Draft

Dependent lenses

Nils Anders Danielsson
University of Gothenburg
nad@cse.gu.se

Abstract

Very well-behaved lenses provide a convenient mechanism for defining setters and getters for nested records (among other things). However, they do not work very well for dependent records, in which one field's type can depend on the value of a previous field.

This paper discusses dependent lenses, a generalisation of ordinary lenses with better support for dependent records. It is shown that a certain notion of composition cannot be defined for such lenses (under certain assumptions), but that another, different notion of composition can be defined.

Keywords Lenses, Dependent types, Agda

1. Introduction

Very well-behaved lenses (Foster et al. 2005)—in this paper I only consider the total variant—are used to make programs that work with nested record structures more elegant. Consider a record type R_1 containing a field $\mathsf{f}_1 : \mathsf{R}_2$, where the record type R_2 contains a field $\mathsf{f}_2 : \mathsf{R}_3$, and the record type R_3 contains a boolean field f_3:

```
record R₁ : Set where
  field f₁ : R₂

record R₂ : Set where
  field f₂ : R₃

record R₃ : Set where
  field f₃ : Bool
```

(Here Set is a type of small types.) Given a value $x : \mathsf{R}_1$, how do you invert the boolean value contained within it? Manual code can easily become somewhat awkward:

```
record x { f₁ = record (R₁.f₁ x) { f₂ =
  record (R₂.f₂ (R₁.f₁ x))
    { f₃ = not (R₃.f₃ (R₂.f₂ (R₁.f₁ x))) } } }
```

(The notation $\mathsf{R}_1.\mathsf{f}_1\ x$ stands for the projection of the field $\mathsf{R}_1.\mathsf{f}_1$ from the record value x.) Consider also the additional complication if the field f_3 is a finite map from strings to Bool, and you want to update the boolean that the string `"foo"` maps to (if any).

With lenses these updates are easy. Assume that three lenses are given, r_1 corresponding to f_1, r_2 corresponding to f_2, and r_3 corresponding to f_3:

```
r₁ : Lens R₁ R₂
r₂ : Lens R₂ R₃
r₃ : Lens R₃ Bool
```

Then one can perform the inversion of the boolean value contained in x by composing the three lenses:

```
modify (r₃ ∘ r₂ ∘ r₁) not x
```

The question addressed in this paper is whether very well-behaved lenses work equally well for *dependent* records, where the types of later fields can depend on the values of previous fields, like the following type:

```
record ℚ : Set where
  field
    dividend  : ℤ
    divisor   : ℕ
    non-zero  : ¬ divisor ≡ 0
```

The answer, in short:

- No matter how the type family of very well-behaved dependent lenses is defined, if it satisfies a certain, natural property, then it is *impossible* to define a general composition operator for these lenses (see Section 5).
- However, one can define a family of types DLens S V of dependent lenses, where V depends on S, in such a way that getters and setters, and functions like modify, can be defined (Section 6).
- Furthermore it is possible to define a notion of composition that seems to work well in practice (Section 6.4).

These results have been formalised using Agda. This document is a literate Agda file, every piece of code—except for inline code in text paragraphs—has been type-checked. (I have hidden some pieces of code, and changed the order in which the code is displayed.) There is also other, supporting formalisation for several statements mentioned in the text.

The formalisation has been developed with the K rule turned off. Equality of functions is assumed to be extensional, and I assume that a propositional truncation operator (The Univalent Foundations Program 2013) is available. Some statements have been proved under the assumption of univalence (The Univalent Foundations Program 2013), and other statements under the assumption of uniqueness of identity proofs for Set. These two principles are not consistent with each other, so in the text I point out when I make use of them. In the formalisation I track their use using types.

2. Non-dependent lenses

Very well-behaved *non-dependent* lenses from a source type S to a view type V are often specified to consist of two functions, a getter and a setter, satisfying three laws. In Agda we can phrase this as a dependent record:

```
record Lens (S V : Set) : Set where
  field
    get     : S → V
    set     : S → V → S
    get-set : ∀ s v → get (set s v) ≡ v
    set-get : ∀ s → set s (get s) ≡ s
    set-set : ∀ s v₁ v₂ → set (set s v₁) v₂ ≡ set s v₂
```

The getter provides a view of the source value, and the setter modifies a source value with a possibly different view. The get-set law states that set really updates the view, the set-get law states that if you update the view with the original view, then nothing happens, and the set-set law states that if you update the view twice (without intervening changes), then the first update is ignored.

Lenses defined in this way form a monoid. Defining an identity lens is easy (the proofs are omitted):

```
id : {S : Set} → Lens S S
id = record
  { get = λ x   → x
  ; set = λ _ s → s
  }
```

(The notation $\{S : \mathsf{Set}\} \to \ldots$ means that S is an *implicit* argument, that does not need to be given explicitly if Agda can infer it.) We can also define composition of lenses:

```
_∘_ : {S V₁ V₂ : Set} →
      Lens V₁ V₂ → Lens S V₁ → Lens S V₂
l₁ ∘ l₂ = record
  { get = λ s      → get l₁ (get l₂ s)
  ; set = λ s v₂  →
            let v₁ = set l₁ (get l₂ s) v₂ in
            set l₂ s v₁
  }
```

(Here I write get l_1 s rather than Lens.get l_1 s. Further down I also write get s, omitting the lens argument.) One can also prove that composition is associative and that id is a left and right unit for composition (even in the absence of the K rule), but the proofs are omitted here.

