Towards Optic-Based Algebraic Theories: the Case of Lenses*

J. López-González[1,2] and Juan M. Serrano[1,2]

[1] Habla Computing, S.L.
[2] Universidad Rey Juan Carlos
j.lopezgo@alumnos.urjc.es
juanmanuel.serrano@urjc.es

Abstract Optics provide rich abstractions and composition patterns to access and manipulate immutable data structures. However, the state of real applications is mostly handled through databases, caches, web services, etc. In this effectful setting, the usefulness of optics is severely limited, whereas algebraic theories, thanks to their potential to abstract away from particular infrastructures, shine. Unfortunately, there is a severe lack of standard algebraic theories, e.g. like MonadState, that programmers can reuse to avoid writing their domain repositories from scratch. This paper argues that optics can serve as a fruitful metaphor to design a rich catalogue of state-based algebras, and focuses on the paradigmatic case of lenses. It shows how lenses can be generalised into an algebraic theory; how compositionality of these algebraic theories can be founded on lens composition; and how to exploit the resulting abstractions in the modular design of data layers. The paper systematically uses Coq for all its definitions and proofs.

Keywords: lenses, optics, algebras, monads, state, repositories, coq

1 Introduction

Optics provide rich abstractions and patterns to access and manipulate immutable data structures. They are also known as *functional references*, since they point at *parts* which are contained or determined by a *whole*. For instance, a *lens* [1] is an optic that points at a single value which is always available from the context; an *affine* points at a single value that is not necessarily available; a *traversal* [2] points at a sequence of values; and so on. Each optic comes equipped with an interface which is specialized in evolving the whole data structure by evolving the parts it is pointing at. We will use the following records to illustrate the idea, where the details about `person` are irrelevant[3]:

```
Record department := mkDepartment
{ budget : nat
```

* Research Article. Main author is a research student.
[3] Coq will be used throughout the paper.

```
; lecturers : list person }.

Record university := mkUniversity
{ name : string
; departments : list department
; students : list person }.
```

For instance, we could define `budgetLn`, a lens that points at the `budget` field of `department`[4], and invoke its modifying operator ($\sim_{ln}$) to duplicate its budget:

```
Definition duplicateDepBudget : department → department :=
  budgetLn ∼ln (λ b ⇒ b * 2).
```

We may also define a lens for fields that refer to a sequence of values such as `departments` or `lecturers`. For instance, a lens `departmentsLn` would allow us to access and modify the *whole* list of departments of a given university. But traversals offer a much better interface in this case. In particular, given a traversal `departmentTr` and the modifying method provided by traversals ($\sim_{tr}$), we may duplicate the budget of every single department as follows:

```
Definition duplicateUnivBudgets : university → university :=
  departmentsTr ∼tr duplicateDepBudget.
```

One of the most attractive features of optics is that they compose heterogeneously. For instance, we may use the combinator $\triangleright_{tr}$ to compose a lens and a traversal to obtain a wider traversal, and, similarly, we may use the combinator $\triangleright_{ln}$ to compose a traversal with a lens to obtain a finer-grained traversal. As an example, consider the following alternative implementation of the last function, where we emphasize composability:

```
Definition duplicateUnivBudgets : university → university :=
  (departmentsLn ▷tr each ▷ln budgetLn) ∼tr (λ b ⇒ b * 2).
```

Here, we show how to create a traversal that points at each single department budget, by combining two lenses with *each*, a widespread traversal that points to the elements of a given list.

In sum, we can appreciate two major benefits from optics, namely the diversity of abstractions and their compositional capabilities. However, optics are restricted to work with immutable data structures, whereas the state of real applications, like university information systems, is mostly handled through databases, web services, caches and so forth. In this setting, the optic approach to handle state is not useless, but severely limited. Instead, one of the most prominent techniques in the realm of effectful systems are algebraic theories [3], which we refer to as algebras.

Basically, algebras allow us to strive for generality and define a *data layer* that abstracts away from the particular infrastructures that actually handle state (caches, databases, immutable state transformers, etc.). For instance, we could

[4] For the moment, we will ignore the details about lens representations.

build an *ad hoc* repository algebra to deal with university state in a general way. We encode it as a typeclass [4]:

```
Class UniversityAlg (p : Type → Type) :=
{ getName : p string
; modifyName (f : string → string) : p unit
; getDepartments : p (list department)
; modifyDepartments (f : department → department) : p unit }.
```

This algebra defines the class of computational effects `p` that we may use to access and manipulate the name and departments of a given university. Building our business logic upon algebraic repositories of this kind is great, since we remain unaware of particular infrastructures details. Unfortunately, this approach has several difficulties, that we show in the next paragraphs.

Firstly, despite the fact that `UniversityAlg` hides the state of the university behind `p`, some of their methods are exposing the immutable `department` structure in their signatures. The information associated to a department could be large enough to make it non-optimal or even impractical to instantiate such a value. In these circumstances, the obvious choice is describing the data layer of a department in a separate algebra, which would describe the accessors to the department fields, and would have its own computational effect `q`. The problem with this arrangement is that it becomes actually very cumbersome to compose the heterogeneous programs `p`, `q`, etc., generated by the different algebras, specially when a large number of them are involved.

Secondly, we are not using standard abstractions to describe the university data layer, and therefore `UniversityAlg` contains very fine-grained methods. This contrasts with the optic approach, where lens or traversal provide standardized means to get or modify fields of domain entities. In this regard, we could replace both `getName` and `modifyName` with a `MonadState` evidence with focus on the name. This standard algebra fits perfectly to view and update this field in a general way. However, it does not provide specialized methods to handle sequences of values. Unluckily, we do not know of any other standard algebra, analogous to traversal, to deal with multiple values at this abstract level. Even if we had these standard algebras available, it would not be easy to determine a notion of composition for them.

