BiGUL: A Formally Verified Core Language for Putback-Based Bidirectional Programming

Hsiang-Shang Ko
National Institute of Informatics, Japan
hsiang-shang@nii.ac.jp

Tao Zan Zhenjiang Hu
SOKENDAI (The Graduate University for Advanced Studies), Japan
National Institute of Informatics, Japan
{zantao,hu}@nii.ac.jp

Abstract

Putback-based bidirectional programming allows the programmer to write only one putback transformation, from which the unique corresponding forward transformation is derived for free. The logic of a putback transformation is more sophisticated than that of a forward transformation and does not always give rise to well-behaved bidirectional programs; this calls for more robust language design to support development of well-behaved putback transformations. In this paper, we design and implement a concise core language BiGUL for putback-based bidirectional programming to serve as a foundation for higher-level putback-based languages. BiGUL is completely formally verified in the dependently typed programming language AGDA to guarantee that any putback transformation written in BiGUL is well-behaved.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Specialized Application Languages

Keywords Putback-based bidirectional transformations, dependent types, formal verification

1. Introduction

Bidirectional transformations (BXs) [5] provide a novel mechanism for maintaining consistency between two pieces of related information, one referred to as the source and the other as the view. A bidirectional transformation consists of a pair of transformations — a forward *get* which extracts information from a source to construct an abstract view, and a backward *put* which embeds information of a view back into a source, producing an updated source — and this pair of transformations should be *well-behaved*, i.e., they should satisfy two round-tripping laws (*PutGet* and *GetPut*; see Section 2). Since writing both transformations while guaranteeing their well-behavedness can take a lot of effort, many bidirectional programming languages [2, 3, 8, 9, 13–15, 18, 23] have been designed to aid the user in writing bidirectional transformations, with which the programmer only needs to write one program that can be interpreted either as *get* or *put*, and the two interpretations are guaranteed to be well-behaved.

Recently there have been investigations into the feasibility of writing bidirectional programs by describing solely their putback directions [16, 24, 25]. The rationale behind this *putback-based* approach is that the *put* component of a bidirectional transformation uniquely determines its *get* component [7, Lemma 2.2.5]. The combinator library PUTLENSES [24] follows this approach, providing a set of putback-based *lens* combinators [8] for writing complex BX programs. Based on PUTLENSES and the functional XML update language FLUX [4], a bidirectional XML update language BIFLUX [25] has also been designed and implemented. Putback-based programming is more delicate than previous approaches which centre around writing *get*, not only because *put* — being an update — is inherently more complex than *get*, but also because not all updates give rise to well-behaved BXs. This introduces a new challenge of ensuring the well-behavedness of a *put*, in the sense that a unique *get* can indeed be derived from the *put* to form a well-behaved BX. PUTLENSES and BIFLUX meet this challenge by inserting dynamic (runtime) checks around programmer-specified actions used in *put*, and the programmer is supposed to supply only actions that can pass these dynamic checks. While both languages are carefully designed, the intricate nature of these dynamic checks can still make people cast doubt on their correctness.

In this paper, we aim to build a clean and solid foundation for higher-level putback-based languages (e.g., BIFLUX) by focusing on a small yet sufficiently powerful putback-based core language BiGUL (for *Bidirectional Generic Update Language*), which is completely formally verified to guarantee that any *put* specified in the language is well-behaved. In more detail:

- Firstly, BiGUL is a clean revision of the core of BIFLUX, and is designed to be closer to practical programming languages (than, e.g., PUTLENSES). It consists of a set of essential and orthogonal bidirectional statements, such as alignment of source and view lists, pattern matching–based independent source updates and invertible view computation, and case analyses on either source or view. Although currently BiGUL only has a dependently typed abstract syntax, which is not intended to be programmer-friendly, BiGUL programs are pretty straightforward to write (or compile to), thanks to its native support of familiar programming constructs like pattern matching and case statements.
- Secondly, the semantics of BiGUL is concretely defined in terms of monadic programs, and all dynamic checks for guaranteeing well-behavedness appear explicitly in the programs. BiGUL is thus differentiated from the more abstract treatment by, e.g., Foster et al. [8], in which partiality is dealt with implicitly and various well-behavedness conditions are specified set-theoretically, creat-

PEPM'16, January 18–19, 2016, St. Petersburg, FL, USA
ACM. 978-1-4503-4097-7/16/01...
http://dx.doi.org/10.1145/2847538.2847544

ing a gap between the definitions and their implementations. The concrete semantics of BiGUL, in contrast, can be faithfully ported to practical programming languages like HASKELL without having to fill the gap between abstract mathematics and concrete implementation.

- Most importantly, BiGUL and its well-behavedness have been completely formally modelled and verified using the dependently typed programming language AGDA [20, 21]. (Full source code is available at `http://www.prg.nii.ac.jp/project/bigul`, and has been checked by AGDA version 2.4.2.4 with standard library version 0.11.) Since BiGUL's semantics are concrete programs and directly proved correct, we can highly confidently guarantee that these programs, when ported to other languages, are indeed well-behaved.
- The main technique for facilitating the construction of well-behavedness proofs is worth mentioning: We reify computation steps of monadic programs as easily manipulable data, so well-behavedness proofs — which have to cover all possible execution traces by analysing program structure — can be carried out just like doing ordinary functional programming.

For the rest of the paper, after explaining how to formalise and prove well-behavedness of BXs in AGDA in Section 2, we present BiGUL's dependently typed syntax and proof-carrying bidirectional semantics in Section 3 and a showcase example in Section 4, and conclude with a few remarks in Section 5. This paper is typeset with a modified version of lhs2TEX such that all global AGDA identifiers are hyperlinked to their definitions, as an aid to those who read this paper electronically.

2. Formalisation of Well-Behaved BXs in AGDA

Before presenting the BiGUL language in Section 3, we first give an account of our AGDA formalisation of the underlying notion of well-behaved partial bidirectional transformations, and also demonstrate by a simple example how well-behavedness proofs are constructed in our setting (Section 2.2). The key to the formalisation is the reification of computation steps of monadic programs as a type family $\_\mapsto\_$ (Section 2.1), which makes the construction of the proofs as easy as ordinary functional programming.

We follow Foster et al.'s classic *lens* approach [8], but for the *PutGet* and *GetPut* laws we adopt the variant used by Pacheco et al. [24]. A lens between a *source* type S and a *view* type V is defined in AGDA as the following record type:[1]

```
record Lens (S V : Set) : Set₁ where
  field
    put : S → V → Par S
    get : S → Par V
    PutGet : {s s′ : S} {v : V} → (put s v ↦ s′) → (get s′ ↦ v)
    GetPut : {s : S} {v : V} →    (get s ↦ v) → (put s v ↦ s)
```

That is, a lens is a pair of functions *put* and *get* satisfying the *PutGet* and *GetPut* laws. The two laws are referred to as the *well-behavedness* properties. Precise definitions of Par and $\_\mapsto\_$ will be presented later in Section 2.1. Here we offer an informal explanation first: Both *put* and *get* are partial functions that may or may not successfully produce a result. Execution of *put s v*, if successful, produces an updated version of the source *s* by embedding all information of the view *v* into *s*, while *get s* extracts the view embedded in *s*. The *PutGet* law states that, for all s, $s' : S$ and $v : V$, if *put s v* successfully computes to s', then subsequently *get* s' should successfully compute to *v*; this ensures that *put* does the embedding properly — after *put* updates a source with a view, *get* can completely recover the view from the updated source. Conversely, the *GetPut* law states that, for all $s : S$ and $v : V$, if *get s* successfully computes to *v*, then subsequently *put s v* should successfully compute to *s*; this ensures that *put* does not perform excessive updates — if the view used to update the source is directly extracted from the source by *get*, then the source will remain unchanged after the update.

Note that our definition of lenses is stronger than the definition of well-behaved lenses given by Foster et al. [8, Definition 3.2]: Our *PutGet* law says that successful computation of *put* guarantees success of the subsequent computation of *get*, whereas in Foster et al.'s definition there is no such guarantee; the *GetPut* law is similar.

2.1 Reification of Monadic Combinators and Computation Steps

The kind of partial transformations we need are actually total functions which wrap their results in the standard Maybe datatype with two constructors just : $A \to$ Maybe A and nothing : Maybe A — the result of a successful computation is wrapped in the just constructor, while failed computation is represented by nothing. It is thus always possible to know whether a computation succeeds or not in a finite amount of time. This notion of "decidable partiality" should be distinguished from the usual partiality as formalised in terms of continuous functions between complete partial orders: When defining the semantics of BiGUL (more specifically, for view case analysis in Section 3.5), we need to write programs that produce some result or fail respectively when a sub-program fails or produces some result, but this is not possible if failed computation is represented as a least element of a complete partial order.

