Calculating with Lenses

Optimising Bidirectional Transformations

Hugo Pacheco
DI-CCTC, Universidade do Minho, Braga, Portugal
hpacheco@di.uminho.pt

Alcino Cunha
DI-CCTC, Universidade do Minho, Braga, Portugal
alcino@di.uminho.pt

Abstract

This paper presents an equational calculus to reason about bidirectional transformations specified in the point-free style. In particular, it focuses on the so-called *lenses* as a bidirectional idiom, and shows that many standard laws characterising point-free combinators and recursion patterns are also valid in that setting. A key result is that uniqueness also holds for bidirectional folds and unfolds, thus unleashing the power of fusion as a program optimisation technique. A rewriting system for automatic lens optimisation is also presented, to prove the usefulness of the proposed calculus.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Data Types and Structures,Recursion; D.3.4 [*Programming Languages*]: Processors—Optimization; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Algebraic approaches to semantics; I.1.1 [*Symbolic and Algebraic Manipulation*]: Expressions and Their Representation

General Terms Design, Languages, Theory

Keywords Program calculation, Bidirectional transformation, Point-free programming

1. Introduction

In an heterogeneous world of data formats and programming languages, data transformation frameworks play an essential role in facilitating the sharing of information among software applications. Modifications to the data often break the consistency between source and target data models; and a key problem with most of these frameworks is the lack of proper bidirectional mechanisms to synchronise them. Typically, we end up with ad-hoc solutions like manually engineering two unidirectional transformations together so that they are consistent according to the required synchronisation policy. This introduces a severe maintenance problem: any change in one of the data models implies a redefinition of both transformations and a new consistency verification.

In response to this problem, intrinsic *bidirectional transformation* [11] frameworks have become increasingly popular in various computer science domains, including heterogeneous data synchronisation [4, 14], software model transformation [12, 15, 29], graph transformations [16], schema evolution [2, 10], relational databases [3] and functional programming [21, 22, 25, 30, 31]. Most of these approaches encompass the design of domain-specific bidirectional languages in which one expression denotes a connected pair of transformations, whose consistency is guaranteed in the respective semantic space.

A distinctive contribution to the field is the *Focal* bidirectional tree transformation language, proposed by Foster *et al* [14], whose building blocks are the so-called *lenses*. A lens computes a view A from a concrete data model C, and encompasses three functions: $get : C \to A$, that abstracts details from a concrete model; $create : A \to C$, that enriches an abstract model into a new concrete model; and $put : A \times C \to C$, that synchronises a modified view with the original concrete model. As an example, consider that the concrete source data model is a list of people containing name and gender. A possible lens over this data type is the transformation that counts the number of women, whose get function could be trivially defined (for example, in *Haskell*) as follows:

$$\begin{array}{l}\mathbf{type}\ Person = (Name, Gender)\ \mathbf{data}\ Gender = M \mid F \\ \mathbf{data}\ Nat = Zero \mid Succ\ Nat \qquad \mathbf{type}\ Name = String \\ get_w :: [Person] \to Nat \\ get_w\ [\,] \qquad\quad = Zero \\ get_w\ ((nm, M) : ps) = get_w\ ps \\ get_w\ ((nm, F) : ps)\ = Succ\ (get_w\ ps)\end{array}$$

Using a traditional non-bidirectional approach, the programmer should now define the remaining functions according to sensible synchronisation requirements. For example, the $create$ function should, given a natural number, generate a list of women of that length, according to the lens properties (as formally presented in Section 2). Given that no other information is available, the only choice is to generate default names. A possible implementation is

$$\begin{array}{l}create_w\ :: Nat \to [Person] \\ create_w\ Zero \quad\ = [\,] \\ create_w\ (Succ\ n) = (\texttt{"Eve"}, F) : create_w\ n\end{array}$$

Since put is expected to reconcile view updates with the source, it is reasonable to require that, likewise to $create$, the number of women in the output must be equal to the updated view. However, whenever possible, the names of women from the original list should also be restored. Men names should be restored at least up to to the original length. Given these requirements, the definition of put is substantially more intricate. A possible implementation could be written as follows, but it is no longer trivial to check that it satisfies all above requirements.

$$\begin{array}{l}put_w\ :: (Nat, [Person]) \to [Person] \\ put_w\ (Zero, [\,]) \qquad\quad = [\,] \\ put_w\ (Zero, (nm, F) : ps) \quad = [\,] \\ put_w\ (Zero, (nm, M) : ps) \quad = (nm, M) : put_w\ (Zero, ps)\end{array}$$

PEPM'11, January 24–25, 2011, Austin, Texas, USA.

$$
\begin{aligned}
put_w\ (Succ\ n, [\,]) &= (\texttt{"Eve"}, F) : create_w\ n \\
put_w\ (Succ\ n, (nm, F) : ps) &= (nm, F) : put_w\ (n, ps) \\
put_w\ (Succ\ n, (nm, M) : ps) &= (nm, M) : put_w\ (Succ\ n, ps)
\end{aligned}
$$

These three functions could be packed together as a lens:

$$
\begin{aligned}
&\mathbf{data}\ Lens\ c\ a = Lens \\
&\quad \{\, get :: c \to a, put :: (a, c) \to c, create :: a \to c \,\} \\
&women = Lens\ get_w\ put_w\ create_w
\end{aligned}
$$

Given the requirements on put, it should be now clear that this conventional approach is bound for failure as model complexity increases. As such, *Focal* provides a rich set of lens combinators that allow users to combine primitive lenses into sophisticated bidirectional transformations, that are guaranteed to be consistent (or *well-behaved*) according to a precise synchronisation policy. In [25], we have taken a similar approach, by bidirectionalising some well-known point-free combinators and recursion patterns: any get function defined using those combinators specifies a *well-behaved* lens, where adequate $create$ and put functions are generated for free. Using this language, we could redefine the above lens just as

$$women = length \circ filter\_l \circ map\ (out_G \circ \pi_2{}^{const\ \texttt{"Eve"}})$$

where *filter_l* filters all left-alternatives from a list, $out_G : Gender \to 1 + 1$ exposes the top-level structure of the $Gender$ data type as a sum-of-products, and π_2 projects the second component of a pair (parameterised by the function $const$ `"Eve"` to generate the other component whenever necessary).

Unfortunately, despite the convenience of these compositional approaches, the resulting transformation can suffer from poor efficiency, due to the cluttering of intermediate data structures; and if manual design of bidirectional transformations is tedious and error-prone, manual optimisation is a much more thankless (not to say impossible) task. Deeply related to this problem is the lack of an algebraic calculus to reason directly about lenses: proving bidirectional properties generally requires independent proofs for each of the three components. In particular, the lack of bidirectional *fusion* laws prevents the use of the typical optimisation techniques for functional programs. Quoting Foster et al [14]:

> "Is there an algebraic theory of lens combinators that would underpin optimisation of lens expressions in the same way that the relational algebra and its algebraic theory are used to optimise relational database queries? [...] This algebraic theory will play a crucial role in a more serious implementation effort."

The point-free style is characterised by a rich set of algebraic laws, making it very amenable for program calculation. Thus, we are particularly well positioned to answer this question: the first goal of this paper is precisely to determine which algebraic laws characterising the point-free combinators (and recursion patterns) can be lifted to lenses. Apart from few side-conditions to control the non-determinism in backward transformations, this algebraic theory will enable us to calculate with lenses using only the get point-free specification. For example, using the bidirectional fold fusion law presented in Section 3, we will be able to show that the map-fusion law is also valid on lenses, by performing the conventional proof:

$$map\ (\delta \circ \tau) = map\ \delta \circ map\ \tau \qquad map\text{-FUSION}$$

The second goal of this paper is to employ this bidirectional calculus as the kernel of an automatic optimisation tool for point-free lenses, thus combining the simplicity and elegance of a combinatorial approach with the efficiency of manually-crafted transformations. For instance, we can show that, after optimisation, the point-free specification of the $women$ example performs neck-to-neck with the original handwritten definition. The implementation of this tool builds on a successful rewrite system for transformation of point-free programs [6, 9], extending it to support lenses. Of course, optimisation could be attempted independently at the three components of a lens, since they are also defined in the point-free style: we initially followed that approach but, unfortunately, the complexity of the put function prevented the automatic spotting of many optimisation opportunities. We ended up with a mixed approach: the bulk of optimisations is performed directly at the lens level (namely, all fusions involving recursion patterns), with some minor optimisations performed later at each component separately.

