{\, \_\ vs \mid \mathit{length}\ vs = n \,\}\!\}$ *keyAlign* ... $\{\!\{\, ss \mid \mathit{length}\ ss = n \,\}\!\}$.) Recursive invocations can be made in the derivation, and the estimated output range for a recursive invocation is $P'\ m$ where m is the size of the current source and view, provided that m is strictly less than n — otherwise, the estimated output range will be empty. The conclusion of Theorem 6.2 justifies the soundness of RangeRecursion, as it implies that every output in $\bigcup_{n\,:\,\mathbb{N}} P'\ n$ can be produced from some input of the right size in R as long as f is expanded enough times.

Having the two recursion rules, we are now ready to verify *keyAlign*.

7 VERIFYING KEY-BASED LIST ALIGNMENT

Alignment is one of the most important issues in synchronisation. In this paper, we consider the specialised (and asymmetric) setting where both the source and view are lists, which are consistent exactly when they have the same length and an element-level consistency relation is satisfied by each pair of the source and view elements at the same position (for example, the source elements can be rectangles and the view elements are their widths). View elements may be inserted, deleted, modified, or reordered. To put the updated view list back into the source list, we need to align the two lists, i.e., decide for each view element to which source element it corresponds (if any), before we can invoke an element-level consistency restorer on the right pairs of source and view elements.

Several variants of list alignment have been implemented in BiGUL (Mendes et al. 2016; Zan et al. 2016). In this paper we choose to verify a variant that is non-trivial and yet not overly complicated: key-based alignment, an implementation of which was presented and explained with a concrete scenario by Hu and Ko (2017, Section 6.2). Their program *keyAlign* is shown in Figure 3. The types of source and view elements are S and V respectively. The program takes two functions $ks : S \to K$ and $kv : V \to K$ as arguments, which can be used to extract a key value of type K from every source or view element, and the type K should support decidable equality. Two more arguments b and c are needed to deal with two of the three possible situations that can result from an alignment:

- A view element v is deemed to correspond to a source element s only if their keys match, i.e., $ks\ s = kv\ v$; an element-level synchroniser $b : S \hookleftarrow V$ will then be invoked on this pair of source and view elements.
- If no source element has the same key as a view element, a function $c : V \to S$ will be used to create a temporary corresponding source; this temporary source will then be fully synchronised with the view using b.
- A source element will be deleted if there is no corresponding view element.

Here is a quick overview of the program: The first branch is the base case. The second branch deals with "happy coincidences": the head elements in the source and view lists match, so we can simply synchronise the heads and recursively process the tails. The third branch deletes everything in the source when the view is empty. The fourth branch is the most interesting: the head view element v has a corresponding source element (which is not at the head position), so we *extract* the first source element with the same key as v and put it at the head (intending to re-enter the second branch afterwards). When there is no source element corresponding to the head view element, the fifth and last branch uses c to create a temporary corresponding source element. Note that the program uses some partial functions, which are fine in Haskell but not in Agda: *extract*, for example, misses two cases for empty source lists, and these cases have to be added for verification in Agda. These missing cases are irrelevant, however, since *keyAlign* does not invoke *extract* on empty source lists.

To verify *keyAlign*, we need to make some assumptions about its arguments. For simplicity, we assume that b can compute successfully for any pair of source and view elements as long as they have the same key; also, b should guarantee that the updated source and the view will have the same key, apart from any other postcondition $R' : \mathcal{P}(S \times S \times V)$ it can establish. As a putback triple:

$$\{\, s\ v \mid ks\ s = kv\ v \,\}\ \ b\ \ \{\, R' \cap \langle\, s'\ \_\ v \mid ks\ s' = kv\ v \,\rangle \,\}$$

We will abbreviate $R' \cap \langle\, s'\ \_\ v \mid ks\ s' = kv\ v \,\rangle$ as T'. For the source-creating function c, since a created source will be further processed by b, we require the key of the created source to be the same as that of the view, i.e.,

$$\forall v.\ \ ks\ (c\ v) = kv\ v$$

We can then derive:

$$\forall n.\quad \{\, \_\ vs \mid length\ vs = n \,\}\ \ keyAlign\ ks\ kv\ b\ c\ \ \{\, ss'\ ss\ vs \mid \exists\, \widetilde{ss}.\ T'^{\star}\ ss'\ \widetilde{ss}\ vs \wedge Retentive\ ss\ vs\ \widetilde{ss} \,\}$$