In this paper, we argue that optics can serve as a fruitful metaphor to design a rich catalogue of state-based algebras, that serve as an *embedded domain specific language* (EDSL) [5] for the implementation of data layers, aimed at the software industry. This EDSL would allow programmers to write code at an algebraic level of abstraction while using a rich catalogue of standard and composable abstractions, analogous to that supplied by optics.

This paper sets out to establish the essential pillars in pursuit of this golden inventory of so-called optic algebras. In particular, it focuses on lenses and `MonadState` as the playing field to expose the paradigmatic relations between optics and state-based algebras. Moreover, we also aim at identifying a methodology that can be applied to the design of other state-based algebras, analogous to traversals, affines, and so forth. These are the main contributions of the paper:

- We show that `MonadState` strictly generalises lenses. Particularly, we show that lenses can be represented as the state monad instance of this algebra. Since `MonadState` distills the algebraic essence of a lens, we will refer to it as *lens algebra* (Sect. 3).
- We show how to generalise the state monad morphism representation of lenses [6] into *lens algebra homomorphisms* (Sect. 4). This abstraction enables composition between lens algebras.
- We provide a design pattern to implement the data layer and the business logic of applications, using lens algebras (Sect. 3.1) and lens algebra homomorphisms (Sect. 4.1) as building blocks. We use the university example as a guide for this task.

As mentioned in the last paragraphs, Sect. 3 and Sect. 4 contain the bulk of the paper. Section 5 reviews related work in the context of our goals, particularly monadic lenses [7], entangled state monads [6,8] and profunctor optics [9]. Section 6 concludes by outlining a methodology to generalize other optics into their optic algebra counterparts, and points to *Stateless*, a work-in-progress EDSL implemented in Scala that aims to bring optic algebras to the software industry. All definitions, examples and propositions in this paper have been formalized using Coq and are supplied as complementary material in Sect. A. Before going into further detail, we will provide a brief background in Sect. 2.

2 Background

For the sake of brevity, we will illustrate the background definitions with an idealized version of Coq, where we adopt a notation closer to math. For instance, *fun* is replaced by λ , *forall* is replaced by $\forall$, and so on. In addition, we deliberately ignore *level* and *associativity* from `Notation` for simplicity. Additional adjustments will be mentioned as they arise.

2.1 Natural Transformations

Definition 1 (natural transformation). *Given functors `f` and `g`, a natural transformation [10] is a family of morphisms that turns objects on `f` into objects on `g`. We can represent it as a polymorphic function.*

```
Class natTrans f g `{Functor f, Functor g} := mkNatTrans
{ runNatTrans : ∀ X, f X → g X }.
Notation "f ⇝ g" := natTrans f g.
```

The transformation must preserve the following commuting property.

```
Class natTransLaws f g `{Functor f, Functor g} (φ : f ⇝ g) :=
{ natTrans_comm : ∀ A B (fa : f A) (g : A → B),
                    φ (fmap g fa) = fmap g (φ fa) }.
```

Note that we replaced `runNatTrans φfx` by `φ fx` to improve the readability of definitions. Natural transformations are composable.

```
Definition composeNT {f g h} `{Functor f, Functor g, Functor h}
                     (φ : g ⇝ h) (ψ : f ⇝ g) : f ⇝ h :=
  mkNatTrans (λ _ fx ⇒ φ (ψ fx)).
Notation "φ · ψ" := composeNT φ ψ
```

The following definition arises naturally when we pay attention to monadic functors.

Definition 2 (monad morphism). *A monad morphism is a natural transformation between monadic type constructors that also satisfies these laws.*

```
Record monad_morphism {f g} `{Monad f, Monad g} (φ : f ⇝ g) :=
{ returnMap : ∀ X (x : X),
                φ (ret x) = ret x
; distrBind : ∀ A B (fa : f A) (f : A → f B),
                φ (fa ≫= f) = φ fa ≫= (λ a ⇒ φ (f a)) }.
```

Broadly, these laws are telling us that a monad morphism should distribute over ≫= and that mapping `ret` from the original monad should produce `ret` in the destination monad. In effect, a monad morphism allows us to *push* whole computations down into `g` and lifts afterwards the result back to `f`.

2.2 State

We recall the state monad [11]:

Definition 3 (state monad). *State is a data type that transforms a value into a new version of it, while providing an additional output along with the transformed value.*

```
Record state (S Out : Type) := mkState
{ runState : S → Out * S }.
```

State transformations can be composed using `Monad` *combinators:*

```
Instance Monad_state {S : Type} : Monad (state S) :=
{ ret  := λ A o ⇒ mkState (λ s ⇒ (o, s))
; bind := λ A B m f ⇒ mkState (λ s ⇒ let (o, s') := runState m s
                                    in runState (f o) s') }.
```

The output provided by a state transformation can be used to include additional information about the transformation which is taking place.

We also provide a pair of convenience methods which will be helpful in further definitions:

```
Definition evalState {S Out} (st : state S Out) (s : S) : Out :=
  fst (runState st s).

Definition execState {S Out} (st : state S Out) (s : S) : S :=
  snd (runState st s).
```

2.3 Lens

As stated in [1], lenses approach the view update problem for tree-structured data.

Definition 4 (asymmetric monomorphic lens). *An asymmetric* lens *consists of a pair of functions, one of them* views *the part from the whole and the other one* updates *the part from an old whole, returning a new whole. This is the concrete monomorphic lens representation:*

```
Record lens (S A : Type) := mkLens
{ view : S → A
; update : S → A → S
; modify (f : A → A) : S → S := λ s ⇒ update s (f (view s)) }.
Notation "ln ∼ln f" := modify ln f.
```

The type parameters S and A serve as the whole and the part, respectively. Lens operations must obey certain laws [12] to be considered very well-behaved[5]*:*

```
Record very_well-behaved {S A} (ln : lens S A) :=
{ view_update :   ∀ s,       update ln s (view ln s) = s
; update_view :   ∀ s a,     view ln (update ln s a) = a
; update_update : ∀ s a1 a2, update ln (update ln s a1) a2 =
                             update ln s a2 }.
```

From now on, we will assume asymmetric very well-behaved monomorphic lenses when we refer to *lens*, unless specified otherwise.

