Verified Profunctor Optics in Idris

Oliver Balfour

October 23, 2021

Abstract

Optics are a commonly used design pattern in industrial functional programming. They are convenient combinators for reading and updating fields in composite data structures. Common implementations such as Edward Kmett's Haskell `lens` library are highly complex. We discuss profunctor optics, a modern formulation of optics which is more flexible than the more common van Laarhoven formulation. This report discusses the implementation and formal verification of optics in Idris, a dependently typed functional programming language and theorem prover. TODO summarise results, discussion and conclusion

Contents

Introduction

> The Introduction should give us an overview of optics and what they're used for (without going into any detail), a brief overview of formal verification, and the general merits of formally verifying optics. It should then tell us briefly what will be discussed in each section and what we ultimately hope to achieve.

Optics are a pure functional solution to the view-update problem (Foster et al. 2005), the problem of how to neatly read and write small components of large composite data structures.

In industrial functional programming the view-update problem is especially pernicious. Data structures representing components in real world systems frequently have dozens of fields and nested data structures with additional complexity. In a pure functional language, to create a new instance of a structure with only one field updated requires a great deal of boilerplate as below:

```
record Teacher where
  constructor MkTeacher
  name : String

record Student where
  constructor MkPerson
  name : String
  age : Int
  email : String
  teacher : Teacher

changeTeacherName : String -> Student -> Student
changeTeacherName tname (MkStudent sname age email (MkTeacher _)) =
  MkStudent sname age email (MkTeacher tname)
```

As data structures become more complex, writing simple getters and setters becomes extremely tedious. TODO write optics for the above types, noting they probably aren't any better than what you've written

In imperative languages, objects are generally mutated in-place, circumventing this issue altogether. However, even in imperative languages there are often many benefits from using immutable objects. In JavaScript for example, there is an increasing trend towards pure functional state management for designing user interfaces, termed "declarative UI programming." (TODO source) Libraries such as Redux.js (TODO cite) use an immutable state object with a group of actions that act on the state type whenever an event is triggered by a user interaction or network event. This presents numerous benefits such as simple control flow, undo/redo functionality and debugging. However, this requires a new state object to be created whenever an action acts on the existing one, an example of the view-update problem in an imperative setting. The conventional approach in JavaScript is to use Immer.js, which rather than using pure optics exploits esoteric language features to emulate mutability on immutable objects, however there is no fundamental reason optics would not work as well. TODO this isn't really the view-update problem is it? You need to define it with reference to literature. TODO cite https://github.com/immerjs/immer etc

Background

> The background should be written as if the audience was also a computer science student, but not necessarily familiar with these areas. This means most of the work you have in the introduction should really belong in the background. You should introduce (and provide motivating examples of) Functors, Profunctors, Lenses, Optics, Dependent Types, Propositions as Types; Proofs as Programs, and finally - Idris as a functional programming language (like Haskell) and as a theorem prover. You also need supporting references for all concepts discussed here.

Idris

Idris is a Haskell-like functional programming language with first-class support for dependent types. It is an actively developed experimental research language. Syntactically Idris and Haskell are almost identical, the most notable difference is that `:` is used to declare types and `::` is the list `cons` constructor. Additionally, types are first class citizens so functions may accept or return types (values may depend on types), a strict generalisation of Haskell which only allows types to depend on types (type constructors).

Idris additionally has linear types based on quantitative type theory which allow types to be annotated with requirements that they must be used exactly 0 or 1 times at runtime (Brady 2021). Idris also has implicit (inferred) arguments. Unlike Haskell, Idris does not possess type inference, as type inference is undecidable in general for dependent types with non-empty typing contexts (Dowek 1993).

Idris is unique in that it is a practical and simple functional programming language to understand given prerequisite Haskell experience, and it doubles up as a theorem prover. The type system is powerful enough to encode theorems about equalities between expressions and universal and existential quantifiers. This allows programmers to express and prove complex properties and invariants of their programs alongside their code, which makes languages like Idris a good candidate language for critical infrastructure and similar systems.

Dependent Types

Dependent types are types that depend on values. For example, the Idris type `Vect 3 Int` is inhabited by vectors of precisely 3 integers. We say the type is indexed by the value `3`.

Some other languages have equivalent types such as `std::array<int, 3>` in C++. However, in C++, non-type template parameters (that is, values the type depends on) must be statically evaluated because generic types are monomorphised at compile time (ISO 2020, 14.1.4). This means template arguments cannot be non-trivial expressions as in Idris.