In the precondition, we use the length of the view list as the termination measure, but otherwise impose no restrictions on the source and view lists. In the postcondition, the relation $T'^{\star} : \mathcal{P}([S] \times [S] \times [V])$ is defined inductively by the following two rules:

$$\begin{array}{l} T'^{\star}\ []\qquad\quad []\qquad\ \ []\\ T'^{\star}\ (s' :: ss')\ (\tilde{s} :: \widetilde{ss})\ (v :: vs)\ \ \Leftarrow\ \ T'\ s'\ \tilde{s}\ v \wedge T'^{\star}\ ss'\ \widetilde{ss}\ vs \end{array}$$

The postcondition thus guarantees that the updated source list ss' will have the same length as vs, and for each pair of source element s' in ss' and view element v in vs at the same position, T' will be established for s', some source element $\tilde{s}$, and v. The elements $\tilde{s}$ are collected into a list $\widetilde{ss}$, and $Retentive\ ss\ vs\ \widetilde{ss}$ says that $\widetilde{ss}$ contains those source elements in *ss* that correspond to some view element in *vs*. This *Retentive* relation turns out to be slightly tricky to define, especially when we do not require that keys in a list are all unique; our definition of *Retentive* says that if a key appears *n* times in the view list, then the first *n* elements with that key in the original source list will be retained.

Due to space restrictions, we only sketch the verification of the second normal branch and the second adaptive branch. The assertions for the second normal branch are as follows, where we omit the invocations of the **rearrS**, **rearrV**, and product rules since they are straightforward in this case:

```
$\textbf{normal}\ (s :: \_)\ (v :: \_) \mid ks\ s = kv\ v\ \textbf{exit}\ (\_ :: \_)$
    $\{\, (s :: \_)\ (v :: vs) \mid 1 + length\ vs = n \wedge ks\ s = kv\ v \,\}_6$
    $\textbf{rearrS}\ (s :: ss) \to (s, ss)$
        $\textbf{rearrV}\ (v :: vs) \to (v, vs)$
                $\{\, s\ v \mid ks\ s = kv\ v \,\}$
                $b$
                $\{\, T' \,\}$
            $*$ $\{\, \_\ vs \mid 1 + length\ vs = n \,\}_1$
                $\{\, \_\ vs \mid length\ vs = pred\ n \wedge pred\ n < n \,\}_2$
                $keyAlign\ ks\ kv\ b\ c$
                $\{\, ss'\ ss\ vs \mid \exists\, \widetilde{ss}.\ T'^{\star}\ ss'\ \widetilde{ss}\ vs \wedge Retentive\ ss\ vs\ \widetilde{ss} \,\}_3$
    $\{\, (s' :: ss')\ (s :: ss)\ (v :: vs) \mid T'\ s'\ s\ v \wedge \exists\, \widetilde{ss}.\ T'^{\star}\ ss'\ \widetilde{ss}\ vs \wedge Retentive\ ss\ vs\ \widetilde{ss} \,\}_4$
    $\{\, (s' :: ss')\ ss\ (v :: vs) \mid \exists\, \widetilde{ss}.\ T'^{\star}\ (s' :: ss')\ \widetilde{ss}\ (v :: vs) \wedge Retentive\ ss\ (v :: vs)\ \widetilde{ss} \wedge ks\ s' = kv\ v \,\}_5$
```

We first look at how the PutbackRecursion rule is applied. Assertion 1 is the actual precondition for the recursive invocation, and we rewrite it into assertion 2, saying that the length of the view list at this point is *pred n*, which is strictly less than *n* since assertion 1 says that *n* is a successor. (The predecessor function is defined by *pred* zero = zero and *pred* (suc *n*) = *n*, so *pred n* is not necessarily less than *n*.) The recursive invocation will thus succeed and establish assertion 3. For the postconditions, the **rearrS**, **rearrV**, and product rules give us assertion 4, while we need to prove assertion 5. The last conjunct $ks\ s' = kv\ v$ in assertion 5 is part of $T'\ s'\ s\ v$ in assertion 4 by definition. From $T'\ s'\ s\ v$ and $\exists\, \widetilde{ss}.\ T'^{\star}\ ss'\ \widetilde{ss}\ vs$ in assertion 4, we see that we should use $s :: \widetilde{ss}$ as the new $\widetilde{ss}$ in assertion 5. We are left to prove $Retentive\ (s :: ss)\ (v :: vs)\ (s :: \widetilde{ss})$ (since *ss* in assertion 5 is $s :: ss$ in assertion 4), which is implied by $Retentive\ ss\ vs\ \widetilde{ss}$ in assertion 4 and $ks\ s = kv\ v$ in assertion 6.

