Generic Point-free Lenses

Hugo Pacheco and Alcino Cunha

DI-CCTC, Universidade do Minho, Braga, Portugal
{hpacheco,alcino}@di.uminho.pt

Abstract. Lenses are one the most popular approaches to define bidirectional transformations between data models. A bidirectional transformation with *view-update*, denoted a *lens*, encompasses the definition of a forward transformation projecting *concrete models* into *abstract views*, together with a backward transformation instructing how to translate an abstract view to an update over concrete models. In this paper we show that most of the standard point-free combinators can be lifted to lenses with suitable backward semantics, allowing us to use the point-free style to define powerful bidirectional transformations by composition. We also demonstrate how to define generic lenses over arbitrary inductive data types by lifting standard recursion patterns, like folds or unfolds. To exemplify the power of this approach, we "lensify" some standard functions over naturals and lists, which are tricky to define directly "by-hand" using explicit recursion.

Keywords: point-free, bidirectional transformations, lenses, recursion patterns, inductive types.

1 Introduction

With the ever growing list of programming languages and application development frameworks, transforming a data format into a different format is essential to "bridge the gap" between technology layers and ensure sharing of information among software applications. Moreover, users generally expect transformations to be bidirectional, in the sense that changes made to one of the models can be safely propagated to its connected pair (imagine the synchronization of a laptop's and a cellphone's contact list).

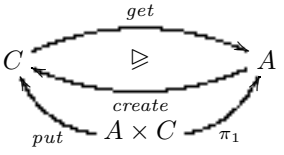
The naive way to create a *bidirectional transformation* is to engineer two unidirectional transformations together and manually prove that they are somehow consistent. This is likely to cause a maintenance problem, besides being a notoriously expensive and error-prone task. Any change in a data format implies a redefinition of both transformations, and a new consistency proof.

A better approach is to design a domain-specific language in which one expression denotes both transformations, which are then guaranteed to be consistent by construction in the respective semantic space. Following this notion, approaches to bidirectional transformations have emerged in the most diverse computing domains, including heterogeneous data synchronization [15,6], software model

C. Bolduc, J. Desharnais, and B. Ktari (Eds.): MPC 2010, LNCS 6120, pp. 331–352, 2010.

transformation [26], schema evolution [9,4], constraint maintenance for graphical user interfaces [21], interactive structure editing [19] and relational databases [7]. By restricting the domain-specific language to particular domains, these approaches overcome the difficulty of designing bidirectional transformations, and achieve a neat balance between expressiveness and the robustness imposed by the consistency constraints.

One of the most successful approaches to bidirectional transformations are the so-called *lenses*, proposed by Foster *et al.* [15] to solve the classical *view-update problem* originating from database theory [3]: when a concrete data model is abstracted into a view, how can changes made to the view be propagated back as updates to the original model? According to the following diagram, a *lens* comprises the definition of three functions involving a concrete data model C and its abstract counterpart A:



The first ingredient of a lens consists of the definition of a view: function $get : C \to A$ abstracts away details from the concrete model that are irrelevant for a specific purpose. Since this abstraction implies loss of information, the backwards transformation $put : A \times C \to C$ is augmented with knowledge of the original concrete instance, rendering it capable to restore some information no longer present in the view. As this is not always possible, a default concrete model is sometimes reconstructed by applying $create : A \to C$ to the view.

Of course, these three functions should be somehow consistent in order to define a *well-behaved* lens. First, *get* must be an abstraction function, i.e, A shall contain at most as much information as C. In a sense, a lens is a dual concept of refinement [23,25]. Second, the lens should be *acceptable*, i.e, updates to a view cannot be ignored and must be translated exactly. Finally, the lens should be *stable*, i.e, if the view does not change, then neither should the source. These properties will be formally defined in the next section.

As an example, let's consider as a concrete data model lists of natural numbers. A possible lens over this data type is determined by the length of the list. In Haskell we could define the *get* function trivially as follows:

$$\begin{aligned}
&\textbf{data}\ Nat = Zero \mid Succ\ Nat\\
&get :: [Nat] \to Nat\\
&get\ [\,] \quad\ = Zero\\
&get\ (x : xs) = Succ\ (get\ xs)
\end{aligned}$$

Given a natural number, *create* must generate a default list of that length:

$$\begin{aligned}
&create :: Nat \to [Nat]\\
&create\ Zero \quad\ = [\,]\\
&create\ (Succ\ n) = Zero : create\ n
\end{aligned}$$

For the lens to be well-behaved, *create* could use other defaults but cannot create a list with a different length. The *put* function is a bit more tricky. A possible definition that guarantees well-behavedness is:

$$
\begin{aligned}
&put :: (Nat, [Nat]) \to [Nat] \\
&put\ (Zero, \_) &&= [\,] \\
&put\ (Succ\ n, [\,]) &&= Zero : put\ (n, create\ n) \\
&put\ (Succ\ n, x : xs) &&= x : put\ (n, xs)
\end{aligned}
$$

If the view (i.e, the length of the list) remains the same or decreases, elements of the original list must be used in the new concrete value. If the length increases, defaults are invented for the new elements.

For more complex data formats and abstractions, the definition of a *put* function that guarantees well-behavedness becomes highly complex. As such, Foster *et al.* [15] propose a combinatorial approach to the definition of lenses over generalized trees: complex lenses are defined by composition of more simpler lenses using a standard set of combinators and recursion. This combinatorial approach is also central to the so-called *point-free* style of programming, popularized by John Backus in his 1977 Turing award lecture [2]. In this variable-free style, functions are defined by composition using a standard set of higher-order combinators, characterized by a rich set of algebraic laws that make this style particularly amenable for program calculation.

In this paper we explore precisely this connection and develop a library of point-free lens combinators. In the next section, we establish that most of the standard point-free combinators define well-behaved lenses. This opens interesting perspectives towards a lens calculus, with practical applications for the optimization of complex lenses defined by composition. In the point-free style of programming, general recursion is usually deterred in favor of more calculation-friendly recursion patterns. In Section 3 we show how to define generic lenses over arbitrary inductive data types by lifting standard recursion patterns, namely folds and unfolds. In principle, this makes it simpler to establish that a lens is well-behaved, when compared to the general recursion approach followed by Foster *et al.* [15], where non-trivial conditions must be proved every time a lens is defined by recursion. In Section 4 we discuss some relevant related work, and we conclude in Section 5 with a synthesis of the main contributions and pointers for future work.

2 Point-free Combinators as Lenses

The rather standard set of point-free combinators that we will use in this paper is shown in Figure 1. Although we give our examples in Haskell, the semantic domain will be the Set category, where objects are sets (types) and arrows are total functions. The most fundamental combinators are the *composition* of $f : B \to C$ after $g : A \to B$, denoted by $f \circ g : A \to C$, and the *identity* function, denoted by $id : A \to A$. The *projections* $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ project out the left and right components of a pair, respectively, and the *split*

$$
\begin{array}{ll}
id & : A \to A \\
\circ & : (B \to C) \to (A \to B) \to (A \to C) \\
\pi_1 & : A \times B \to A \\
\pi_2 & : A \times B \to B \\
\vartriangle & :: (A \to B) \to (A \to C) \to (A \to B \times C) \\
\times & : (A \to B) \to (C \to D) \to (A \times C \to B \times D) \\
i_1 & : A \to A + B \\
i_2 & : B \to A + B \\
\nabla & : (A \to C) \to (B \to C) \to (A + B \to C) \\
+ & : (A \to B) \to (C \to D) \to (A + C \to B + D) \\
! & : A \to 1 \\
\underline{\ } & : B \to (A \to B)
\end{array}
$$

Fig. 1. Point-free combinators

combinator $f \vartriangle g : A \to B \times C$ builds a pair by applying $f : A \to B$ and $g : A \to C$ to the same input value. The derived *product* combinator $f \times g : A \times C \to B \times D$ applies $f : A \to B$ and $g : C \to D$ to the left and right elements, respectively, of the input pair in order to build a new pair. The *injections* $i_1 : A \to A + B$ and $i_2 : B \to A + B$ build left and right alternatives of a disjoint sum, respectively, and the *either* combinator $f \nabla g : A + B \to C$ applies $f : A \to C$ if the input is a left alternative or $g : B \to C$ otherwise. The derived *sum* combinator $f + g : A + C \to B + D$ uses $f : A \to B$ to build a left alternative from a left alternative or $g : C \to D$ otherwise. The *bang* combinator $! : A \to 1$ returns the single element of the terminal object, and, given a constant $b : B$, $\underline{b}$ always returns b. Some of the laws governing these combinators are presented in Appendix A, and can easily be derived from their uniqueness laws. For more information on point-free program calculation in general see [5,17,22,11].