3. No first projection lens

Let us now consider whether there can be a non-dependent lens corresponding to the first projection from a Σ-type (a pair type where the type of the second component can depend on the value of the first component).

It is not so hard to see that this is impossible. Consider the following Σ-type, consisting of a boolean and a proof requiring this boolean to be true:

```
Unit : Set
Unit = Σ b : Bool · b ≡ true
```

There is exactly one value in this type (people familiar with homotopy type theory may recognise it as a singleton type):

```
unit : Unit
unit = (true , refl)

all-values-equal : (u₁ u₂ : Unit) → u₁ ≡ u₂
all-values-equal (true , refl) (true , refl) = refl
```

(Here refl is the equality type's constructor.)

Assume for a contradiction that a first projection lens exists for the Unit type:

```
first : Lens Unit Bool
```

By using the get-set law twice and all-values-equal once we get a contradiction:[1]

```
true                          ≡⟨ by (get-set first) ⟩
get first (set first unit true)  ≡⟨ by all-values-equal ⟩
get first (set first unit false) ≡⟨ by (get-set first) ⟩
false                         ∎
```

Note that this contradiction could be proved without assuming that get *first* *p* is the first projection of the pair *p*. Furthermore the proof used only one of the lens laws, get-set.

4. Alternative definitions

Before moving on to dependent lenses, let us consider some alternative definitions of very well-behaved non-dependent lenses.

[1] Here by is a tactic that, given a lemma, tries to prove an equality.

Pierce and Schmitt (2003) note that for very well-behaved lenses as defined above (but in set theory), satisfying the extra assumption that the range of the set function is the entire source set, the source set is in bijective correspondence with the cartesian product of the view set and some other set. It has been suggested[2] that one can define the type of very well-behaved lenses as a dependent pair (Σ-type) consisting of a type and a bijection (denoted with a double arrow below):

$$\begin{array}{l}\mathsf{Bijection\text{-}lens} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}_1 \\ \mathsf{Bijection\text{-}lens}\ S\ V = \Sigma\ R : \mathsf{Set} \cdot S \leftrightarrow R \times V\end{array}$$

However, this type is not in general isomorphic to the one defined above: Lens ⊥ ⊥ (where ⊥ is the empty type) is isomorphic to the unit type, but Bijection-lens ⊥ ⊥ is isomorphic to Set.

Capriotti (2014) has introduced *higher lenses* (if you do not understand this definition, do not worry about it, it is not essential to understand the rest of the text):

$$\begin{array}{l}\mathsf{Higher\text{-}lens} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}_1 \\ \mathsf{Higher\text{-}lens}\ S\ V = \\ \quad \Sigma\ get\ : (S \to V) \cdot \\ \quad \Sigma\ P\ \ : (\|\ V\ \| \to \mathsf{Set}) \cdot \\ \qquad (\lambda\ v \to \Sigma\ s \cdot get\ s \equiv v) \equiv (\lambda\ v \to P\ |\ v\ |)\end{array}$$

Here $\|\ V\ \|$ stands for the *propositional truncation* of V. This type is isomorphic to the set[3] quotient of V with the trivial relation; the constructor $|\_| \ :\ V \to \|\ V\ \|$ shows that $\|\ V\ \|$ is inhabited if V is. When the source type is a set the type of higher lenses is isomorphic to the type of non-dependent lenses given in Section 2, assuming univalence. (Capriotti (2014) did not present a complete proof. For a formal proof developed by Capriotti, Andrea Vezzosi and myself, see this paper's accompanying code.)

In the presence of univalence these higher lenses are isomorphic to the following definition of "equivalence-lenses" (found by Andrea Vezzosi and myself), which are similar to the bijection-lenses mentioned above:

$$\begin{array}{l}\mathsf{ELens} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}_1 \\ \mathsf{ELens}\ S\ V = \Sigma\ R : \mathsf{Set} \cdot (S \simeq (R \times V)) \times (R \to \|\ V\ \|)\end{array}$$

Here $A \simeq B$ is the type of *equivalences* between the types A and B. When A and B are sets this concept from homotopy type theory is in bijective correspondence with the type of bijections between A and B. The extra component requiring the view type to be ("merely") inhabited whenever the remainder type R is inhabited ensures that, in the presence of univalence, ELens ⊥ ⊥ is isomorphic to the unit type, so this type can be seen as a more well-behaved variant of Bijection-lens. More generally, still in the presence of univalence, ELens S V is isomorphic to Higher-lens S V, and, if S is a set, to Lens S V. The definition of dependent lenses in Section 6 will be based on this type.

Finally I present an isomorphic variant of Equivalence-lens, that may provide a somewhat different perspective on the definition:

$$\begin{array}{l}\mathsf{Equivalence\text{-}lens'} : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}_1 \\ \mathsf{Equivalence\text{-}lens'}\ S\ V = \\ \quad \Sigma\ get\qquad\quad : (S \to V) \cdot \\ \quad \Sigma\ R\qquad\quad\ \ : \mathsf{Set} \cdot \\ \quad \Sigma\ remainder\ : (S \to R) \cdot \\ \qquad \mathsf{Is\text{-}equivalence}\ (\lambda\ s \to remainder\ s\ ,\ get\ s) \times \\ \qquad \mathsf{Surjective}\ remainder\end{array}$$

Here Is-equivalence f means that f is an equivalence; this holds if and only if f has a left and right inverse. The type states that a lens is a getter function, a remainder type, and a *surjective* function from the source type to the remainder type, satisfying the property that the given combination of the remainder function and the getter is an equivalence.