Now we turn to the second adaptive branch:

```
$\textbf{adaptive}\ ss\ (v :: \_) \mid kv\ v \in map\ ks\ ss$
    $\{\, ss\ (v :: \_) \mid kv\ v \in map\ ks\ ss \,\}_1$
    $\{\, ss\ (v :: vs) \mid \langle\, (s :: \_)\ (v :: \_) \mid ks\ s = kv\ v \,\rangle\ (extract\ ks\ kv\ v\ ss)\ (v :: vs) \,\}_2$
    $\lambda\ ss\ (v :: \_) \to extract\ ks\ kv\ v\ ss$
    $\{\, ss'\ ss\ (v :: vs) \mid \langle\, ss'\ ss\ vs \mid \exists\, \widetilde{ss}.\ T'^{\star}\ ss'\ \widetilde{ss}\ vs \wedge Retentive\ ss\ vs\ \widetilde{ss} \,\rangle\ ss'\ (extract\ ks\ kv\ v\ ss)\ (v :: vs) \,\}_3$
    $\{\, ss'\ ss\ vs \mid \exists\, \widetilde{ss}.\ T'^{\star}\ ss'\ \widetilde{ss}\ vs \wedge Retentive\ ss\ vs\ \widetilde{ss} \,\}_4$
```

Assertion 1 (where we omit the negations of the previous main conditions) implies assertion 2, which is the condition for re-entering the second normal branch: the source list *ss* must be non-empty so that *extract ks kv v ss* will successfully produce a non-empty list, whose head will then have the same key as the head view element *v*. After adaptation, the rerunning of the **case** establishes assertion 3, which should be shown to imply assertion 4, the final postcondition. Assertion 3 says that $T'^{\star}$ holds for the updated source list *ss*′, some list $\widetilde{ss}$ of source elements, and the view list, and we can directly use $\widetilde{ss}$ as the witness and establish the first conjunct in assertion 4. Assertion 3 also says that $\widetilde{ss}$ retains certain elements from the adapted source list, which is just the original source list with its first source element having key $kv\ v$ moved to the head position, so we know that $\widetilde{ss}$ retains the same elements in the original source list as well. This is an illustrative example showing that adaptation should be done cautiously to maintain sufficient links between the adapted source and the original source, or otherwise it can be immensely difficult to prove the implication from the adapted postcondition to the final postcondition.

We should not forget to derive a range triple for *keyAlign*. For simplicity, let us derive:

$$\forall n. \quad \{\!\{\ \_\ vs \mid length\ vs = n\ \}\!\}\ \ keyAlign\ ks\ kv\ b\ c\ \ \{\!\{\ ss \mid length\ ss = n\ \}\!\}$$

assuming:

$$\{\!\{\ s\ v \mid ks\ s = kv\ v\ \}\!\}\ \ b\ \ \{\!\{\ \_\ \}\!\}$$

That is, if b is capable of producing everything, then $keyAlign\ \ ks\ \ kv\ \ b\ \ c$ can also produce everything. If $T' \subseteq \langle\ s'\ \_\ v \mid C\ s'\ v\ \rangle$ for some element-level consistency relation $C : \mathcal{P}(S \times V)$, then by Theorem 5.4 we know that for any source list, *get* (*keyAlign ks kv b c*) will successfully produce a view list of the same length, and each pair of source and view elements at the same position will satisfy C. For the derivation, again due to space restrictions we can only give a quick sketch: We only need to derive ranges for the two normal branches. The output range of the first branch can be derived as $\langle\ [] \mid 0 = n\ \rangle$ — that is, the branch can produce empty lists when $0 = n$; as for the output range of the second branch, we can derive $\langle\ (\_ :: ss) \mid 1 + length\ ss = n\ \rangle$. Their union is the output range of the entire **case**, and is indeed $\langle\ ss \mid length\ ss = n\ \rangle$.

8 DISCUSSION

How powerful is BiGUL compared with previous bidirectional languages? Regarding our version of BiGUL, most of its expressive power stems from the **case** construct, which has gone beyond Foster et al. (2007)'s "general conditional" in terms of expressiveness, allowing, in particular, key-based list alignment to be implemented directly. Note that the case analysis constructs in Ko et al. (2016)'s original BiGUL were essentially the same as Foster et al.'s conditionals, and Ko et al. had to provide list alignment as an extra and complex primitive.