In the next section (Section 2), we briefly review the point-free lens combinators first presented in [25] and introduce the algebraic laws that characterise them. Section 3 reviews the construction of recursive lenses and studies the uniqueness properties of bidirectional folds and unfolds. In Section 4, we show how to harness these laws, in particular fusion, into an effective rewrite system for the simplification of bidirectional transformations and discuss the implementation of this rewrite system (using the functional programming language Haskell), together with the speedup results for a complex transformation scenario. Section 5 compares related work and Section 6 concludes the paper with a synthesis of the main contributions and directions for future work.

$$
\begin{aligned}
id &: A \to A \\
(\circ) &: (B \to C) \to (A \to B) \to (A \to C) \\
(\vartriangle) &: (A \to B) \to (A \to C) \to (A \to B \times C) \\
\pi_1 &: A \times B \to A \\
\pi_2 &: A \times B \to B \\
(\times) &: (A \to B) \to (C \to D) \to (A \times C \to B \times D) \\
(\triangledown) &: (A \to C) \to (B \to C) \to (A + B \to C) \\
i_1 &: A \to A + B \\
i_2 &: B \to A + B \\
(+) &: (A \to B) \to (C \to D) \to (A + C \to B + D) \\
ap &: B^A \times A \to B \\
\bar{\cdot} &: (A \times B \to C) \to (A \to C^B) \\
\cdot^\bullet &: (B \to C) \to (B^A \to C^A) \\
! &: A \to 1 \\
\underline{\cdot} &: A \to (1 \to A)
\end{aligned}
$$

Figure 1. Point-free combinators

2. Point-free Combinators as Lenses

Before presenting the building blocks of our lens language, we need to set the semantic space where they exist. In this work, the semantic domain for functions will be the SET category, with sets (types) as objects and total functions as arrows. Being a "well-behaved" category, with products, co-products and terminal object, we immediately get a powerful set of point-free combinators for building complex functions out of simpler ones (Figure 1). Moreover, these constructs enjoy universal laws that enable reasoning by calculation. For example, the existence of categorical products ensures that there is a unique way to combine two functions with a shared domain into a function that applies both in order to produce a pair:

$$f = g \vartriangle h \Leftrightarrow \pi_1 \circ f = g \wedge \pi_2 \circ f = h$$

As expected, not every combination of get, put and $create$ functions builds a reasonable lens. Several properties will be required of these functions in order to get a "well-behaved" lens (henceforward denoted just as "lens"). Using the point-free style, we can now give a precise definition of such properties.

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

$$get_\delta \circ create_\delta = id \qquad \textsc{CreateGet}$$
$$get_\delta \circ put_\delta = \pi_1 \qquad \textsc{PutGet}$$
$$put_\delta \circ (get_\delta \vartriangle id) = id \qquad \textsc{GetPut}$$

Property CREATEGET guarantees that get is an abstraction function, this is, A contains at most as much information as C. PUTGET guarantees that the lens is *acceptable*, i.e., updates to a view cannot be ignored and must be translated exactly. Finally, GETPUT states that the lens should be *stable*, i.e, if the view does not change, then neither should the source.

In [25] our research question was: which morphisms in SET also denote lenses? In order to answer this question, each primitive function and combinator of Figure 1 was scrutinised and, whenever possible, bidirectionalised to a primitive lens or lens combinator. Being the point-free style so intertwined with algebraic calculation, the followup question must necessarily be: are the laws characterising the standard point-free combinators also valid in the lifted lenses? One of the objectives of this paper is precisely to answer this question. In the remaining of this section we will briefly recall the work presented in [25], but now stating for each bidirectionalised combinator the laws that characterise it. To avoid introducing a different notation, the lifted lens combinators are represented using the same notation but, to disambiguate lens laws from the standard point-free laws, lens variables will be denoted by greek letters ($\delta, \tau, \phi, \psi, ...$).

2.1 Basic lens combinators

The fundamental point-free combinators are identity and function composition. Both can be lifted to lenses as follows:

$$\begin{array}{ll} id : C \trianglerighteq A & \forall \delta : B \trianglerighteq A, \tau : C \trianglerighteq B.\ (\delta \circ \tau) : C \trianglerighteq A \\ get = id & get = get_\delta \circ get_\tau \\ put = \pi_1 & put = put_\tau \circ (put_\delta \circ (id \times get_\tau) \vartriangle \pi_2) \\ create = id & create = create_\tau \circ create_\delta \end{array}$$

The *identity* and *associativity* axioms that characterise these combinators are also valid for the lifted versions:

$$id \circ \delta = \delta = \delta \circ id \qquad id\textsc{-Nat}$$
$$\delta \circ (\tau \circ \phi) = (\delta \circ \tau) \circ \phi \qquad \circ\textsc{-Assoc}$$

Since both these laws are valid, a category of lenses can be defined, whose objects are the same objects of SET and morphisms are well-behaved lenses.

Proof. Two lenses δ and τ are equal iff $get_\delta = get_\tau$, $put_\delta = put_\tau$, and $create_\delta = create_\tau$. The first equality is always trivially true because the get function has exactly the same point-free definition as the lens itself. The trickiest part is always proving that both puts are equal (especially when involving lots of compositions). For example, for $\circ$-ASSOC such proof can done as follows:

$$\begin{array}{l} put_{\delta \circ (\tau \circ \phi)} \\ = \{ \text{definition of } put \} \\ put_{\tau \circ \phi} \circ (put_\delta \circ (id \times get_{\tau \circ \phi}) \vartriangle \pi_2) \\ = \{ \text{definition of } put \} \\ put_\phi \circ (put_\tau \circ (id \times get_\phi) \vartriangle \pi_2) \circ (put_\delta \circ (id \times get_{\tau \circ \phi}) \vartriangle \pi_2) \\ = \{ \times\textsc{-Fusion}; \times\textsc{-Absor}; \times\textsc{-Cancel} \} \\ put_\phi \circ (put_\tau \circ (put_\delta \circ (id \times get_{\tau \circ \phi}) \vartriangle get_\phi \circ \pi_2) \vartriangle \pi_2) \\ = \{ \text{definition of } get \} \\ put_\phi \circ (put_\tau \circ (put_\delta \circ (id \times get_\tau \circ get_\phi) \vartriangle get_\phi \circ \pi_2) \vartriangle \pi_2) \\ = \{ \times\textsc{-Fusion}; \times\textsc{-Functor-Comp}; \times\textsc{-Cancel} \} \\ put_\phi \circ (put_\tau \circ (put_\delta \circ (id \times get_\tau) \vartriangle \pi_2) \circ (id \times get_\phi) \vartriangle \pi_2) \\ = \{ \text{definition of } put \} \\ put_\phi \circ (put_{\delta \circ \tau} \circ (id \times get_\phi) \vartriangle \pi_2) \\ = \{ \text{definition of } put \} \\ put_{(\delta \circ \tau) \circ \phi} \end{array}$$

Note that these equalities are proven using laws valid for functions on SET (see [25] for a compendium). Throughout the paper, we will use the same name to denote laws valid both on lenses and functions. Disambiguation should be trivial from the context. The proofs for $create$ of the law id-NAT are trivial and will be elided. Due to space constrains, the proofs of the remaining laws introduced in this section will not be presented. Although some of them are a bit more complex than the one above, they are still fairly easy, at least for someone experienced with the point-free style. □

The *bang* combinator is a lens that ignores all the concrete information:

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

Here, $f : 1 \to C$ is a function that generates default concrete values. Due to this parameter function, we cannot state that 1 is a proper terminal object of our category of lenses, because there is more than one lens with type $C \trianglerighteq 1$. Nonetheless, we can phrase a lifted version of the uniqueness law for !:

$$\delta = !^{create_\delta} \Leftrightarrow \delta : A \trianglerighteq 1 \qquad !\textsc{-Uniq}$$

2.2 Products

Unfortunately, our category of lenses is not as "well-behaved" as SET. For example, as discussed in [25], there are no categorical products because in general it is not possible to define the split lens $\delta \vartriangle \tau : A \trianglerighteq B \times C$, given lenses $\delta : A \trianglerighteq B$ and $\tau : A \trianglerighteq C$; there are no lenses of type $A \trianglerighteq A \times A$ (unless A is the unit type 1), because the view is not an abstraction of the source. Thus, although $id : A \trianglerighteq A$ is a well-behaved lens, $id \vartriangle id : A \trianglerighteq A \times A$ is not. However, we do have the product bi-functor and projections as valid lenses, defined as follows:

$$\begin{array}{ll} \forall f : A \to B.\ {\pi_1}^f : A \times B \trianglerighteq A & \forall f : B \to A.\ {\pi_2}^f : A \times B \trianglerighteq B \\ get \quad = \pi_1 & get \quad = \pi_2 \\ put \quad = id \times \pi_2 & put \quad = swap \circ (id \times \pi_1) \\ create = id \vartriangle f & create = f \vartriangle id \end{array}$$

$$\begin{array}{l} \forall \delta : C \trianglerighteq A, \tau : D \trianglerighteq B.\ \delta \times \tau : C \times D \trianglerighteq A \times B \\ get \quad = get_\delta \times get_\tau \\ put \quad = (put_\delta \times put_\tau) \circ distp \\ create = create_\delta \times create_\tau \end{array}$$

In the projections, the parameter f is a function that generates default values for the deleted component of the pair. $swap : A \times B \to B \times A$ and $distp : (A \times B) \times (C \times D) \to (A \times C) \times (B \times D)$ are standard isomorphisms.

