Practical Verification of Lenses

Marnix Massar
m.p.massar@student.tudelft.nl

CSE3000 Research Project

Supervisors: Jesper Cockx & Lucas Escot

TU Delft

1. Background

- Agda is a total, dependently typed functional programming language, mostly used for proofs
- Haskell is a functional programming language used for real world applications
- AGDA2HS is a tool for translating code from Agda to Haskell, since libraries written in Agda can contain their own proofs, and libraries written in Haskell cannot

2. Lenses

- A lens is a first class getter/setter for functional programming languages
- Code written with lenses is easier to read and maintain, as is demonstrated in Fig. 1

Is it possible to implement a formally verified version of lenses using AGDA2HS?

```
healPlayer :: Int -> Game -> Game
healPlayer points game = game { _player =
    (_player game) { _status =
        (_status (_player game)) { _health =
            _health (_status (_player game)) + points }}}

healPlayer' :: Int -> Game -> Game
healPlayer' points game = over (player ⊙ status ⊙ health) game (+ points)
```

Fig. 1 - An example use case for lenses. The function healPlayer increments a player's health without lenses. The much smaller and more readable function healPlayer' does the same by composing the lenses player, status, and health. The lens implementations have been left out for brevity.

3. Implementation

Lenses are defined by their ability to put and get, not their implementation. Some options are:

- As a type synonym using functors, so function composition can be reused (Fig. 2)
- As a record type, losing regular functional composition (Fig. 3)

4. Verification

There are certain provable *lens laws* a lens needs to obey to be a *very well-behaved* lens:

- PutGet: Get(Put(v, o)) ≡ v
- GetPut: Put(Get(o), o) ≡ o
- PutPut: Put(v2, Put(v1, o)) ≡ Put(v2, o)

5. Results and conclusions

- A working implementation of lenses as records has been implemented, including verified lenses for tuples and records (Fig. 4, 5 & 6)
- Lenses for list indices were not implemented, due to the difficulty created by the dynamic size of lists
- One potential research direction is the implementation of lenses for dynamically sized structures such as lists
- To implement type-synonym lenses, AGDA2HS needs to support explicit "forall" in types. This too warrants further research

$$\forall f.\mathrm{Functor} f \Rightarrow (a \to \ fa) \to \ s \to \ fs$$

Fig. 2 - Lenses as a type synonym.

$$(s \to \ a) \times (s \to \ a \to \ s)$$
$$(s \to \ a) \times (s \to \ (a \to \ a) \to \ s)$$

Fig. 3 - Lenses as record types.

```
record Lens (s a : Set) : Set where
    field
        get  : s → a
        over : s → (a → a) → s
open Lens public
{-# COMPILE AGDA2HS Lens #-}

put : {s a : Set} → Lens s a → s → a → s
put l o v = (over l) o (const v)
{-# COMPILE AGDA2HS put #-}

one : {q r : Set} → Lens (q × r) q
one = record { get  = fst
             ; over = (λ o f → f (fst o) , (snd o)) }
{-# COMPILE AGDA2HS one #-}
```

Fig. 4 - An example lens 'one' in Agda, which operates on the first element of a tuple.

```
data Lens s a = Lens{get :: s -> a, over :: s -> (a -> a) -> s}

put :: Lens s a -> s -> a -> s
put l o v = over l o (const v)

one :: Lens (q, r) q
one = Lens fst (\ o f -> (f (fst o), snd o))
```

Fig. 5 - Translation of Fig. 4 using AGDA2HS.

```
myTuple :: (Int, String)
myTuple = (2, "mpm")

myUpdatedTuple :: (Int, String)
myUpdatedTuple = over one myTuple (* 11)
-- myUpdatedTuple is now (22, "mpm")
```

Fig. 6 - Example usage of the translation from Fig. 5.