We can choose to write our transformations directly as Maybe-programs, usually structured with monadic combinators [19, 27] like "return" and "bind". This approach does not work too satisfactorily, though (see the second half of Section 2.2 for an example), when we consider the kind of proofs we need to construct for these programs: When proving *PutGet* and *GetPut*, given is a proof that a program — consisting of sub-programs bound together by the combinators — computes successfully, and we need to analyse it to obtain proofs saying that each of the sub-programs computes successfully, and then reassemble these proofs to show that a related composite program also computes successfully. This task would be much easier if a proof of a successful composite computation is exactly a bunch of proofs that all its sub-computations succeed. To achieve this, we can reify the monadic binding structure of Maybe-programs by deeply embedding the monadic combinators as the constructors of a datatype, so that we can define types of proofs of successful computation by induction on the binding structure.

We thus define a datatype Par, which is a deep embedding of the operators that we use to write partial transformations:[2]

```
data Par : Set → Set₁ where
  return : {A : Set} →   A → Par A
  _>>=_ : {A B : Set} → Par A → (A → Par B) → Par B
  fail   : {A : Set} →   Par A
```

There are also other deeply embedded operators for error handling (see Section 3.5) and assertion, which we omit here. The meaning of the constructors of Par is formally given by the following interpreter to Maybe:[3]

[1] Arguments wrapped in curly braces are *implicit*, and are typeset in a less obtrusive style in this paper. We do not need to supply the implicit arguments when applying a function if AGDA can infer what those arguments should be.

[2] Identifiers with underscores can be used either in prefix or infix form. For example, we can write either $\_\gg\!=\_$ *mx f* or *mx* $\gg\!=$ *f*.

[3] The **with** notation evaluates an expression (here *runPar mx*) and matches the result with an additional pattern (appearing on the right of '|').

```
runPar : {A : Set} → Par A → Maybe A
runPar (return x) = just x
runPar (mx >>= f) with runPar mx
runPar (mx >>= f) | just x    = runPar (f x)
runPar (mx >>= f) | nothing = nothing
runPar fail = nothing
```

We say that *mx* : Par *A* computes to *x* : *A* exactly when *runPar mx* ≡ just *x*, and that *mx* fails to compute exactly when *runPar mx* ≡ nothing. The equality *runPar mx* ≡ just *x* is already a family of types of proofs of successful computation, but these equality proofs are not so straightforward to analyse and construct (see the second half of Section 2.2). We hence move on to define an equivalent family of types, whose proofs are easier to manipulate.

For every *mx* : Par *A* and *x* : *A*, we define a type *mx* ↦ *x* whose inhabitants explain how the sub-programs in *mx* compute successfully one by one, eventually producing *x*:[4]

```
_↦_ : {A : Set} → Par A → A → Set
(return x)   ↦ x′ = x ≡ x′
(mx >>= f) ↦ y  = (x : _) × (mx ↦ x) × (f x ↦ y)
fail           ↦ x  = ⊥
```

In prose: return *x* computes to *x′* when *x* is exactly *x′*; *mx* >>= *f* computes to *y* when there exists a value *x* such that *mx* computes to *x* and *f x* computes to *y*; and it is impossible for fail to compute successfully. Here our use of the term "computes to" is justified, since we can prove that a term of type *mx* ↦ *x* can be constructed if and only if *runPar mx* ≡ just *x*, meaning that our reification of *runPar mx* ≡ just *x* as *mx* ↦ *x* is accurate. We are thus entitled to use _↦_ in the statements of *PutGet* and *GetPut*.

2.2 A Sample Well-Behavedness Proof

To give the reader a taste of how well-behavedness proofs are constructed in our AGDA setting, let us try to define an operator _⋄_ for lens composition:[5]

```
_⋄_ : {A B C : Set} → Lens A B → Lens B C → Lens A C
l ⋄ r = record
  { put = λ a c → Lens.get l a >>= λ b →
                         Lens.put r b c >>= Lens.put l a
  ; get = λ a → Lens.get l a >>= Lens.get r
  ; PutGet = ? ; GetPut = ?}
```

The *put* and *get* functions for the composite lens *l* ⋄ *r* can be pretty straightforwardly constructed from the *put* and *get* functions of the component lenses *l* and *r*. As part of the definition of _⋄_, we also need to prove that *PutGet* and *GetPut* hold for the pair of *put* and *get* we provide. For *PutGet* we need to show that, for all *a*, *a′* : *A*, and *c* : *C*, from a proof of *put a c* ↦ *a′* we can construct a proof of *get a′* ↦ *c*. That is, we need to write a function that converts an inhabitant of the type *put a c* ↦ *a′* to one of the type *get a′* ↦ *c*. AGDA automatically expands the two types by the definition of _↦_, the first one to

```
(b : B) × (Lens.get l a ↦ b) ×
   (b′ : B) × (Lens.put r b c ↦ b′) × (Lens.put l a b′ ↦ a′)
```

and the second one to

```
(b : B) × (Lens.get l a′ ↦ b) × (Lens.get r b ↦ c)
```

[4] We write dependent pair types $\Sigma(x \in A)\, B\, x$ as $(x : A) \times B\, x$. The underscore in $(x : \_) \times (mx \mapsto x) \times (f\, x \mapsto y)$ leaves the type of x for AGDA to infer.

[5] A field of a record is extracted by applying the field's accessor function to the record. For example, the accessor function Lens.*get* has type {*S V* : Set} → Lens *S V* → *S* → Par *V*.

Thus all we need to do is convert a five-tuple to a three-tuple. This is just ordinary functional programming: Applying Lens.*PutGet l* to the fifth component of type Lens.*put l a b′* ↦ *a′* we obtain an inhabitant of type Lens.*get l a′* ↦ *b′*, and applying Lens.*PutGet r* to the fourth component of type Lens.*put r b c* ↦ *b′* we obtain an inhabitant of type Lens.*get r b′* ↦ *c*. Hence the proof term for *PutGet* is simply

```
λ { (b , x , b′ , y , z) → (b′ , Lens.PutGet l z , Lens.PutGet r y)}  (∗)
```

With a similar reasoning, we can also construct a proof term for *GetPut*:

```
λ { (b , x , y) → (b , x , b , Lens.GetPut r y , Lens.GetPut l x)}
```

completing the definition of _⋄_.

For the rest of this paper we will omit the well-behavedness proofs, but a majority of these proofs — while requiring more complicated case analyses — are technically as simple as the above proofs for lens composition.

Working directly with equality proofs. To emphasise the advantage of reasoning with Par and _↦_, let us consider how the well-behavedness proofs for _⋄_ would be constructed if we wrote partial transformations simply as Maybe-programs and stated well-behavedness in terms of AGDA's equality type _≡_. The *put* and *get* functions would be defined in exactly the same way except that the bind operator used is the Maybe version. The *PutGet* law now reads

```
pg : {a a′ : A} {c : C} →
       (Lens.get l a >>= (λ b →
          Lens.put r b c >>= Lens.put l a) ≡ just a′) →
       Lens.get l a′ >>= Lens.get r ≡ just c
```

The logic of proving *pg* is exactly the same as before, but the machinery involved is more complex. To analyse the antecedent equality proof, we can prove a lemma:

```
bind-computes :
   {A B : Set} (mx : Maybe A) {f : A → Maybe B} {y : B} →
   mx >>= f ≡ just y →
   (x : A) × (mx ≡ just x) × (f x ≡ just y)
```

Applying the lemma twice analyses the antecedent into three smaller equality proofs, achieving the same effect as the pattern matching in the λ-expression (∗). As for establishing the consequent, a probably easier way is to rewrite Lens.*get l a′* to just *b′* with an application of Lens.*PutGet l* so the consequent type simplifies to Lens.*get r b′* ≡ just *c*, which can then be discharged by an application of Lens.*PutGet r*. The whole proof is

```
pg {a = a} p with bind-computes (Lens.get l a) p
pg {c = c} p | b , x , q with bind-computes (Lens.put r b c) q
pg             p | b , x , q | b′ , y , z rewrite Lens.PutGet l z
                                                                    = Lens.PutGet r y
```

which is apparently more heavyweight than the λ-expression (∗). For more complicated well-behavedness proofs, the above approach quickly becomes unmanageable, whereas our approach scales nicely.

3. BiGUL and Its Formally Verified Lens Semantics

Having explained the infrastructure of our AGDA formalisation, we now move on to the BiGUL language itself. Following the putback-based approach, our language BiGUL describes *put* functions, i.e., updates to a source using a view. Figure 1 shows (a simplified version of) the main definition of BiGUL (which refers to some auxiliary definitions that will appear in later sub-sections). In

```
data BiGUL : U → U → Set_1 where
  replace : {S : U} →    BiGUL S S
  fail    : {S V : U} → BiGUL S V
  skip    : {S : U} →    BiGUL S one
  caseS   : {S V : U} → List (CaseSBranch^B S V) → BiGUL S V
  caseV   : {S V : U} → List (CaseVBranch^B S V) → BiGUL S V
  align   : {S V : U} → (source-condition : ⟦ S ⟧ → Par Bool)
                        (match : ⟦ S ⟧ → ⟦ V ⟧ → Par Bool)
                        (b : BiGUL S V)
                        (create : ⟦ V ⟧ → Par ⟦ S ⟧)
                        (conceal : ⟦ S ⟧ → Par (Maybe ⟦ S ⟧)) →
                        BiGUL (list S) (list V)
  update : {S : U} →     (p : Pattern S) (bs : BiGULs p) →
                        BiGUL S (Views p bs)
  rearr   : {S V V' : U} → (p : Pattern V) (q : Pattern V') →
                          Paths p q → BiGUL S V' → BiGUL S V
```

Figure 1. Top-level definition of BiGUL (simplified)

general, a BiGUL update proceeds by decomposing the source into several parts to be updated independently (update; Section 3.2), and accordingly rearranging the view to match with these parts (rearr; Section 3.3), until the source and the view are reduced to the extent that basic operations can be applied (replace, fail, and skip; Section 3.1). When both the source and the view are lists, there is a powerful alignment operation for synchronising the two lists (align; Section 3.6). And we can do case analyses on either the source (caseS; Section 3.4) or the view (caseV; Section 3.5) for describing more complex update strategies.