While we are on the subject of alignment, the reader might be interested in a comparison with Barbosa et al. (2010)'s matching lenses. While some of the more sophisticated matching strategies offered by matching lenses can be hard to implement nicely in BiGUL so far, matching lenses are special-purpose and require the invention of a new framework in which everything is built from scratch. In contrast, BiGUL is designed with the ultimate aim of expressing all lenses using just a fixed set of simple primitives, like what we can do in general-purpose programming languages — hence the need for a general reasoning principle like the Hoare-style logic.

BiGUL claims to be "putback-based" but is still a lens language. Is there really a fundamental difference between BiGUL and previous "get-based" lens languages? When it comes to language definition, BiGUL programs denote lenses and have to be defined in both directions, exactly the same as other lens languages. It is when it comes to *using* the lenses that the distinction between the *get*- and *put*-based approaches becomes meaningful. The majority of lens languages are *get*-based, as explained by Foster (2009) below his Lemma 2.2.6:

> [l]ens programmers often feel like they are writing the forward transformation (because the names of primitives typically connote the forward transformation) and getting the backward transformation "for free"[.]

Matsuda and Wang (2015), for example, explicitly state that their language adopts this design. Foster et al. (2007)'s Figure 8 also clearly shows that their lens programs are supposed to be constructed like writing *get*, and Bohannon et al. (2006)'s relational lenses are written like database queries, which are *get* transformations. Inevitably they have to enrich their programs with putback information to gain more control, but that makes constructing and understanding such programs more awkward, having to break the abstraction and reason in both directions in terms of the underlying semantics. In contrast, BiGUL's putback-based design lets the programmer construct programs purely in the *put* direction. The Hoare-style logic makes the distinction between the two approaches absolutely clear for the first time — it is possible to precisely reason about bidirectional behaviour purely in the putback direction, whereas it is unthinkable that the same effect can be achieved from the opposite direction, which is perhaps the reason that no similar reasoning principles have been proposed for *get*-based languages.

Doesn't Theorem 5.4 say that we need to reason in both directions anyway? We have consistently explained how range triples can be derived by thinking in the putback direction. Even if the sceptical reader insisted that deriving range triples requires thinking in the forward direction, they would still have to admit that it is much easier to prove that *get* is contained in the precondition for *put* (as required by Theorem 5.4) than to prove that it is contained in the consistency relation, which is usually much smaller than the precondition for *put*. The indisputable fact is that the major work is done in derivations of putback triples, making the reasoning putback-based.

Where is lens composition? In terms of consistency, the behaviour of lens composition is just relational composition; on the other hand, the retentive behaviour of lens composition is rather chaotic and hard to reason about. We can formulate a rule like:

$$\frac{\begin{array}{l}\{\, a\ b' \mid \exists b, c.\ R\ a\ c \land R'\ a\ b \land U'\ b'\ b\ c \,\}\ \ l\ \ \{\, \langle\, a'\ \_\ b \mid R'\ a'\ b \,\rangle \cap T' \,\} \\ \{\!\{\, a\ b' \mid \exists b, c.\ R\ a\ c \land R'\ a\ b \land U'\ b'\ b\ c \,\}\!\}\ \ l\ \ \{\!\{\, P' \,\}\!\} \\ \{\, b\ c \mid \exists a.\ R'\ a\ b \land R\ a\ c \,\}\ \ r\ \ \{\, U' \,\}\end{array}}{\{\, R \cap \langle\, a\ \_ \mid P'\ a \,\rangle \,\}\ \ l \circ r\ \ \{\, a'\ a\ c \mid \exists b, b'.\ T'\ a'\ a\ b' \land U'\ b'\ b\ c \,\}}$$

and we have actually proved that the rule is sound, but the form of the rule is too complex to be easily usable. We will need to find a sweet spot and design a composition rule that is perhaps not as general as the above one but can still say enough about the retentive behaviour; most importantly, this rule should give guidance on how composition can be effectively used, and only with such a rule can composition be justified as a worthy construct. It should be noted that while composition is included in other languages like Foster et al. (2007)'s, the problem with controlling the retentive behaviour of composition has always existed, as discussed by, e.g., Diskin et al. (2011, Section 2.2).