The following laws guarantee that the product lens is also a bi-functor in the category of lenses:

$$id \times id = id \qquad \times\textsc{-Functor-Id}$$
$$(\delta \times \tau) \circ (\phi \times \psi) = \delta \circ \phi \times \tau \circ \psi \qquad \times\textsc{-Functor-Comp}$$

Projections also enjoy a kind of naturality law, with a precise characterisation of how the default generation function must be adapted.

$${\pi_1}^f \circ (\delta \times \tau) = \delta \circ {\pi_1}^{create_\tau \circ f \circ get_\delta} \qquad \pi_1\textsc{-Nat}$$
$${\pi_2}^f \circ (\delta \times \tau) = \tau \circ {\pi_2}^{create_\delta \circ f \circ get_\tau} \qquad \pi_2\textsc{-Nat}$$

2.3 Sums

Similarly to products, categorical sums do not exist in the category of lenses, this time because the injections i_1, i_2 are not surjective functions and thus not definable as lenses. Notwithstanding, the sum bi-functor and either combinators denote valid lenses, as defined below:

$$\begin{array}{l}\forall_{p:A\to 2}, \delta : C \trianglerighteq A, \tau : B \trianglerighteq A.\ (\delta \nabla \tau)^p : C + B \trianglerighteq A \\ get \quad = get_\delta \nabla get_\tau \\ put \quad = (put_\delta + put_\tau) \circ distr \\ create = (create_\delta + create_\tau) \circ p\ ?\end{array}$$

$$\begin{array}{l}\forall_{f:A\times D\to C,\ g:B\times C\to D}, \delta : C \trianglerighteq A, \tau : D \trianglerighteq B. \\ \quad {\scriptstyle get_\delta \circ f = \pi_1\ \wedge\ get_\tau \circ g = \pi_1} \Rightarrow (\delta + \tau)^{f,g} : C + D \trianglerighteq A + B \\ get \quad = get_\delta + get_\tau \\ put \quad = (put_\delta \nabla f + g \nabla put_\tau) \circ dists \\ create = create_\delta + create_\tau\end{array}$$

In these definitions, $distr : (A + B) \times C \to A \times C + B \times C$ and $dists : (A + B) \times (C + D) \to (A \times C + A \times D) + (A \times C + A \times D)$ are isomorphisms that distribute products over sums, while $(?)$ lifts a predicate of type $A \to 2$ to a more useful input-preserving function of type $A \to A + A$. The either combinator is parameterised by a predicate p that dictates the choice of left or right alternatives in $create$. We will denote by $\overset{\bullet}{\nabla}$ and $\nabla_\bullet$ the versions when the predicate always returns true or false, respectively. In the sum combinator, the parameter functions specify how to reconstruct concrete values whenever the abstract and concrete sums are "out of sync". The conditions $get_\delta \circ f = \pi_1$ and $get_\tau \circ g = \pi_2$ force these functions to be acceptable (likewise to put), i.e., the view cannot be ignored when computing the defaults. Useful candidates that satisfy these restrictions are $create_\delta \circ \pi_1$ and $create_\tau \circ \pi_1$, respectively, and when superscripts are omitted from the sum these are assumed to be the parameters.

Likewise to its functional counterpart, the either lens combinator also satisfies fusion and absorption laws:

$$\delta \circ (\tau \nabla \phi)^p = (\delta \circ \tau \nabla \delta \circ \phi)^{p \circ create_\delta} \qquad \text{+-FUSION}$$

$$(\delta \nabla \tau)^p \circ (\phi + \psi)^{f,g} = (\delta \circ \phi \nabla \tau \circ \phi)^p \qquad \text{+-ABSOR}$$

Note how the first law constrains the new predicate to be coherent with the $create$ of the fused lens. Compositions of sums can be fused according to the following law, that states how the new parameter functions can be deduced:

$$\begin{array}{l}(\delta + \tau)^{f,g} \circ (\phi + \psi)^{h,i} = (\delta \circ \phi + \tau \circ \psi)^{j,k} \\ \qquad\qquad \Leftrightarrow \\ {\scriptstyle j = h \circ (f \circ (id \times get_\psi) \vartriangle \pi_2)\ \wedge\ k = i \circ (g \circ (id \times get_\phi) \vartriangle \pi_2)}\end{array} \qquad \text{+-COMP}$$

If the parameters are the standard $create \circ \pi_1$ we have the following simplified version:

$$(\delta + \tau) \circ (\phi + \psi) = \delta \circ \phi + \tau \circ \psi \qquad \text{+-FUNCTOR-COMP}$$

Together with the following law, this ensures that the sum combinator is also a bi-functor.

$$(id + id)^{f,g} = id \qquad \text{+-FUNCTOR-ID}$$

2.4 Isomorphisms as lens combinators

The simplest cases of bidirectional transformations are isomorphisms. Given a bijective function $f : A \to B$ (with inverse $f^{-1} : B \to A$), we can trivially define a *lens isomorphism* $f : A \trianglerighteq B$ as:

$$\begin{array}{l}get \quad = f \\ put \quad = f^{-1} \circ \pi_1 \\ create = f^{-1}\end{array}$$

It is trivial to prove that $f \circ f^{-1} = f^{-1} \circ f = id$ is also valid at the lens level. There are many useful examples of such lens isomorphisms, such as the following:

$$\begin{array}{ll}swap & : A \times B \trianglerighteq B \times A \\ assoc & : A \times (B \times C) \trianglerighteq (A \times B) \times C \\ coswap & : A + B \trianglerighteq B + A \\ coassoc & : A + (B + C) \trianglerighteq (A + B) + C \\ distl & : (A + B) \times C \trianglerighteq A \times C + B \times C \\ distr & : A \times (B + C) \trianglerighteq A \times B + A \times C\end{array}$$

Since splits and injections are not valid lenses, these lens isomorphisms play an important role in extending the expressivity of our point-free lens language. For example, all lenses that rearrange nested pairs can be defined as compositions of $swap$, $assoc$, $assoc^{-1}$ and products [23].

A *natural lens* η between functors F and G, denoted by $\eta : F \mathrel{\dot{\trianglerighteq}} G$, is a lens 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 $\eta_A : F\ A \trianglerighteq G\ A$ such that, for any lens $\delta : A \trianglerighteq B$, the following naturality condition holds:

$$\eta \circ F\ \delta = G\ \delta \circ \eta \qquad \eta\text{-NAT}$$

This concept can be generalised to functors of higher arity. If a natural lens η is also an isomorphism, then it is called a *natural lens isomorphism*. Such is the case of all the above lenses. For example, the following bidirectional naturality laws are also valid:

$$swap \circ (\delta \times \tau) = (\tau \times \delta) \circ swap \qquad swap\text{-NAT}$$

$$coswap \circ (\delta + \tau)^{f,g} = (\tau + \delta)^{g,f} \circ coswap \qquad coswap\text{-NAT}$$

The naturality law for $distl$ is a bit more tricky:

$$\begin{array}{l}distl \circ ((\delta + \tau)^{f,g} \times \phi) = (\delta \times \phi + \tau \times \phi)^{h,i} \circ distl \\ \qquad\qquad \Leftrightarrow \\ {\scriptstyle h = (f \times put_\phi) \circ distp\ \wedge\ i = (g \times put_\phi) \circ distp}\end{array} \qquad distl\text{-NAT}$$

Many other useful laws can be proved about these lens isomorphisms, such as the following cancelation laws:

$$swap\text{-CANCEL} \qquad {\pi_1}^f \circ swap = {\pi_2}^f \wedge {\pi_2}^f \circ swap = {\pi_1}^f$$

$$coswap\text{-CANCEL} \qquad (\delta \nabla \tau)^p \circ coswap = (\tau \nabla \delta)^{coswap \circ p}$$

$$distl\text{-SND-CANCEL} \qquad ({\pi_2}^f \nabla {\pi_2}^g)^p \circ distl = {\pi_2}^{(f+g)\circ p?}$$

2.5 Higher-order lens combinators

Higher-order lenses are also definable in our category of lenses through exponentiation. The exponentiation type B^A denotes all functions with domain A and codomain B, and is characterised by an operation $f^A : B^A \to C^A$, where $f : B \to C$. Replacing the type superscript by the symbol $\bullet$, the exponentiation lens can be defined as follows:

$$\begin{array}{l}\forall \delta : B \trianglerighteq C.\ \delta^\bullet : B^A \trianglerighteq C^A \\ get = get_\delta{}^\bullet \\ put = put_\delta{}^\bullet \circ \hat{\vartriangle} \\ create = create_\delta{}^\bullet\end{array}$$

Here $\hat{\vartriangle} = \overline{(ap \times ap) \circ ((\pi_1 \times id) \vartriangle (\pi_2 \times id))}$ denotes the uncurried version of the split combinator [7]. Again, exponentiation is a functor in the lens category:

$$id^\bullet = id \qquad \bullet\text{-FUNCTOR-ID}$$

$$(\delta \circ \tau)^\bullet = \delta^\bullet \circ \tau^\bullet \qquad \bullet\text{-FUNCTOR-COMP}$$

The ap combinator can also be lifted to a lens. The point-free definition is a bit tricky, and thus we just present the point-wise version for better comprehension:

$$
\begin{array}{ll}
\forall_{f:B\to A}.\ ap^f : B^A \times A \trianglerighteq B & \\
get\ (g,x) & = g\ x \\
put\ (y,(g,x)) & = (\lambda z \to \mathbf{if}\ x = z\ \mathbf{then}\ y\ \mathbf{else}\ g\ x, x) \\
create\ y & = (const\ y, f\ y)
\end{array}
$$

Note how the put function updates the original function with a new result for the input value that was applied. The parameter function f is used in $create$ to choose a value of the domain A. Application cancels exponentiation, according to the law:

$$ap^f \circ (\delta^\bullet \times id) = \delta \circ ap^{f \circ get_\delta} \qquad \bullet\text{-CANCEL}$$

Unfortunately, we also do not have categorical exponentiation in the category of lenses because the curry of a well-behaved lens may not be a well-behaved lens. For example, note that, although $\pi_2 : A \times B \trianglerighteq B$ is a lens, $\overline{\pi_2} : A \to B^B$ is not surjective and thus cannot be made into a lens (given a value of type A it returns the function id).