There are two main kinds of dependent types. Π types generalise the `Vect 3 Int` example above. The type $\Pi x.Px$, which is expressed as `(x:a) -> P x` in Idris for some `P : (x:a) -> Type` is a function type where the codomain type depends on the value of the argument `x`. This allows functions to dynamically compute their return types in a type-safe manner. For instance, the `replicate` function in the Idris standard library has the type `replicate : (len : Nat) -> a -> Vect len a`, using a Π type to construct a length `len` vector of copies of an object.

The other kind is Σ types, which in Idris are known as dependent pairs. The type $\Sigma x.Px$ corresponds with the dependent pair `(x:a ** P x)` which is a pair of a value and a type where the type may depend on the value. Dependent pairs are outside the scope of this report.

As types can depend on values, Idris has an equality type `=` indexed by two values. It has one constructor `Refl : x = x` (reflexivity). An instance of `Refl : a = b` in some cases is obtainable using type rewriting rules discussed later, in which case the expressions `a` and `b` share the same normal form and are intensionally equal.

Dependent types are useful because they allow programmers to express more sophisticated types such as length indexed vectors, which allow programmers to write total matrix multiplication functions. Additionally, logical propositions correspond with types, and dependent types are expressive enough to allow a language to be used as a theorem prover and formally verify properties of programs.

Propositions as Types

The Curry-Howard correspondence, also known as *Propositions as Types*, is the observation that propositions in a logic correspond with types in a language and proofs correspond with function definitions (Wadler 2015). This observation underpins theorem provers like Idris, Coq and Lean. The theorem statement or goal is encoded in a type signature. The function body is a proof of the goal. If the program is well-typed, the proof is correct.

Every consistent type system encodes some set of logical propositions. Dependent types are expressive enough that they can encode an intuitionistic or constructive logic complete with implications, conjunction, disjunction, negation, quantifiers and equalities.

In Idris, the type `a` is interpreted as a proposition a, where a is true iff `a` as a type is inhabited. A proof of a is simply an object of type `a`. The function type `a -> b` is interpreted as a logical implication $a \implies b$. Intuitively, if a total function of type `a -> b` exists then the existence of an `a` guarantees the existence of a `b`. Logical negation is encoded as `a -> Void` where `Void` is uninhabited.

The equality type is especially useful in conjunction with. If $a = b$ and a constructive proof of this exists then `a = b` is a singleton type, and if no proof exists it is uninhabited and thus false.

A is tabulated below. Σ types are encoded using a construct called dependent pairs, which is not discussed in this report. Π types are encoded with function types where the return type depends on the argument.

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`
$a \Rightarrow b$	$a \to b$	`a -> b`
$\forall x.Px$	$\Pi x.Px$	`(x:a) -> P x`
$\exists x.Px$	$\Sigma x.Px$	`(x:a ** P x)`
$\neg p$	$p \to \bot$	`p -> Void`
$a = b$	$a = b$	`a = b`

Table 1: A complete list of connectives and quantifiers. Note that the predicates in Idris are of the form `P : (x : a) -> Type` where `P x = ()` or `P x = Void`.

Proof Techniques

Idris will reduce values in types to their normal form by applying function definitions. It will attempt to unify both sides of equality types as well by reducing either side until it coincides with the other. This allows proofs to skip many intermediate simplification steps. Idris will generally reduce values in types to their normal form, analogous to simplifying mathematical expressions. For instance `3 + 7 = 11` will be rewritten to `10 = 11` (which of course is uninhabited).

This allows us to write simple proofs as below, which are analogous to unit tests.

```
fact : Nat -> Nat
fact Z = 1
fact (S n) = (S n) * fact n

factTheorem : fact 5 = 120
factTheorem = Refl

factTheorem2 : (S n) * fact n = fact (S n)
factTheorem2 = Refl
```

The main proof techniques in Idris are structural induction, rewriting types and ex falso quodlibet.

Structural induction is the most common tool. This entails case splitting a theorem over each constructor and recursively invoking the theorem on smaller components of an inductively defined structure. If Idris can determine the theorem is total as the recursive calls eventually reach the base case, the proof will type check. Recursive calls are analogous to inductive hypotheses.