9 CONCLUSION

Based on Theorem 2.3, it has been argued that "putback" is the essence of bidirectional programming (Fischer et al. 2015b). We would like to amend this statement: putback-based *reasoning* is the essence of bidirectional programming. This paper has shown, with the Hoare-style logic for BiGUL, what such reasoning can look like. More specifically, we have demonstrated how an adequate understanding of a BiGUL program's bidirectional behaviour can be achieved by reasoning exclusively about its putback behaviour.

Bidirectional programming has been highly declarative, in the sense that the programmer writes down only a consistency specification (perhaps with some behavioural details) and relies on the system to produce a well-behaved but otherwise arbitrary implementation. However, it has long been realised that declarative approaches are hardly enough for practical bidirectional applications (see, e.g., Stevens 2010, Section 4.1). The bidirectional transformations community currently concentrates on the exploration of more forms of well-behavedness laws (see, e.g., Cheney et al. 2015), but we should not be satisfied with only well-behavedness guarantees. Instead, we should also start aiming to precisely characterise the behaviour of bidirectional programs like what the BiGUL programmer can now do with the Hoare-style logic, treating bidirectional programming as seriously as we treat general-purpose programming.

More broadly, we believe that programming languages should be shipped with reasoning principles — even domain-specific languages deserve domain-specific reasoning principles! In the case of BiGUL, we have demonstrated how the precision of the Hoare-style logic can complement the informal description of BiGUL's semantics, and how the Hoare-style logic can be designed domain-specifically to work seamlessly for BIGUL programs. Moreover, the evolution of BIGUL is partly prompted by the development of the Hoare-style logic, whose eventual simplicity justifies BIGUL's current design. If, as Dijkstra (1974) argued, programs and their correctness proofs should grow hand in hand, then programming languages and their reasoning principles ought to be developed together as well. BIGUL and its Hoare-style logic make a nice example of this statement.

REFERENCES

Davi M. J. Barbosa, Julien Cretin, Nate Foster, Michael Greenberg, and Benjamin C. Pierce. 2010. Matching Lenses: Alignment and View Update. In *International Conference on Functional Programming (ICFP'10)*. ACM, 193–204. DOI: http://dx.doi.org/10.1145/1863543.1863572

Aaron Bohannon, Benjamin C. Pierce, and Jeffrey A. Vaughan. 2006. Relational Lenses: A Language for Updatable Views. In *Principles of Database Systems (PODS'06)*. ACM, 338–347. DOI: http://dx.doi.org/10.1145/1142351.1142399

Venanzio Capretta. 2005. General Recursion via Coinductive Types. *Logical Methods in Computer Science* 1, 2 (2005), 1–28. DOI: http://dx.doi.org/10.2168/LMCS-1(2:1)2005

James Cheney, Jeremy Gibbons, James McKinna, and Perdita Stevens. 2015. Towards a Principle of Least Surprise for Bidirectional Transformations. In *International Workshop on Bidirectional Transformations (BX'15)*. CEUR-WS, 66–80. http://ceur-ws.org/Vol-1396/p66-cheney.pdf

Krzysztof Czarnecki, J. Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James F. Terwilliger. 2009. Bidirectional Transformations: A Cross-Discipline Perspective. In *International Conference on Model Transformation (Lecture Notes in Computer Science)*, Vol. 5563. Springer, 260–283. DOI: http://dx.doi.org/10.1007/978-3-642-02408-5_19

Edsger W. Dijkstra. 1974. Programming as a Discipline of Mathematical Nature. *Amer. Math. Monthly* 81, 6 (1974), 608–612. DOI: http://dx.doi.org/10.2307/2319209

Zinovy Diskin, Yingfei Xiong, and Krzysztof Czarnecki. 2011. From State- to Delta-Based Bidirectional Model Transformations: the Asymmetric Case. *Journal of Object Technology* 10 (2011), 6:1–25. DOI: http://dx.doi.org/10.5381/jot.2011.10.1.a6

Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. 2015a. A Clear Picture of Lens Laws. In *Mathematics of Program Construction (Lecture Notes in Computer Science)*, Vol. 9129. Springer, 215–223. DOI: http://dx.doi.org/10.1007/978-3-319-19797-510

Sebastian Fischer, Zhenjiang Hu, and Hugo Pacheco. 2015b. The Essence of Bidirectional Programming. *SCIENCE CHINA Information Sciences* 58, 5 (2015), 1–21. DOI: http://dx.doi.org/10.1007/s11432-015-5316-8