Figure 2 gives a running example of a BiGUL program. (A more complex example, in particular involving caseS and caseV, is given in Section 4.) The source is a list of book data of type $\mathsf{String} \times \mathsf{String} \times \mathbb{N} \times \mathbb{N} \times \mathsf{Bool}$ representing a book's title, author, publication year, price, and whether it is in stock, and we intend to update the price of those books published in 2016 that are still in stock. We thus prepare a list of pairs of book title and price of type $\mathsf{String} \times \mathbb{N}$ as the view. The BiGUL program then describes how to update the source with the view: We focus on those books in the source list which are published in 2016 and still in stock (line 1) and align them with the view list by title (line 2); for every matched pair of source and view books, we rearrange the view (lines 3–4) to match with the shape of the source, decompose the source by pattern matching (line 5), and replace the title and price of the source with those from the view (line 6); for every unmatched view book, we create a source book with default author information and publication year 2016 and mark it as in stock (line 7), and its title and price will later be updated by the action for matched pairs (i.e., lines 3–6); finally, for every unmatched source book, we keep it in the source list but mark it as out of stock (line 8).

The types of the BiGUL constructors are indexed by the source and view types of the operations represented by the constructors, and ultimately we can write a well-typed interpreter to Lens:

```
interp : {S V : U} → BiGUL S V → Lens ⟦ S ⟧ ⟦ V ⟧
```

(U and ⟦_⟧ are defined in Section 3.2.) Since well-behavedness is built into the definition of Lens, being able to construct *interp* means that we have not only translated BiGUL to *put* and *get* functions, but also proved formally that *PutGet* and *GetPut* are satisfied, in the manner explained in Section 2.2.

Below we will go through each of the BiGUL operations, starting with the three most basic ones in Section 3.1. We should warn the reader that the subsequent Section 3.2 is more about the datatype-generic mechanism underlying a major part of BiGUL and less about BXs, and we choose to present it earlier because it will be used in later sections, and also help to clarify most of Figure 2 sooner. Section 3.4 will be the first section that presents the development of a nontrivial bidirectional operation in detail.

3.1 Replacing, Failure, and Skipping

The three most basic operations are replace, fail, and skip. The replace operation is applicable when the source and view are of the same type. Its *put* semantics is discarding the original source and returning the view as the updated source, while its *get* semantics is just the identity. This bidirectional semantics can in fact be derived from a *partial isomorphism*: We define partial isomorphisms by

```
record Iso (A B : Set) : Set_1 where
  field
    to   : A → Par B
    from : B → Par A
    to-from-inverse : {x : A} {y : B} → (to x ↦ y) → (from y ↦ x)
    from-to-inverse : {x : A} {y : B} → (from y ↦ x) → (to x ↦ y)
```

of which Lens is a generalisation, as we can easily convert Iso *A B* to Lens *A B*:

```
iso-lens : {A B : Set} → Iso A B → Lens A B
iso-lens iso = record { put    = λ _ b → Iso.from iso b
                      ; get    = Iso.to iso
                      ; PutGet = Iso.from-to-inverse iso
                      ; GetPut = Iso.to-from-inverse iso}
```

(Note that the *put* ignores its source argument.) The lens semantics of replace can then be derived from the identity isomorphism:

```
id-iso : {S : Set} → Iso S S
id-iso = record {to = return ; from = return ; ... }
interp replace = iso-lens id-iso
```

We can also derive the lens semantics for fail — whose *put* and *get* simply fail to compute for any input — from the empty isomorphism in the same way:

```
empty-iso : {S V : Set} → Iso S V
empty-iso = record { to   = λ _ → fail
                   ; from = λ _ → fail ; ... }
interp fail = iso-lens empty-iso
```

On the other hand, the *put* semantics of the skip operation is discarding the view and returning the original source, and its lens semantics is not an instance of *iso-lens*:

```
skip-lens : {S : Set} → Lens S ⊤
skip-lens = record { put = λ s _ → return s
                   ; get = λ _ → return tt ; ... }
interp skip = skip-lens
```

Note that the view type is specified as the unit type ⊤ (whose only inhabitant is tt : ⊤) — if the view type were more complex, there would be no way for *get* to decide what to return. In general, *put* must embed all view information into the updated source so *get* can recover the entire view from the updated source, establishing *PutGet*. ⊤ is a type with no information, and hence Lens.*put skip-lens* can safely ignore its view argument of type ⊤, while Lens.*get skip-lens* can simply return tt.

3.2 Source Update

A fundamental operation is decomposing a source and updating its parts, represented by the update constructor of BiGUL. We follow

```
1 align (λ { (_ , _ , year , _ , instock) → return (year == 2016 ∧ instock) })
2       (λ { (stitle , _) (vtitle , _) → return (stitle == vtitle) })
3       (rearr (prod var var) (prod var (prod unit (prod unit (prod var       unit))))
4                             (        inj₁ refl ,  tt ,        tt ,         inj₂ refl , tt)
                  -- the two lines above are a deeply embedded representation of the function λ { (title , price) → (title , tt , tt , price , tt) }
5            (update (prod var (prod     var (prod  var (prod  var            var))))
6                    ((_ ,  replace) , (_ , skip) , (_ , skip) , (_ , replace) , (_ , skip))))
7       (λ _ → return ("" , "(to be updated)" , 2016 , 0 , true))
8       (λ { (title , author , year , price , instock) → return (just (title , author , year , price , false)) })
```

Figure 2. Updating the prices of the books published in 2016

the approach taken by FLUX [4], which requires that updates must be independent — that is, the same part of a source cannot be updated more than once. (This is for ensuring *PutGet*: Consider, for example, the scenario in which the view is a pair $(v\ ,\ v')$, whose components have the same type as the source; if we allowed the source to be replaced twice, first with v and second with v', then v would be overwritten by the second replacing and could not be retrieved by a subsequent invocation of *get*.) It turns out that a pattern matching notation suits this task perfectly: Pattern matching is arguably the most intuitive way to decompose a source, and we can specify manifestly independent updates by writing them at the variable positions in a pattern. For example, at lines 5–6 of Figure 2 we use update to match a source book of type String × String × ℕ × ℕ × Bool with a five-variable pattern and update its five components respectively by replace, skip, skip, replace, and skip. The idea itself is straightforward; what follows, despite its slight complexity (especially if the reader is not familiar with dependently typed programming), is merely our datatype-generic and strongly typed implementation of the idea in AGDA [1, 10]. The definitions for update will be laid out in three levels: We first define the class of datatypes that we wish to process with BiGUL; for every datatype, we define the patterns applicable to the inhabitants of the datatype; finally, every pattern induces a "container" type, whose inhabitants can be used to store inner BiGUL statements at the variable positions of the pattern.

Since AGDA is fully dependently typed, we can naturally define patterns in a type-directed way such that nonsensical patterns are ruled out by construction. This task starts from the construction of a *universe* U, i.e., a datatype whose inhabitants are interpreted as types. U has appeared in the definition of BiGUL in Figure 1, where it is used as the type of the indices representing the source and view types of the operations. The actual AGDA implementation of BiGUL uses a universe capable of expressing mutually inductive datatypes, but, to make the presentation easier to follow, the universe U that we define below — along with its interpretation ⟦_⟧ — is only a much simplified version:

```
data U : Set₁ where          ⟦_⟧ : U → Set
  k    : Set → U             ⟦ k A    ⟧ = A
  one  : U                   ⟦ one    ⟧ = ⊤
  _⊕_ : U → U → U           ⟦ F ⊕ G ⟧ = ⟦ F ⟧ ⊎ ⟦ G ⟧
  _⊗_ : U → U → U           ⟦ F ⊗ G ⟧ = ⟦ F ⟧ × ⟦ G ⟧
  list : U → U               ⟦ list F ⟧ = List ⟦ F ⟧
```

The types that we can encode within U are those expressible in terms of atomic types, the unit type ⊤, sum types _⊎_ (whose two constructors are $\mathsf{inj}_1 : A \to A \uplus B$ and $\mathsf{inj}_2 : B \to A \uplus B$), product types _×_, and List; applying ⟦_⟧ to an inhabitant of the universe decodes it to the type it represents.