Using these combinators, we can give a precise point-free characterization of well-behaved lenses.

Definition 1 (Lens). *A well-behaved lens l, denoted by $l : C \trianglerighteq A$, is a bidirectional transformation that comprises three total functions $get : C \to A$, $put : A \times C \to C$ and $create : A \to C$, satisfying the following properties:*

$$get \circ create = id \qquad \textsc{CreateGet}$$

$$get \circ put = \pi_1 \qquad \textsc{PutGet}$$

$$put \circ (get \vartriangle id) = id \qquad \textsc{GetPut}$$

Property CreateGet [6] guarantees that the lens is an abstraction, ensuring that *get* is a surjection and *create* an injection. PutGet and GetPut [15] guarantee that the lens is acceptable and stable, respectively.

We will now show how to lift most of the point-free combinators of Figure 1 to lenses. To avoid introducing new notation, we will denote the lens corresponding to a particular combinator using the same syntax. From the context it should be clear if we are referring to the lens or the point-free combinator. For some lenses there is some freedom in the design of backwards transformations (namely,

create and *put*). As such, they can receive extra parameters to plugin in contexts were such freedom exists.

The simplest cases of bidirectional transformations are isomorphisms. Given a bijective function $f : A \to B$ with inverse $f^{-1} : B \to A$, there exists a lens $f : A \trianglerighteq B$ with:

$$\begin{aligned} get &= f \\ put &= f^{-1} \circ \pi_1 \\ create &= f^{-1} \end{aligned}$$

A primitive combinator that falls under this category is the identity function $id : A \trianglerighteq A$. Similarly, all usual isomorphisms involving sums and products are lenses. Here are some that we will use throughout the paper:

$$\begin{aligned} swap &: A \times B \trianglerighteq B \times A \\ assocr &: (A \times B) \times C \trianglerighteq A \times (B \times C) \\ assocl &: (A \times (B \times C)) \trianglerighteq ((A \times B) \times C) \\ coassocl &: A + (B + C) \trianglerighteq (A + B) + C \\ distr &: A \times (B + C) \trianglerighteq (A \times B) + (A \times C) \\ distl &: (A + B) \times C \trianglerighteq (A \times C) + (B \times C) \end{aligned}$$

One of the most important properties of lenses is composability. In point-free, the composition of two lenses, first defined in [15], can be restated as follows:

$$\begin{aligned} &\forall f : B \trianglerighteq A,\ g : C \trianglerighteq B.\ \ (f \circ g) : C \trianglerighteq A \\ &\quad get = get_f \circ get_g \\ &\quad put = put_g \circ (put_f \circ (id \times get_g) \triangle \pi_2) \\ &\quad create = create_g \circ create_f \end{aligned}$$

If the concrete domain of f and the abstract domain of g have the same type, then f and g are composable and $f \circ g$ is a lens with the concrete domain of g and the abstract domain of f. In the *get* and *create* directions, the composed transformation is just the composition of the respective transformations from f and g. In the *put* direction, in order to apply the *put* functions in sequence, the original concrete value is duplicated. Note that, while put_g consumes the original concrete value with type C, the concrete value passed to put_f, with type B, is calculated by applying the function get_g to the original concrete value.

The projections π_1 and π_2 will be the main ingredients in defining more complex lenses that project away components of a concrete data type:

$$\begin{aligned} &\forall b \in B.\ {\pi_1}^b : A \times B \trianglerighteq A \\ &\quad get = \pi_1 \\ &\quad put = id \times \pi_2 \\ &\quad create = id \triangle \underline{b} \end{aligned} \qquad \begin{aligned} &\forall a \in A.\ {\pi_2}^a : A \times B \trianglerighteq B \\ &\quad get = \pi_2 \\ &\quad put = swap \circ (id \times \pi_1) \\ &\quad create = \underline{a} \triangle id \end{aligned}$$

Since π_1 and π_2 project the corresponding elements of the product, the backward transformations have to reconstruct the projected out elements. In *create*, a

default value is inserted for the "lost" value of the pair, while *put* copies it from the original pair. Therefore, the derived lenses accept additional parameters (the constants a and b), represented using superscript.

In general, the split of two lenses $f : C \trianglerighteq A$ and $g : C \trianglerighteq B$ sharing the same domain is not a well-behaved lens $f \triangle g : C \trianglerighteq A \times B$. For example, the duplication combinator $id \triangle id : A \trianglerighteq A \times A$ would be a valid lens iff the invariant $\pi_1 = \pi_2$ was imposed on the codomain $A \times A$, stating that both components of the pair are always equal. However, if f and g project distinct concrete information from C then it is possible to define a well-behaved lens. An example of such a lens is the *swap* isomorphism $\pi_2 \triangle \pi_1 : A \times B \trianglerighteq B \times A$. When this *non-interference* between f and g exists, updates to the view can be propagated back to the concrete model by independent inspection of both components of the pair. In practice this means that, when defining $put : (A \times B) \times C \to C$ the order of application of $put_f : A \times C \to C$ and $put_g : B \times C \to C$ should be irrelevant. Formally, this non-interference condition can be expressed by the following equality:

$$put_f \circ (id \times put_g) \circ assocr = put_g \circ (id \times put_f) \circ assocr \circ (swap \times id)$$

Given a split $f \triangle g$ where this non-interference condition is valid, it should be possible to lift it into a well-behaved lens by defining *put* as any of the above expressions (for example, $put_f \circ (id \times put_g) \circ assocr$). Unfortunately, we are unaware of a general definition for *create* that obeys the CreateGet law, which prevents us from giving a generic definition of split as a well-behaved lens. For *swap* it is rather easy to show that the non-interference condition is valid, that the suggested definition for *put* is equal to the expected $swap \circ \pi_1$ (according to the previous generic definition of a bijection as a well-behaved lens), and that *create* can be done using *swap* itself.

Another instance of split that satisfies the non-interference condition is the product combinator $f \times g = f \circ \pi_1 \triangle g \circ \pi_2$. Again, it is easy to show that any of the above alternative definitions for *put* is equivalent to $(put_f \times put_g) \circ distp$, where *distp* is the isomorphism given by:

$$distp : (C \times D) \times (A \times B) \to (C \times A) \times (D \times B)$$
$$distp = (\pi_1 \times \pi_1) \triangle (\pi_2 \times \pi_2) \qquad \text{Distp-Def}$$

For this particular split, creating a concrete value from an abstract one can be done by independently creating both components of the pair, leading to the following definition:

$$\begin{aligned} &\forall f : C \trianglerighteq A,\ g : D \trianglerighteq B.\ f \times g : C \times D \trianglerighteq A \times B \\ &\quad get \quad = get_f \times get_g \\ &\quad put \quad = (put_f \times put_g) \circ distp \\ &\quad create = create_f \times create_g \end{aligned}$$

In practice, most expressions involving split that satisfy non-interference can be transformed into point-free expressions using other valid lens product combinators and isomorphisms (like $\times$ or *swap*).