The very well-behaved class of lenses forms a category, where lenses position themselves as morphisms. The identity lens is the one where part and whole correspond to the same value:

```
Definition identityLn {A} : lens A A :=
  mkLens id (λ _ ⇒ id).
```

The composition of concrete lenses is clumsily achieved by the following function:

```
Definition composeLn {S A B}
    (ln1 : lens S A) (ln2 : lens A B) : lens S B :=
  mkLens (view ln2 · view ln1)
         (λ s a′ ⇒ update ln1 s (update ln2 (view ln1 s) a′)).
Notation "ln1 ▷ln ln2" := composeLn ln1 ln2.
```

There are other lens representations that improve composability, the most popular ones being *van laarhoven* [2] and *profunctor* [9] approaches. In this paper, however, our preferred choice is a lens representation based on monad morphisms [6], that also enjoys similar benefits with respect to compositionality:

Lemma 1. *A monad morphism* `state A ⇝state S` *is isomorphic to a very well-behaved* `lens S A`.

```
Definition lens′ (S A : Type) := state A ⇝ state S.
```

[5] If we discard `update_update`, we can talk about well-behaved lenses.

Informally, the state monad morphism representation can be understood as a morphism from a program that evolves the part into a program that evolves the whole.

2.4 MonadState

We introduce now the `MonadState` typeclass [3], as an algebra to manipulate state in a general way:

Definition 5 (MonadState typeclass). *MonadState classifies all those effects which are able to access and manipulate an inner state. It supplies a pair of methods, the first of them* gets *the current state and the second one* puts *a new state, by replacing the existing one.*

```
Class MonadState (A : Type) (m : Type → Type) `{Monad m} :=
{ get : m A
; put : A → m unit }.
```

These are the properties that should be held by `MonadState` *instances to be considered lawful:*

```
Record MonadState_Laws {A m} `{MonadState A m} :=
{ get_get : get ≫= (λ s1 ⇒ get ≫= (λ s2 ⇒ ret (s1, s2))) =
            get ≫= (λ s ⇒ ret (s, s))
; get_put : get ≫= put = ret tt
; put_get : ∀ s, put s ≫ get = put s ≫ ret s
; put_put : ∀ s1 s2, put s1 ≫ put s2 = put s2 }.
```

Among the effects that are able to instantiate this typeclass, `state` is probably the most widespread one:

```
Instance MonadState_state {S} : MonadState S (state S) :=
{ get := mkState (λ s ⇒ (s, s))
; put := λ s ⇒ mkState (λ _ ⇒ (tt, s)) }.
```

Note how the state hidden by `state S` matches with the focus `S` we are providing access to via the `MonadState` interface.

3 The Algebraic Theory for Lenses

There is a degree of overlap and a number of similarities between `lens S A` and `MonadState A m`. Firstly, they provide analogous methods: both `view` and `get` share the notion of reading a value from the current state; analogously, both `update` and `put` share the notion of writing or replacing the state with a new version of it. Secondly, both abstractions declare a set of properties that must be held by their instances to be considered lawful. Letting aside `get_get` from `MonadState`, it is easy to observe a strong correspondence among the rest of them. Thirdly, the university example from Sect. 1 enabled us to appreciate that both `lens` and `MonadState` are suited for accessing and manipulating a

single state which is always available from the context. Throughout this section, we will show the precise connections between them, starting with an informal derivation to pave the way.

If we pay attention to `view : S →A` and `update : S →A →S` from lens, we can see that both methods contain an origin source `S` as first parameter. However, the rest of the signature does not look homogeneous. In fact, `view` is producing an output `A` and `update` requires an additional input `A` and produces a new version of the state `S` as result. Now, we will try to homogenise both methods, so they both contemplate the notion of input, output and resulting state, aiming at abstracting away the common parts. We show the process with the following informal derivation[6]:

```
  ( view   : S      → A
  , update : S → A →     S)
≃ [functional extensionality]
  ( λ s   ⇒ view s     : S      → A
  , λ s a ⇒ update s a : S → A →     S)
≃ [add contrived input to view]
  ( λ s _ ⇒ view s     : S → 1 → A
  , λ s a ⇒ update s a : S → A →     S)
≃ [add contrived resulting state to view]
  ( λ s _ ⇒ (view s, s) : S → 1 → A * S
  , λ s a ⇒ update s a  : S → A →     S)
≃ [add contrived output to update]
  ( λ s _ ⇒ (view s, s)      : S → 1 → A * S
  , λ s a ⇒ (tt, update s a) : S → A → 1 * S)
≃ [flip parameters]
  ( λ _ s ⇒ (view s, s)      : 1 → S → A * S
  , λ a s ⇒ (tt, update s a) : A → S → 1 * S)
≃ [abstract with state monad]
  ( λ _ ⇒ mkState (λ s ⇒ (view s, s))      : 1 → state S A
  , λ a ⇒ mkState (λ s ⇒ (tt, update s a)) : A → state S 1)
```

As can be seen, this normalisation process leads us to a pair of kleisli arrows for the state monad with `S` as inner value. Having detected this common structure, we could abstract away `state S` from their signatures, which would result in `1 →m A` for the derived `view` and `A →m 1` for the derived `update`. These are exactly the signatures of the methods `get` and `put` supplied by `MonadState`. So, thinking of lens as an instance of `MonadState` – where `A` is the focus and `state S` is the monadic effect – seems plausible. Indeed, converting concrete lenses into `MonadState` instances, and vice versa, turns out to be straightforward.

```
Instance lens_2_ms {S A} (ln : lens S A)
    : MonadState A (state S) :=
{ get   := mkState (λ s ⇒ (view ln s, s))
; put a := mkState (λ s ⇒ (tt, update ln s a)) }.

Definition ms_2_lens {S A} (ms : MonadState A (state S))
```

[6] We represent `unit` as `1` to simplify signatures.

```
    : lens S A :=
{| view s     := evalState get s
;  update s a := execState (put a) s |}.
```

These methods evidence the following statement.