3. Recursion Patterns as Lenses

Concerning recursion, most inductive datatypes can be defined as fixpoints of polynomial functors (sums of products). Each datatype comes equipped with an isomorphism $out_F : \mu F \to F\ \mu F$ that can be used to expose its top-level structure (in a sense, encoding pattern matching over that type), and its converse $in_F : F\ \mu F \to \mu F$ that determines how values of that type can be constructed. Being isomorphisms, these functions can trivially be lifted to lenses. Besides uniquely determining a type (up to isomorphism), a functor also dictates a unique way of consuming and producing values of that type: the well-known recursion patterns fold and unfold. For example, given an algebra $g : F\ A \to A$, the fold $(\!|g|\!) : \mu F \to A$ is the unique function satisfying the following universal law:

$$f = (\!|g|\!) \Leftrightarrow f \circ in_F = g \circ F\ f$$

From this we can derive the well known fold fusion law. As we will see in this section, this universal law is also valid for lenses: it will enable us to apply fusion directly to lenses, thus streamlining the optimisation process.

Building on the results of the previous section, we first define a polytypic (polynomial) functor map over lenses:

$$
\begin{array}{l}
\forall \delta : C \trianglerighteq A.\ F\ \delta : F\ C \trianglerighteq F\ A \\
Id\ \delta = \delta \\
\underline{T}\ \delta = id \\
(F \otimes G)\ \delta = F\ \delta \times G\ \delta \\
(F \oplus G)\ \delta = F\ \delta + G\ \delta \\
(F \odot G)\ \delta = F\ (G\ \delta)
\end{array}
$$

This definition trivially satisfies the following laws:

$$F\ id = id \qquad \text{FUNCTOR-ID}$$

$$F\ \delta \circ F\ \tau = F\ (\delta \circ \tau) \qquad \text{FUNCTOR-COMP}$$

In order to bidirectionalise recursion patterns, in [25] we introduced a polytypic functor zipping function $fzip_F : (A \to C) \to F\ A \times F\ C \to F\ (A \times C)$ that satisfies the following laws:

$$put_{F\ \delta} = F\ put_\delta \circ fzip_F\ create_\delta \qquad fzip\text{-PUT}$$

$$F\ \pi_1 \circ fzip_F\ f = \pi_1 \qquad fzip\text{-CANCEL}$$

$$fzip_F\ f \circ (F\ g \vartriangle F\ h) = F\ (g \vartriangle h) \qquad fzip\text{-SPLIT}$$

One of the key results in [25] is that the fold can be bidirectionalised using an unfold for the put function:

$$
\begin{array}{ll}
\forall \delta : F\ A \trianglerighteq A.\ (\!|\delta|\!)_F : \mu F \trianglerighteq A & \\
get & = (\!|get_\delta|\!)_F \\
put & = \mathbf{let}\ g = put_\delta \circ (id \times F\ get) \vartriangle \pi_2 \\
 & \quad \mathbf{in}\ \ [\!|fzip_F\ create \circ g \circ (id \times out_F)|\!]_F \\
create & = [\!|create_\delta|\!]_F
\end{array}
$$

In [25] we have also shown that this definition yields a well-behaved lens whenever the unfold terminates. We will now prove that it also has uniqueness:

$$\delta = (\!|\tau|\!)_F \Leftrightarrow \delta \circ in_F = \tau \circ F\ \delta \qquad (\!|\cdot|\!)\text{-UNIQ}$$

Proof. This proof can be factorised in the following three lemmas:

$$get_\delta = get_{(\!|\tau|\!)_F} \Leftrightarrow get_{\delta \circ in_F} = get_{\tau \circ F\ \delta}$$

$$create_\delta = create_{(\!|\tau|\!)_F} \Leftrightarrow create_{\delta \circ in_F} = create_{\tau \circ F\ \delta}$$

$$put_\delta = put_{(\!|\tau|\!)_F} \Leftrightarrow put_{\delta \circ in_F} = put_{\tau \circ F\ \delta} \quad \Leftarrow \quad \begin{array}{c} get_\delta = get_{(\!|\tau|\!)_F} \\ create_\delta = create_{(\!|\tau|\!)_F} \end{array}$$

Again, the first follows directly from the unidirectional uniqueness. The proof of the remaining is presented in Figure 2. □

In [25] we have also shown how to bidirectionalise the unfold using an hylomorphism (a composition of a fold after an unfold) for put. Due to space constrains we will not present the definition. Likewise to the fold, it is possible to prove that the bidirectional version of unfold also has uniqueness:

$$\delta = [\!|\tau|\!]_F \Leftrightarrow out_F \circ \delta = F\ \delta \circ \tau \qquad [\!|\cdot|\!]\text{-UNIQ}$$

From uniqueness it is trivial to derive the following laws, more amenable for equational reasoning:

$$(\!|in_F|\!)_F = id \qquad (\!|\cdot|\!)\text{-REFLEX}$$

$$(\!|\tau|\!)_F \circ in_F = \tau \circ F\ (\!|\tau|\!)_F \qquad (\!|\cdot|\!)\text{-CANCEL}$$

$$\delta \circ (\!|\tau|\!)_F = (\!|\phi|\!)_F \Leftarrow \delta \circ \tau = \phi \circ F\ \delta \qquad (\!|\cdot|\!)\text{-FUSION}$$

$$[\!|out_F|\!]_F = id \qquad [\!|\cdot|\!]\text{-REFLEX}$$

$$out_F \circ [\!|\tau|\!]_F = F\ [\!|\tau|\!]_F \circ \tau \qquad [\!|\cdot|\!]\text{-CANCEL}$$

$$[\!|\tau|\!]_F \circ \delta = [\!|\phi|\!]_F \Leftarrow \tau \circ \delta = F\ \delta \circ \phi \qquad [\!|\cdot|\!]\text{-FUSION}$$

3.1 Algebraic laws for lenses over lists

Lists are ubiquitous in functional programming. As a demonstration of the usefulness of our bidirectional calculus, we will now show how some standard operations over lists can be defined in our language and prove some properties about them. In particular, the lenses used in our introduction example can be defined as follows:

$$
\begin{array}{ll}
length^A : [A] \trianglerighteq Nat & \\
length^v & = (\!|in_N \circ (id + \pi_2 \overset{v \circ !}{-})|\!)_L \\
map & : (A \trianglerighteq B) \to ([A] \trianglerighteq [B]) \\
map\ \delta & = (\!|in_L \circ (id + \delta \times id)|\!)_L \\
filter\_l & : [A + B] \trianglerighteq [A] \\
filter\_l & = (\!|(in_L \blacktriangledown \pi_2) \circ coassoc \circ (id + distl)|\!)_L
\end{array}
$$

The parameter in $length$ is the default value of type A to be inserted in the source list when the target length increases. In the introductory example it was omitted because it was the sole inhabitant of type 1. Many more lenses over lists can be found in the Haskell `pointless-lenses` library introduced in [25]. Some of the usual laws that can be proved about these functions are presented in Figure 3. Since map is defined exactly as its unidirectional version, the following map fusion laws can be trivially proven from uniqueness.

$$(\!|\tau|\!) \circ map\ \delta = (\!|\tau \circ (id + \delta \times id)|\!) \qquad (\!|\cdot|\!)\text{-MAP-FUSION}$$

$$map\ \delta \circ [\!|\tau|\!] = [\!|(id + \delta \times id) \circ \tau|\!] \qquad [\!|\cdot|\!]\text{-MAP-FUSION}$$