Moving to sums, we have two alternative ways to generically lift the either combinator into a well-behaved lens:

$$\begin{array}{l}\forall f : A \to C,\ g : B \to C.\ \ f \mathbin{\bullet\nabla} g : A + B \trianglerighteq C \\ \quad get \quad = get_f \nabla get_g \\ \quad put \quad = (put_f + put_g) \circ distr \\ \quad create = i_1 \circ create_f \\ \forall f : A \to C,\ g : B \to C.\ \ f \mathbin{\nabla\bullet} g : A + B \trianglerighteq C \\ \quad get \quad = get_f \nabla get_g \\ \quad put \quad = (put_f + put_g) \circ distr \\ \quad create = i_2 \circ create_g\end{array}$$

When putting back, put_f is used if the concrete value is a left alternative and put_g otherwise. For *create* we have two alternatives – either output a left or a right alternative – originating left-biased ($\bullet\nabla$) and right-biased ($\nabla\bullet$) versions of this lens. Assuming that predicates are represented using sums (for example, using $p : A \to A + A$ instead of $p : A \to Bool$), this lens corresponds to a point-free formulation of the concrete conditional combinator *ccond* from [15].

The sum injections $i_1 : A \to A + B$ and $i_2 : B \to A + B$ are non-surjective functions and classic examples of refinements [9]. The only way to lift them into lenses would be by imposing an invariant on the codomain $A + B$, constraining its values to be all left or all right alternatives, respectively. Since this semantic constraint is not supported by standard type systems, unrestricted usage of the injections will be disallowed for well-behaved lenses. Notwithstanding, if injections are used inside an expression that is jointly surjective, they can sometimes build up well-behaved lenses. Two particular useful cases are the lenses $i_1 \nabla f : A + C \trianglerighteq A + B$ and $f \nabla i_2 : C + B \trianglerighteq A + B$, where $f : C \to A + B$ is any lens. Notice that these eithers are necessarily surjective because f, being a well-behaved lens, is already surjective. For example, the first lens can be defined as follows:

$$\begin{array}{l}\forall f : C \trianglerighteq A + B.\ \ i_1 \nabla f : A + C \trianglerighteq A + B \\ \quad get \quad = i_1 \nabla get_f \\ \quad put \quad = ((id + create_f \circ i_2) \circ \pi_1 \nabla i_2 \circ put_f) \circ distr \\ \quad create = id + create_f \circ i_2\end{array}$$

For the sum combinator we can have the following lifting into a lens:

$$\begin{array}{l}\forall f : C \trianglerighteq A,\ g : D \trianglerighteq B.\ \ f + g : C + D \trianglerighteq A + B \\ \quad get \quad = get_f + get_g \\ \quad put \quad = (put_f \nabla create_f \circ \pi_1 + create_g \circ \pi_1 \nabla put_g) \circ dists \\ \quad create = create_f + create_g\end{array}$$

where *dists* is the following distribution combinator over sums:

$$\begin{array}{ll} dists : (A + B) \times (C + D) \to (A \times C + A \times D) + (B \times C + B \times D) & \\ dists = (distr + distr) \circ distl & \textsc{Dists-Def}\end{array}$$

In the definition of *put*, *dists* is first used to span the four possible cases. If the abstract and concrete values match (cases $A \times C$ and $B \times D$), then put_f and put_g are applied as expected. Otherwise (cases $A \times D$ and $B \times C$), we ignore the "out of sync" concrete values and use $create_f$ and $create_g$ to generate concrete values of the correct type. The definition of *create* is trivial and is merely the sum of the *create* functions of f and g. This sum combinator is essentially the point-free homologous of the abstract conditional combinator *acond* from [15].

Actually, $+$ can be lifted into a well-behaved lens in many different ways. Another alternative is the following, for arbitrary functions $h : A \times D \to C$ and $i : B \times C \to D$, although the first definition gives more natural results in most cases and will be used by default:

$$\begin{array}{l} \forall f : C \trianglerighteq A,\ g : D \trianglerighteq B.\ \ (f + g)^{h,i} : C + D \trianglerighteq A + B \\ \quad get \quad = get_f + get_g \\ \quad put \quad = (put_f + put_g) \circ (id \nabla (\pi_1 \triangle h) + (\pi_1 \triangle i) \nabla id) \circ dists \\ \quad create = create_f + create_g \end{array}$$

As with projections, in order to lift $! : C \to 1$ into a lens, a default value must be provided to be returned by the *create* function. The definition is trivial:

$$\begin{array}{l} \forall c \in C.\ !^c : C \trianglerighteq 1 \\ \quad get \quad = ! \\ \quad put \quad = \pi_2 \\ \quad create = \underline{c} \end{array}$$

Likewise to sum injections, the constant combinator $\underline{\cdot} : B \to (A \to B)$, that given a value $b \in B$ returns b for all input values, cannot be lifted into a well-behaved lens unless an invariant is imposed on the abstract type stating that all its values are equal to b. Again, there exist particular expressions in which this combinator forms a well-behaved lens, such as $\underline{b} \nabla f : A + C \trianglerighteq B$ or $f \nabla \underline{b} : C + A \trianglerighteq B$, where $f : C \trianglerighteq B$ is any other well-behaved lens.

3 Recursion Patterns as Lenses

In this section, we investigate recursive lenses over inductive data types. Most user defined data types can be defined as the fixpoint of a polynomial functor. Given a *base functor* F, the inductive type generated by its least fixpoint will be denoted by μF. A polynomial functor is either the identity functor Id (denoting recursive invocation), the constant functor $\underline{A}$ or the lifting of the sum $\oplus$ and product bifunctors $\otimes$. For example, for lists we have $[A] = \mu L_A$, where $L_A = \underline{1} \oplus \underline{A} \otimes Id$, and for naturals $Nat = \mu N$, where $N = \underline{1} \oplus Id$. Associated with each data type μF we also have two unique functions $in_F : F\ \mu F \to \mu F$ and $out_F : \mu F \to F\ \mu F$, that are each other's inverse. They allow us to encode and inspect values of the given type, respectively. The application of *out* to a type results on a one-level unfolding to a sum-of-products representation capable of being processed with point-free combinators.

Given a functor F and a function $f : A \to B$, the functor mapping $F\ f : F\ A \to F\ B$ is a function that preserves the functorial structure and modifies all the instances of the type argument A into instances of type B. It can be defined inductively on the functor F, such that the argument f is applied to the recursive occurrences inside the sums-of-products structure and constants are left unchanged:

$$\begin{aligned} F\ f &: F\ A \to F\ B \\ Id\ f &= f \\ \underline{T}\ f &= id \\ (F \otimes G)\ f &= F\ f \times G\ f \\ (F \oplus G)\ f &= F\ f + G\ f \end{aligned}$$

