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
  - Linear types
  - Dependent types (and thus no type inference)
- Unique: theorem prover and practical language for Haskell programmers

Dependent Types

- Types can depend on values, eg `Vect 3 Bool`
- Π types: similar to universal quantifiers
  Example: `zeroes : (n : Nat) -> Vect n Int`
- Σ types: similar to existential quantifiers, called dependent pairs
  Example: `filterPos : Vect n Int -> (m:Nat ** Vect m 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

```
-- 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
  -- Goal is S n = S n
  in Refl
```

Proof Techniques

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

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

Categories

- Many functional programming design patterns are categorical
- Profunctor optics and van Laarhoven optics use profunctors and functors
- Categories consist of objects and morphisms
- Morphisms can be composed associatively
- Each object has an identity morphism
- Types/sets and total functions form a category
- $Hom(A, B)$: set of morphisms from A to B (assuming locally small)

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 $F(\mathsf{id}_A) = \mathsf{id}_{F(A)}$ 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: predicates of type `a -> Bool`

Applicatives and Monads

Monad

- Special type of endofunctor with `return : a -> m a` and `join : m (m a) -> m a`
- Gives rise to a Kleisli category where morphisms are `a -> m b` and composition uses `join`
  `compose f g = join . fmap f . g`
- Monads can encode side effects in a type safe way

Applicative

- Between a functor and a monad, has `ap : m (a -> b) -> (m a -> m b)` and `return`

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 $C^{op} \times C \to C$
- Encoded as `Type -> Type -> Type` with `dimap : (a -> b) -> (c -> d) -> p b c -> p a d`
- Intuition: data pipeline where you stick an extra stage at the beginning and end
- Examples: Hom profunctor, Hom in any Kleisli category, `Const`, `Forget r` (soon)

Optics

- Types of optics
  - Lenses: optics for product types. Support viewing and updating fields of composite structures
  - Prisms: optics for sum types. Support pattern matching to view if the field is present, updating if the field is present, and constructing sum types from one component
  - Adapters: optics for isomorphic types, e.g. different representations of the same data
  - Traversals: optics for containers like lists and trees
  - Not all optics fall into these categories: what's an optic for a `Maybe (a, Int)`?
- Encodings
  - Simple algebraic data types for lenses, prisms, etc.
  - van Laarhoven functor transformer lenses
  - Profunctor optics
  - ...and many others (isomorphism/residual lenses, etc.)

Simple Optics

```
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
```

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)

In Haskell, where `a` is a composite type and `b` is the field type

```
type LaarhovenLens a b = forall f. Functor f =>
  (b -> f b) -> (a -> f a)
```

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

van Laarhoven Optics

```
newtype Const b a = { unConst :: b } deriving Functor
newtype Id a = { unId :: a } deriving Functor

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

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

- Verified lots of functors, applicatives, profunctors
- Built a small profunctor optics library with lenses, prisms and traversals for pairs, sums, lists, trees, ...
- Verified some optics

Difficulties

- Profunctors are often functions (but not always, see `Const`)
- Profunctor 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 and Future Work

Several papers proved correspondence between types of optics

Optics on dependently typed structures like type indexed syntax trees would be very powerful

Conclusion

Examples

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

Bits and pieces are verified, but it's far from comprehensive

Extensionality issues