$create_\delta = create_{(\!|\tau|\!)_F}$
$\Leftrightarrow \{ \text{definition of } create \}$
$create_\delta = [\!|create_\tau|\!]_F$
$\Leftrightarrow \{ [\!|\cdot|\!]\text{-UNIQ} \}$
$out_F \circ create_\delta$
$= F\ create_\delta \circ create_\tau$
$\Leftrightarrow \{ \text{definition of } create \}$
$create_{in_F} \circ create_\delta$
$= create_{F\ \delta} \circ create_\tau$
$\Leftrightarrow \{ \text{definition of } create \}$
$create_{\delta \circ in_F} = create_{\tau \circ F\ \delta}$

$put_\delta = put_{(\!|\tau|\!)_F}$
$\Leftrightarrow \{ \text{definition of } put \}$
$put_\delta = fzip_F\ create_{(\!|\tau|\!)_F} \circ (put_\tau \circ (id \times F\ get_{(\!|\tau|\!)_F}) \vartriangle \pi_2) \circ (id \times out_F)$
$\Leftrightarrow \{ [\!|\cdot|\!]\text{-UNIQ} \}$
$out_F \circ put_\delta = F\ put_\delta \circ fzip_F\ create_{(\!|\tau|\!)_F} \circ (put_\tau \circ (id \times F\ get_{(\!|\tau|\!)_F}) \vartriangle \pi_2) \circ (id \times out_F)$
$\Leftrightarrow \{ get_{(\!|\tau|\!)_F} = get_\delta; create_{(\!|\tau|\!)_F} = create_\delta \}$
$out_F \circ put_\delta = F\ put_\delta \circ fzip_F\ create_\delta \circ (put_\tau \circ (id \times F\ get_\delta) \vartriangle \pi_2) \circ (id \times out_F)$
$\Leftrightarrow \{ fzip\text{-PUT}; \text{definition of } get \}$
$out_F \circ put_\delta = put_{F\ \delta} \circ (put_\tau \circ (id \times get_{F\ \delta}) \vartriangle \pi_2) \circ (id \times out_F)$
$\Leftrightarrow \{ \text{definition of } put; \times\text{-REFLEX} \}$
$out_F \circ put_\delta = put_{\tau \circ F\ \delta} \circ (id \times out_F)$
$\Leftrightarrow \{ in\text{-ISO}; \times\text{-FUNCTOR-COMP}; \text{LEIBNIZ} \}$
$out_F \circ put_\delta \circ (id \times in_F) = put_{\tau \circ F\ \delta}$
$\Leftrightarrow \{ \times\text{-CANCEL}; \text{definition of } put \}$
$put_{in_F} \circ (put_\delta \circ (id \times in_F) \vartriangle \pi_2) = put_{\tau \circ F\ \delta}$
$\Leftrightarrow \{ \text{definition of } get; \text{definition of } put \}$
$put_{\delta \circ in_F} = put_{\tau \circ F\ \delta}$

Figure 2. Proof of $(\!|\cdot|\!)$-UNIQ

$map\ id = id$	map-ID
$map\ \delta \circ map\ \tau = map\ (\delta \circ \tau)$	map-FUSION
$map\ \delta \circ cat = cat \circ (map\ \delta \times map\ \delta)$	map-CAT
$map\ \delta \circ concat = concat \circ map\ (map\ \delta)$	map-CONCAT
$filter\_l \circ map\ (\delta + \tau)^{f,g} = map\ \delta \circ filter\_l$	$filter\_l$-MAP
$length^v \circ cat = plus \circ (length^v \times length^v)$	$length$-CAT
$length^v \circ map\ \delta = length^{create_\delta\ v}$	$length$-MAP
$length^v \circ concat = sum \circ map\ length^v$	$length$-CONCAT

Figure 3. Lens laws for common operations over lists.

For example, using $(\!|\cdot|\!)$-MAP-FUSION the proof of $length$-MAP can be done as follows:

$length^v \circ map\ \delta$
$= \{ length\text{-DEF}; (\!|\cdot|\!)\text{-MAP-FUSION} \}$
$(\!|in_N \circ (id + \pi_2{}^{v \circ !}) \circ (id + \delta \times id)|\!)_L$
$= \{ +\text{-FUNCTOR-COMP}; \pi_2\text{-NAT} \}$
$\quad create_\delta \circ \underline{v} \circ ! \circ get_{id}$
$\quad = \{ \underline{\ }\text{-COMP}; \text{definition of } get \}$
$\quad \underline{create_\delta\ v} \circ !$
$(\!|in_N \circ (id + \pi_2{}^{\underline{create_\delta\ v} \circ !})|\!)_L$
$= \{ length\text{-DEF} \}$
$length^{create_\delta\ v}$

Some of these properties can be generalised for arbitrary recursive data types. For example, we can define generic mapping lens ([25]) for polymorphic types viewed as fixed points of bi-functors. In the next section, we will discuss how bidirectional laws can be harnessed into a rewrite system for lens optimisation. These list lenses and laws will be particularly useful in order to agilise the definition and optimisation of lenses over lists.

4. Implementation

The main difference between equational reasoning and term rewriting [1] is that bidirectional equations of the form $f = g$ are adapted into unidirectional rewrite rules of the form $f \rightsquigarrow g$ (read f leads to g), indicating that a term f can be substituted by a term g, but not otherwise. For the goal of simplification, the general idea is to substitute terms by simpler terms (for most cases). In our case, this corresponds to view the equational laws from Sections 2,3 as rules oriented from left-to-right. In this section, we explain the implementation (in Haskell) of the rewrite system that is in the core of our lens optimisation tool, walk through a complex transformation scenario, and compare the performance of optimised and non-optimised lenses to demonstrate the usefulness of the tool.

4.1 Mechanising Fusion

Laws like the ones presented in Figure 3 allow us to optimise lenses over recursive data types without using the fusion laws $(\!|\cdot|\!)$-FUSION and $[\!|\cdot|\!]$-FUSION directly. This is particularly useful because this fusion laws imply "guessing" the algebra (or co-algebra) of the resulting fold (or unfold). To be more specific, consider the bidirectional fold fusion law.

$$\delta \circ (\!|\tau|\!)_F = (\!|\phi|\!)_F \Leftarrow \delta \circ \tau = \phi \circ F\ \delta$$

Reading this law as a rewrite rule, in order to perform the reduction $\delta \circ (\!|\tau|\!)_F \rightsquigarrow (\!|\phi|\!)_F$, one must compute a lens ϕ such that $\delta \circ \tau = \phi \circ F\ \delta$ holds. Unfortunately, we cannot always avoid the need to use fusion, and thus some technique must be implemented in order to mechanise it. This research topic received some attention in the past: one of the most successful implementations is the MAG system [28], which views the guessing step as a *higher-order matching* problem. However, MAG is not fully automatic and thus not suitable for our optimisation tool: the user must have some idea of the steps of the proof to provide sufficient hints to proceed with the derivation.

Similarly to [27], our approach is to reduce the hard guessing step to a simple rewriting problem that, although not as general as MAG, is fully automatic and works in practice for many examples. In the above fusion law, if the converse of δ could be computed as δ°, then ϕ could trivially be defined as $\delta \circ \tau \circ F\ \delta^\circ$. Of course, this is just an alternative formulation of the guessing step and useless *per se*. However, if δ° is left opaque (just denoting the tagging of expression δ), and by applying our standard rewrite system, temporarily augmented with rules $\delta \circ \delta^\circ \rightsquigarrow id$ and $\delta^\circ \circ \delta \rightsquigarrow id$, we manage to get rid of δ°, then we get the desired algebra. This

idea is embodied in the following rewrite rule, where we test that δ° does not occur in the normal form of $\delta \circ \tau \circ F\ \delta^\circ$.

$$\delta \circ (\![\tau]\!)_F \rightsquigarrow (\![\phi]\!)_F \Leftarrow \delta \circ \tau \circ F\ \delta^\circ \overset{*}{\rightsquigarrow} \phi \wedge \delta^\circ \notin \phi \quad (\![\cdot]\!)\text{-FUSION}$$

As an example of this technique, Figure 4 presents the rewrite trace for one of the fusions needed to optimise the example in the introduction (indentation in the trace indicates the rewriting of side-conditions). To make the presentation clear, in this trace we mention the inverse of some rules (corresponding to the respective laws oriented from right-to-left). Obviously, to ensure termination, these rules are not encoded as such in our rewrite system. Instead, we have generalised rule versions that covers additional cases such as the following for $distl$-NAT (the definitions of f and g can be easily computed, but are omitted to simplify the presentation):

$$distl \circ (id \times \delta) \rightsquigarrow (id \times \delta + id \times \delta)^{f,g} \circ distl$$

$$distl \circ ((\delta + \tau) \circ \phi \times \psi) \rightsquigarrow (\delta \times \psi + \tau \times \psi)^{f,g} \circ distl \circ (\phi \times id)$$

4.2 Encoding in Haskell

In previous work [25], we have developed the `pointless-lenses` library for defining complex lens transformations as point-free lenses in Haskell. A straightforward method to optimise these lenses would be to specify the laws described in this paper as GHC rewrite rules [18], and allow their use by the GHC compiler. However, this approach provides little control over the rewrite strategy and is not capable of implementing laws such as !-UNIQ and $(\![\cdot]\!)$-FUSION, since it does not support type-directed rewriting nor side-conditions. In order to harness the full power of our algebraic laws, we instead recover a successful type-safe, type-directed rewriting system for transformation of point-free programs [6, 9], and extend it to support bidirectional lenses. Instead of the *shallow embedding* of lenses as Haskell functions used in `pointless-lenses`, this rewrite system makes use of a *deep embedding*, where the objects and arrows of the target lens language are encoded as Haskell datatypes.