On the other side, a *natural transformation* η between functors F and G, denoted by $\eta : F \dot{\to} G$, is a function that transforms instances of F into instances of G while preserving the inner instances of the polymorphic type argument. It assigns to each type A an arrow $\eta_A : F\ A \to G\ A$ such that, for any function $f : A \to B$, the following naturality condition holds:

$$G\ f \circ \eta_A = \eta_B \circ F\ f \qquad \textsc{Nat-Swap}$$

Instead of defining lenses by general recursion, we resort to well-known recursion patterns, and use their powerful algebraic laws (see Appendix A) to prove that the resulting lenses are well-behaved. The most fundamental combinator is the fold or *catamorphism* that encodes the recursion pattern of iteration. Given an algebra $g : F\ A \to A$, the catamorphism $(\!|g|\!)_F : \mu F \to A$ is the unique function that makes the hereunder diagram commute:

$$\begin{array}{ccc} \mu F & \xrightarrow{out_F} & F\ \mu F \\ {\scriptstyle (\!|g|\!)_F}\downarrow & & \downarrow{\scriptstyle F\ (\!|g|\!)_F} \\ A & \xleftarrow[g]{} & F\ A \end{array}$$

A catamorphism recursively consumes a data type μF by replacing its constructors with the given algebra g. A well-known example is the $length : [A] \to Nat$ function presented in the introduction, that can be defined as the following catamorphism:

$$length = (\!|in_N \circ (id + \pi_2)|\!)_{L_A}$$

Another example of a catamorphism is the function $filter\_left : [A + B] \to [A]$ that filters all the left alternatives from a list of optional elements:

$$\begin{aligned} &filter\_left :: [Either\ a\ b] \to [a] \\ &filter\_left\ [\,] &&= [\,] \\ &filter\_left\ (Left\ x : xs) &&= x : filter\_left\ xs \\ &filter\_left\ (Right\ x : xs) &&= filter\_left\ xs \end{aligned}$$

Using the basic isomorphisms presented before, it is not difficult to put together a point-free algebra with the intended behaviour:

$$filter\_left = (\!|(in_{L_A} \nabla \pi_2) \circ coassocl \circ (id + distl)|\!)_{L_{A+B}}$$

The dual recursion pattern of catamorphism is the unfold or *anamorphism*. Given a coalgebra $h : A \to F\ A$, the anamorphism $[\!(h)\!]_F : A \to \nu F$ is a function that, given an element of A, builds a (possibly infinite) element of the coinductive datatype νF (the greatest fixpoint of F). The coalgebra h is used to decide when generation stops and, in case it proceeds, which "seeds" should be used to generate the recursive occurrences of νF. Here we will only be interested in a specific kind of unfolds, namely those that always terminate. Not only we want all our lenses to terminate, but we also want to be able to freely compose them with catamorphisms - this composition is not always well-defined because anamorphisms can generate infinite values that are not part of the least fixed point consumed by catamorphisms. If we restrict ourselves to *recursive* [8] (or *reductive* [1]) coalgebras, the resulting morphism (to be denoted by *recursive anamorphism*) is guaranteed to halt in finitely many steps. A recursive coalgebra $h : A \to F\ A$ is essentially one that guarantees that all As contained in the resulting $F\ A$ are somehow smaller than its input. Capretta *et al.* [8] give a nice formal definition and provide a set of constructions for building recursive coalgebras out of simpler ones. Since $out_F : \mu F \to F\ \mu F$ is a final recursive coalgebra, we can safely compose catamorphisms with recursive anamorphisms. Given a recursive coalgebra $h : A \to F\ A$, the recursive anamorphism $[\![h]\!]_F : A \to \mu F$ is the unique function that makes the hereunder diagram commute:

$$\begin{array}{ccc} \mu F & \xleftarrow{in_F} & F\ \mu F \\ {\scriptstyle [\![h]\!]_F}\uparrow & & \uparrow{\scriptstyle F\ [\![h]\!]_F} \\ A & \xrightarrow[h]{} & F\ A \end{array}$$

Given this uniqueness property, the recursive anamorphism obeys the same laws as the normal anamorphism, namely fusion. A trivial example of a recursive anamorphism to naturals is again the length function:

$$length = [\![(id + \pi_2) \circ out_{L_A}]\!]_N$$

Notice that $id + \pi_2 : L_A \dot{\to} N$ and that a composition of a natural transformation with a recursive coalgebra is again a recursive coalgebra [8, Proposition 3.9], so this is clearly a recursive anamorphism. In fact, every catamorphism $(\!|in_G \circ \eta|\!)_F$, where $\eta : F \dot{\to} G$ is a natural transformation can also be defined by a recursive anamorphism $[\![\eta \circ out_F]\!]_G$, and vice-versa. A classical example of a recursive anamorphism that cannot be defined using a catamorphism is the function $zip : [A] \times [B] \to [A \times B]$, that zips two lists together into a list of pairs:

$$
\begin{aligned}
&zip :: ([a], [b]) \to [(a, b)] \\
&zip\ (x : xs, y : ys) = (x, y) : zip\ (xs, ys) \\
&zip\ \_ \qquad\qquad\ = [\,]
\end{aligned}
$$

In the point-free style it can be redefined as follows:

$$zip = [\![(! + distp) \circ coassocl \circ dists \circ (out_{L_A} \times out_{L_B})]\!]_{L_{A \times B}}$$

The coalgebra guarantees that the output list stops being generated when at least one of the inputs is empty. Otherwise, both tails are used as "seed" to recursively generate the tail of the output list.

The composition of a catamorphism after an anamorphism is known as *hylomorphism*, but as mentioned above, this composition is not always well-defined in SET. Here, we will be interested in hylomorphisms that are guaranteed to terminate, namely those where the cata is composed with a recursive anamorphism:

$$[\![g, h]\!]_F = (\!|g|\!)_F \circ [\![h]\!]_F \qquad \text{HYLO-SPLIT}$$

These *recursive hylomorphisms* (the unique *coalgebra-to-algebra morphisms* of [8]) are quite amenable to program calculation because they enjoy a uniqueness law similar to the other recursion patterns:

$$[\![g, h]\!]_F = f \Leftrightarrow g \circ F\ f \circ h = f \qquad \text{HYLO-UNIQ}$$

3.1 Functor Mapping

We can lift functor mapping into a lens combinator by applying regular functor mapping to each component of a lens, as follows:

$$
\begin{aligned}
&\forall f : C \trianglerighteq A.\ \ F\ f : F\ C \trianglerighteq F\ A \\
&get \quad\ = F\ get_f \\
&put \quad\ = F\ put_f \circ fzip_F\ create_f \\
&create = F\ create_f
\end{aligned}
$$

The interesting snippet is the $fzip_F$ combinator, responsible for zipping abstract and concrete instances of the same F-structure, and that is defined below:

$$
\begin{aligned}
&fzip_F : (A \to C) \to F\ A \times F\ C \to F\ (A \times C) \\
&fzip_{Id}\ f = id \\
&fzip_{\underline{T}}\ f = \pi_1 \\
&fzip_{(F \otimes G)}\ f = (fzip_F\ f \times fzip_G\ f) \circ distp \\
&fzip_{(F \oplus G)}\ f = (fzip_F\ f \nabla F\ (id \triangle f) \circ \pi_1 + G\ (id \triangle f) \circ \pi_1 \nabla fzip_G\ f) \circ dists
\end{aligned}
$$

As usual, *fzip* gives preference to the values from the abstract data type. In the case of sums (similarly to the definition of the + lens), *fzip* is applied recursively to the sub-functors F and G, and whenever the abstract and concrete values are "out of sync", the abstract value is preserved and a new concrete value is created from the abstract value, by invoking the argument function.

