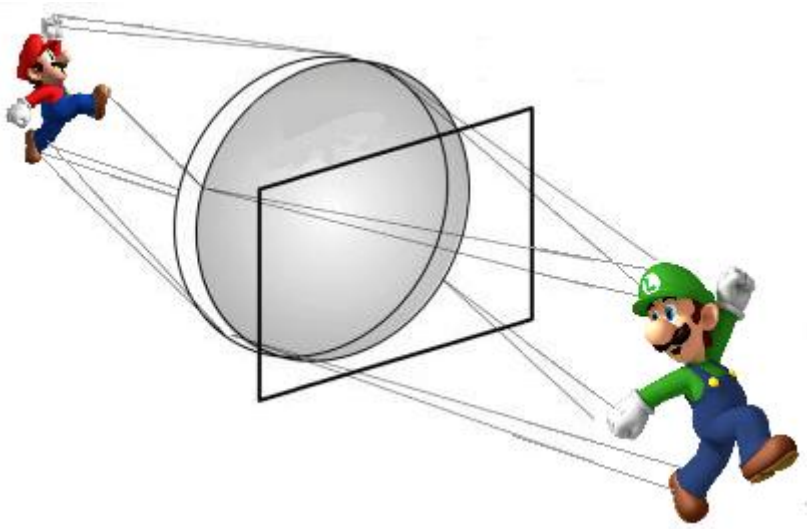
Generic Point-free Lenses

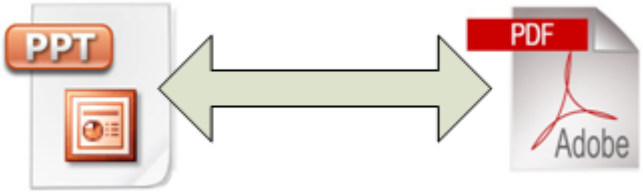


Frederico Valente

Outline

- Problem
- Lenses Overview
- PF Notation and Combinators
- Recursion Patterns on Lenses
- Related Work
- Conclusions

Problem Overview



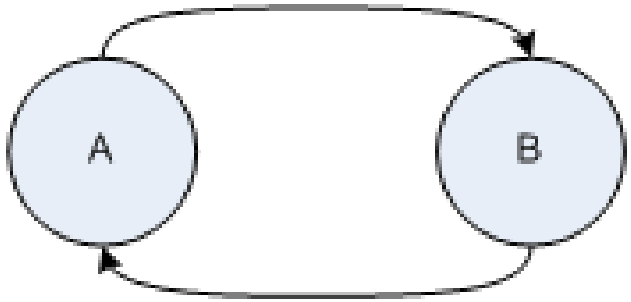
laptop	brand	size	price
MacBook Air	Apple	13.3	1700
VAIO PMT9300	Sony	13.3	2200
ThinkPad Z61m	IBM	15.4	2000

$\geq$

laptop	price
MacBook Air	1700
VAIO PMT9300	2200
ThinkPad Z61m	2000

Lenses

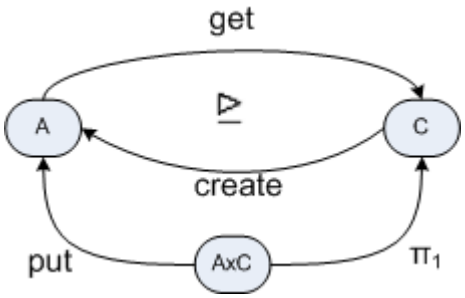
- Transforming a data format into another is essencial to "*bridge the gap"* between technologies
- Bi-directional transformations
- The naive approach is to assemble two one-way transformations



Lenses - properties

- A well-behaved lens has the following properties:
  - *get* must be an abstraction function
  - A lens should be acceptable
  - A lens should be stable

Lenses - Definition



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

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

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

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

PF combinators as lenses

- For complex data-formats the definition of a "*put*" function that guarantees well-behavedness becomes highly complex
  - A combinatory approach is desirable
  - This approach is also central to the point-free style
  - The authors explore this connection and develop a library of PF lens combinators

Point-free combinators as lenses

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

Lens Composability

- The point-free composability of two lenses can be restated as:

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

Lens sum

- We can lift the "*sum*" combinator into a lens:

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

Where dists is defined as:

$$dists : (A + B) \times (C + D) \to (A \times C + A \times D) + (B \times C + B \times D)$$

Functor Mapping

- Functors are special mappings between categories
- A functor *F* from *C* to *D* is a mapping that associates each object X ϵ C an object F(X) ϵ D
- Must preserve identity morphisms and composition of morphisms

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

Recursion patterns as lenses

- Instead of defining lenses by general recursion the authors resort to recursion patterns
- Allows for lenses over inductive data types
  - values that may contain other values of the same type
  - data List a = Nil | Cons a (List a)

Recursion patterns as lenses

- Catamorphism
  - Generalization of folds on lists to arbitrary algebraic data types
  - Encodes the recursion pattern of iteration
  - Filter_left: [A+B] -> A
  - Dual of anamorphisms

Recursion patterns as lenses

- Combinators
  - Anamorphism
    - generalizes the list-producing *unfold* functions to arbitrary algebraic data types
    - Must restrict to finite co-algebras -> guaranteed to halt
    - Dual of catamorphism

Related Work

- Lens over trees – Harmony
- 2LT framework by the authors
- http://hackage.haskell.org -> pointless-lenses
- Tokio researchers

Conclusions

- Authors show how to lift most standard point-free operators and recursion patterns to well-behaved lenses
- An extendable Haskell library was created using point-free lenses and the studied combinators
- Poor performance for complex transformations