For example,

```
-- forall n : Nat. n + 0 = n
natPlusZeroId : (n : Nat) -> n + 0 = n
natPlusZeroId Z = Refl
natPlusZeroId (S n) = cong S (natPlusZeroId n)

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

These proofs invoke a lemma in the Idris Prelude, `cong : (f:t->u) -> (a = b) -> (f a = f b)`, which is analogous to the rule $\forall f.\ a = b \Longrightarrow f(a) = f(b)$ in mathematics.

Idris also provides a facility for rewriting the goal type using an equality. For example:

```
trans' : a = b -> b = c -> a = c
trans' p1 p2 =
  -- goal: a = c
  rewrite p1 in  -- replace `a` with `b` in `a = c`
    -- new goal: b = c
    p2
```

Rewriting can be convenient, however using a number of rewrites makes proofs difficult to follow. Prelude functions such as `trans`, `sym`, `cong` and `replace` accomplish the same task in a more conventional proof structure.

As intuitionistic logics do not have the law of the excluded middle or double negation, proof by contradiction is not possible. Instead, ex falso quodlibet, the principle of explosion, must be used. In some cases a function has cases which are not possible but must have a well-typed proof for those cases to satisfy the totality checker. In this case, rather than deriving a contradiction to show the state is not possible, the contradiction can be used with the function `void : Void -> a` to derive the proof goal.

Limitations

Dependently typed theorem provers are intuitionistic in nature, which is strictly less powerful than classical logic. There exist theorems which can be proven with classical logic for which no constructive proof in Idris exists.

Double negation cancellation is not true in general, as there is no canonical map `((a -> Void) -> Void) -> a`. Existence statements cannot be proven without finding a witness to the proof, so a proof by contradiction that $\neg\forall x.P(x)$ does not imply $\exists x.\neg P(x)$. Instead, a dependent pair containing an explicit x satisfying $\neg P(x)$ must be constructed, which may not be possible.

Additionally, there is a distinction between intensional and extensional equality of functions. In mathematics, the statements $f = g$ and $\forall x.f(x) = g(x)$ are equivalent. In Idris however, only the forward implication is true. Function equality is intensional, meaning functions are equal iff the normal form of their lambda expressions are α-equivalent, so they are the equal up to renaming bound variables. In many cases extensionally equal functions are not intensionally equal, so the Idris equality type may not be helpful. It is possible to use the built-in `believe_me : a -> b` proof to introduce an extensionality axiom, however Idris cannot rewrite types if they invoke axioms as there essentially is no definition to substitute.

These limitations mean that many theorems of interest either cannot be proven or are much more difficult to prove in Idris. TODO examples

Optics

Optics are data accessors that ease reading and writing into composite data structures. Industrial programs tend to have very complex and deeply nested data structures, which makes tasks like copying and updating a single field in a deeply nested data structure in a functional style very cumbersome. Optics are an elegant solution to this problem. They are objects which represent a view into a field of a data structure which can be composed for nested structures and used to view and update the field they model.

There are two common encodings of optics, van Laarhoven optics and profunctor optics. Most established implementations use the older van Laarhoven design, however these have many limitations. One of the principal benefits of optics is that they compose elegantly, however the van Laarhoven encoding makes a distinction between lenses (optics for product types) and prisms (optics for sum types) and does not allow composition of lenses and optics, so you cannot express an optic that reads and writes the integer in a 'Maybe (Integer, String)'.

However, profunctor optics generalise optics to work around these issues. As with many other functional programming design patterns, they are inspired by category theory, specifically the notion of a profunctor. Profunctor optics are generic over the typeclass of profunctors, so they allow choice of profunctor in which to use an optic. This allows programmers to not just use optics to view and update, but to also accumulate side effects in the process.

A major concern is that optics are very complicated. Formal verification of profunctor optics is thus a natural application of dependent types.

TODO what are common/useful optics with examples? can we verify optics? what are lenses vs prisms vs traversals vs adapters?

Functors

Before discussing profunctors, we discuss categories and functors. A category is a mathematical object which consists of a collection of objects and between any two objects a collection of arrows or morphisms (Mac Lane 1970). We will only discuss locally small categories so we may assume these collections of morphisms are sets, called Hom-sets. The only properties categories must have is an associative composition operation on morphisms and an identity morphism on each object. Categories are a useful abstraction as they generalise objects and structure preserving maps between them from many different fields. There is a category of sets where objects are sets and Hom-sets contain functions, $\mathrm{Hom}(A, B) = \{f : A \to B \text{ is a function}\}$. Morphism composition is function composition, and there exists an identity function on each set. In group theory, there is a category of groups where objects are groups and morphisms are group homomorphisms. Many other examples exist, such as partially ordered sets form categories where objects are elements and exactly one morphism exists between every ordered pair.