Representation of objects and arrows As defined in [9], the typed representation of objects (types and functors) uses *generalised algebraic data types* (GADTs) [19]:

```
data Type a where
  Int   :: Type Int
  One   :: Type 1
  ...
  Prod  :: Type a → Type b → Type (a, b)
  Data  :: String → Functor (F a) → Type a
  Fun   :: Type a → Type b → Type (a → b)
  Lens  :: Type a → Type b → Type (Lens a b)

data Functor (f :: * → *) where
  I     :: Functor Id
  K     :: Type c → Functor c
  (⊠)   :: Functor f → Functor g → Functor (f ⊗ g)
  (⊞)   :: Functor f → Functor g → Functor (f ⊕ g)
  (⊡)   :: Functor f → Functor g → Functor (f ⊙ g)
```

The above definitions provide value-level constructors for base types, sums, products, user-defined types, functions, lenses, and polynomial functors. For example, the value $Prod\ Int\ Int$ represents the type (Int, Int). Note that the $Functor$ value in the definition of $Data$ is not arbitrary: somehow, we must ensure that it is the base functor of the user-defined type a. This relation is established by the *type family* [26] $F\ a$, that acts as a type-level function from types to their base functors, as exemplified for lists:

```
type family F a :: * → *
type instance F [a] = 1 ⊕ a ⊗ Id
```

For example, $Data$ `"[Int]"` $(K\ 1 \boxplus K\ Int \boxtimes I)$ is the representation of datatype $[Int]$. Moreover, when applying a functor to a type, we want to get an isomorphic sum-of-products type capable of being processed with point-free combinators. To this extent, we add the $Rep\ f\ a$ type family that, given a functor f and a type a, returns the equivalent "flat" type:

```
type family Rep (f :: * → *) a :: *
type instance Rep Id a = a
type instance Rep (g ⊗ h) a = (Rep g a, Rep h a)
...
```

Point-free expressions can also be represented in a type-safe manner using a GADT:

```
data PF f where
  Id     :: PF (c → c)
  Fst    :: PF ((a, b) → a)
  Bang   :: PF (a → 1)
  ...
  IdL    :: PF (Lens c c)
  FstL   :: PF (a → b) → PF (Lens (a, b) a)
  BangL  :: PF (1 → c) → PF (Lens c One)
  CataL  :: PF (Lens (Rep (F a) b) b) → PF (Lens a b)
  ...
```

Note that the inhabitants of type $PF\ f$ are the point-free representations of both unidirectional functions and bidirectional lenses.

Rewrite rules and systems In our implementation, rules are represented by monadic type-preserving functions that, given a type representation and a point-free expression, return a new expression of the same type:

```
type Rule = ∀f ∘ Type f → PF f → RewriteM (PF f)
```

$RewriteM$ is a stateful monad that keeps a trace of the applied rules and is an instance of $MonadPlus$, thus modelling partiality in rule application: the monadic function $success$ updates the $RewriteM$ monad to keep trace of a successful reduction; failure is signalled with $mzero$. For example, we can encode a rewrite rule for !-UNIQ as follows:

```
bang_uniq :: Rule
bang_uniq (Lens _ _) (BangL f) = mzero
bang_uniq (Lens a One) l = do
  let createl = createof (Lens a One) l
  g ← optimise_fun (Fun One a) createl
  success "!-Uniq" (BangL g)
bang_uniq _ _ = mzero
```

The first case of this rule avoids a rewriting loop (application of $banqUniq$ to ! itself. The third catch-all case indicates that the rule fails for any other input. The second case reveals the two-layered architecture of our rewrite system: the strategy $optimise\_fun$ simplifies function representations, of the form $PF\ (a \to b)$, and the strategy $optimise\_lens$ rewrites lens representations, of the form $PF\ (Lens\ a\ b)$. To mediate between these two classes, the procedures $getof$,$createof$ and $putof$ take the representation of a lens and return the representations of the corresponding get,put and $create$ functions. As a general methodology, whenever a unidirectional function is created inside a lens rule, we apply $optimise\_fun$ to simplify it.

The rewrite systems themselves are built using strategic term rewriting [20], where the combination of a standard set of basic rules allows the simple design of flexible rewriting strategies. Some standard strategic combinators are $\gg$, $\oslash$ and nop, that encode sequential composition, choice and identity. From these, other combinators can be derived, such as $try\ r = r\ \oslash\ nop$. Examples

$$
\begin{array}{l}
\mathit{length} \circ \mathit{filter\_l} \\
\rightsquigarrow \{\mathit{filter\_l}\text{-DEF}; (\![\cdot]\!)\text{-FUSION}; +\text{-FUNCTOR}\} \\
\quad \mathit{length} \circ (\mathit{in}_L \bigtriangledown \pi_2{}^{\underline{b} \circ !}) \circ \mathit{coassoc} \circ (\mathit{id} + \mathit{distl}) \circ (\mathit{id} + \mathit{id} \times \mathit{length}^\circ)) \\
\quad \rightsquigarrow \{+\text{-FUNCTOR-COMP}; +\text{-FUNCTOR-ID}^{-1}; \mathit{distl}\text{-NAT}; +\text{-FUNCTOR-COMP}^{-1}\} \\
\quad \mathit{length} \circ (\mathit{in}_L \bigtriangledown \pi_2{}^{\underline{b} \circ !}) \circ \mathit{coassoc} \circ (\mathit{id} + (\mathit{id} \times \mathit{length}^\circ + \mathit{id} \times \mathit{length}^\circ)^{(\pi_1 \times \mathit{put}_{\mathit{length}^\circ}) \circ \mathit{distp}, (\pi_1 \times \mathit{put}_{\mathit{length}^\circ}) \circ \mathit{distp}}) \circ (\mathit{id} + \mathit{distl}) \\
\quad \rightsquigarrow \{\mathit{coassoc}\text{-NAT}; +\text{-ABSOR}\} \\
\quad \mathit{length} \circ (\mathit{in}_L \circ (\mathit{id} + \mathit{id} \times \mathit{length}^\circ) \bigtriangledown \pi_2{}^{\underline{b} \circ !} \circ (\mathit{id} \times \mathit{length}^\circ)) \circ \mathit{coassoc} \circ (\mathit{id} + \mathit{distl}) \\
\quad \rightsquigarrow \{ + \text{-FUSION}; \mathit{length}\text{-DEF}; (\![\cdot]\!)\text{-CANCEL}\} \\
\quad (\mathit{in}_N \circ (\mathit{id} + \pi_2{}^!) \circ (\mathit{id} + \mathit{id} \times \mathit{length}) \circ (\mathit{id} + \mathit{id} \times \mathit{length}^\circ) \bigtriangledown \mathit{length} \circ \pi_2{}^{\underline{b} \circ !} \circ (\mathit{id} \times \mathit{length}^\circ)) \circ \mathit{coassoc} \circ (\mathit{id} + \mathit{distl}) \\
\quad \rightsquigarrow \{ + \text{-FUNCTOR-COMP}; \times\text{-FUNCTOR-COMP}; \mathit{length} \circ \mathit{length}^\circ \rightsquigarrow \mathit{id}\} \\
\quad (\mathit{in}_N \circ (\mathit{id} + \pi_2{}^!) \bigtriangledown \mathit{length} \circ \pi_2{}^{\underline{b} \circ !} \circ (\mathit{id} \times \mathit{length}^\circ)) \circ \mathit{coassoc} \circ (\mathit{id} + \mathit{distl}) \\
\quad \rightsquigarrow \{\pi_2\text{-NAT}; \mathit{length} \circ \mathit{length}^\circ \rightsquigarrow \mathit{id}\} \\
\quad\quad \mathit{create}_{\mathit{id}} \circ \underline{b} \circ ! \circ \mathit{get}_{\mathit{length}^\circ} \rightsquigarrow \{\text{definition of } \mathit{create}; !\text{-UNIQ}\}\ \underline{b} \circ ! \\
\quad (\mathit{in}_N \circ (\mathit{id} + \pi_2{}^!) \bigtriangledown \pi_2{}^{\underline{b} \circ !}) \circ \mathit{coassoc} \circ (\mathit{id} + \mathit{distl}) \\
(\![(\mathit{in}_N \circ (\mathit{id} + \pi_2{}^!) \bigtriangledown \pi_2{}^{\underline{b} \circ !}) \circ \mathit{coassoc} \circ (\mathit{id} + \mathit{distl})]\!)_L
\end{array}
$$

Figure 4. Fusion mechanisation example

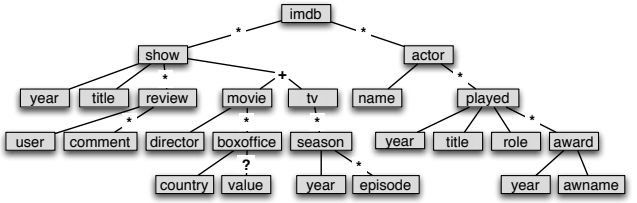

Figure 5. A movie database schema, inspired by IMDb.