We can polytypically prove (in the style of [18]) the following laws about *fzip*:

$$F\ \pi_1 \circ fzip_F\ f = \pi_1 \qquad \textsc{Fzip-Cancel}$$
$$fzip_F\ f \circ (F\ g \vartriangle F\ h) = F\ (g \vartriangle h) \qquad \textsc{Fzip-Split}$$

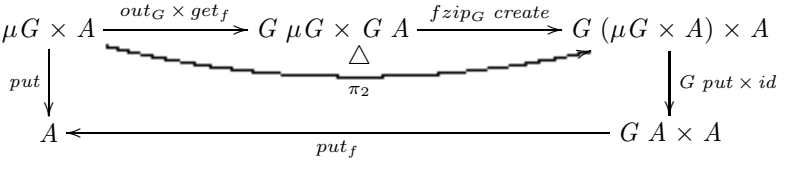
The first states that $fzip_F$ cannot modify the shape of the abstract type, nor the data contained in it. The second states that zipping two "in sync" values can be trivially done just by mapping. The proof of the first property can be found in Appendix B. The other proof is similar and is omitted.

3.2 Anamorphism

At this point, we have enough ingredients to "lensify" anamorphisms. For the resulting lens to be well-behaved, the coalgebra must be recursive and itself a well-behaved lens. The generic definition is as follows:

$$\begin{aligned}
&\forall f : A \trianglerighteq G\ A.\ \ [\![f]\!]_G : A \trianglerighteq \mu G \\
&\quad get\ \ = [\![get_f]\!]_G \\
&\quad put\ \ = [\![put_f, fzip_G\ create \circ (out_G \times get_f) \vartriangle \pi_2]\!]_{G \otimes \underline{A}} \\
&\quad create = (\!|create_f|\!)_G
\end{aligned}$$

Knowing that $create_f$ is an algebra with type $G\ A \to A$, *create* is trivially defined using a catamorphism. The generic definition of *put* uses an accumulation technique implemented as a recursive hylomorphism: it proceeds inductively over the abstract value, using the concrete value as an accumulator. The function that propagates the accumulator to recursive calls is $fzip_G\ create \circ (out_G \times get_f)$. The diagram for this hylomorphism is the following:



The proof that this lens is well-behaved is given in Appendix B. The proof of laws CreateGet and PutGet can be done using the fusion law for anamorphisms. The proof of law GetPut uses hylomorphism fusion and Hylo-Uniq.

By applying this definition to the *zip* function, we get the expected definitions for *create* and *put*. For better understanding, we present them using Haskell syntax and explicit recursion (easily derivable from the original point-free definition):

$$\begin{aligned}
&create :: [(a, b)] \to ([a], [b]) \\
&create\ [\,] \qquad\quad = ([\,], [\,]) \\
&create\ ((x, y) : t) = \mathbf{let}\ (xs, ys) = create\ t\ \mathbf{in}\ (x : xs, y : ys)
\end{aligned}$$

$$
\begin{aligned}
&put :: ([(a,b)],([a],[b])) \to ([a],[b])\\
&put\ ([],([],r)) &&= ([],r)\\
&put\ ([],(l,[])) &&= (l,[])\\
&put\ ((x,y):t,(\_:l,\_:r)) &&= \mathbf{let}\ (xs,ys) = put\ (t,(l,r))\ \mathbf{in}\ (x:xs,y:ys)\\
&put\ (l,\_) &&= create\ l
\end{aligned}
$$

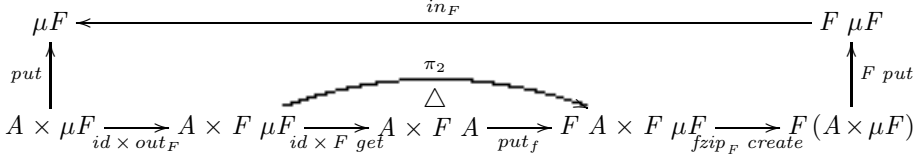
The *create* induced by this lens is just the *unzip* function, a fold that recursively splits a list of pairs into two lists. The *put* has a more intricate behaviour: it only recovers elements of one of the original concrete lists when the updated abstract list is smaller than it but with the exact same length of the other concrete list. This guarantees that zipping the result again yields the same view. For example, $put\ ([(1,2),(3,4)],([4,5],[6,7,8,9]))$ returns $([1,3],[2,4,8,9])$. Notice how the elements 8 and 9 of the bigger list are recovered.

3.3 Catamorphism

Catamorphisms can be lifted to well-behaved lenses as follows:

$$
\begin{aligned}
&\forall f : F\ A \trianglerighteq A.\ \ (\!|f|\!)_F : \mu F \trianglerighteq A\\
&\quad get &&= (\!|get_f|\!)_F\\
&\quad put &&= [\!|fzip_F\ create \circ (put_f \circ (id \times G\ get) \vartriangle \pi_2) \circ (id \times out_F)|\!]_F\\
&\quad create &&= [\!|create_f|\!]_F
\end{aligned}
$$

Notice how *put*, encoded as an anamorphism, still gives preference to abstract values by using *fzip*, as depicted in the following diagram:



To prove that $(\!|f|\!)$ is a well-behaved lens, we must prove that both *put* and *create* are recursive anamorphisms. Given these conditions, the proof is very similar to the proof for anamorphisms and is omitted.

The *filter_left* function, using the left-biased version of the either combinator, is an example of a fold lens where it is not difficult to prove that both the *create* and *put* are indeed recursive. Its corresponding *create* is a simple unfold that maps a list into a list of left alternatives:

$$
\begin{aligned}
&create :: [a] \to [Either\ a\ b]\\
&create\ [] &&= []\\
&create\ (x:xs) &&= Left\ x : create\ xs
\end{aligned}
$$

The *put* function restores right alternatives from the original concrete list:

$$
\begin{aligned}
&put :: ([a], [Either\ a\ b]) \to [Either\ a\ b] \\
&put\ (xs, Right\ y : ys) &&= Right\ y : put\ (xs, ys) \\
&put\ ([\,], \_) &&= [\,] \\
&put\ (x : xs, [\,]) &&= Left\ x : put\ (xs, create\ xs) \\
&put\ (x : xs, Left\ y : ys) &&= Left\ x : put\ (xs, ys)
\end{aligned}
$$

For an example of a catamorphism lens whose *create* function is not recursive, just replace the left-biased either combinator by the right-biased version in the point-free definition presented above. This change yields the following reshaping of *create*, assuming b to be the default constant that parameterizes π_2:

$$
\begin{aligned}
&create :: [a] \to [Either\ a\ b] \\
&create\ xs = Right\ b : create\ xs
\end{aligned}
$$

3.4 Natural Transformations

A special case of the previous lenses occurs when the forward transformation is both expressible as a catamorphism and an anamorphism with the same natural transformation in the recursive gene. We name a lens f describing a bidirectional natural transformation between functors F and G a *natural lens* and type it with the signature $f : F \dot{\trianglerighteq} G$, where $get : F \dot{\to} G$, $put : G \otimes F \dot{\to} F$ and $create : G \dot{\to} F$. Unlike the previous cases, where we still have to check that the coalgebras are recursive, given a natural lens $\eta : F \dot{\trianglerighteq} G$, both $(\!|in_G \circ \eta|\!)_F$ and $[\!(\eta \circ out_F)\!]_G$ immediately determine well-behaved lenses between μF and μG because, as mentioned before, termination is guaranteed.

There are several examples of these lenses. As seen before, the *length* function is a well-known example that can be expressed either as a catamorphism on lists or an anamorphism to naturals. Instantiating the input type to lists of naturals and the default constant that parameterizes π_2 to *Zero*, the forward and backward functions induced by this lens are exactly the same as the ones presented in the introduction. Another lens that establishes a natural transformation between base functors is mapping over lists:

$$\forall f : C \trianglerighteq A.\ \ map\ f = (\!|in_{L_A} \circ (id + f \times id)|\!)_{L_C} : [\,C\,] \trianglerighteq [\,A\,]$$

This definition can be generalized for any parametric type D defined inductively over a bifunctor B:

$$\forall f : C \trianglerighteq A.\ \ gmap\ f = (\!|in_{B\ A} \circ B\ f\ id|\!)_{B\ C} : D\ C \trianglerighteq D\ A$$

3.5 Hylomorphisms

It is well known that most recursive functions can be encoded using hylomorphisms over polynomial functors. Given that Hylo-Split allows us to factorize a hylomorphism into the composition of a catamorphism after an anamorphism,

the range of recursive functions that we can lift to well-behaved lenses is considerably enlarged. Of course the algebras and coalgebras of the hylomorphism must themselves be lenses (for example, built using the combinators presented in Section 2), and the coalgebras must be recursive.