We can now define a family of types Pattern : U → Set such that, for any D : U, the type Pattern D contains exactly those patterns sensible for ⟦ D ⟧. Below is a simplified version covering the patterns used in this paper:

```
data Pattern : U → Set₁ where
  var   : {D : U} → Pattern D
  unit  : Pattern one
  left  : {D E : U} → Pattern D → Pattern (D ⊕ E)
  right : {D E : U} → Pattern E → Pattern (D ⊕ E)
  prod  : {D E : U} → Pattern D → Pattern E → Pattern (D ⊗ E)
```

On all types we can use the var pattern which matches anything, while on sum types we can use the left and right patterns matching inj_1 and inj_2 constructors but not the prod pattern, which only makes sense for product types. Also we have the unit pattern for matching tt : ⊤.

Next, patterns are given a different interpretation as "containers" for storing values — whose types depend on the types of the variable sub-patterns — at their variable positions. For example, a pair pattern prod var var : Pattern $(D \otimes E)$ can be interpreted as a container storing a pair of type $f\ D \times f\ E$ for any given type-computing function f : U → Set. The types of such containers can be defined by induction on Pattern:

```
VarPositions :
    {l : Level} {D : U} → Pattern D → (U → Set l) → Set l
VarPositions (var {D}) f = f D
VarPositions unit      f = ⊤
VarPositions (left  p) f = VarPositions p f
VarPositions (right p) f = VarPositions p f
VarPositions (prod p q) f = VarPositions p f × VarPositions q f
```

A first situation where the notion of pattern-induced containers is helpful is when formalising pattern matching: The result of a successful pattern matching can be filled into such a container, by putting the resulting components at their respective variable positions. Defining the types of pattern matching results as an instance of VarPositions:

```
PatResult : {D : U} → Pattern D → Set
PatResult p = VarPositions p ⟦_⟧
```

we can formulate pattern matching as a partial isomorphism:

```
pat-iso : {D : U} → (p : Pattern D) → Iso ⟦ D ⟧ (PatResult p)
```

whose definition is shown in Figure 3. The *to* component of the isomorphism performs pattern matching, and the *from* component reverses pattern matching by evaluating a pattern as if it is an expression, using the input of type PatResult p as an environment for values at variable positions. (This pattern matching isomorphism will also play an important role in Section 3.3.)

Now we can explain the syntax of the source updating operation, i.e., the update constructor of the BiGUL datatype, and define how it is interpreted, i.e., the update case of *interp*. By specialising

```
pat-iso : {D : U} → (p : Pattern D) → Iso ⟦ D ⟧ (PatResult p)
pat-iso p = record { to    = deconstruct p
                    ; from  = return ∘ construct p ; ... }
  where
    deconstruct :
       {D : U} (p : Pattern D) → ⟦ D ⟧ → Par (PatResult p)
    deconstruct var        x       = return x
    deconstruct unit       _       = return tt
    deconstruct (left  p)  (inj₁ x) = deconstruct p x
    deconstruct (left  p)  (inj₂ y) = fail
    deconstruct (right p)  (inj₁ x) = fail
    deconstruct (right p)  (inj₂ y) = deconstruct p y
    deconstruct (prod p q) (x , y)  = deconstruct p x >>= λ l →
                                      deconstruct q y >>= λ r →
                                      return (l , r)
    construct : {D : U} (p : Pattern D) → PatResult p → ⟦ D ⟧
    construct var        x       = x
    construct unit       _       = tt
    construct (left  p)  x       = inj₁ (construct p x)
    construct (right p)  y       = inj₂ (construct p y)
    construct (prod p q) (x , y) = construct p x , construct q y
    ...
```

Figure 3. Definition of the pattern matching isomorphism

VarPositions, we can place BiGUL statements — along with their view types — at the variable positions of a pattern:

```
BiGULs : {D : U} → Pattern D → Set₁
BiGULs p = VarPositions p (λ D → (E : U) × BiGUL D E)
```

The update constructor of BiGUL then takes a pattern p and a bunch of BiGUL statements collected in *bs* : BiGULs p. Its view type is a product of the view types in *bs*, whose code in U is computed by

```
Views : {D : U} (p : Pattern D) → BiGULs p → U
Views var        (E , _)   = E
Views unit       _         = one
Views (left  p)  bs        = Views p bs
Views (right p)  bs        = Views p bs
Views (prod p q) (bs , bs′) = Views p bs ⊗ Views q bs′
```

The lens semantics of update is given by

```
interp (update pat bs) =
  iso-lens (pat-iso pat) ⋄ interp-update pat bs
```

where *interp-update* has type

```
interp-update : {D : U} (p : Pattern D) (bs : BiGULs p) →
                Lens (PatResult p) ⟦ Views p bs ⟧
```

which inductively interprets and composes the BiGUL statements in *bs* into one lens. In prose, the *put* semantics of update decomposes the source by matching it with the pattern p, updates its components at the variable positions using the BiGUL statements in *bs*, and reassembles the updated components, while its *get* semantics again decomposes the source by matching it with p and extracts views from its components using *bs*.

3.3 View Rearrangement

The operation rearr is for rearranging the view to a specific shape suitable for subsequent updates. For example, the update operation in Section 3.2 demands that the view must be in a shape that matches the source pattern (as computed by Views), and this is usually achieved by an outer rearr, as is the case at lines 3–4 of Figure 2. In essence, rearr performs an invertible computation, while its *get* semantics performs the inverse of the computation. The invertible computation used is usually just moving things around and not complicated (hence the name "rearrangement"), so instead of letting the programmer write general invertible functions (probably along with their inverses), we ask instead for syntactic descriptions of simple invertible computations, which are more restrictive but allow automatic inversion.

The kind of invertible computation we wish to express is a decomposition of an old view followed by an assembling of a new view using all components of the old view — for example, lines 3–4 of Figure 2 simply express the computation:

```
λ { (title , price) → (title , tt , tt , price , tt) }
```

When it comes to decomposition, pattern matching should come to mind again; assembling can also be handled by pattern matching, as we have formulated pattern matching as an isomorphism *pat-iso* in Section 3.2. We thus require two patterns p : Pattern D and q : Pattern E, the former for decomposing the old view of type ⟦ D ⟧ and the latter for assembling the new view of type ⟦ E ⟧. For the latter pattern q, we also need to specify which component of the old view will be used as the value at each variable position. This last piece of information is represented by specialising VarPositions again:

```
Paths : {D E : U} → Pattern D → Pattern E → Set₁
Paths p q = VarPositions q (Path p)
```

where Path p T, defined below, is the type of all paths that lead from the root of a PatResult p to a component of type ⟦ T ⟧ at a variable position:

```
Path : {D : U} → Pattern D → U → Set₁
Path (var {D}) T = D ≡ T
Path unit      T = ⊥
Path (left  p) T = Path p T
Path (right p) T = Path p T
Path (prod p q) T = Path p T ⊎ Path q T
```

If a given bunch of *paths* : Paths p q use up all components of the old view, i.e., all possible paths for p are contained in *paths*, then p, q, and *paths* together specify a computation that can be automatically inverted — since all components of the old view are present somewhere in the new view, we can always reassemble the old view from the new view (unless there is inconsistency — a component of the old view might be copied into more than one positions of the new view, and the values at these positions of the new view must be the same when doing the reassembling).

In more detail, we can construct an isomorphism:

```
rearr-iso : {D E : U} →
  (p : Pattern D) (q : Pattern E) (paths : Paths p q) →
  Invertible p q paths → Iso ⟦ E ⟧ ⟦ D ⟧
```

where Invertible p q *paths* is defined to state that *paths* can pass a check for ensuring that all components of the old view are used. The *from* direction matches an inhabitant of ⟦ D ⟧ with the pattern p and uses the components as an environment for evaluating the pattern q to an inhabitant of ⟦ E ⟧. The *to* direction is more delicate: It creates an empty container of type VarPositions p (Maybe ∘ ⟦_⟧) (all of whose variable positions are nothing) and fills it with components obtained by matching the input ⟦ E ⟧ with q. If the container can be completely and consistently filled up — that is, every position in the container is filled with one and only one value — then we turn the container into a "necessarily full" container of type VarPositions p ⟦_⟧ — i.e., PatResult p — by stripping all the just tags, and use it as an environment for evaluating p to an inhabitant of $[\![D]\!]$.

