# Verified Profunctor Optics in Idris

# Introduction

- The view-update problem is hard in pure functional languages
- Optics are pure functional data accessors and come in many flavours
- Optics solve the view-update problem
- Profunctor optics are a nice encoding but they're complicated
- Formal verification of profunctor optics would be nice

# Idris

- Dependently typed functional programming language and theorem prover
- Very similar to Haskell. Some key differences:
  - `:` and `::` are swapped
  - Dependent types (and thus no type inference)
  - Linear types
- Unique: theorem prover and practical language for Haskell programmers

# Dependent Types

- Types can depend on values, eg `Vect 3 Bool`
- $\Pi$ types: similar to universal quantifiers
  Example: `zeroes : (n : Nat) -> Vect n Int`
  ($\Pi n : \text{Nat}.\text{Vect } n \text{ Int}$ in type theory)
- $\Sigma$ types: similar to existential quantifiers, called dependent pairs
  Example: `filterPos : Vect n Int -> (m:Nat ** Vect m Nat)`
  (the return type is $\Sigma m : \text{Nat}.\text{Vect } m \text{ Nat}$)
- Type inference is undecidable
- Can create an equality type constructor = with one constructor `Refl : x = x`

# Propositions as Types

- Curry-Howard correspondence: logical propositions correspond to types in programming languages
- Propositions are types, valid proofs are well-typed programs
- Dependently typed languages can express first order logic and equalities between expressions

# Propositions as Types

| Logic | Type Theory | Idris Type |
|---|---|---|
| $T$ | $\top$ | `()` |
| $F$ | $\bot$ | `Void` |
| $a \wedge b$ | $a \times b$ | `(a, b)` |
| $a \vee b$ | $a + b$ | `Either a b` |
| $a \Rightarrow b$ | $a \rightarrow b$ | `a -> b` |
| $\forall x.Px$ | $\Pi x.Px$ | `(x:a) -> P x` |
| $\exists x.Px$ | $\Sigma x.Px$ | `(x:a ** P x)` |
| $\neg p$ | $p \rightarrow \bot$ | `p -> Void` |
| $a = b$ | $a = b$ | `a=b` |

Note that the Idris predicates are of the form `P : (x : a) -> Type` where `P x = ()` or `P x = Void`

# Proof Techniques

- Structural induction
- Rewriting types
- Ex Falso Quodlibet
- Boolean reflection

# Proof Techniques

Structural induction and rewriting

Rewriting changes the goal by substitution

```
-- forall n : Nat. n + 0 = n
natPlusZeroId : (n : Nat) -> n + 0 = n
natPlusZeroId Z = Refl
natPlusZeroId (S n) =
  -- Goal is S (n + 0) = S n
  rewrite natPlusZeroId n  -- rewrite n + 0 to n
  -- Goal is S n = S n
  in Refl
```

# Proof Techniques

Structural induction and `cong : (f:t->u) -> (a=b) -> (f a=f b)` (congruence of equality)

```
-- forall xs : List a. xs ++ [] = xs
listConcatRightNilId : (xs : List a) -> xs ++ [] = xs
listConcatRightNilId [] = Refl
listConcatRightNilId (x::xs) =
  -- Goal is x :: (xs ++ []) = x :: xs
  cong (x::)
    -- Subgoal is xs ++ [] = xs
    (listConcatRightNilId xs)
```

# Categories

$$
\begin{tikzcd}
A \arrow[loop left, "\mathrm{id}_A"] \arrow[r, "f"] \arrow[dr, "g \circ f"] & B \arrow[loop right, "\mathrm{id}_B"] \arrow[d, "g"] \\
 & C \arrow[loop below, "\mathrm{id}_C"]
\end{tikzcd}
$$

# Functors

- Structure preserving maps between categories
- $F : C \to D$ maps
  - objects $C \ni A \mapsto F(A) \in D$, and
  - morphisms $Hom(A, B) \ni f \mapsto F(f) \in Hom(F(A), F(B))$
- Satisfy $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$ and $F(f \circ g) = F(f) \circ F(g)$
- In Idris:
  - Type constructors map objects, `fmap : (a -> b) -> (f a -> f b)` maps morphisms
  - Generic containers like lists, trees, pairs are endofunctors
  - `a->` is a functor
- Contravariant functors: functors in the dual category
  - A functor $C^{op} \to C$ in Idris is a functor with a reversed `fmap`, called `contramap : (b -> a) -> (f a -> f b)`
  - Example: boolean predicates of type `a -> Bool`