Take as an example the natural number addition function $plus : Nat \times Nat \to Nat$:

$$\begin{aligned} &plus :: (Nat, Nat) \to Nat \\ &plus\ (Zero, m) \quad = m \\ &plus\ (Succ\ n, m) = Succ\ (plus\ (n, m)) \end{aligned}$$

Although it is not a fold neither an unfold, it can be defined as the following hylomorphism, where both the algebra and the recursive coalgebra are lenses:

$$plus = [\![in_N \circ (out_N \nabla\, i_2), (\pi_2 + id) \circ distl \circ (out_N \times id)]\!]_{\underline{Nat} \oplus Id}$$

In order to lift this function into a well-behaved lens, the *create* and *put* functions should guarantee that the sum of the generated pair of numbers equals the abstract value. The *create* automatically induced by the techniques presented above simply creates a pair with the abstract value and a *Zero* as the second element:

$$\begin{aligned} &create :: Nat \to (Nat, Nat) \\ &create\ n = (n, Zero) \end{aligned}$$

As usual, the induced *put* function is a bit more tricky: if the abstract value is greater than the first element of the concrete pair, that element is preserved and the second element becomes the difference between both; if the abstract value is smaller it just pairs, it with zero likewise to *create*:

$$\begin{aligned} &put :: (Nat, (Nat, Nat)) \to (Nat, Nat) \\ &put\ (Zero, \_) \qquad\qquad\quad = (Zero, Zero) \\ &put\ (n, (Zero, o)) \qquad\quad = \mathbf{let}\ (a, b) = create\ n\ \mathbf{in}\ (b, a) \\ &put\ (Succ\ n, (Succ\ m, o)) = \mathbf{let}\ (a, b) = put\ (n, (m, o))\ \mathbf{in}\ (Succ\ a, b) \end{aligned}$$

4 Related Work

Our point-free framework can be seen as a domain-specific language similar to the language for lenses over trees first developed by Foster *et al.* [15]. While our generic combinators rely on the sum-of-products representation of inductive types, they represent all recursive types as generalized trees. They also devise a complex set-based type system with invariants to precisely define the domains for which their combinators are well-behaved. Using such *semantic* types [16], they are able to define well-behaved lenses like $\underline{c}$, for an arbitrary constant c. We rely in a *syntactic* and more standard (and decidable) type system that is implemented in functional programming languages like Haskell or ML, with the

counterpart that we have a more limited set of well-behaved lenses. Additionally, we identify precise termination conditions to verify in order to guarantee that recursive lenses like folds and unfolds constitute well-behaved lenses. We believe that using the techniques presented in [1,8] these conditions are easier to verify than the conditions stated in [15] concerning general recursion.

On a more algebraic tone, researchers from the University of Tokyo have studied the automatic inversion of forward transformations defined in a point-free language of injective functions, to be used in a structure XML editor supporting views [24]. The idea is to move the burden of information preservation from *put* to *get* as to make $put : A \to C$ stateless and $get : C \to A$ injective. In practice this setting resembles that of data refinement, as attested by the required $put \circ get = id$ property. In order to deal with duplication and structural changes, editing tags are introduced in the domain of *put*. In the case of data duplication, if only one element of the resulting pair is updated (and thus marked with an edit tag), a view-to-view roundtrip is then able to propagate the modification to the other element and restore the invariant in the view. In addition, a weaker version of PutGet, baptized PutGetPut ($put \circ get \circ put \sqsubseteq put$), is required, to ensure that, when editing a view, applying *put* to update the source and computing the new view with *get* is sufficient to synchronize all the changes. The preorder $\sqsubseteq$ reflects the partiality of *put*, given that the domain of *get* may be larger than the range of *put*.

A follow-up work approached the automatic derivation of backward transformations based on a notion of view-update under *constant complement* [20]. Instead of assuming forward injective functions, they now take any $get : C \to A$ and derive an explicit complement function $get^c : C \to H$, such that the tupled function $get \vartriangle get^c : C \to A \times H$ is injective. The *put* function is then calculated from the specification $(get \vartriangle get^c)^{-1} \circ (id \times get^c)$. The bidirectional properties follow those of *closed* view-updating [3], where the source is hidden from the users when the view is updated. Besides the fundamental stability condition, there are undoability and composability conditions that, as shown by Diskin [14], yield precisely the *very well-behaved lenses* first presented in [15]. However, these additional properties are defined modulo partiality of *put* since, according to the constant complement approach, *put* should forbid any changes to the information that the complement has kept. For instance, inserting and removing elements are forbidden updates in their running example of a filtering lens.

To avoid restricting the syntax of the forward transformations, Voigtländer allows normal Haskell functions to be used in lens definitions [27]. In this scenario, reasonable backward transformations can be derived by observing the runtime behaviour of the forward transformations. A higher-order bidirectionalizer *bff* is defined that receives a polymorphic *get* function and returns the corresponding *put* function, ensuring bidirectional properties similar to [20]. Although, for example, the mapping lens is not definable in this framework, there are some lenses supported by *bff* that are not expressible with our combinators, namely polymorphic functions that duplicate information. However, likewise to [20], it is debatable how much bidirectionalization is truly achieved, concerning the degree

of partiality of the backward transformation. For example, in this framework the *put* function of the *length* lens would not try to synchronize updates that change the shape of the abstract view, and thus would only be defined for the cases when the length of the original list remains constant.

Wang *et al.* [28] propose a language of right-invertible point-free combinators denoting total transformations, in order to define a *view* mechanism on datatypes that enables sound equational reasoning at the view level. However, they only consider pure abstractions (i.e, without a *put* function that takes into account old concrete values), and the chosen right-inverses of most of their combinators essentially coincide with the *create* functions of our lens combinators. Since their language also includes non-surjective datatype constructors as primitives, an additional compile-time check is required to test the joint surjectivity of programs involving constructors. The inclusion of a fold combinator also raises concerns regarding the termination of anamorphisms as right-inverses, and, likewise to our approach, additional constraints on the coalgebras must be checked.

In previous work, we have proposed a two-level bidirectional transformation framework (2LT) for *data refinement* [9,4], where forwards and backwards transformations were also specified in the point-free style, and type-safeness of the value migration functions was ensured with a deep embedding in Haskell. Later, we have shown how point-free program calculation could be used for the optimization of large compositions of bidirectional transformations and structure-shy query migration from the source to the target types [12,13]. In this paper, we tackle the dual problem of abstraction, using similar techniques to define generic point-free lenses: we intend to incorporate them into the 2LT framework, in order to enlarge the scope of model transformation scenarios to which it can be applied, and benefit from the optimization strategies previously implemented.

5 Conclusion

In this paper we have shown how to lift most of the standard point-free combinators and recursion patterns to well-behaved lenses. This enables the definition of elegant, generic, and, hopefully, intuitive lenses over inductive data types. Concerning recursion, we have identified precise termination conditions that allow folds and unfolds to be lifted to well-behaved lenses. Notice that we can also tackle arbitrary recursive lenses by expressing them as hylomorphisms, i.e the composition of a fold after an unfold. Using the techniques described in [10], we have also implemented a Haskell library, with the combinators presented in this paper and some more, to aid the construction of functional bidirectional programs by composition. The library is extensible, by supporting user-defined lens combinators, and is available through the Hackage package repository (`http://hackage.haskell.org`) under the name `pointless-lenses`, honoring a common joke about point-free programming.