We can now define the semantics of rearr. A natural choice is to use lens composition defined in Section 2.2:

$$\begin{array}{l} \mathit{interp}\ (\mathsf{rearr}\ p\ q\ \mathit{paths}\ b)\ = \\ \quad \mathit{interp}\ b \diamond \mathit{iso\text{-}lens}\ (\mathit{rearr\text{-}iso}\ p\ q\ \mathit{paths}\ \boxed{\ldots}) \end{array}$$

This definition is not good enough, however. The reason is subtle: The intended *put* semantics of rearr is to transform the view and then use b with the new view, but the above semantics performs one extra step that uses the *get* semantics of b to compute an intermediate source, which is then ignored according to the definition of *iso-lens*. This extra step turns out to be not only redundant but also detrimental: It is possible that the *put* semantics of b can accept sources that are not in the domain of its *get* semantics (for example, when b is a caseS with adaptive branches (Section 3.4)), and when that is the case, the intended *put* semantics of rearr should compute successfully, while the above semantics will fail to compute due to the extra invocation of *get*. We hence need to introduce a more specialised operator for composing a lens with an isomorphism:

$$\begin{array}{l} \_\triangleleft\_ : {}^{\{A\ B\ C\ :\ \mathsf{Set}\}\ \to}\ \mathsf{Lens}\ A\ B \to \mathsf{Iso}\ B\ C \to \mathsf{Lens}\ A\ C \\ l \triangleleft \mathit{iso}\ =\ \textbf{record} \\ \quad \{\ \mathit{put}\ =\ \lambda\ a\ c \to \mathsf{Iso}.\mathit{from}\ \mathit{iso}\ c\ \gg\!=\ \mathsf{Lens}.\mathit{put}\ l\ a \\ \quad ;\ \mathit{get}\ =\ \lambda\ a \to \mathsf{Lens}.\mathit{get}\ l\ a\ \gg\!=\ \mathsf{Iso}.\mathit{to}\ \mathit{iso}\ ;\ \boxed{\ldots}\ \} \end{array}$$

Note that its *put* does not invoke Lens.*get* l on a. The semantics of rearr can then be defined by

$$\mathit{interp}\ (\mathsf{rearr}\ p\ q\ \mathit{paths}\ b)\ =\ \mathit{interp}\ b \triangleleft \mathit{rearr\text{-}iso}\ p\ q\ \mathit{paths}\ \boxed{\ldots}$$

This is still not the actual definition, in fact — note that the invertibility proof is omitted. To be able to supply the invertibility proof, *interp* needs an additional argument consisting of invertibility proofs for all rearr operations appearing in the program being interpreted. This additional argument is omitted from the presentation for brevity.

3.4 Case Analysis on Source

The next construct caseS is for doing case analysis on the source. The basic idea of caseS's semantics is to choose from a list of branches the first one that matches the source, and then execute the BiGUL statement of that branch. It is, however, difficult to find a *get* to pair with this *put* semantics: The obvious *get* which selects the first branch that matches its source argument does not work, because, when we consider *PutGet*, it may well happen that the updated source produced by *put* matches a different branch from the one matching the original source, and hence in general we would have to guarantee *PutGet* for *put* and *get* coming respectively from any two branches, which is unmanageable. A more manageable solution is to insert a dynamic check at the end of *put* to ensure that the updated source must match the same branch as the original source, which is the solution adopted by Foster et al.'s concrete conditional lens [8, Section 6.1].

3.4.1 Enriching Source Case Analysis with Adaptive Branches

The above solution, however, is sometimes too restrictive in practice. For example, in our putback-based language BiYacc [28] for generating well-behaved pairs of parsers and "reflective" printers, a program specifies how a printer *updates* a concrete syntax tree (the source) to become consistent with an abstract syntax tree (the view). The printer tries to retain the structure of a source wherever possible, but when the structure of the source is radically different from that of the view, we need to change the source completely, and this change of source structure would fail the aforementioned dynamic check that prevents branch switching. We therefore need a more flexible case analysis on source. (See Section 4 for a simpler scenario where we also need this increased flexibility.)

Our solution is to add a special kind of branches called *adaptive* branches to source case analysis, in addition to *normal* branches. The two kinds of branches are defined by the following datatype:

$$\begin{array}{l} \textbf{data}\ \mathsf{CaseSBranchType}\ (S\ V : \mathsf{Set}) : \mathsf{Set}_1\ \textbf{where} \\ \quad \mathsf{normal}\ \ \ : \mathsf{Lens}\ S\ V \qquad \to \mathsf{CaseSBranchType}\ S\ V \\ \quad \mathsf{adaptive} : (S \to \mathsf{Par}\ S) \to \mathsf{CaseSBranchType}\ S\ V \end{array}$$

A normal branch is just a lens, while an adaptive branch is a source transformation. With each branch we also need to associate a decidable predicate on the source type specifying when a source matches the branch, so the complete definition of the type of branches is

$$\begin{array}{l} \mathsf{CaseSBranch} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}_1 \\ \mathsf{CaseSBranch}\ S\ V\ =\ (S \to \mathsf{Par}\ \mathsf{Bool}) \times \mathsf{CaseSBranchType}\ S\ V \end{array}$$

The lens semantics of caseS is then given by

$$\mathit{caseS\text{-}lens} : (S\ V : \mathsf{Set}) \to \mathsf{List}\ (\mathsf{CaseSBranch}\ S\ V) \to \mathsf{Lens}\ S\ V$$

which is built from a list of branches.

3.4.2 The *put* and *get* Semantics

The *put* and *get* components of *caseS-lens* are shown in Figure 4. For the *put* semantics, we might think of a source case analysis as still consisting mainly of normal branches, but we can now specify additional cases such that when a source matches none of the normal branches, the source can still be accepted, which is then transformed — or adapted — to one that will match a normal branch. More explicitly, the case analysis may be run twice: If a normal branch is chosen the first time, then the case analysis succeeds and we execute the branch; otherwise, we adapt the source and rerun the case analysis, and must match a normal branch this time. A single round of the case analysis is performed by the function *put-with-adaptation*, which takes a continuation of type $S \to \mathsf{Par}\ S$ that, when an adaptive branch is matched, is invoked on the adapted source. The *put* semantics, which consists of up to two rounds of the case analysis, is *put-with-adaptation* with a second instance of *put-with-adaptation* as the continuation, and the continuation for the latter is the computation that always fails — that is, the second round cannot fall into an adaptive branch. In either round, if a normal branch is matched and executed, we must ensure that the updated source satisfies the predicate of the branch and also none of the predicates of the previous branches, so a subsequent execution of *get* on the updated source — which chooses a branch in the same way as *put* — will go through the same branch as *put*, establishing *PutGet*. In *put-with-adaptation*, there is an accumulating parameter bs' keeping hold of the mismatched branches; if a normal branch is matched and executed, the function *check-diversion* will be invoked to ensure that the updated source matches none of the branches in bs'.

The *get* semantics, on the other hand, simply tries to match the source with each of the branches and execute the first matched branch if the branch is normal, or fail if the branch is adaptive. A quick reasoning for the behaviour about adaptive branches is as follows: An abstract reading of the well-behavedness laws says that the domain of *get* must coincide with the range of updated sources produced by *put*. Since a successfully updated source necessarily comes out from, and will again match, a normal branch, *get* must fail for any source matching an adaptive branch.

Example. Of course, we still need to prove the well-behavedness properties to actually say that the *put* and *get* functions are defined correctly. But before doing so, let us look at a minimal example that uses adaptation. Suppose that the source is a natural number and the view has type $\top$. That is, there is no information in the view to be

```
caseS-lens : (S V : Set) → List (CaseSBranch S V) → Lens S V
caseS-lens S V bs = record { put = λ s v → put-with-adaptation bs [] s v (λ s′ → put-with-adaptation bs [] s′ v (λ _ → fail))
                           ; get = get bs ; ... }
  where
    branch : {l : Level} {X : Set l} → (Lens S V → X) → ((S → Par S) → X) → CaseSBranchType → X
    branch f g (normal   l ) = f l
    branch f g (adaptive u) = g u

    check-diversion : List (CaseSBranch S V) → S → Par ⊤
    check-diversion []             s = return tt
    check-diversion ((p , _) :: bs) s = p s >>= λ matched → if matched then fail else check-diversion bs s

    put-with-adaptation : List (CaseSBranch S V) → List (CaseSBranch S V) → S → V → (S → Par S) → Par S
    put-with-adaptation []               bs′ s v cont = fail
    put-with-adaptation ((p , b) :: bs) bs′ s v cont =
      p s >>= λ matched → if matched then branch (λ l → Lens.put l s v >>= λ s′ → p s′ >>= λ matched′ →
                                                     if matched′ then check-diversion bs′ s′ >>= (λ _ → return s′) else fail)
                                                 (λ u → u s >>= cont) b
                          else  put-with-adaptation bs ((p , b) :: bs′) s v cont

    get : List (CaseSBranch S V) → S → Par V
    get []             s = fail
    get ((p , b) :: bs) s = p s >>= λ matched → if matched then branch (λ l → Lens.get l s) (λ _ → fail) b else get bs s

    ...
```

Figure 4. Definitions of *put* and *get* for source case analysis

put back. There is, however, a requirement that the source should be "normalised" to an even number. We can turn this requirement into a dynamic check by wrapping skip in a caseS:

```
caseS ((return ∘ isEven , normal skip) :: [])
    -- isEven : ℕ → Bool
```

Both the *put* and *get* semantics of this BiGUL program compute successfully if and only if the input source is even. However, we can make *put* also accept odd sources by adding an adaptive branch:

```
toEven : BiGUL (k ℕ) one
toEven = caseS
  ((return ∘ isEven        , normal   skip) ::
   ((λ _ → return true) , adaptive (λ n → return (n − 1))) :: [])
```

That is, normally we expect the source to be an even number, but if an odd source does show up, *put* will be capable of "normalising" it: The odd source will fall into the second adaptive branch and get transformed into an even number, which will then match the first normal branch, and be returned as the updated source.