5. Composition is impossible

Let us now consider *dependent* lenses. In this section I will show that, given certain assumptions, it is impossible to define composition for such lenses.

I assume that the type of dependent lenses has the following type signature:

$$\mathsf{DLens} : (S : \mathsf{Set})\ (V : S \to \mathsf{Set}) \to \mathsf{Set}$$

Note that the view type has been replaced by a family of view types, dependent on the source type. I also assume that there is a (dependently typed) getter function:

$$\mathsf{dget} : \forall\ \{S\ V\} \to \mathsf{DLens}\ S\ V \to (s : S) \to V\ s$$

Finally I assume that *non-dependent* dependent lenses are not very different from regular non-dependent lenses, by requiring that there is a retraction (split surjection) from the former to the latter, relating dget and get in the following way:

$$\begin{array}{lll}\mathsf{to} & : & \forall\ \{S\ V\} \to \mathsf{DLens}\ S\ (\lambda\ \_ \to V) \to \mathsf{Lens}\ S\ V \\ \mathsf{from} & : & \forall\ \{S\ V\} \to \mathsf{Lens}\ S\ V \to \mathsf{DLens}\ S\ (\lambda\ \_ \to V) \\ \mathsf{to\text{-}from} & : & \forall\ \{S\ V\}\ (l : \mathsf{Lens}\ S\ V) \to \mathsf{to}\ (\mathsf{from}\ l) \equiv l \\ \mathsf{get\text{-}to} & : & \forall\ \{S\ V\}\ (l : \mathsf{DLens}\ S\ (\lambda\ \_ \to V))\ (s : S) \to \\ & & \quad \mathsf{get}\ (\mathsf{to}\ l)\ s \equiv \mathsf{dget}\ l\ s\end{array}$$

Here is the partial specification of composition that I am using, parametrised by some source and view types:

$$\begin{array}{l}\textbf{record}\ \mathsf{Specification\text{-}of\text{-}composition} \\ \qquad\quad (S : \mathsf{Set})\ (V_1 : S \to \mathsf{Set}) \\ \qquad\quad (V_2 : (s : S) \to V_1\ s \to \mathsf{Set}) : \mathsf{Set}\ \textbf{where} \\ \textbf{field} \\ \quad \mathsf{dcomp}\qquad : (l_1 : \mathsf{DLens}\ S\ V_1) \\ \qquad\qquad\qquad\ (l_2 : \forall\ s \to \mathsf{DLens}\ (V_1\ s)\ (V_2\ s)) \to\end{array}$$

[2] I do not know who first suggested this.

[3] A concept from homotopy type theory. A type is a set if and only if it satisfies the principle of uniqueness of identity proofs, see Section 6.3.

```
                    DLens S (λ s → V₂ s (dget l₁ s))
dget-dcomp  : ∀ l₁ l₂ s →  dget (dcomp l₁ l₂) s ≡
                           dget (l₂ s) (dget l₁ s)
```

The composition operator takes a lens and a family of lenses to a lens, and the getter for the resulting lens must be expressible in terms of the getters for the argument lenses.

I will prove that Specification-of-composition is uninhabited when all the type parameters are instantiated with (constant functions returning) Bool:

```
no-composition-operator :
  ¬ Specification-of-composition  Bool (λ _ → Bool)
                                  (λ _ _ → Bool)
```

Assume for a contradiction that this instance of the specification is inhabited, giving us access to the dcomp function and the dget-dcomp law. The existence of the retraction mentioned above ensures that we can define a corresponding composition operator for non-dependent lenses:

```
comp :  Lens Bool Bool → (Bool → Lens Bool Bool) →
        Lens Bool Bool
comp l₁ l₂ = to (dcomp (from l₁) (λ b → from (l₂ b)))
```

A calculation shows that this operator satisfies a law corresponding to dget-dcomp, see Figure 1.

The comp function will be used together with the following family of lenses, that is the identity for true and swap for false:

```
id-or-swap : Bool → Lens Bool Bool
id-or-swap true  = id
id-or-swap false = swap
```

Here swap is defined in the following way (proofs omitted):

```
swap : Lens Bool Bool
swap = record
  { get  = not
  ; set  = λ _ v → not v
  }
```

A simple case analysis shows that id-or-swap satisfies the following property:

```
get-id-or-swap : ∀ b → get (id-or-swap b) b ≡ true
get-id-or-swap true  = refl
get-id-or-swap false = refl
```

Now we can use composition to construct the following strange lens:

```
strange : Lens Bool Bool
strange = comp id id-or-swap
```

This lens is strange because its getter is constant:

```
get-strange : ∀ b → get strange b ≡ true
get-strange b =
  get strange b                 ≡⟨⟩
  get (comp id id-or-swap) b    ≡⟨ by get-comp ⟩
  get  (id-or-swap b)
       (get id b)               ≡⟨⟩
  get (id-or-swap b) b          ≡⟨ by (get-id-or-swap b) ⟩
  true                          ∎
```

In combination with the get-set law this leads to a contradiction:

```
true                          ≡⟨ by get-strange ⟩
get strange
  (set strange true false)    ≡⟨ by (get-set strange) ⟩
false                         ∎
```

Note that the proof above only uses one lens law, get-set (just like the proof in Section 3). Dropping the set-get or set-set laws from the definition of non-dependent lenses would not help.