Complex lens transformations suffer from degraded performance due to the cluttering of intermediate structures originated from the combination of smaller transformations. In the short run, we intend to apply the techniques developed

for point-free refinement optimization [12,13] to the optimization of complex lenses defined by composition. Oliveira [25] already showed that a relational point-free calculus can be a more natural setting to formalize bidirectional transformations. This relational calculus can provide several advantages, such as reasoning about termination, computing inverses of arbitrary transformations, and expressing structural invariants over data-types. The latter is of utmost importance to statically calculate the domain on which a put function is well-defined, thus widening the set of potential well-behaved lenses to combinators like split or the injections.

Acknowledgments

We would like to thank the anonymous referees for the many insightful comments, and José Bacelar Almeida for the helpful discussions concerning termination.

References

1. Backhouse, R., Doornbos, H.: Mathematics of recursive program construction. Manuscript (2001), `http://www.cs.nott.ac.uk/rcb/MPC/papers`
2. Backus, J.: Can programming be liberated from the von Neumann style? a functional style and its algebra of programs. Communications of the ACM 21(8), 613–641 (1978)
3. Bancilhon, F., Spyratos, N.: Update semantics of relational views. ACM Transactions on Database Systems 6(4), 557–575 (1981)
4. Berdaguer, P., Cunha, A., Pacheco, H., Visser, J.: Coupled schema transformation and data: Conversion for XML and SQL. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 290–304. Springer, Heidelberg (2006)
5. Bird, R., de Moor, O.: The Algebra of Programming. Prentice-Hall, Englewood Cliffs (1997)
6. Bohannon, A., Foster, J.N., Pierce, B.C., Pilkiewicz, A., Schmitt, A.: Boomerang: resourceful lenses for string data. In: Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08), pp. 407–419. ACM, New York (2008)
7. Bohannon, A., Pierce, B.C., Vaughan, J.A.: Relational lenses: a language for updatable views. In: Proceedings of the 25th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS'06), pp. 338–347. ACM, New York (2006)
8. Capretta, V., Uustalu, T., Vene, V.: Recursive coalgebras from comonads. Information and Computation 204(4), 437–468 (2006)
9. Cunha, A., Oliveira, J.N., Visser, J.: Type-safe two-level data transformation. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 284–299. Springer, Heidelberg (2006)
10. Cunha, A., Pacheco, H.: Algebraic specialization of generic functions for recursive types. In: Proceedings of the 2nd Workshop on Mathematically Structured Functional Programming (MSFP'08). ENTCS, Elsevier Science Publishers B. V., Amsterdam (2008)

11. Cunha, A., Pinto, J.S., Proença, J.: A framework for point-free program transformation. In: Butterfield, A., Grelck, C., Huch, F. (eds.) IFL 2005. LNCS, vol. 4015, pp. 1–18. Springer, Heidelberg (2006)
12. Cunha, A., Visser, J.: Strongly typed rewriting for coupled software transformation. Electronic Notes in Theoretical Computer Science 174(1), 17–34 (2007)
13. Cunha, A., Visser, J.: Transformation of structure-shy programs with application to XPath queries and strategic functions. Science of Computer Programming (to appear, 2010)
14. Diskin, Z.: Algebraic Models for Bidirectional Model Synchronization. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 21–36. Springer, Heidelberg (2008)
15. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM Transactions on Programming Languages and Systems 29(3), 17 (2007)
16. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. Journal of the ACM 55(4), 1–64 (2008)
17. Gibbons, J.: Calculating functional programs. In: Blackhouse, R., Crole, R.L., Gibbons, J. (eds.) Algebraic and Coalgebraic Methods in the Mathematics of Program Construction. LNCS, vol. 2297, pp. 149–203. Springer, Heidelberg (2002)
18. Hinze, R.: Generic programs and proofs. Bonn University, Habilitation (2000)
19. Hu, Z., Mu, S.-C., Takeichi, M.: A programmable editor for developing structured documents based on bidirectional transformations. Higher Order and Symbolic Computation 21(1-2), 89–118 (2008)
20. Matsuda, K., Hu, Z., Nakano, K., Hamana, M., Takeichi, M.: Bidirectionalization transformation based on automatic derivation of view complement functions. In: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07), pp. 47–58. ACM, New York (2007)
21. Meertens, L.: Designing constraint maintainers for user interaction (1998), Manuscript available at, `http://www.kestrel.edu/home/people/meertens`
22. Meijer, E., Fokkinga, M., Paterson, R.: Functional programming with bananas, lenses, envelopes and barbed wire. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 124–144. Springer, Heidelberg (1991)
23. Morgan, C., Gardiner, P.H.B.: Data refinement by calculation. Acta Informatica 27(6), 481–503 (1990)
24. Mu, S.-C., Hu, Z., Takeichi, M.: An algebraic approach to bi-directional updating. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 2–20. Springer, Heidelberg (2004)
25. Oliveira, J.N.: Data transformation by calculation. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) Generative and Transformational Techniques in Software Engineering II. LNCS, vol. 5235, pp. 134–195. Springer, Heidelberg (2008)
26. Stevens, P.: Bidirectional model transformations in QVT: Semantic issues and open questions. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 1–15. Springer, Heidelberg (2007)
27. Voigtländer, J.: Bidirectionalization for free! (Pearl). In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'09), pp. 165–176. ACM, New York (2009)
28. Wang, M., Gibbons, J., Matsuda, K., Hu, Z.: Translucent Abstraction: Safe Views through Invertible Programming. Draft (2010), `http://web.comlab.ox.ac.uk/files/2280/total.pdf`

A Point-free Laws

$$f \circ id = f \wedge id \circ f = f \qquad \text{Id-Nat}$$

$$f \circ (g \circ h) = (f \circ g) \circ h \qquad \circ\text{-Assoc}$$

$$f \circ h = g \circ h \Leftarrow f = g \qquad \text{Leibniz}$$

$$\pi_1 \vartriangle \pi_2 = id \qquad \times\text{-Reflex}$$

$$\pi_1 \circ (f \vartriangle g) = f \wedge \pi_2 \circ (f \vartriangle g) = g \qquad \times\text{-Cancel}$$

$$(f \vartriangle g) \circ h = f \circ h \vartriangle g \circ h \qquad \times\text{-Fusion}$$

$$(f \times g) \circ (h \vartriangle i) = f \circ h \vartriangle g \circ i \qquad \times\text{-Absor}$$

$$(\pi_1 \triangledown \pi_1) \circ distr = \pi_1 \qquad \text{Distr-Fst}$$

$$(\pi_1 + \pi_1) \circ distl = \pi_1 \qquad \text{Distl-Fst}$$

$$(f \times g) \circ (h \times i) = f \circ h \times g \circ i \qquad \times\text{-Functor-Comp}$$

$$f \circ (g \triangledown h) = f \circ g \triangledown f \circ h \qquad +\text{-Absor}$$

$$(f + g) \circ (h + i) = f \circ h + g \circ i \qquad +\text{-Functor-Comp}$$

$$in_F \circ out_F = id_{\mu F} \wedge out_F \circ in_F = id_{F\ \mu F} \qquad \text{In-Out-Iso}$$

$$F\ f \circ F\ g = F\ (f \circ g) \qquad \text{Functor-Comp}$$

$$F\ id = id \qquad \text{Functor-Id}$$

$$(\!|g|\!)_F \circ in_F = g \circ F\ (\!|g|\!)_F \qquad \text{Cata-Cancel}$$

$$f \circ (\!|g|\!)_F = (\!|h|\!)_F \Leftarrow f \circ g = h \circ F\ f \qquad \text{Cata-Fusion}$$

$$[\!|out_F|\!]_F = id \qquad \text{Ana-Reflex}$$

$$out_F \circ [\!|h|\!]_F = F\ [\!|h|\!]_F \circ h \qquad \text{Ana-Cancel}$$

$$[\!|g|\!]_F \circ f = [\!|h|\!]_{\mu F} \Leftarrow g \circ f = F\ f \circ h \qquad \text{Ana-Fusion}$$

$$[\!|g, h|\!]_F = g \circ F\ [\!|g, h|\!]_F \circ h \qquad \text{Hylo-Cancel}$$

$$g \circ [\!|h, i|\!]_F \circ j = [\!|k, l|\!]_F \Leftarrow g \circ h = k \circ F\ g\ \wedge\ i \circ j = F\ j \circ l \qquad \text{Hylo-Fusion}$$

B Proofs

Proof ($[\!|f|\!]_G$ is a well-behaved lens).