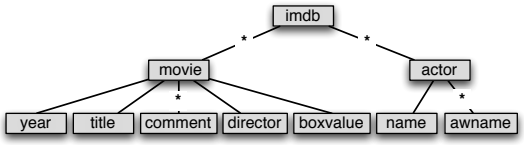

Figure 6. A view of the original schema.

of strategic combinators that traverse the structure of datatypes are *once*, that applies its argument rule exactly once according to a top-down traversal and *outermost*, that performs exhaustive rule application. Using these strategic combinators, we can construct complete transformation strategies, such as

$$
\begin{array}{rl}
\mathit{optimise\_lens} = & \mathit{outermost}\ \mathit{opt} \circledcirc \mathit{rec} \\
\textbf{where}\ \mathit{opt} & = \mathit{nat\_id} \oslash \mathit{bang\_uniq} \oslash ... \\
\mathit{rec} & = \mathit{try}\ (\mathit{once}\ \mathit{fuse} \circledcirc \mathit{optimise\_lens}) \\
\mathit{fuse} & = \mathit{cata\_fusion} \oslash \mathit{ana\_fusion} \oslash ...
\end{array}
$$

This strategy exhaustively applies the set of rewrite rules for lenses described across this paper, and some more. Some of these rules (particularly fusion rules) are evidently more expensive, due to the intermediate rewriting of side-conditions, and are deferred inside the strategy until no other rule can be applied.

4.3 Application scenario

We will now present some examples and compare the performance between automatically optimised lenses and their original point-free specifications. Consider the XML schema shown in Figure 5 for storing information about movies and actors in an IMDb like database. By representing this schema in Haskell (with sequences represented by left-nested tuples, multiple occurrences by lists, choices by left-nested *Either*s, and optional elements by *Maybe*), we can use our lens language to define views of the data. As an example, imagine that we want to summarise the information about movies and actors stored in our IMDb, according to the following lens transformation:

$$
\begin{array}{ll}
\mathit{imdb} & = \mathit{shows} \times \mathit{actors} \\
\mathit{shows} & = \mathit{map}\ ((\mathit{id} \times \mathit{reviews}) \times \mathit{id}) \circ \mathit{filter\_l} \\
& \quad \circ\ \mathit{map}\ \mathit{distr} \circ \mathit{map}\ (\mathit{id} \times (\mathit{movie} + \mathit{tv}))
\end{array}
$$

$$
\begin{array}{ll}
\mathit{reviews} & = \mathit{length}^{\mathit{creview}} \circ \mathit{concat} \circ \mathit{map}\ \pi_2{}^{\mathit{duser}} \\
\mathit{movie} & = \mathit{id} \times \mathit{boxoffices} \\
\mathit{boxoffices} & = \mathit{sum} \circ \mathit{filter\_r} \circ \mathit{map}\ (\mathit{out}_M \circ \pi_2{}^{\mathit{dcountry}}) \\
\mathit{tv} & = \mathit{concat} \circ \mathit{map}\ \pi_2{}^{\mathit{dyear}} \\
\mathit{actors} & = \mathit{map}\ (\mathit{id} \times \mathit{awards}) \\
\mathit{awards} & = \mathit{map}\ \pi_2{}^{\mathit{dyear}} \circ \mathit{concat} \circ \mathit{map}\ \pi_2{}^{\mathit{dytr}}
\end{array}
$$

Here, *duser*, *dyear*, *creview*, ..., denote default functions and values, and *dytr* accounts for $(\mathit{dyear} \bigtriangleup \mathit{dtitle}) \bigtriangleup \mathit{drole}$; $\mathit{out}_M : \mathit{Maybe}\ A \to 1 + A$ is the deconstructor for the type *Maybe A*. The resulting schema is depicted in Figure 6. Our transformation comprises two main lenses applied in parallel to the lists of shows and actors. For each show, we first calculate the total box office value if it is a movie, or collect a list of episodes if it is a TV series. Then, we select only movies, count the number of comments for each review, and return the resulting list of movies. For each actor, we gather a list of all names of the awards he/she has won.

Obviously, the above specification is not very efficient: not only it relies on a heavily compositional style, but it also contains a redundant transformation. Fortunately, our rewrite system can apply several optimisations to this example. For instance, using fold fusion, it is able to reduce the *boxoffices* lens to a single traversal. Also, it can fuse consecutive mappings in *shows* and discard the redundant computation over TV series.

We have measured space and time consumption for this example, and the results are presented in Figure 7. Most of a lens inefficiency comes from the complex synchronising behaviour of its *put* function. Therefore, to better quantify the speedup achieved by our optimisation technique, we have compared the runtime behaviour of the *put* function before (`specification`) and after optimisation (`optimised`). We compiled each function using `GHC 6.12.1` with optimisation flag `O2`. Each example was tested with pre-compiled input databases of increasing size (measured in MBytes needed to store their Haskell definitions), randomly gener-

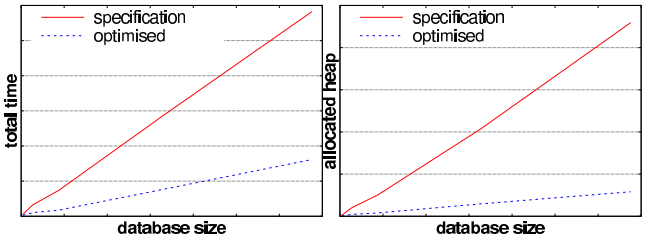

Figure 7. Benchmark results for the $imdb$ example.

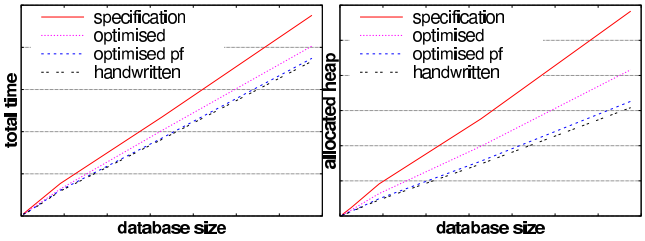

Figure 8. Benchmark results for the $women$ example.

ated with the *QuickCheck* testing suite [5]. As expected, the original specification performs much worse than the optimised lens, by factors of 3.6 in time and 7.9 in space for the biggest sample, and the loss factor grows with the database size.

The reader may question why the benchmark results do not include a comparison with an handwritten definition. The answer is embarrassingly simple: it is extremely complex to hand-code the put function of the $imdb$ lens, let alone an efficient version. However, we did compare the point-free (`specification`) and handwritten (`handwritten`) definitions for the $women$ example from Section 1 and post the results in Figure 8. Below the optimised lens (`optimised`), we introduce another measure for the output of a second optimisation phase performed on the point-free definition of the put function (`optimised pf`). Even for this simple example, the optimised put allocates nearly half the memory and performs very close to the handwritten definition, both in time and space. This additional speedup reported in `optimised pf` is mainly due to the optimisation of expressions involving "opaque" lens isomorphisms, such as $assoc$ or $distl$.

5. Related Work

This paper builds on the work first presented in [25], stating for each lens combinator the laws that characterise it and harnessing these into a lens optimisation framework. This quest for algebraic laws lead to pithy changes in the definitions of some combinators: we generalised the either combinator to subsume the left- and right-biased instances; the parameters of the sum combinator were also generalised but restricted to be *acceptable*, a desired property for calculation; the $i_1 \nabla id$ and $id \nabla i_2$ primitive combinators were eliminated because they can be defined as $(id \,\dot{\nabla}\, id + id) \circ coassoc$ and $(id + id \,\nabla_{\bullet}\, id) \circ coassoc^{-1}$, respectively. The exponentiation combinators are also new to this paper.

Unlike bidirectional interpreters, such as [21, 30], that execute the backward transformation by stepwise interpretation of the forward specification, algebraic bidirectionalisation techniques are concerned with deriving backward implementations by calculation. In [23], researchers from the University of Tokyo propose a point-free language of injective functions whose put functions can be calculated by inverting the specification of get. In [22], they bidirectionalise a restricted first-order functional language based on a notion of view-update under *constant complement*: after deriving a complement function get^c, put can be calculated from the specification $(get \vartriangle get^c)^{-1} \circ (id \times get^c)$, by resorting to tupling and inversion techniques. In principle, the inferred backward transformation could also be optimised using a rewrite system similar to ours, but that path was not explored and no clues are given about how to reason directly at the bidirectional level.