One can perhaps object that the specification of composition is too strong: why should the family of lenses l_2 be allowed to inspect its argument? In Section 6.4 a variant of composition is defined. This notion restricts the family of lenses by not giving it access to values from the source type, but only from the remainder type.

6. Dependent lenses

Notwithstanding the negative result in the previous section, this section contains a definition of dependent lenses. The definition is based on the equivalence-lenses from Section 4. These equivalence-lenses contain an equivalence between the source type S and the cartesian product $R \times V$ of the remainder type R and the view type V, indicating that V is somehow a part of S.

For dependent lenses the view type is replaced by a family of types, so one tentative idea may be to replace the cartesian product with a Σ-type $\Sigma\, r : \mathsf{R} \cdot \mathsf{V}\, r$. This does not work, because V does not depend on R, but on S. However, if V is seen as a part of S, then it might seem reasonable that it should not in general depend on "all" of S, but only on the remainder type, R. The following definition therefore introduces an alternative view family V' that depends on R, and is restricted to be a variant of V:

```
record DLens (S : Set) (V : S → Set) : Set₁ where
  field
    R          : Set
    V′         : R → Set
    equiv      : S ≃ (Σ r : R · V′ r)
    inhabited  : ∀ r → ∥ V′ r ∥

  remainder : S → R
  remainder s = fst (to equiv s)
```

```
get-comp : ∀ l₁ l₂ b → get (comp l₁ l₂) b ≡ get (l₂ b) (get l₁ b)
get-comp l₁ l₂ b =
  get (comp l₁ l₂) b                                ≡⟨⟩
  get (to (dcomp (from l₁) (λ b → from (l₂ b)))) b  ≡⟨ by get-to ⟩
  dget (dcomp (from l₁) (λ b → from (l₂ b))) b      ≡⟨ by dget-dcomp ⟩
  dget (from (l₂ b)) (dget (from l₁) b)             ≡⟨ by get-to ⟩
  get (to (from (l₂ b))) (dget (from l₁) b)         ≡⟨ by get-to ⟩
  get (to (from (l₂ b))) (get (to (from l₁)) b)     ≡⟨ by to-from ⟩
  get (l₂ b) (get l₁ b)                             ∎
```

Figure 1. The get-comp law. Note that if the explanation has been omitted in an equational reasoning step, as in … ≡⟨⟩ …, then the two sides are equal by definition.

```
  field
    variant  : ∀ {s} → V′ (remainder s) ≡ V s
```

Here to equiv is the forward component of the equivalence. For every source value s we can compute a remainder value remainder s, and the last field requires V' (remainder s) to be equal to $V\,s$.

6.1 Getters and setters

Given a dependent lens we can define a getter and a setter:

```
get′ : (s : S) → V′ (remainder s)
get′ s = snd (to equiv s)

get : (s : S) → V s
get s = cast variant (get′ s)

set′ : (s : S) → V′ (remainder s) → S
set′ s v′ = from equiv (remainder s , v′)

set : (s : S) → V s → S
set s v = set′ s (cast (sym variant) v)
```

These definitions make use of the cast function to "transport" a value from one type to another via an equality. The cast function is in turn defined using subst:

```
subst :  ∀ {a p} {A : Set a} (P : A → Set p) →
         ∀ {x y : A} → x ≡ y → P x → P y
subst _ refl p = p

cast : {A B : Set} → A ≡ B → A → B
cast = subst id
```

The last definition also uses symmetry:

```
sym : ∀ {a} {A : Set a} {x y : A} → x ≡ y → y ≡ x
sym refl = refl
```

(Some functions above have been given universe-polymorphic types, because they are used with "large" arguments.)

We can also define a function corresponding to the modify function used in the introduction:

```
modify : (s : S) → (V s → V s) → S
modify s f = set s (f (get s))
```

6.2 Lens laws

It is possible to prove variants of the usual lens laws. The set-get law can be established using simple equational reasoning, see Figure 2. This proof uses a lemma, cast-sym-cast, which states that if you cast in one direction, and then in the opposite direction, then you get back what you started with:

```
cast-sym-cast :  {A B : Set} (eq : A ≡ B) {x : A} →
                 cast (sym eq) (cast eq x) ≡ x
cast-sym-cast refl = refl
```

The following variant of cast-sym-cast will be used below:

```
cast-cast-sym :  {A B : Set} (eq : A ≡ B) {x : B} →
                 cast eq (cast (sym eq) x) ≡ x
cast-cast-sym refl = refl
```

The other two non-dependent lens laws are not well-typed in this setting. For instance, get′ (set′ $s\,v'$) has type V' (remainder (set′ $s\,v'$)), not V' (remainder s). However, we can get around this problem by observing that set and set′ do not change the remainder:

```
remainder-set′ :  ∀ s v′ →
                  remainder (set′ s v′) ≡ remainder s
remainder-set′ s v′ =
  remainder (set′ s v′)        ≡⟨⟩
  fst (to equiv (from equiv s′)) ≡⟨ cong fst (to-from equiv _) ⟩
  fst s′                       ≡⟨⟩
  remainder s                  ∎
  where
```

```
set′-get′ : ∀ s → set′ s (get′ s) ≡ s
set′-get′ s =
  set′ s (get′ s)                                ≡⟨⟩
  from equiv (remainder s , snd (to equiv s))    ≡⟨⟩
  from equiv (fst (to equiv s) , snd (to equiv s)) ≡⟨⟩
  from equiv (to equiv s)                        ≡⟨ by (from-to equiv) ⟩
  s                                              ∎

set-get : ∀ s → set s (get s) ≡ s
set-get s =
  set s (get s)                                  ≡⟨⟩
  set′ s (cast (sym variant) (cast variant (get′ s))) ≡⟨ by (cast-sym-cast variant) ⟩
  set′ s (get′ s)                                ≡⟨ by set′-get′ ⟩
  s                                              ∎
```