Proposition 1. *There is an isomorphism between any lawful instance of* `MonadState A (state S)` *and a very well-behaved* `lens S A`*.*

We claim therefore that `MonadState` is a generalisation of `lens`. Although we have not seen this connection elsewhere, it seems to be evident in [6], where `BX` is instantiated with `state` to recover (a)symmetric lenses. Broadly, `MonadState` captures the algebra to view and update a single state which is always available, but at a higher level of abstraction. This is the reason why we will refer to `MonadState` as *lens algebra*.

Definition 6. *Lens algebra is an alternative way of referring to* `MonadState`*, emphasizing the fact that it generalizes a* `lens` *by distilling its algebraic essence.*

```
Record lensAlg (p : Type → Type) (A : Type) `{M : Monad p} :=
{ view : p A
; update : A → p unit
; modify (f : A → A) : p unit := view ≫= (update · f) }.
Notation "ln ∼ln f" := modify ln f.
```

Very well-behaved lens algebra laws are exactly `MonadState` *laws.*

```
Record lensAlgLaws {p A} `{Monad p} (ln : lensAlg p A) :=
{ view_view :
    view ln ≫= (λ s1 ⇒ view ln ≫= (λ s2 ⇒ ret (s1, s2))) =
    view ln ≫= (λ s ⇒ ret (s, s))
; view_update : view ln ≫= update ln = ret tt
; update_view : ∀ s, update ln s ≫ view ln = update ln s ≫ ret s
; update_update : ∀ s1 s2, update ln s1 ≫ update ln s2 =
                           update ln s2 }.
```

If we discard `update_update` *from the set we can talk about well-behavelens algebras.*

Remark 1. Normally, instances of `MonadState` where the focus and the state hidden by the effect differ are unusual in the functional programming community. This is probably a consequence of the functional dependency between the effect and the focus that some languages impose to this typeclass in order to avoid ambiguity while resolving instances [13]. We are not interested in resolving lens algebras implicitly, that is the reason why we use a `Record` to represent them.

Section 2.4 showed `MonadState_state` as the most widespread instance of `MonadState`. Its type is `MonadState S (state S)`, where the types of focus and inner value of `state` match. In fact, this instance corresponds to a well-known lens.

Corollary 1. `MonadState_state` *(Sect. 2.4) is isomorphic to* `identityLn` *(Sect. 2.3).*

Once we have positioned lens algebra, or `MonadState`, as a standard abstraction, analogous to lens, but at a more general setting, we will use it to implement the data layer of the university example.

3.1 Data Layer Design with Lens Algebras

Since this paper put focus on lenses, we will simplify the original example, ignoring traversable structures. Thereby, instead of having a list of departments hanging from the university, we will assume a unique math department in the university repository:

```
Record UniversityAlg p Univ `{id : MonadState Univ p} :=
{ getName : p string
; modName (f : string → string) : p unit
; getMathDep : p department
; modMathDep (f : department → department) : p unit }.
```

This is similar to the repository we presented in Sect. 1, but there are several differences. Firstly, we use a record instead of a class, for analogous reasons to the ones mentioned in Remark 1. Secondly, we supply a new type parameter `Univ`, which appears in the signature of the implicit parameter `id` exclusively. This parameter positions itself as the identity lens algebra, i.e. the one where the inner state of `p` and the focus `Univ` match. `Univ` seems contrived at this point, but it will be relevant when we approach composition further on.

Lens algebras encapsulate the functionality that we need to get or modify a particular field. Thereby, we replace the corresponding fine-grained accessors from the university repository with this abstraction.

```
Record UniversityAlg p Univ `{id : MonadState Univ p} :=
{ nameLn : lensAlg p string
; mathDepLn : lensAlg p department }.
```

Moreover, as introduced in Sect. 1 it is not recommended to expose the immutable structure `department` in the data layer. To avoid that, we should segregate department into a new repository, and provide an evidence to it from the university algebra. It would result in the following data layer:

```
Record DepartmentAlg p Dep `{id : MonadState Dep p} :=
{ budgetLn : lensAlg p nat }.

Record UniversityAlg p Univ `{id : MonadState Univ p} :=
{ nameLn : lensAlg p string
; q : Type → Type
; Dep : Type
; ev : `{DepartmentAlg q Dep}
; mathDepLn : lensAlg p Dep }.
```

As a result, we get a new algebra `DepartmentAlg` parameterized with its own effect `p` and a concrete type `Dep`. For its part, `UniversityAlg` has been

extended to establish the link to the new algebra[7]. Note that `Dep` does not necessarily correspond to the data structure `department`. Instead, it determines the minimal information that we need to know in order to interoperate with the particular infrastructure. For instance, it could be an *url*, the primary key in a database table, or any other kind of index. The same intuition applies to `Univ`.

Given this implementation of the data layer, we should be able to program the business logic that duplicates the budget of a department.

```
Definition duplicateDepBudget p Dep
   `{MonadState Dep p}
    (data : DepartmentAlg p Dep) : p unit :=
  budgetLn data ∼ln (λ b ⇒ b * 2).
```

As can be seen, we have to supply the department repository as an additional parameter. Then, we simply extract the lens algebra from it and invoke its modifying operator. The result of this function is a `p unit`, a program that achieves a modification over the department and outputs nothing.