3.4.3 Sketch of the Well-Behavedness Proofs

As explained in Section 2.2, the well-behavedness proofs for *caseS-lens* proceed by analysing all possible ways for *put* or *get* to compute successfully, and then showing how their successful computation guarantees successful computation of the corresponding *get* or *put*. The only slight complication about *caseS-lens* is that *put-with-adaptation* has an accumulating parameter, which implies that we need to prove generalised versions of the well-behavedness statements. For *PutGet*, the main statement we prove is

```
PutGet-with-adaptation :
  (bs bs′ : List (CaseSBranch S V))
  {s s′ : S} {v : V} {cont : S → Par S} →
  ({s : S} → (cont s ↦ s′) → (get (revcat bs′ bs) s′ ↦ v)) →
  (put-with-adaptation bs bs′ s v cont ↦ s′) →
  (get (revcat bs′ bs) s′ ↦ v)
```

where *revcat* is defined by

```
revcat : {l : Level} {A : Set l} → List A → List A → List A
revcat []        ys = ys
revcat (x :: xs) ys = revcat xs (x :: ys)
```

which is the usual way of implementing linear-time list reversal, and coincides with how branches are moved from *bs* to *bs′* in *put-with-adaptation*. *PutGet* is then proved by setting *bs′* to [] and applying *PutGet-with-adaptation* to itself in a way similar to how we define *put* in terms of *put-with-adaptation*. For *GetPut*, the generalisation we need is relatively simpler:

```
GetPut-with-adaptation :
  (bs bs′ : List (CaseSBranch S V))
  {cont : S → Par S} {s : S} {v : V} →
  (check-diversion bs′ s ↦ tt) →
  (get bs s ↦ v) → (put-with-adaptation bs bs′ s v cont ↦ s)
```

GetPut is then proved by setting *bs′* to [], in which case the premise about *check-diversion* becomes trivially true.

3.4.4 Fitting the Lens to the Interpreter

The final step is to define a corresponding constructor, caseS, in the BiGUL datatype. We start with defining a variant of CaseSBranchType whose source and view type parameters are codes from the universe U (Section 3.2) and whose normal constructor takes a BiGUL statement instead of a Lens:

```
data CaseSBranchTypeᴮ (S V : U) : Set₁ where
  normal   : BiGUL S V          → CaseSBranchTypeᴮ S V
  adaptive : (⟦ S ⟧ → Par ⟦ S ⟧) → CaseSBranchTypeᴮ S V
```

and also a variant of CaseSBranch:

```
CaseSBranchᴮ : U → U → Set₁
CaseSBranchᴮ S V =
  (⟦ S ⟧ → Par Bool) × CaseSBranchTypeᴮ S V
```

```
-- Assume S : Set and V : Set
get-with-check : (V → Par Bool) → Lens S V → S → Par V
get-with-check p l s = Lens.get l s >>= λ v →
                       p v >>= λ matched →
                       if matched then return v else fail
put : (bs : List (CaseVBranch S V)) → S → V → Par S
put []              s v = fail
put ((p , l) :: bs) s v =
  p v >>= λ matched →
  if matched then Lens.put l s v
             else  put bs s v >>= λ s' →
                   catch (get-with-check p l s') (λ _ → fail)
                         (return s')
get : (bs : List (CaseVBranch S V)) → S → Par V
get []              s = fail
get ((p , l) :: bs) s =
  catch (get-with-check p l s) return
        (get bs s >>= λ v → p v >>= λ matched →
         if matched then fail else return v)
```

Figure 5. Definitions of *put* and *get* for view case analysis

As shown in Figure 1, the caseS constructor of BiGUL takes a list of CaseSBranch^{B}s. Its interpretation is, unsurprisingly, *caseS-lens*:

```
mutual
  ...
  interp (caseS {S} {V} bs) =
    caseS-lens ⟦ S ⟧ ⟦ V ⟧ (interp-caseS bs)
  ...
  interp-caseS : {S V : U} → List (CaseSBranch^B S     V  ) →
                             List (CaseSBranch ⟦ S ⟧ ⟦ V ⟧)
  interp-caseS []                        = []
  interp-caseS ((p , normal b        ) :: bs) =
               (p , normal (interp b)) :: interp-caseS bs
  interp-caseS ((p , adaptive u      ) :: bs) =
               (p , adaptive u       ) :: interp-caseS bs
```

The helper function *interp-caseS* simply maps *interp* into the normal branches. To help AGDA see that *interp* is terminating, though, we need to define *interp-caseS* explicitly (instead of using the standard *map* function on lists) and put the definition along with *interp* in a **mutual** block.

3.5 Case Analysis on View

For caseV, we still try to match the view with a list of branches, and execute the first branch that matched, like for caseS. Unlike caseS, though, there are no adaptive branches for caseV, so the definition of branches is more straightforward:

```
CaseVBranch : Set → Set → Set_1
CaseVBranch S V = (V → Par Bool) × Lens S V
```

The lens semantics we assign to caseV then has type

```
caseV-lens : (S V : Set) → List (CaseVBranch S V) → Lens S V
```

Its *put* and *get* are shown in Figure 5. The definition of *put* is basically what one would expect. As for *get*, there is no obvious way to decide which branch to go — unlike *put*, we are not given a view to match with the branches. Here we try to execute each of the branches until we reach one that computes a view successfully, which is the approach taken by Pacheco et al. [24]. This requires Par to be extended with one more constructor:

```
catch : {A B : Set} → Par A → (A → Par B) → Par B → Par B
```

interpreted by

```
runPar (catch mx f my) with runPar mx
runPar (catch mx f my) | just x   = runPar (f x)
runPar (catch mx f my) | nothing  = runPar my
```

That is, catch *mx f my* behaves like *mx* $\gg\!=$ *f* when *mx* computes successfully, or becomes *my* when *mx* fails. Execution of *get* can then be specified to try subsequent branches if the current branch fails to compute.

Example. Suppose that the source is a natural number and the view is $\top \uplus \top$, and the source should be even if the view is inj$_1$ tt, or odd if the view is inj$_2$ tt. To establish this relationship, we can use the *toEven* program presented at the end of Section 3.4.2 and an analogous program *toOdd* inside a caseV:

```
caseV ((return ∘ isInj_1 ,    -- isInj_1 : {A B : Set} → A ⊎ B → Bool
          rearr (left   unit) unit tt toEven) ::
        ((λ _ → return true) ,
          rearr (right unit) unit tt toOdd ) :: [])
```

If the view is inj$_1$ tt, we rearrange it to tt and invoke *toEven* to update the source to an even number; the case where the view is inj$_2$ tt is analogous. This is a common pattern for caseV (which will appear again in Section 4): Each branch begins with a view rearrangement which gets rid of some information that has been used to choose this branch, and then uses a caseS to update the source to correspond to the view. As for the *get* semantics, an even source will successfully compute to tt through *toEven* and then to inj$_1$ tt by the rearrangement, while an odd source will fail to compute through *toEven* and fall to the second branch, eventually producing inj$_2$ tt.

Well-behavedness. For the well-behavedness properties, *PutGet* is the more difficult one to establish, as we cannot easily guarantee that executing *put* followed by *get* will both go through the same branch. Our compromised solution is to make *put* do an expensive check after a branch is matched and executed, requiring that the *get* for each of the previous branches must fail. For the programmer, this means that, in order for a caseV to be able to pass this check, the ranges of the *put* semantics of the branches (which coincide with the domains of *get*) should be disjoint.

3.6 List Alignment

For the semantics of align, which is the key operation used in Figure 2 (and in the BIFLUX language [25]), we employ a particular parametrised strategy for updating a list of sources of type *S* with a list of views of type *V*, which tries to align/match the sources with the views, synchronise the matched pairs of sources and views by an inner lens, and process the unmatched sources and views in some programmer-specified ways. The purpose of align is similar to Barbosa et al.'s matching lenses [2], but we do not intend align to be as expressive as matching lenses yet — a few of the design choices made below will appear somewhat arbitrary and inflexible, but the semantics is already interesting enough such that formal verification is desirable.

The lens semantics of align has type

```
align-lens : {S V : Set} →
  (source-condition : S → Par Bool)
  (matching-condition : S → V → Par Bool)
  (l : Lens S V)
  (create : V → Par S)
```

$(conceal : S \to \mathsf{Par}\ (\mathsf{Maybe}\ S)) \to$
$\mathsf{Lens}\ (\mathsf{List}\ S)\ (\mathsf{List}\ V)$

To explain its *put* semantics more concretely, below we will use the program in Figure 2 to put the view list

```
("Pottery" , 850) :: ("Hiking" , 550) :: []
```

into the source list

```
("Pottery" , "Rowling" , 2016 , 950 , true) ::
("Habits"  , "Tolkien" , 1937 , 450 , true) ::
("Nanny"   , "Lewis"   , 2016 , 650 , true) :: []
```

step by step.