Figure 2. The set′-get′ and set-get laws.

```
  s′ = (remainder s , v′)

remainder-set : ∀ s v → remainder (set s v) ≡ remainder s
remainder-set s v =
  remainder (set s v)                 ≡⟨⟩
  remainder
    (set′ s (cast (sym variant) v))   ≡⟨ remainder-set′ _ _ ⟩
  remainder s                         ∎
```

These proofs are not proved using the by tactic, because later results depend on the structure of the proofs. Instead I have used the cong lemma, which expresses the fact that every non-dependent function respects equality:

```
cong : ∀ {a b} {A : Set a} {B : Set b} →
         (f : A → B) {x y : A} → x ≡ y → f x ≡ f y
cong _ refl = refl
```

The get′-set′ law can now be stated by using the subst function to transport from one type to another:

```
get′-set′ :
  ∀ s v′ →
  subst V′ (remainder-set′ s v′) (get′ (set′ s v′)) ≡ v′
```

This statement of the get′-set′ law can be proved by using equational reasoning, see Figure 3. The proof makes use of two general lemmas. The first one relates subst and cong:

```
subst-∘ :
  ∀ {a b p} {A : Set a} {B : Set b} {x y : A}
    {P : B → Set p} {f : A → B} {p : P (f x)}
  (eq : x ≡ y) →
  subst (P ∘ f) eq p ≡ subst P (cong f eq) p
subst-∘ refl = refl
```

The second one makes it possible to remove an instance of subst (the name of this lemma is taken from The Univalent Foundations Program (2013)):

```
apd : {A : Set} {B : A → Set} {x y : A}
      (f : (x : A) → B x) (eq : x ≡ y) →
      subst B eq (f x) ≡ f y
apd _ refl = refl
```

The V-set lemma states that the view type stays unchanged when the source parameter is changed using set:

```
V-set : ∀ s v → V (set s v) ≡ V s
V-set s v =
  V (set s v)                ≡⟨ sym variant ⟩
  V′ (remainder (set s v))   ≡⟨ cong V′ (remainder-set s v) ⟩
  V′ (remainder s)           ≡⟨ variant ⟩
  V s                        ∎
```

This lemma can be used to state the get-set law, see Figure 4. Here I have relegated some steps of equational reasoning to an omitted lemma.

The set′-set′ law can be stated by using remainder-set′ to “correct” the type of the second view argument, see Figure 5. The law is proved by using Σ-≡, which gives a general way to prove that two values from a Σ-type are equal:

```
Σ-≡ :
  {A : Set} {B : A → Set} {x y : Σ x : A · B x} →
  (eq : fst x ≡ fst y) →
  subst B eq (snd x) ≡ snd y →
  x ≡ y
Σ-≡ refl refl = refl
```

Finally we can state and prove the set-set law, see Figure 6. Here I have again relegated some steps of equational reasoning to an omitted lemma.

```
get′-set′ s v′ =
   subst V′ (remainder-set′ s v′) (get′ (set′ s v′))     ≡⟨⟩
   subst V′ (cong fst eq) (get′ (set′ s v′))             ≡⟨ by (subst-∘ eq) ⟩
   subst (V′ ∘ fst) eq (get′ (set′ s v′))                ≡⟨⟩
   subst (V′ ∘ fst) eq (snd (to equiv (from equiv s′))) ≡⟨ by (apd snd eq) ⟩
   snd s′                                                ≡⟨⟩
   v′                                                    ∎
   where
   s′ = (remainder s , v′)
   eq = to-from equiv s′
```

Figure 3. The get′-set′ law.

```
get-set : ∀ s v → cast (V-set s v) (get (set s v)) ≡ v
get-set s v =
   cast (V-set s v) (get (set s v))                                      ≡⟨⟩
   cast (V-set s v) (cast variant (get′ (set s v)))                      ≡⟨ by lemma ⟩
   cast variant (cast (cong V′ (remainder-set s v)) (get′ (set s v)))   ≡⟨ by (subst-∘ (remainder-set s v)) ⟩
   cast variant (subst V′ (remainder-set s v) (get′ (set s v)))          ≡⟨⟩
   cast variant (subst  V′ (remainder-set s v) (get′ (set′ s (cast (sym variant) v)))) ≡⟨ by get′-set′ ⟩
   cast variant (cast (sym variant) v)                                   ≡⟨ by (cast-cast-sym variant) ⟩
   v                                                                     ∎
```

Figure 4. The get-set law.

```
set′-set′ : ∀ s v₁′ v₂′ → set′ (set′ s v₁′) v₂′ ≡ set′ s (subst V′ (remainder-set′ s v₁′) v₂′)
set′-set′ s v₁′ v₂′ =
   set′ (set′ s v₁′) v₂′                                          ≡⟨⟩
   from equiv (remainder (set′ s v₁′) , v₂′)                      ≡⟨ by (Σ-≡ (remainder-set′ s v₁′) refl) ⟩
   from equiv (remainder s , subst V′ (remainder-set′ s v₁′) v₂′) ≡⟨⟩
   set′ s (subst V′ (remainder-set′ s v₁′) v₂′)                   ∎
```

Figure 5. The set′-set′ law.