# Profunctors

- Let $C$ be the category of Idris types, and pretend it's the same as the category of sets
- Morally, a profunctor is a functor $P : C^{op} \times C \to C$
- Encoded as `Type -> Type -> Type` with `dimap : (b -> a) -> (c -> d) -> p a c -> p b d`
- Example: Hom profunctor (others soon)
- Laws: $P(\text{id}, \text{id}) = \text{id}$ and $P(f' \circ f, g \circ g') = P(f, g) \circ P(f', g')$

# Optics

- Lots of types: lenses, prisms, etc.
- Lots of encodings: we'll look at 3

# Simple Optics

Simple algebraic data type encoding

```
record PrimitiveLens a b s t where
  constructor MkPrimLens
  view : s -> a
  update : (b, s) -> t

record PrimitivePrism a b s t where
  constructor MkPrimLens
  match : s -> Either t a
  build : b -> t
```

Lenses are for product types
Prisms are for sum types

# Simple Optics

```
-- Left projection lens
_1 : PrimitiveLens a b (a,c) (b,c)
_1 = MkPrimLens fst update where
  update : (b, (a, c)) -> (b, c)
  update (x', (x, y)) = (x', y)

view _1 (2, List Int) == 2
update _1 ("hello", (2, True)) == ("hello", True)
```

# Simple Optics

Problem: how do we compose optics to get views into composite structures?

Solution 1: van Laarhoven optics

# van Laarhoven Optics (Functor Transformers)

Where `a` is a composite type and `s` is the field type:

```
LaarhovenLens : {f : Type -> Type} -> VFunctor f
  => Type -> Type -> Type
LaarhovenLens a s = (a -> f a) -> (s -> f s)
```

Generic over the functor typeclass/interface. Each functor makes the optic do something different

# van Laarhoven Optics

```
record Const b a where
  constructor MkConst
  unConst : b
Functor (Const b) where ...

record Id a where
  constructor MkId
  unId : a
Functor Id where ...

view : LaarhovenLens a s -> (s -> a)
view optic structure = unConst $
  optic (\x => MkConst x) structure

update : LaarhovenLens a s -> ((a, s) -> s)
update optic (field, structure) = unId $
  optic (\x => MkId field) structure
```

# van Laarhoven Optics

Now composition of lenses is function composition!

But how do we compose lenses and prisms when they're different types entirely?

Solution 2: profunctor optics!

# Profunctor Optics

Profunctor optics are generic over the profunctor typeclass

Cartesian profunctors: have a map `first : p a b -> p (a,c) (b,c)`
Cocartesian profunctors: have a map `left : p a b -> p (Either a c) (Either b c)`

```
Optic : (Type -> Type -> Type) -> Type -> Type -> Type
  -> (Type -> Type)
Optic p a b s t = p a b -> p s t

Lens : Type -> Type -> Type -> Type -> Type
Lens a b s t = {p : Type -> Type -> Type} ->
  Cartesian p => Optic p a b s t

Prism : Type -> Type -> Type -> Type -> Type
Prism a b s t = {p : Type -> Type -> Type} ->
  Cocartesian p => Optic p a b s t
```

# Profunctor Optics

```
-- _1 : {p : Type -> Type -> Type} -> Cartesian p =>
--        p a b -> p (a, c) (b, c)
_1 : Lens a b (a, c) (b, c)
_1 = first
```

Remember: `first : p a b -> p (a,c) (b,c)`

A lens is an optic which can only be defined for Cartesian profunctors, so it needs `first`/`second` as above

# Profunctor Optics

- We can now compose lenses and prisms (the result requires a Cartesian and Cocartesian profunctor)
- Profunctor optics are difficult to write
- There's a correspondence between van Laarhoven and profunctor optics
- Best of both worlds: write simple optics and map them to profunctor optics