We could also be interested in implementing the logic to duplicate the university budgets[8]. To do so, we should be able to compose `mathDepLn` with `budgetLn` somehow. However, we do not have the means to adapt `q` programs, the ones that evolve a department, into `p` programs, the ones that evolve the whole university. We need new abstractions, close to natural transformations, to enable this kind of composability, which will be introduced in the next section.

4 Composable Lens Algebras

So far, we have taken concrete lenses to a more general setting by abstracting away `state S` from them, resulting in lens algebras. This abstraction provides an analogous interface to the one we find in lenses, but it does not contemplate the notion of composition. On the other hand, Sect. 1 introduced an alternative lens representation with better composition guarantees, encoded as a state monad morphism. In this section, we will achieve an analogous abstraction over this representation, aiming at enabling composition between lens algebras at this general, algebraic setting. Equipped with the new abstractions, we will evolve the university data layer to its final version, which will allow us to implement the logic to duplicate the university budgets.

Fortunately, the alternative lens representation contains an explicit mention to `state S`, so we do not need additional derivations to be able to abstract this structure away. As a result, we get a new definition.

Definition 7. *`lensAlg'` is a monad morphism `state A ⇝p` that adapts `state A` programs – which evolve the focus – into `p` programs – that contextualize and evolve the whole.*

[7] Here, we assume that `ev` also collects evidences for `Monad` and `MonadState`.
[8] Note that this would only involve the math department in the new configuration.

```
Definition lensAlg′ (p : Type → Type) (A : Type) `{Monad p} :=
  state A ⇝ p.
```

As the name suggests, this abstraction is directly connected with the original lens algebra, and hence `MonadState`. Indeed, we could recover `view` and `update` methods by turning the new representation into the original one:

```
Definition lensAlg′_2_lensAlg {p A} `{Monad p}
  (φ : lensAlg′ p A) : lensAlg p A :=
{| view      := φ (mkState (λ a ⇒ (a, a)))
;  update a′ := φ (mkState (λ a ⇒ (tt, a′))) |}.
```

The connection also holds in the opposite direction.

Lemma 2. *There is an isomorphism between a lawful* `lensAlg p A` *and a monad morphism* `lensAlg′p A`.

Despite having defined an alternative representation for lens algebra which consists of a unique morphism, we face severe limitations when we try to compose a pair of them. Particularly, we need to fix the type of program for the second lens algebra to `state X`, where `X` corresponds to the type of focus for the first lens algebra. By doing so, the only program that we are able to generalize is the one which corresponds to the leftmost lens algebra in the composition chain. We need to overcome this strong limitation.

We have abstracted away `state S` from the alternative lens definition, but there remains an additional reference to `state A` in the resulting abstraction. Is it feasible to abstract it away as well? Interestingly, the state programs that are passed through the monad morphism in `lensAlg′_2_lensAlg` are exactly the ones that we defined in `MonadState_state` for `get` and `set`. Given this situation, we could abstract the reference to `state A` away while retaining the lens algebra interface, as long as we supply a `MonadState` evidence in exchange for it.

Definition 8. *Lens algebra homomorphisms abstract away any reference to* `state A` *from* `lensAlg′p A`*, resulting in a new higher kinded type parameter* `q` *which must be an instance of* `MonadState A q`.

```
Definition lensAlgHom p q A `{Monad p} `{MonadState A q} :=
  q ⇝ p.
```

The new definition does not contain explicit mentions to `state` any more, providing a general `q` instead. The `MonadState` evidence is all that we need to recover the interface of lens algebras.

```
Definition lensAlgHom_2_lensAlg {p q A}
 `{Monad p} `{MonadState A q}
  (φ : lensAlgHom p q A) : lensAlg p A :=
{| view      := φ get
;  update a′ := φ (put a′) |}.
```

Proposition 2. *A lens algebra homomorphism* `lensAlgHom p q A` *induces a lawful lens algebra* `lensAlg p A`*.*

As we can see, φ maps `get` into `view` and `put` into `update`. If we take Definition 6 into account, by doing so, we are actually mapping a lens algebra into another one. That is the reason why we refer to the new abstraction as *lens algebra homomorphism*. Particularly, we are turning an instance of `lensAlg q A` —or `MonadState A q`— into an instance of `lensAlg p A`, where the type of programs differ, but the focus `A` is exactly the same. In fact, the homomorphism is turning `q` programs that evolve the focus from certain context into `p` programs that evolve the very same focus from a broader one. This idea reassembles the notion of whole and part from ordinary lenses.

Remark 2. If lens algebra is an alternative way of representing `MonadState`, why do not we use the former in the definition of lens algebra homomorphisms? As we said in Remark 1, and contrary to lens algebras, `MonadState` usually declares a functional dependency that evidences that the type of program determines the type of focus. This is exactly the behaviour that we are interested in for the homomorphism constraint, since we want to keep the inner `q` program as close to the focus as possible. In this context, it is fine to exploit the mechanism for implicit typeclass resolution provided by the language.

Having abstracted `state A` from the new definition, composition becomes trivial. In fact, it is basically natural transformation composition:

```
Definition composeLnAlgHom {p q r A B}
   `{MonadState B r} `{MonadState A q} `{Monad p}
    (φ : lensAlgHom p q A)
    (ψ : lensAlgHom q r B) : lensAlgHom p r B :=
  φ · ψ.
Notation "hom1 ▷ln hom2" := composeLnAlgHom hom1 hom2
```

The composition function is closed under composition and preserves identity and associativity laws, which leads us to the following lemma.

Corollary 2. *Lens algebra homomorphisms conform a category.*

All in all, a lens algebra homomorphism is a composable abstraction that supplies the interface of lens algebras. This abstraction is more general than lens algebras, but we can overwrite their alternative representation when we instantiate `q` to a `state` program on the focus:

```
Definition lensAlg′ p A := lensAlgHom p (state A) A
```

Now, we will exploit lens algebra homomorphisms to evolve the university data layer to its final version.