6.3 Relation to non-dependent lenses

The principle of uniqueness of identity proofs (UIP) for a given type A states that, for any two values of type A, all equality proofs between those two values are equal:

```
Uniqueness-of-identity-proofs :
   ∀ {a} → Set a → Set a
Uniqueness-of-identity-proofs A =
   {x y : A} → (p q : x ≡ y) → p ≡ q
```

If we assume that this property holds for Set, the type of small types, then non-dependent dependent lenses are isomorphic to equivalence-lenses:

```
Uniqueness-of-identity-proofs Set →
{S V : Set} →
Σ eq : DLens S (λ _ → V) ≃ ELens S V ·
   ∀ l s → get l s ≡ eget (to eq l) s
```

(Here eget is the get function for equivalence-lenses.) Note that the assumption that Set satisfies UIP is not compatible with univalence (but it holds in Agda if the K rule is turned on). I have not managed to determine if non-dependent dependent lenses are isomorphic to equivalence-lenses in the presence of univalence.

The argument in Section 5 showing that composition is impossible for dependent lenses was carried out using regular lenses, Lens. It was also restricted to dependent lenses living

```
set-set : ∀ s v₁ v₂ → set (set s v₁) v₂ ≡ set s (cast (V-set s v₁) v₂)
set-set s v₁ v₂ =
  set (set s v₁) v₂                                                  ≡⟨⟩
  set′ (set′ s (cast (sym variant) v₁)) (cast (sym variant) v₂)      ≡⟨ by set′-set′ ⟩
  set′ s (subst  V′ (remainder-set′ s (cast (sym variant) v₁))
                 (cast (sym variant) v₂))                            ≡⟨⟩
  set′ s (subst V′ (remainder-set s v₁) (cast (sym variant) v₂))     ≡⟨ by (subst-∘ (remainder-set s v₁)) ⟩
  set′ s (cast (cong V′ (remainder-set s v₁)) (cast (sym variant) v₂)) ≡⟨ lemma ⟩
  set′ s (cast (sym variant) (cast (V-set s v₁) v₂))                 ≡⟨⟩
  set s (cast (V-set s v₁) v₂)                                       ■
```

Figure 6. The set-set law.

in Set, but the dependent lenses defined in this section live in Set_1. However, the argument works also for equivalence-lenses and larger dependent lenses, even in the absence of univalence. Thus, in the presence of UIP for Set, the strong form of composition discussed in Section 5 cannot be defined for DLens.

It is also possible to show that, in the presence of UIP for set, the first projection lens cannot always be defined. Let us assume that UIP holds:

```
uip : Uniqueness-of-identity-proofs Set
```

In this case casts between a type and itself can be removed:

```
drop-cast : {A : Set} (eq : A ≡ A) (x : A) →
            cast eq x ≡ x
drop-cast eq x =
  cast eq x   ≡⟨ by uip ⟩
  cast refl x ≡⟨⟩
  x           ■
```

Let us now, as in Section 3, assume that there is a first projection lens for the Unit type:

```
first : DLens Unit (λ _ → Bool)
```

We again get a contradiction, see Figure 7. I do not know if a corresponding result can be proved in the presence of univalence.

6.4 Composition

If composition cannot be defined, what is the point of dependent lenses? Fortunately it is possible to define a weaker variant of composition. As mentioned in Section 5 the notion of composition defined here does not allow the family of lenses to inspect values from the source type, but only from the remainder type. Composition is defined in Figure 8. The remainder type combines the two remainder types, and the variant of the view type family is defined in terms of the lens family's variant type families. The equivalence is defined using the argument equivalences, using the following associativity-like property for Σ-types:

```
Σ-assoc :
  {A : Set} {B : A → Set}
  {C : Σ x : A · B x → Set} →
  (Σ x : A · Σ y : B x · C (x , y)) ≃
  (Σ p : (Σ x : A · B x) · C p)
```

Finally the inhabitance and variant proofs are defined using the lens family's inhabitance and variant proofs.

The following minor variant, in which the lens family takes an implicit argument instead of an explicit one, will be used in the examples below:

```
_⨾_ :
  {S : Set} {V₁ V₂ : S → Set} →
  (l₁ : DLens S V₁) →
  (∀ {r} →
     DLens (V′ l₁ r)
           (λ v′ → V₂ (from (equiv l₁) (r , v′)))) →
  DLens S V₂
l₁ ⨾ l₂ = dcomp l₁ (λ _ → l₂)
```