- First, from the source list we can select a sub-list that we intend to align with the views by specifying a decidable predicate *source-condition* : $S \to \mathsf{Par\ Bool}$. For our example, the sub-list consists of the entries for the books *Pottery* and *Nanny*.
- The sub-list of sources satisfying *source-condition* are then aligned with the list of views by finding pairs of source and view satisfying a given binary predicate *matching-condition* : $S \to V \to \mathsf{Par\ Bool}$. The matching order is fixed in our strategy: For each view we find the first matching source that has not been matched with a previous view. For our example, there is one matching pair, namely the source and view entries for *Pottery*.
- On each matched pair of source and view we execute an inner lens $l : \mathsf{Lens}\ S\ V$ to update the source with the view. For our example, l updates the title and price, so the price of *Pottery* is set to 850 in the updated source.
- For unmatched views, we should create corresponding sources in the source list (so a subsequent *get* can produce these views). This is done by first applying a programmer-specified function *create* : $V \to \mathsf{Par}\ S$ to the view to create a temporary source, and then updating this source with the view by l. For our example, *Hiking* in an unmatched view, so we create a temporary source and update its title and price to "Hiking" and 550 respectively, producing an entry

  ```
  ("Hiking" , "(to be updated)" , 2016 , 550 , true)
  ```

- For unmatched sources, we can choose to simply delete them or transform them such that they do not satisfy *source-condition*. This is specified by a function *conceal* : $S \to \mathsf{Par}\ (\mathsf{Maybe}\ S)$: An unmatched source is deleted if applying *conceal* to it produces nothing, or transformed if *conceal* produces a new source wrapped in just. For our example, *Nanny* is an unmatched source, and we conceal the entry by setting its *instock* flag to false. These transformed sources can be placed anywhere in the source list; currently we simply place all of them towards the front of the list.
- Finally, the list of updated sources is merged with those not satisfying *source-condition* in the beginning. The merging order is chosen such that *GetPut* can be established automatically. For our example, the resulting list is

  ```
  ("Nanny"   , "Lewis"           , 2016 , 650 , false) ::
  ("Habits"  , "Tolkien"         , 1937 , 450 , true ) ::
  ("Pottery" , "Rowling"         , 2016 , 850 , true ) ::
  ("Hiking"  , "(to be updated)" , 2016 , 550 , true ) :: []
  ```

The corresponding *get* semantics extracts from the input source list all the sources satisfying *source-condition* and then uses Lens.*get* l to get a list of views from these sources. For example, running the *get* direction of Figure 2 on the updated source above will first extract the entries for *Pottery* and *Hiking*, and then get their title and price, producing the view list we started with (and thus conforming to the *PutGet* law).

To guarantee well-behavedness, a few checks about l and *conceal* need to be inserted: After a pair of source and view are synchronised by l, the updated pair should again satisfy *matching-condition*, and also the updated source should still satisfy *source-condition*. And a source produced by *conceal* must not satisfy *source-condition*. The program in Figure 2 is written such that these checks will always be successful: Since the lens l updates the title, the *matching-condition* — which compares the titles — will definitely hold for the updated source and view. Also, the updated source will continue to satisfy *source-condition* because the publication year and *instock* flag are not changed by l. Finally, *conceal* sets the *instock* flag to false, necessarily invalidating *source-condition*.

4. A Showcase Example: Transatlantic Corporation

In this section we look at an example in detail, which requires nontrivial putback logic, in particular involving caseS and caseV. Suppose that a transatlantic corporation regularly relocates employees between its UK and US offices, and pays them in the local currency. The payroll database stores for each employee their name, salary (a number interpreted as pounds or dollars depending on which country the employee works in), and current office location:

$$Employee = Name \otimes (Salary \otimes Location) : \mathsf{U}$$

where $Name : \mathsf{U}$ and $Salary : \mathsf{U}$ are datatypes representing name and salary encoded as elements of U (e.g., $\mathsf{k\ String}$ and $\mathsf{k}\ \mathbb{N}$). *Location* is defined as a disjoint sum:

$$Location = UKLocation \oplus USLocation : \mathsf{U}$$

where $UKLocation : \mathsf{U}$ and $USLocation : \mathsf{U}$ are, again, encoded datatypes representing UK and US office locations. To manage relocation, we extract from the payroll database each employee's current office location:

$$Office = Name \otimes Location : \mathsf{U}$$

We would like to add, remove, or relocate employees by modifying the extracted list and put the modified list back to the payroll database. Most interestingly, when an employee is relocated to a different country, their salary should also be converted to the local currency. Below we describe this putback logic in BiGUL.

At top level, we align a list of *Employee*s with a list of *Office*s:

```
relocate-employees : BiGUL (list Employee) (list Office)
relocate-employees = align
  (λ _ → return true)
  (λ {(sname , _) (vname , _) → return (sname == vname)})
  relocate-employee
  (λ {(_ , loc) → return ("" , 0 , loc)})
  (λ _ → return nothing)
```

We are considering all *Employee*s in the list, so the source condition is always true. Entries from the two lists are matched by employee name. Pairs of matched *Employee* and *Office* are synchronised by an inner BiGUL statement *relocate-employee*, to be defined later. An unmatched *Office* means that a new employee is inserted, so we create a temporary *Employee* in the source list, which is then updated by *relocate-employee* using the unmatched *Office*. (We put the view location in the temporary *Employee* merely for efficiency, as *relocate-employee* would not have to deal with mismatching source and view locations.) An unmatched *Employee* means that the employee is removed from the view, so we conceal the entry by deleting it from the source.

Matched pairs of *Employee* and *Office* are synchronised by

```
relocate-employee : BiGUL Employee Office
relocate-employee =
```

```
    update (prod var              var)
           ((_ ,  replace) , (_ , adjust-salary-and-office)))
```

We decompose the source by pattern matching to replace the name in the source with the name in the view — this looks like a redundant step but is required because BiGUL enforces that all view information, in particular the name, must be put into the source. The rest of the source, i.e., salary and location, is further synchronised with the remaining view by

```
adjust-salary-and-office : BiGUL (Salary ⊗ Location) Location
adjust-salary-and-office =
  caseV ((return ∘ isInj₁ ,
          rearr (left  var) (prod unit var) (tt , refl)
                  -- λ {(inj₁ loc) → (tt , loc)}
                relocate-to-UK) ::
         ((λ _ → return true) ,
          rearr (right var) (prod unit var) (tt , refl)
                  -- λ {(inj₂ loc) → (tt , loc)}
                relocate-to-US ) :: [])
```

In *adjust-salary-and-office*, we use caseV to distinguish which country the employee will work in, and use rearr to get rid of the inj_1 or inj_2 tag and also insert a tt as the view to be used to update the *Salary* in the source later. If the employee will work in the UK, the update to the source is

```
relocate-to-UK : BiGUL (Salary ⊗ Location    )
                       (one    ⊗ UKLocation)
relocate-to-UK = caseS
  ((return ∘ isInj₁ ∘ proj₂ ,   -- proj₂ : {A B : Set} → A × B → B
    normal  (update (prod var (left   var))
                    ((_ ,  skip) , (_ , replace)))) ::
   ((λ _ → return true) ,
    adaptive (λ {(salary , _) →
                   return ($⇒£ salary , inj₁ "")})) :: [])
```

We probe which country the employee is in originally with a caseS. If the original country is the UK, the employee is not being relocated to a different country, so we can skip updating the salary and just replace the location. If it is the US, we adapt the source by changing the salary from dollars to pounds (by a function $\$\!\Rightarrow\!£ : \mathit{Salary} \rightarrow \mathit{Salary}$ defined elsewhere) and change the location to an empty one in the UK, which will be replaced by the view location in the second round of the source case analysis following the adaptation. The other branch *relocate-to-US* is defined symmetrically.

To ensure totality of *put*, we should now check whether all constraints for passing dynamic checking are met, working from the inner statements towards the outer ones. For the caseS statements in *relocate-to-UK* and *relocate-to-US*, the normal branches do not change the source country and thus do not switch branch, and the adaptive branches do change the source so it matches a normal branch. For the caseV in *adjust-salary-and-office*, the ranges of *relocate-to-UK* and *relocate-to-US* are disjoint since they respectively produce UK and US locations. For the align in *relocate-employees*, since the source condition is always true, whatever *relocate-employee* produces necessarily satisfies the source condition, and we must conceal unmatched sources by deleting them, which is indeed what we specify; finally, we must guarantee that the updated source produced by *relocate-employee* and the view again satisfy the matching condition, even when the view is an unmatched one and the original source is a temporary one created from the view, and this is indeed guaranteed because *relocate-employee* puts the name in the view into the source.