4.1 Extending Data Layer Design with Homomorphisms

In Sect. 3.1 we faced some limitations to compose lens algebras, aggravated by the separation of algebras for university and department. Here, we will overcome

those limitations using the new representations of lens algebras outlined in the last section. Firstly, we will encode `nameLn` and `budgetLn` as `lensAlg′` structures. Secondly, we will replace `mathDepLn` with a lens algebra homomorphism, acting as the nexus between university and department. The resulting data layer is represented as follows:

```
Record DepartmentAlg p Dep `{id : MonadState Dep p} :=
{ budgetLn : lensAlg′ p nat }.

Record UniversityAlg p Univ `{id : MonadState Univ p} :=
{ nameLn : lensAlg′ p string
; q : Type → Type
; Dep : Type
; ev : `{DepartmentAlg q Dep}
; mathDepLn : lensAlgHom p q Dep }.
```

Now, we have all the ingredients to implement the business logic to duplicate university budgets.

```
Definition duplicateMathBudget p Univ
   `{MonadState Univ p}
    (data : UniversityAlg p Univ): p unit :=
  (mathDepLn data ▷ln budgetLn (ev data)) ∼ln (λ b ⇒ b * 2).
```

The implementation of this method reflects the same elegance which is customary for optic-based designs. First of all, the data accessors are composable, so we can combine the lens algebra that focuses on the university math department with the lens algebra that focuses on the department budget. Second of all, the resulting lens algebra can invoke the modifying operator ($\sim_{ln}$), passing the function that duplicates the budget as argument. In contrast with common optic-based designs, however, this business logic is implemented once and for all, not only for immutable data structures, but for alternative state-based infrastructures (e.g. a relational database) as well.

Now imagine we are interested in knowing the resulting budget after duplicating it. There is an obvious implementation for that logic: first we modify the value and then we consult it. Lens algebras provide such operations, and the monadic effect allows us to compose their results:

```
Definition duplicateMathBudgetR p Uni `{MonadState Uni p}
    (data : UniversityAlg p Uni) : p nat :=
  let ln := mathDepLn data ▷ln budgetLn (ev data)
  in ln ∼ln (λ b ⇒ b * 2) ≫ view ln.
```

However, if we take into account the complexity of the potential underlying infrastructures, this implementation would not be optimal, since it requires accessing to the department twice. We could solve this situation by exploiting the distributive property over `bind` provided by lens algebra homomorphisms:

```
Definition duplicateMathBudgetR′ p Uni `{MonadState Uni p}
    (data : UniversityAlg p Uni) : p nat :=
  let bLn := budgetLn (ev data)
  in (mathDepLn data) (bLn ∼ln (λ b ⇒ b * 2) ≫ view bLn).
```

Instead of performing two small operations – involving a particular field – over the whole state, this version performs a unique larger operation over the inner focus, and then lifts the result. This way of programming is essentially enabled by adopting natural transformations, and it turns out to be extremely handy for many situations, even when different fields are involved. Moreover, it may lead to significant optimizations in performance, since this strategy allows us in principle to push down whole programs to the underlying infrastructure of particular repositories, and reduce the number of interactions between them.

5 Related Work

5.1 Profunctor Lenses

There is a lens representation based on the notion of *profunctors*, which can be seen as a generalisation of functions, that has acquired major significance recently [9]. It is encoded as follows.

```
Definition pLens S T A B := ∀ p `{Cartesian p}, p A B → p S T.
```

Note that this representation corresponds with a polymorphic lens, hence the four type parameters. To recover the monomorphic version, we should restrict instances to `pLens S S A A`. If we ignore the `Cartesian` constraint, which is just a member of the profunctor hierarchy, we appreciate that this definition consists of a pure function that turns `p A B`, a generalized function on the part, into `p S T`, a generalized function on the whole.

Despite the generality provided by profunctors, we cannot use this lens representation to replace lens algebras. For instance, let us assume a profunctor lens focusing on the budget of a department, whose type is `pLens Dep Dep nat nat`, where `Dep` corresponds to an index, as introduced in Sect. 3.1. As any other profunctor lens, it works for all `Cartesian` instances. Pickering *et al.* illustrate that *UpStar* is an instance of that typeclass. If we combine it with the *Constant* functor, we recover the classic `view` functionality from the lens. In this context, we obtain a pure function from `Dep` to `nat`. This stands in conflict with `Dep` being an index, which imposes an effectful computation as the only way to retrieve the budget.

All in all, profunctor lens is yet another lens representation, and despite its great composition capabilities, it is also generalized by lens algebras (provided that we constraint ourselves to the monomorphic version of lenses). If we want to relate lens algebras with other abstractions, they must contemplate computational effects in their definition somehow.

5.2 Monadic Lenses

Monadic lenses are a novel approach to combine lenses with monadic effects [7]. Its definition is very similar to `lens`, though the updating method returns an effect as result.

```
Record mLens S A m `{Monad m} := mkMLens
{ mview   : S → A
; mupdate : S → A → m S }.
```

Note that `mview` is lacking an effect in its result deliberately, since including it would lead to severe composition problems. The following laws hold for a well-behaved monadic lens:

```
Record mLensLaws {S A m} `{Monad m}
    (mln : mLens S A m) := mkMLensLaws
{ mview_mupdate : ∀ s, mupdate mln s (mview mln s) = ret s
; mupdate_mview : ∀ B (k : S → A → m B) s a,
    mupdate mln s a >>= (λ s' ⇒ k s' (mview mln s')) =
    mupdate mln s a >>= (λ s' ⇒ k s' a) }.
```

These laws establish that no effect should be produced when updating the whole with the current part, and that viewing the part should be consistent with the last update. If we set `m` to `Id` we recover an ordinary well-behaved lens.