$get \circ create = id$
$\Leftrightarrow$ {definition of get; Ana-Reflex}
$[\!|get_f|\!]_G \circ create = [\!|out_G|\!]_G$
$\Leftarrow$ {Ana-Fusion}
$get_f \circ create = G\ create \circ out_G$
$\Leftrightarrow$ {definition of $create$; Cata-Cancel}
$get_f \circ create_f \circ G\ create \circ out_G = G\ create \circ out_G$
$\Leftarrow$ {Leibniz}
$get_f \circ create_f = id$
$\Leftrightarrow$ {CreateGet}
TRUE

$get \circ put$
$=$ {definition of get}

$[\![get_f]\!]_G \circ put$
$= \{$ANA-FUSION; definition of $put\}$
$\quad get_f \circ [\![put_f, fzip_G\ create \circ (out_G \times get_f) \triangle \pi_2]\!]_{G \otimes \underline{A}} = G\ put \circ h$
$\quad \Leftrightarrow \{$HYLO-CANCEL$\}$
$\quad get_f \circ put_f \circ (G \otimes \underline{A})\ put \circ (fzip_G\ create \circ (out_G \times get_f) \triangle \pi_2) = G\ put \circ h$
$\quad \Leftrightarrow \{$PUTGET; $\times$-FUNCTOR-COMP$\}$
$\quad \pi_1 \circ (G\ put \circ fzip_G\ create \circ (out_G \times get_f) \triangle \pi_2)$
$\quad \Leftrightarrow \{\times$-CANCEL$\}$
$\quad G\ put \circ fzip_G\ create \circ (out_G \times get_f) = G\ put \circ fzip_G\ create \circ (out_G \times get_f)$
$[\![fzip_G\ create \circ (out_G \times get_f)]\!]_G$
$= \{$ANA-FUSION$\}$
$\quad G\ \pi_1 \circ fzip_G\ create \circ (out_G \times get_f) = out_G \circ \pi_1$
$\quad \Leftrightarrow \{$FZIP-CANCEL; $\times$-CANCEL$\}$
$\quad out_G \circ \pi_1 = out_G \circ \pi_1$
$[\![out_G]\!]_G \circ \pi_1$
$= \{$ANA-REFLEX$\}$
π_1

$put \circ (get \triangle id)$
$= \{$definition of $put\}$
$[\![put_f, (fzip_G\ create \circ (id \times get_f) \triangle \pi_2) \circ (out_G \times id)]\!]_{G \otimes \underline{A}} \circ (get \triangle id)$
$= \{$HYLO-FUSION$\}$
$\quad (fzip_G\ create \circ (id \times get_f) \triangle \pi_2) \circ (out_G \times id) \circ (get \triangle id)$
$\quad = (G \otimes \underline{A})\ (get \triangle id) \circ (get_f \triangle id)$
$\quad \Leftrightarrow \{\times$-ABSOR; $\times$-ABSOR; $\times$-CANCEL$\}$
$\quad (fzip_G\ create \circ (out_G \circ get \triangle get_f) \triangle id)$
$\quad = (G\ (get \triangle id) \times id) \circ (get_f \triangle id)$
$\quad \Leftrightarrow \{$ANA-CANCEL; $\times$-FUSION$\}$
$\quad (fzip_G\ create \circ (G\ get \triangle id) \circ get_f \triangle id)$
$\quad = (G\ (get \triangle id) \times id) \circ (get_f \triangle id)$
$\quad \Leftrightarrow \{$FZIP-SPLIT; $\times$-ABSOR$\}$
$\quad (G\ (get \triangle id) \circ get \triangle id) = (G\ (get \triangle id) \circ get_f \triangle id)$
$[\![put_f, get_f \triangle id]\!]_{G \otimes \underline{A}} = id$
$= \{$HYLO-UNIQ; GETPUT$\}$
id

*Proof (*FZIP-CANCEL*).*

$Id\ \pi_1 \circ fzip_{Id}\ f$
$= \{$FZIP-DEF; ID-NAT$\}$
π_1

$\underline{T}\ \pi_1 \circ fzip_{\underline{T}}\ f$
$= \{$FZIP-DEF; ID-NAT$\}$
π_1

$(F \otimes G)\ \pi_1 \circ fzip_{F \otimes G}\ f$
$= \{$FZIP-DEF$\}$

$$
\begin{array}{l}
(F\ \pi_1 \times G\ \pi_1) \circ (fzip_{\,F}\ f \times fzip_{\,G}\ f) \circ distp \\
= \{ \times\text{-Functor-Comp; Fzip-Cancel; Fzip-Cancel} \} \\
(\pi_1 \times \pi_1) \circ distp \\
= \{ \text{Distp-Def; } \times\text{-Absor; } \times\text{-Cancel; } \times\text{-Cancel} \} \\
\pi_1 \circ \pi_1 \triangle \pi_2 \circ \pi_1 \\
= \{ \times\text{-Fusion; } \times\text{-Reflex; Id-Nat} \} \\
\pi_1
\end{array}
$$

$$
\begin{array}{l}
(F \oplus G)\ \pi_1 \circ fzip_{\,F \oplus G}\ f \\
= \{ \text{Fzip-Def} \} \\
(F\ \pi_1 + G\ \pi_1) \circ (fzip_F\ f \nabla F\ (id \triangle f) \circ \pi_1 + G\ (id \triangle f) \circ \pi_1 \nabla fzip_G\ f) \\
\circ dists \\
= \{ +\text{-Functor-Comp; } +\text{-Absor; } +\text{-Absor} \} \\
(F\ \pi_1 \circ fzip_F\ f \nabla F\ \pi_1 \circ F\ (id \triangle f) \circ \pi_1 + G\ \pi_1 \circ G\ (id \triangle f) \circ \pi_1 \nabla G\ \pi_1 \\
\circ fzip_G) \circ dists \\
= \{ \text{Fzip-Cancel; Fzip-Cancel; Functor-Comp; Functor-Comp} \} \\
(\pi_1 \nabla F\ (\pi_1 \circ (id \triangle f)) \circ \pi_1 + G\ (\pi_1 \circ (id \triangle f)) \circ \pi_1 \nabla \pi_1) \circ dists \\
= \{ \text{Functor-Id; Functor-Id} \} \\
((\pi_1 \nabla \pi_1) + (\pi_1 \nabla \pi_1)) \circ dists \\
= \{ \text{Dists-Def; Distr-Fst; Distr-Fst} \} \\
(\pi_1 + \pi_1) \circ distl \\
= \{ \text{Distl-Fst} \} \\
\pi_1
\end{array}
$$