5. Concluding Remarks

We have given an account of the design and semantics of BiGUL and also how the language and its well-behavedness are formalised and proved in AGDA. Below we conclude this paper by commenting on the putback-based approach to bidirectional transformations, emphasising our formal development and the monad reification trick, discussing issues about dynamic checking, comparing BiGUL with its closest relative PUTLENSES, and talking briefly about porting BiGUL to HASKELL.

One might wonder how BiGUL, being described as a putback-based language, stands out from existing work on lenses — after all, the fundamental definition of Lens is exactly the classic definition which does not emphasise either the *put* or *get* side. We designed BiGUL such that the natural way to understand a BiGUL program is to read it as a description of putback logic, and, more importantly, a BiGUL program can be written by thinking solely in the putback direction (as opposed to programming the *get* direction or, worse, switching between *get* and *put*). (It may appear that the programmer needs to devise a *get* before programming a corresponding *put*, but that impression most likely stems from the confusion between general BX specifications — e.g., consistency *relations* between sources and views — and *get functions* as a much more specialised form of BX specifications. The programmer should of course have a specification in mind before programming *put*, but that specification is not necessarily a *get* function.)

We have to admit, though, that the expressive power of the current BiGUL is not stronger than existing work on lenses. In particular, we came up with the adaptive caseS to solve a practical problem we encountered [28], but realised only later that the idea already existed as Foster et al.'s "fixup functions"; the common pattern using adaptive caseS inside caseV, which appears in the examples in Sections 3.5 and 4, is essentially their general conditional lens [8, Section 6.3]. (Foster et al. did not give an example using their general conditional lens, though, possibly due to the complexity of its putback semantics. It might be argued that, without switching to the putback-based perspective, it is hard to use this kind of more complex lenses. This is only an indirect support of the putback-based approach, though.) We thus believe that the main contribution of this paper should be its completely formal development: Grohne et al. [12] have also worked on formal verification about BXs (in AGDA), but their work was about semantic bidirectionalisation based on parametric polymorphism [26]. Our work is the first to formally describe and verify a collection of nontrivial and practically implemented lenses in a dependently typed language, in particular utilising the power of dependent types to design a logically clean abstract syntax and make well-behavedness proofs much easier to handle.

As explained in Section 2, crucial to the ease of our formal development is the monad reification trick. Essentially, this is another demonstration of the flexibility of deep embedding (see, e.g., [11]). By deeply embedding the monadic program structure, we are able to interpret it in two different but related ways, either as a Maybe-program or as a proposition stating that the Maybe-program computes successfully. We believe that the trick is generally applicable, and may well have been applied elsewhere.

One might be sceptical about the use of dynamic checks to guarantee the well-behavedness of BiGUL, as the programmer can write programs that fail inadvertently and only realise that at runtime. We currently deal with this problem by informally stating the constraints that should be satisfied by the programmer-supplied actions in order to pass the dynamic checks. It is also possible to turn such statements into formal theorems saying that a *put* succeeds if relevant constraints are satisfied. We do not take this step because, as future work, we plan to jump further by switching to a dependently typed setting where constraints on programmer-supplied actions can be directly specified in their types. This will not only eliminate all dynamic checks from the lens semantics, but also help the programmer to write correct putback programs with the type system (as demonstrated by, e.g., Norell [22]).

BiGUL is closely related to PUTLENSES [24]: Both BiGUL and PUTLENSES are datatype-generic and describe partial putback transformations that use dynamic checks to guarantee well-behavedness. BiGUL does not strive for maximal expressiveness like PUTLENSES (in particular, BiGUL does not consider effectful computations in *put*), but instead takes a more cautious approach by focusing on a smaller set of essential features and formally proving their well-behavedness. The dynamic checks are sometimes tricky to get right, and proofs about them can require detailed case analyses that are not so easy to manage. (Indeed, during the formalisation we did not get everything right the first time and had to make corrections after failing to complete the proofs.) Thus the fact that the whole BiGUL language is formally verified is a firm step towards reliable putback-based bidirectional programming: As BiGUL is intended to serve as the (new) core of higher-level putback-based languages like BIFLUX [25] and BIYACC [28], the well-behavedness of the latter languages will be established beyond doubt.

To (re-)implement BIFLUX and BIYACC, we have ported BiGUL to HASKELL. The types need some adaptation because HASKELL is not fully dependently typed (but close enough; see, e.g., Lindley and McBride [17]), while the transformations can be more or less faithfully transcribed. A significant difference between AGDA and HASKELL is that HASKELL freely allows general recursion, whereas AGDA requires every recursive program to be evidently well-founded. We do not consider recursion in the AGDA formalisation, but we need general recursion in BIFLUX and BIYACC, which is a major reason for porting BiGUL to HASKELL. We believe that the total correctness proved in AGDA can be translated into some sort of partial correctness in HASKELL (perhaps along the line of Danielsson et al. [6]); that is, a BiGUL program in HASKELL may not terminate, but when it does, it is guaranteed to be well-behaved.

Acknowledgements

We would like to thank Li Liu, Hugo Pacheco, Jeremy Gibbons, and the anonymous reviewers of APLAS '15 and PEPM '16 for their valuable comments. This work is partially supported by JSPS Grant-in-Aid for Scientific Research (A) No. 25240009 in Japan.

References

[1] T. Altenkirch and C. McBride. Generic programming within dependently typed programming. In *IFIP TC2/WG2.1 Working Conference on Generic Programming*, pages 1–20. Kluwer, B.V., 2003.

[2] D. M. J. Barbosa, J. Cretin, J. N. Foster, M. Greenberg, and B. C. Pierce. Matching lenses: alignment and view update. In *International Conference on Functional Programming*, pages 193–204. ACM, 2010.

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

[4] J. Cheney. FLUX: functional updates for XML. In *International Conference on Functional Programming*, pages 3–14. ACM, 2008.

[5] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. Terwilliger. Bidirectional transformations: a cross-discipline perspective. In *International Conference on Model Transformation*, volume 5563 of *Lecture Notes in Computer Science*, pages 260–283. Springer-Verlag, 2009.

[6] N. A. Danielsson, J. Gibbons, J. Hughes, and P. Jansson. Fast and loose reasoning is morally correct. In *Principles of Programming Languages*, pages 206–217. ACM, 2006.

[7] J. N. Foster. *Bidirectional Programming Languages*. PhD thesis, University of Pennsylvania, 2009.

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

[9] J. N. Foster, A. Pilkiewicz, and B. C. Pierce. Quotient lenses. In *International Conference on Functional Programming*, pages 383–396. ACM, 2008.

[10] J. Gibbons. Datatype-generic programming. In *Spring School on Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 1–71. Springer-Verlag, 2007.

[11] J. Gibbons. Functional programming for domain-specific languages. In *Central European Functional Programming School*, volume 8606 of *Lecture Notes in Computer Science*, pages 1–28. Springer-Verlag, 2015.

[12] H. Grohne, A. Löh, and J. Voigtländer. Formalizing semantic bidirectionalization with dependent types. In *International Workshop on Bidirectional Transformations*, pages 75–81. CEUR-WS, 2014.

[13] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *International Conference on Functional Programming*, pages 205–216. ACM, 2010.

[14] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric lenses. In *Principles of Programming Languages*, pages 371–384. ACM, 2011.

[15] M. Hofmann, B. C. Pierce, and D. Wagner. Edit lenses. In *Principles of Programming Languages*, pages 495–508. ACM, 2012.

[16] Z. Hu, H. Pacheco, and S. Fischer. Validity checking of putback transformations in bidirectional programming. In *International Symposium on Formal Methods*, volume 8442, pages 1–15. Springer-Verlag, 2014.

[17] S. Lindley and C. McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. In *Haskell Symposium*, pages 81–92. ACM, 2013.

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

[19] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[20] U. Norell. *Towards a Practical Programming Language based on Dependent Type Theory*. PhD thesis, Chalmers University of Technology, 2007.

[21] U. Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer-Verlag, 2009.

[22] U. Norell. Interactive programming with dependent types. In *International Conference on Functional Programming*, pages 1–2. ACM, 2013.

[23] H. Pacheco and A. Cunha. Generic point-free lenses. In *Mathematics of Program Construction*, volume 6120 of *Lecture Notes in Computer Science*, pages 331–352. Springer-Verlag, 2010.

[24] H. Pacheco, Z. Hu, and S. Fischer. Monadic combinators for "putback" style bidirectional programming. In *Partial Evaluation and Program Manipulation*, pages 39–50. ACM, 2014.

[25] H. Pacheco, T. Zan, and Z. Hu. BiFluX: a bidirectional functional update language for XML. In *Principles and Practice of Declarative Programming*, pages 147–158. ACM, 2014.

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

[27] P. Wadler. The essence of functional programming. In *Principles of Programming Languages*, pages 1–14. ACM, 1992.

[28] Z. Zhu, H.-S. Ko, P. Martins, J. Saraiva, and Z. Hu. BiYacc: roll your parser and reflective printer into one. In *International Workshop on Bidirectional Transformations*, pages 43–50. CEUR-WS, 2015.