Monadic lenses do not accommodate the stateful infrastructures that we are interested in. In this sense, the absence of effects in `mview` is probably the main evidence of it, in line with the discussion in Sect. 5.1. Broadly, lens algebras and monadic lenses pursuit different goals. On the one hand, lens algebras abstracts away from immutable data structures by parameterizing the computational effect to access and manipulate state. On the other hand, monadic lenses aim at enriching plain lenses with all kind of effects, such as partiality or logging, but they target immutable data structures. We discuss whether lens algebras support such computational effects in Sect. 5.3.

That said, there are interesting observations from the discussion presented by Abou-Saleh *et al.* around effectful `mview`, that are relevant in our case. In particular, there is a question that arises naturally: is it safe to define an effectful `view` for lens algebra as we do? To answer this question, note that lens algebra laws (or `MonadState` laws) impose a strong condition over `view`, which is described as follows:

Lemma 3. *Consider a (very) well-behaved lens algebra `ln` with type `lensAlg p A`. The following property is derived from its laws:*

```
∀ (X : Type) (px : p X), view ln ≫ px = px.
```

This lemma tells us that `view` is constrained to retrieving the state, i.e. it must not produce any further effect. Therefore, lens algebra laws make it safe to declare an effectful `view`. We can find a slight variation of this lemma for the state monad transformer in [6, Lemma 2.7.].

5.3 Entangled State Monads

Entangled state monads [6] emerged in the context of bidirectional transformations – where symmetric lenses have become central – as the initial model of the following algebra:

```
Record BX (p : Type → Type) (A B : Type) : Type :=
{ getL : p A
; getR : p B
; putL : A → p unit
; putR : B → p unit }.
```

We can appreciate that `BX p A B` is `MonadState p A` with an additional focus `B`, and corresponding methods to access and manipulate it. Its associated laws duplicate `MonadState` laws for each focus and append a new law to state that `getL` and `getR` can be commuted. In this regard, the definition does not contemplate commutativity of `putL` and `putR`, since it would break up the entanglement among the states.

There are strong similarities between entangled state monads and our work. Firstly, Abou-Saleh *et al.* recovers a very well-behaved asymmetric `lens S A` as an instance `BX (state S)S A`, which provides methods to deal with both whole `S` and part `A`. This is basically the same approach that we follow in Proposition 1, where we also employ the state monad to recover lens. Secondly, `BX` is closely related to the way we encode data layers. In particular, the constraint `id` that we provide to repository algebras corresponds to the methods `getL` and `putL` , although we ignore them, since we are not interested in evolving the whole directly (at least, in the examples shown in this paper). For its part, `getR` and `putR` corresponds to a lens algebra hosted by the data layer, such as `budgetLn`.

Therefore, why do not we use `BX` to program data layers? First of all, composition is defined for well-behaved stateful BX [6, Definition 3.8.]. A crucial feature of this class of instances is that `getL` and `getR` operations are essentially pure. Unfortunately, this does not hold in our most relevant use cases. For instance, if we need to instantiate `DepartmentAlg` with a state monad transformer to access the information from a relational database, the `getR` operation would not be pure. Second of all, we are interested in deploying different computational effects for the nested repositories, while `BX` composition requires a fixed one. Lens algebra homomorphisms support them, while preserving the look and feel of classic lenses in the implementation of business logic.

There is a relevant takeaway from entangled state monads, involving *overwritability*, that will allow us to answer the question whether lens algebras support additional effects, such as partiality or logging. Overwritability corresponds with the controversial [14] `update_update` law in the context of lens algebras. In this sense, very well-behaved lens algebras inherit the same limitations. In fact, if we aim at enriching instances with additional computational effects, we should embrace the notion of well-behaved lens algebras, where `update_update` law is discarded.

6 Conclusions

Figure 1 summarizes the relationships established in this paper between optics (first row) and algebraic theories (second row), as far as lenses are concerned.

All the relationships in this diagram were formalised as propositions in the Coq proof assistant.

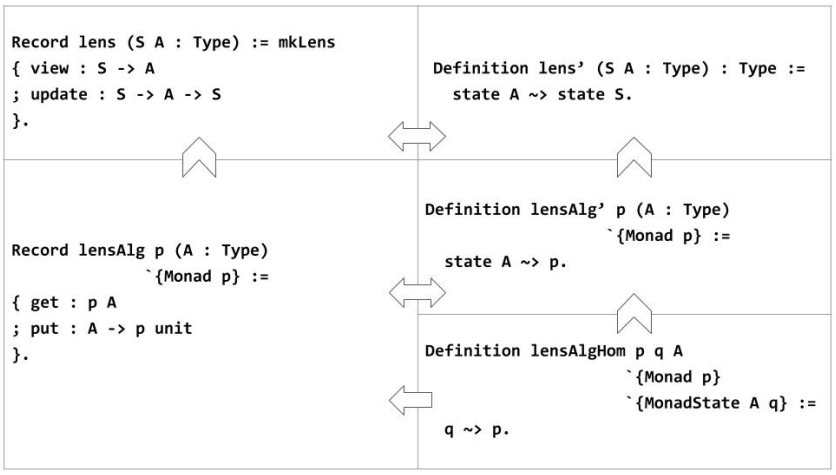

Figure 1. Taking lenses to a higher level of abstraction

The left column of this diagram relates the concrete representation of very well-behaved lenses with the algebraic theory of `MonadState`. This relationship was shown to be a proper generalisation, insofar `lens S A` can be defined as `MonadState A (state S)`. This justifies the alias of `lensAlg` for `MonadState`. The right column essentially deals with compositional issues. There, we start from an alternative representation of lenses with better compositional properties, `state A ⇝state S`, and proceed towards lens algebra homomorphism, a strict generalisation that enjoys full compositional properties with respect to `lensAlg`s. This direct support for compositability at the algebraic level is the major point of departure with respect to `BX`, the closest approach in the literature to our work. Crucially, we showed how this algebraic notion of compositionality can be founded on a strict generalisation of composable lens representations.