6.5 Constructing lenses

When constructing a lens one can often avoid having to construct all the fields above, in many cases it suffices to give an equivalence:

```
equivalence-to-lens :
  {S R : Set} {V : R → Set}
  (eq : S ≃ (Σ r : R · V r)) →
  DLens S (λ s → V (fst (to eq s)))
equivalence-to-lens {S} {R} {V} eq = record
  { R       = Σ r : R · ∥ V r ∥
  ; V′      = λ r → V (fst r)
  ; equiv   =
    S                                  ≃⟨ eq ⟩
    (Σ r : R · V r)                    ≃⟨ Σ-cong ≃×∥∥ ⟩
    (Σ r : R · ∥ V r ∥ × V r)          ≃⟨ Σ-assoc ⟩
    (Σ p : (Σ r : R · ∥ V r ∥) · V (fst p)) □
```

```
true                                                          ≡⟨ by (get-set first) ⟩
cast (V-set first unit true)  (get first (set first unit true))  ≡⟨ by (drop-cast (V-set first unit true)) ⟩
get first (set first unit true)                               ≡⟨ by all-values-equal ⟩
get first (set first unit false)                              ≡⟨ by (drop-cast (V-set first unit false)) ⟩
cast (V-set first unit false)  (get first (set first unit false)) ≡⟨ by (get-set first) ⟩
false                                                         ∎
```

Figure 7. A contradiction.

```
dcomp : {S : Set} {V₁ V₂ : S → Set} →
        (l₁ : DLens S V₁) →
        (∀ r → DLens (V′ l₁ r) (λ v′ → V₂ (from (equiv l₁) (r , v′)))) →
        DLens S V₂
dcomp {S} {V₁} {V₂} l₁ l₂ = record
  { R         = Σ r₁ : R l₁ · R (l₂ r₁)
  ; V′        = λ p → V′ (l₂ (fst p)) (snd p)
  ; equiv     = S                                               ≃⟨ equiv l₁ ⟩
                (Σ r₁ : R l₁ · V′ l₁ r₁)                        ≃⟨ Σ-cong (equiv (l₂ _)) ⟩
                (Σ r₁ : R l₁ · Σ r₂ : R (l₂ r₁) · V′ (l₂ r₁) r₂)  ≃⟨ Σ-assoc ⟩
                (Σ p : (Σ r₁ : R l₁ · R (l₂ r₁)) · V′ (l₂ (fst p)) (snd p)) □
  ; inhabited = λ p → inhabited (l₂ (fst p)) (snd p)
  ; variant   = λ {s} →
                  let r₁ = fst (to (equiv l₁) s) in
                  V′ (l₂ r₁) (remainder (l₂ r₁) (snd (to (equiv l₁) s))) ≡⟨ variant (l₂ r₁) ⟩
                  V₂ (from (equiv l₁) (to (equiv l₁) s))                ≡⟨ by (from-to (equiv l₁)) ⟩
                  V₂ s                                                  ∎
  }
```

Figure 8. Composition for dependent lenses.

```
  ; inhabited = snd
  ; variant   = refl
  }
```

The remainder type R and the variant of the view family V′ are chosen in such a way that it is easy to construct the inhabited and variant proofs. The equivalence equiv is constructed using "equivalence-reasoning". The first step uses the supplied equivalence, and the last step uses Σ-assoc. The second step uses the fact that Σ-types preserve equivalences in their second argument (Σ-cong) and the fact that any type is equivalent to the cartesian product of its propositional truncation and itself:

```
≃×∥∥ : {A : Set} → A ≃ (∥ A ∥ × A)
```

Using equivalence-to-lens it is easy to construct an identity lens:

```
id : {S : Set} → DLens S (λ _ → S)
id {S} = equivalence-to-lens (
           S      ≃⟨ ⊤-× ⟩
           ⊤ × S  □)
```

The proof makes use of the fact that the unit type is a left identity for the cartesian product (⊤-×).

6.6 Examples

Consider the following record types (where Fin n is a type containing n values):

```
record R₁ (n : ℕ) : Set where
  field
    f  : Fin n → Fin n
    x  : Fin n
    lemma : ∀ y → f y ≡ y

record R₂ : Set where
  field
    n  : ℕ
    r₁ : R₁ n
```

We can easily define lenses for the x, and lemma fields (implementation omitted):

```
x      : {$n$ : ℕ} → DLens (R$_1$ $n$) (λ _ → Fin $n$)
lemma  : {$n$ : ℕ} →
         DLens (R$_1$ $n$) (λ $r$ → ∀ $y$ → R$_1$.f $r$ $y$ ≡ $y$)
```

I do not see much point in trying to define a lens for the f field in isolation: if the function is changed, then the lemma may have to be modified. However, we can construct a lens that provides a view of both the function and the lemma:

```
f : {$n$ : ℕ} →
    DLens  (R$_1$ $n$)
           (λ _ → Σ$f$ : (Fin $n$ → Fin $n$) · ∀ $y$ → $f$ $y$ ≡ $y$)
```

It is also possible to define a lens for the r_1 field of R_2:

```
r$_1$ : DLens R$_2$ (λ $r$ → R$_1$ (R$_2$.n $r$))
```

Finally we can compose lenses.

```
n$_2$ : DLens R$_2$ (λ $r$ → Fin (R$_2$.n $r$))
n$_2$ = r$_1$ ⨾ x

f$_2$ : DLens R$_2$ (λ $r$ → Σ $f$ : (Fin (R$_2$.n $r$) → Fin (R$_2$.n $r$)) ·
                                ∀ $y$ → $f$ $y$ ≡ $y$)
f$_2$ = r$_1$ ⨾ f

lemma$_2$ : DLens R$_2$ (λ $r$ → ∀ $y$ → R$_1$.f (R$_2$.r$_1$ $r$) $y$ ≡ $y$)
lemma$_2$ = r$_1$ ⨾ lemma
```

Note that, despite the somewhat complicated type of the composition operator, the composed lenses have reasonable types.