# Generic over Profunctors

Updating: use the `Morphism` (Hom) profunctor
Works for all optics

```
_1 : Lens a b (a, c) (b, c)
_1 {p=Morphism} : (a -> b) -> ((a, c) -> (b, c))
```

# Generic over Profunctors

Getters (`view`): use the `Forget r` profunctor
Works for lenses, deconstructs product types

```
record Forget r a b where
  constructor MkForget -- MkForget : (a->r) -> Forget r a b
  unForget : a -> r    -- unForget : Forget r a b -> (a->r)
```

```
VProfunctor (Forget r) where ...
```

```
_1 {p=Forget a} : Forget a a b -> Forget a (a, c) (b, c)
unForget (_1 {p=Forget a} (MkForget (\x => x)))
 : (a, c) -> a
```

# Generic over Profunctors

Dually, constructing sum types (`build`): use the `Const` profunctor

```
record Const r a where
  constructor MkConst  -- MkConst : a -> Const r a
  unConst : a          -- unConst : Const r a -> a
```

```
op : Prism a b (Maybe a) (Maybe b)
op {p=Const} : Const a b -> Const (Maybe a) (Maybe b)
unConst (op {p=Const} (MkConst x)) : Maybe a   -- x : a
```

# What I Did

- Built a small profunctor optics library with lenses, prisms, etc.
- Verified lots of functors, applicatives, profunctors
- Verified some optics

# Profunctor Optics Library

Examples

```
update (op . _1) (\x=>x*x) (Just (3, True)) = Just (9, True)
```

```
view (_1 . _1) ((3, True), Nat) = 3
```

```
build op 3 = Just 3
```

```
update listTraverse (\x=>x*x) [1,2,3,4] = [1,4,9,16]
```

```
update inorder (\x=>x*x) (Node (Node Null 3 Null) 4 Null)
  = Node (Node Null 9 Null) 16 Null
```

# Verification

`VFunctor` = Verified Functor

```
interface VFunctor (f : Type -> Type) where
  -- fmap maps functions
  fmap : (a -> b) -> (f a -> f b)
  -- fmap respects identity, F(id) = id
  fid : (x : f a)
    -> fmap (\x => x) x = x
  -- fmap respects composition, F(g . h) = F(g) . F(h)
  fcomp : (x : f a) -> (g : b -> c) -> (h : a -> b)
    -> fmap (g . h) x = (fmap g . fmap h) x
```

# Verification

```
interface VProfunctor (p : Type -> Type -> Type) where
  -- dimap maps two morphisms over a profunctor
  dimap : (a -> b) -> (c -> d) -> p b c -> p a d
  -- Identity law, dimap id id = id
  pid : {a, b : Type} -> (x : p a b) ->
    dimap (\x => x) (\x => x) x = x
  -- Composition, dimap (f'.f) (g.g') = dimap f g . dimap f'
  pcomp
    : {a, b, c, d, e, t : Type}
    -> (x : p a b)
    -> (f' : c -> a) -> (f  : d -> c)
    -> (g :  e -> t) -> (g' : b -> e)
    -> dimap (f' . f) (g . g') x =
        (dimap f g . dimap f' g') x
```

# Verification

Verified some optics behave as expected, for example:

```
-- view _1 = fst (extensionally)
forgetLeftProjection : (x : a) -> (y : b)
  -> fst (x, y) = view _1 (x, y)
forgetLeftProjection x y = Refl
```

# Difficulties

Extensionality

- Profunctor types are often function types
- Laws require proving profunctor values are equal
- Intensional equality is difficult to prove and in some cases impossible
- Extensionality axiom used in some places

# Related Work

- The seminal paper is from 2017 so the area is new
- Riley, 2018 focused on codifying lawfulness in different types of optics
- Several authors proved the correspondence between different types of optics
  - Approach 1 (Boisseau et. al. 2018): uses the Yoneda lemma
  - Approach 2 (Román, 2019 and Milewski, 2017): uses a construction called Tambara modules and calls the correspondence the profunctor representation theorem
- Román, 2019 wrote a partial proof of the profunctor representation theorem in Agda

# Conclusion

- The library created is practical but far from comprehensive
- Many functors and other structures were successfully verified
- Extensionality issues caused serious problems
- Future work: optics on dependently typed structures like type indexed syntax trees would be useful