Current work focuses on applying the aforementioned approach that allowed us to turn lenses into algebraic theories, to other members of the optics catalogue. To this extent, in order to identify the algebraic counterparts of affines, traversals, getters, etc., the existence of alternative optic representations based on natural transformations, like the state monad morphism lens representation, becomes essential. Unfortunately, we have not found previous work describing such abstractions for other optics, so we are responsible for building them up.

This paper also provided a design pattern to implement the data layer of applications with better modularity guarantees. Particularly, instead of defining *ad hoc* repository algebras with get/set functions to manipulate particular fields, we promote lens algebras as a standard reusable abstraction that fulfills that

objective. In addition, we have shown how lens algebra homomorphisms provide the glue to relate the different algebras that make up the whole data layer. By defining the data layer using this pattern, we hide the complexity details of natural transformations behind the API of lens algebras.

For the time being, we have been doing some experiments in implementing this design pattern in a Scala library that we affectionately named Stateless[9]. The repository for this project includes several examples, including an extended version of the university, which reflects the potential of the ideas in this paper. Moreover, it shows preliminary versions of the algebraic counterparts for affines, traversals, getters, etc.

Besides the formalization and justification of this catalogue of optic algebras, the second great challenge that we face is guaranteeing that the data layer of programs implemented with Stateless can be interpreted in common state-based infrastructures, with *optimal* levels of performance. This essentially means that we don't incur in performance penalties when translating optic algebra programs to relational databases, microservices, caches, and so forth (including combinations of some of them, as when having a database containing *urls* pointing at the service that storages the actual state of some domain entity). For this purpose, we plan to build upon existing EDSLs optimization techniques [15,16].

Acknowledgments

Partially supported by a Doctorate Industry Program grant to Habla Computing SL, from the Spanish Ministry of Economy, Industry and Competitiveness.

References

1. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bi-directional tree transformations: a linguistic approach to the view update problem. ACM SIGPLAN Notices **40**(1) (2005) 233–246
2. O'Connor, R.: Functor is to lens as applicative is to biplate: Introducing multiplate. arXiv preprint arXiv:1103.2841 (2011)
3. Gibbons, J.: Unifying theories of programming with monads. In: International Symposium on Unifying Theories of Programming, Springer (2012) 23–67
4. Wadler, P., Blott, S.: How to make ad-hoc polymorphism less ad hoc. In: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, ACM (1989) 60–76
5. Van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. ACM Sigplan Notices **35**(6) (2000) 26–36
6. Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Notions of bidirectional computation and entangled state monads. In: International Conference on Mathematics of Program Construction, Springer (2015) 187–214
7. Abou-Saleh, F., Cheney, J., Gibbons, J., McKinna, J., Stevens, P.: Reflections on monadic lenses. CoRR **abs/1601.02484** (2016)

[9] `https://github.com/hablapps/stateless`

8. Cheney, J., McKinna, J., Stevens, P., Gibbons, J., Abou, F., et al.: Entangled state monads. In: BX Workshop. (2014)
9. Pickering, M., Gibbons, J., Wu, N.: Profunctor optics: Modular data accessors. arXiv preprint arXiv:1703.10857 (2017)
10. Pierce, B.C.: Basic category theory for computer scientists. MIT press (1991)
11. Wadler, P.: Monads for functional programming. In: International School on Advanced Functional Programming, Springer (1995) 24–52
12. Fischer, S., Hu, Z., Pacheco, H.: A clear picture of lens laws. In: International Conference on Mathematics of Program Construction, Springer (2015) 215–223
13. Jones, M.P.: Type classes with functional dependencies. In: European Symposium on Programming, Springer (2000) 230–244
14. Johnson, M., Rosebrugh, R.: Lens put-put laws: monotonic and mixed. Electronic Communications of the EASST **49** (2012)
15. Rompf, T., Odersky, M.: Lightweight modular staging: a pragmatic approach to runtime code generation and compiled dsls. In: Acm Sigplan Notices. Volume 46., ACM (2010) 127–136
16. Moors, A., Rompf, T., Haller, P., Odersky, M.: Scala-virtualized. In: Proceedings of the ACM SIGPLAN 2012 workshop on Partial evaluation and program manipulation, ACM (2012) 117–120
17. Sozeau, M., Oury, N.: First-class type classes. In: International Conference on Theorem Proving in Higher Order Logics, Springer (2008) 278–293

A Definitions and Proofs

The claims that we state in this paper require proofs in order to be trustworthy. Unfortunately, it became really tedious to write informal proofs for this research project, specially when other optics were involved. The complexity and length of several of them led us to a feeling of mistrust in our work. That is the reason why we decided to approach a proof assistant. In this sense, we chose The Coq Proof Assistant, given its good support for algebraic theories [17].

All definitions, examples and proofs have been formalized with Coq 8.7.1 (January 2018) and collected in a GitHub repository[10]. Source files have a direct correspondence with the different sections along the paper, that we show in the following description.

Sect1.v definitions for Sect. 1
Sect2.v definitions for Sect. 2
Sect3.v definitions and theories for Sect. 3
Sect4.v definitions and theories for Sect. 4
Sect5.v definitions for Sect. 5

We detected that many of the definitions and theorems appearing in the aforementioned repository could be reused for formalizing other functional programming research projects. Thereby, we decided to create *Koky*[11], an open-sourced typeclass library where we have been collecting them all, which is publicly accessible.

[10] `https://github.com/hablapps/lensalgebra`
[11] `https://github.com/hablapps/koky`