6.7 Lenses from containers

This section contains a (perhaps not very useful) observation about the relationship between unary containers and dependent lenses.

A unary container (Abbott et al. 2005) consists of a type of shapes S and for each shape s a type of positions $P\ s$:

```
record Container : Set$_1$ where
  constructor _▷_
  field
    Shape    : Set
    Position : Shape → Set
```

Given such a container one can construct a type constructor:

```
⟦_⟧ : Container → Set → Set
⟦ $C$ ⟧ $A$ = Σ $s$ : Shape $C$ · (Position $C$ $s$ → $A$)
```

The type ⟦ C ⟧ A consists of pairs of a shape and an indexing function mapping this shape's positions to values of type A. Let us name the projections:

```
shape : {$C$ : Container} {$A$ : Set} →
        ⟦ $C$ ⟧ $A$ → Shape $C$
shape = fst

index : {$C$ : Container} {$A$ : Set}
        ($x$ : ⟦ $C$ ⟧ $A$) → Position $C$ (shape $x$) → $A$
index = snd
```

A standard example of a unary container is the list type family, which can be represented as a lens in the following way:

```
List : Container
List = ℕ ▷ Fin
```

The shape of a list is its length, and a list of length n has n positions. The following container instead gives rise to streams:

```
Stream : Container
Stream = ⊤ ▷ λ _ → ℕ
```

There is only one shape, and the type of positions for this shape is the natural numbers. More generally, in a certain extensional type theory unary containers can represent every strictly positive simple type in one variable (Abbott et al. 2005).

It may be interesting to note that a unary container immediately gives rise to a dependent lens:

```
container-to-lens :
  ($C$ : Container) {$A$ : Set} →
  DLens (⟦ $C$ ⟧ $A$) (λ $xs$ → Position $C$ (shape $xs$) → $A$)
container-to-lens $C$ {$A$} = equivalence-to-lens (
  ⟦ $C$ ⟧ $A$                                ≃⟨⟩
  (Σ $s$ : Shape $C$ · (Position $C$ $s$ → $A$)) □)
```

The source type is the container type, and the view family consists of functions from positions to values. The getter function takes a container object and a matching position, and returns the value at that position:

```
container-get :
  ($C$ : Container) {$A$ : Set} ($xs$ : ⟦ $C$ ⟧ $A$) →
  Position $C$ (shape $xs$) → $A$
container-get $C$ = get (container-to-lens $C$)
```

The setter function does not update the value at a given position, it updates the entire position-to-value function:

```
container-set :
  ($C$ : Container) {$A$ : Set} ($xs$ : ⟦ $C$ ⟧ $A$) →
  (Position $C$ (shape $xs$) → $A$) → ⟦ $C$ ⟧ $A$
container-set $C$ = set (container-to-lens $C$)
```

7. Related work

I am not aware of any prior work on lenses with the kind of dependency described above. Ahman and Uustalu (2014) describe dependently typed update lenses. Such a lens is defined for a given directed container, consisting of a set S, an S-indexed family of sets P, and three operations satisfying certain laws, and consists of a set X of sources and a function $\mathsf{act} : X \to \Sigma s : S.P s \to X$ satisfying some laws. The act function can be decomposed into two functions, $g : X \to S$, and $s : (x : X) \to P\,(g\,x) \to X$. This can be compared to a dependent lens $\mathsf{DLens}\,X\,S$, for which the get function has type $(x : X) \to S\,x$, and the set function has type $(x : X) \to S\,x \to X$. Perhaps it would be interesting to combine dependent lenses and dependently typed update lenses.

Hofmann et al. (2012) describe symmetric edit lenses, and discuss a lifting that takes a lens between two "modules" X and Y to a lens between $T(X)$ and $T(Y)$, where T is constructed from an arbitrary container of a certain kind.

8. Discussion

I have presented a notion of (very well-behaved) dependent lenses, proved that a certain form of composition cannot be defined for such lenses, and provided another form of composition which does work.

Several questions remain unanswered. In Section 6.3 I noticed that, in the presence of UIP for the type of small types, non-dependent lenses are isomorphic to dependent lenses, and the first projection lens cannot be defined for all Σ-types. I do not know whether these properties hold in a setting where UIP is replaced by univalence. Perhaps the given construction does not work very well in this setting.

Acknowledgements

I would like to thank Paolo Capriotti and Andrea Vezzosi (who are also mentioned in the text above).

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n° 247219. It has also received funding from the Swedish Research Council (621-2013-4879).

References

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, 2005. doi:10.1016/j.tcs.2005.06.002.

Danel Ahman and Tarmo Uustalu. Coalgebraic update lenses. In *Proceedings of the 30th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXX)*, volume 308 of *Electronic Notes in Theoretical Computer Science*, pages 25–48, 2014. doi:10.1016/j.entcs.2014.10.003.

Paolo Capriotti. Higher lenses. Homotopy Type Theory blog post, at the time of writing available at `https://homotopytypetheory.org/2014/04/29/higher-lenses/`, 2014.

J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view update problem. In *POPL® 2005: The 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages®*, pages 233–246, 2005. doi:10.1145/1040305.1040325.

Martin Hofmann, Benjamin Pierce, and Daniel Wagner. Edit lenses. In *POPL'12, Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 495–508, 2012. doi:10.1145/2103656.2103715.

Benjamin C. Pierce and Alan Schmitt. Lenses and view update translation. Unpublished, 2003.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. First edition, 2013.