In the context of model-driven development, [29] discusses the inherent problems of existing bidirectional model transformation tools, and [12] proposes an algebraic framework for classifying the properties of bidirectional model transformations. Nevertheless, only consistency properties are considered, and no algebraic laws for calculation and optimization of bidirectional languages are discussed. A lens language for the bidirectionalisation of graph transformations is proposed in [16]. The key construct of this language is the structural recursion operation on graphs, that enjoys a fusion law on the underlying unidirectional graph algebra: two consecutive structural recursions $rec\ e_2 \circ rec\ e_1$ can be replaced by a single structural recursion $rec\ (rec\ e_2 \circ e_1)$ that avoids computing an intermediate result, if the expression e_2 does not depend on its argument graph. This calculational law is applied to the get transformation in [16], before bidirectionalisation. However, unlike in our setting, this optimisation is not stated bidirectionally and may yield different behaviours in the backward transformation.

In independent work, Hoffmann *et al* [17] study the fundamental properties of a symmetric generalisation of traditional lenses, from a category-theoretic perspective. Many of their results corroborate our own conclusions, such as the existence of tensor products and sums (but not categorical sums and products), and the ability to define recursive lenses with folds and unfolds that satisfy uniqueness (given certain termination considerations similar to [25]). Unlike us, categorical exponentiations were not studied and they have not shown how to harness their results into an effective lens optimisation framework.

In previous work, we have proposed a two-level bidirectional framework (2LT) for the point-free specification of *data refinement* transformations [2, 10]. Since the synchronisation behaviour is much simpler, optimisation of these bidirectional transformations can be done independently at the functional level for each component of the transformation [8, 9]. Here we not only tackle the dual problem of *data abstraction*, but also perform optimisation directly at the bidirectional level, due to the high complexity of composite put transformations. In the future, we intend to incorporate our results into the 2LT framework, thus enlarging the scope of model transformation scenarios to which it can be applied.

6. Conclusion

In this paper, we proposed an equational calculus to reason directly about lenses defined in the point-free style. This calculus allows us to hide the complexity of backward transformations, and calculate with lenses by reasoning only about the simple forward specification. The main result is the existence of uniqueness for bidirectional folds and unfolds, thus unleashing the power of fusion to optimise bidirectional programs, while preserving their semantics.

To prove the usefulness of this calculus, we have emloyed it at the kernel of an automatic optimisation tool for point-free lenses. This tool is implemented as an extension of a previous rewrite system [9], and both the updated lens library and the rewrite system for lens optimisation (including the encoding of the algebraic laws) are available through the Hackage package repository under the names `pointless-lenses` and `pointless-rewrite`, honouring a common joke about the point-free style.

In our current language, the nonexistence of splits and injections triggered the definition of several opaque primitive isomorphism combinators to regain expressiveness. In order to alleviate this

problem, we are considering migrating to a point-free relational setting (as shown by Oliveira [24]). In particular, we intend to explore the calculation of invariants over target data structures, so that the above-mentioned combinators can be defined as total lenses for particular domains, with corresponding algebraic laws holding for such domains. Similarly, in *Focal* these restricted domains are built into the framework and enforced by a complex set-based type system, for each lens combinator. We also intend to use the powerful relational calculus to derive backward transformations that are correct by construction, thus avoiding the well-behavedness proofs (an approach similar to [22], but in a relational setting).

Lenses, as a framework for data abstraction, could also be of great value in scything through the complexity of large software systems [13]. However, in order to express transformations over these systems, structure-shyness of the specifications is imperative to reduce the specification cost and foster the reusability. To this extent, it would be interesting to extend our point-free bidirectional language with strategic lens combinators: this could enable the bidirectionalisation and subsequent optimisation of existing generic functional programs and structure-shy querying languages such as XPath.

References

[1] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[2] P. Berdaguer, A. Cunha, H. Pacheco, and J. Visser. Coupled Schema Transformation and Data: Conversion for XML and SQL. In *Proceedings of the 9th International Symposium on Practical Aspects of Declarative Languages*, volume 4085 of *LNCS*, pages 290–304. Springer, 2007.

[3] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *Proceedings of the 25th ACM SIGMOD Symposium on Principles of Database Systems*, pages 338–347. ACM, 2006.

[4] A. Bohannon, J. N. Foster, B. C. Pierce, A. Pilkiewicz, and A. Schmitt. Boomerang: resourceful lenses for string data. In *Proceedings of the 35th ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 407–419. ACM, 2008.

[5] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the 5th ACM SIGPLAN international conference on Functional programming*, pages 268–279. ACM, 2000.

[6] A. Cunha and H. Pacheco. Algebraic Specialization of Generic Functions for Recursive Types. In *Proceedings of the 2nd Workshop on Mathematically Structured Functional Programming*, 2008.

[7] A. Cunha and J. S. Pinto. Point-free program transformation. *Fundamenta Informaticae*, 66(4):315–352, 2005.

[8] A. Cunha and J. Visser. Strongly Typed Rewriting For Coupled Software Transformation. *Electronic Notes in Theoretical Computer Science*, 174(1):17–34, 2007.

[9] A. Cunha and J. Visser. Transformation of structure-shy programs with application to XPath queries and strategic functions. *Science of Computer Programming*, In Press, Corrected Proof, 2010.

[10] A. Cunha, J. N. Oliveira, and J. Visser. Type-safe Two-level Data Transformation. In *Proceedings of the 14th International Symposium on Formal Methods*, volume 4085 of *LNCS*, pages 284–299. Springer, 2006.

[11] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, volume 5563 of *LNCS*, pages 260–283. Springer, 2009.

[12] Z. Diskin. Algebraic Models for Bidirectional Model Synchronization. In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems*, volume 5301 of *LNCS*, pages 21–36. Springer, 2008.

[13] A. Egyed. Automated abstraction of class diagrams. *ACM Transactions on Software Engineering and Methodology*, 11(4):449–491, 2002.

[14] 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.

[15] S. Hidaka, Z. Hu, H. Kato, and K. Nakano. Towards a compositional approach to model transformation for software development. In *Proceedings of the 24th ACM Symposium on Applied Computing*, pages 468–475. ACM, 2009.

[16] S. Hidaka, Z. Hu, K. Inaba, H. Kato, K. Matsuda, and K. Nakano. Bidirectionalizing graph transformations. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 205–216. ACM, 2010.

[17] M. Hofmann, B. C. Pierce, and D. Wagner. Symmetric Lenses, 2010. Submitted for publication.

[18] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, pages 203–233. ACM, 2001.

[19] S. P. Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the 11th ACM SIGPLAN international conference on Functional programming*, pages 50–61. ACM, 2006.

[20] R. Lämmel. Typed generic traversal with term rewriting strategies. *Journal of Logic and Algebraic Programming*, 54(1-2):1 – 64, 2003.

[21] D. Liu, Z. Hu, and M. Takeichi. Bidirectional interpretation of XQuery. In *Proceedings of the 2007 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 21–30. ACM, 2007.

[22] K. Matsuda, Z. Hu, K. Nakano, M. Hamana, and M. Takeichi. Bidirectionalization transformation based on automatic derivation of view complement functions. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 47–58. ACM, 2007.

[23] S.-C. Mu, Z. Hu, and M. Takeichi. An Algebraic Approach to Bi-Directional Updating. In *Proceedings of the 2nd Asian Symposium on Programming Languages and System*, volume 3302 of *LNCS*. Springer, 2004.

[24] J. N. Oliveira. Data Transformation by Calculation. In *Proceedings of the 2nd International Summer School on Generative and Transformational Techniques in Software Engineering*, volume 5235 of *LNCS*, pages 139–198. Springer, 2007.

[25] H. Pacheco and A. Cunha. Generic Point-free Lenses. In *Proceedings of the 10th International Conference on Mathematics of Program Construction*, volume 6120 of *LNCS*, pages 331–352. Springer, 2010.

[26] T. Schrijvers, S. M, S. P. Jones, and M. T. Chakravarty. Towards Open Type Functions for Haskell. In *Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages*, pages 233–251, 2007.

[27] T. Sheard and L. Fegaras. A fold for all seasons. In *Proceedings of the 1993 conference on Functional Programming Languages and Computer Architecture*, pages 233–242. ACM, 1993.

[28] G. Sittampalam and O. de Moor. Mechanising Fusion. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, chapter 5, pages 79–104. Palgrave Macmillan, 2003.

[29] P. Stevens. Bidirectional model transformations in QVT: Semantic issues and open questions. In *Proceedings of the 10th International Conference on Model Driven Engineering Languages And Systems*, volume 4735 of *LNCS*, pages 1–15. Springer, 2007.

[30] J. Voigtländer. Bidirectionalization for free! (Pearl). In *Proceedings of the 36th Annual ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 165–176. ACM, 2009.

[31] J. Voigtländer., Z. Hu, K. Matsuda, and M. Wang. Combining Syntactic and Semantic Bidirectionalization. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, pages 181–192. ACM, 2010.