John Nathan Foster. 2009. *Bidirectional Programming Languages.* Ph.D. Dissertation. University of Pennsylvania.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. 2007. Combinators for Bidirectional Tree Transformations: A Linguistic Approach to the View-Update Problem. *ACM Transactions on Programming Languages and Systems* 29, 3 (2007), 17. DOI: http://dx.doi.org/10.1145/1232420.1232424

Jeremy Gibbons. 2007. Datatype-Generic Programming. In *Spring School on Datatype-Generic Programming (Lecture Notes in Computer Science)*, Vol. 4719. Springer, 1–71. DOI: http://dx.doi.org/10.1007/978-3-540-76786-2_1

C. A. R. Hoare. 1969. An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (1969), 576–580. DOI: http://dx.doi.org/10.1145/363235.363259

Zhenjiang Hu and Hsiang-Shang Ko. 2017. Principle and Practice of Bidirectional Programming in BiGUL. Draft lecture notes for the *Oxford Summer School on Bidirectional Transformations.* (2017). https://bitbucket.org/prl_tokyo/bigul/raw/master/SummerSchool16/paper/BiGUL_tutorial.pdf

Zhenjiang Hu, Hugo Pacheco, and Sebastian Fischer. 2014. Validity Checking of Putback Transformations in Bidirectional Programming. In *International Symposium on Formal Methods (Lecture Notes in Computer Science)*, Vol. 8442. Springer, 1–15. DOI: http://dx.doi.org/10.1007/978-3-319-06410-9_1

Hsiang-Shang Ko, Tao Zan, and Zhenjiang Hu. 2016. BiGUL: A Formally Verified Core Language for Putback-Based Bidirectional Programming. In *Partial Evaluation and Program Manipulation (PEPM'16)*. ACM, 61–72. DOI: http://dx.doi.org/10.1145/2847538.2847544

Nuno Macedo, Hugo Pacheco, Alcino Cunha, and José N. Oliveira. 2013. Composing Least-Change Lenses. In *International Workshop on Bidirectional Transformations (Electronic Communications of the EASST)*. EASST. DOI: http://dx.doi.org/10.14279/tuj.eceasst.57.868

Per Martin-Löf. 1984. *Intuitionistic Type Theory.* Bibliopolis, Napoli.

Kazutaka Matsuda and Meng Wang. 2015. Applicative Bidirectional Programming with Lenses. In *International Conference on Functional Programming (ICFP'15)*. ACM, 62–74. DOI: http://dx.doi.org/10.1145/2858949.2784750

Jorge Mendes, Hsiang-Shang Ko, and Zhenjiang Hu. 2016. *The Under-Appreciated Put: Implementing Delta-Alignment in BiGUL.* Technical Report GRACE-TR 2016-03. GRACE Center, National Institute of Informatics. http://grace-center.jp/wp-content/uploads/2016/04/GRACE-TR-2016-03.pdf

Hugo Pacheco, Zhenjiang Hu, and Sebastian Fischer. 2014a. Monadic Combinators for "Putback" Style Bidirectional Programming. In *Partial Evaluation and Program Manipulation (PEPM'14)*. ACM, 39–50. DOI: http://dx.doi.org/10.1145/2543728.2543737

Hugo Pacheco, Tao Zan, and Zhenjiang Hu. 2014b. BiFluX: A Bidirectional Functional Update Language for XML. In *Principles and Practice of Declarative Programming (PPDP'14)*. ACM, 147–158. DOI: http://dx.doi.org/10.1145/2643135.2643141

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science (LICS'02)*. IEEE, 55–74. DOI: http://dx.doi.org/10.1109/LICS.2002.1029817

Perdita Stevens. 2010. Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. *Software and Systems Modeling* 9 (2010), 7. DOI: http://dx.doi.org/10.1007/s10270-008-0109-9

Tetsuo Yokoyama and Robert Glück. 2007. A Reversible Programming Language and its Invertible Self-Interpreter. In *Partial Evaluation and Program Manipulation (PEPM'07)*. ACM, 144–153. DOI: http://dx.doi.org/10.1145/1244381.1244404

Tao Zan, Li Liu, Hsiang-Shang Ko, and Zhenjiang Hu. 2016. Brul: A Putback-Based Bidirectional Transformation Library for Updatable Views. In *International Workshop on Bidirectional Transformations (BX'16)*. CEUR-WS, 77–89. http://ceur-ws.org/Vol-1571/paper_3.pdf