Notably, types and total functions in Idris form a category similar to the category of sets. For convenience, these categories are assumed the same.

Functors are structure preserving maps between categories (and thus morphisms in the category of categories). They consist of two components mapping objects and morphisms from the domain category to objects and morphisms in the codomain category. Functors respect identities $F(\mathrm{id}_X) = \mathrm{id}_{F(X)}$ and composition $F(f \circ g) = F(f) \circ F(g)$

An endofunctor is a functor which maps into the same category.

It is worth mentioning applicative functors and monads briefly

In Idris, generic containers such as lists and trees are endofunctors. The type constructor 'List : Type -> Type' is the component of the functor mapping objects, and the 'map : (a -> b) -> (List a -> List b)' function is the component mapping morphisms. Additionally, the partially applied arrow type 'a->' is a functor (which we will see is a partially applied Hom profunctor).

Profunctors

Profunctors are a generalisation of functors which relate to Hom-sets. A profunctor from category C to D formally is a functor $D^{op} \times C \to \mathbf{Set}$. (TODO what's a dual category, product category?)

In Idris, profunctors correspond to arrow-like types. For instance, '->' is a profunctor.

Profunctors give rise to a category in which they are the Hom profunctor (is this true in general?)

Intuitively, profunctors are type constructors that construct an function arrow like type.

Hom is a profunctor Hom functor is partially applied hom profunctor

-> is the Hom profunctor Kleisli is the Hom profunctor in the Kleisli category

There are two profunctors of interest: '->' (the standard Hom profunctor in the cat of Idris types) and '(a,b:Type)->(a->f b)' for functor/monad 'f' (the Hom profunctor in the Kleisli category). The latter allows us to accumulate side effects while using our optics

Dictionary / hash map is a profunctor (https://bartoszmilewski.com/2017/03/29/ends-and-coends/)

VProfunctor = verified not v-enriched

Profunctor Optics

what are profunctor optics? why are they good? what are common/useful optics with examples? optics on type indexed data types??? other languages: immer.js, directly writing to relevant fields correspondence with van Laarhoven Boisseau and Gibbons 2018

Related Work

https://bartoszmilewski.com/2017/03/29/ends-and-coends/

http://www.mtm.ufsc.br/ ebatista/2016-2/maclanecat.pdf

https://dl.acm.org/doi/pdf/10.1145/3236779

Research Rabbit

https://dl.acm.org/doi/pdf/10.1145/1040305.1040325

Existing research on profunctor optics

Much existing work is focused on the correspondence between the van Larrhoven and profunctor representations of lenses, prisms, adapters and traversals. Boisseau and Gibbons 2018 provides an elegant proof of the correspondence with the Yoneda lemma, and previous work including Pickering, Gibbons, and Wu 2017 and Milewski 2017 provide similar proofs invoking more complex machinery such as Tambara modules and tensor products.

No prior work is known to have been done on formally verified profunctor optics

Profunctor optics for dependent/indexed types would be lit, eg for type indexed syntax trees (using the Idris type system to make sure your compiler compiles only well typed programs)

todo

We can then have a penultimate section discussing a framework for formal verification of profunctor optics in Idris and discussing the structure and details of your solution (minimal use of source code here). Concretely, this section should properly motivate the formal verification of profunctor optics and then progress through your solution. Talk concepts and give formal examples but avoid using source code (you can refer to sections in the appendix where necessary though).

Finally in the conclusion you can discuss what you've accomplished, what improvements could be made, and other related work in the field (i.e where the current boundaries of formally verified profunctor optics are if any - and general boundaries of profunctor optics research).

For our appendix, we should give the source code - your framework section is welcome to refer to lines or sections here, but should avoid putting the code in directly.

In general you need to go into more detail in the background. What are van Larrhoven optics formally? What problem do they solve, what precisely are their limitations?

Bibliography

Boisseau, Guillaume and Jeremy Gibbons (2018). “What you needa know about Yoneda: Profunctor optics and the Yoneda Lemma (functional pearl)”. In: *Proceedings of the ACM on Programming Languages* 2.ICFP, pp. 1–27.

Brady, Edwin (2021). “Idris 2: Quantitative Type Theory in Practice”. In: *arXiv preprint arXiv:2104.00480*.

Dowek, Gilles (1993). “The undecidability of typability in the $\lambda\Pi$-calculus”. In: *International Conference on Typed Lambda Calculi and Applications*. Springer, pp. 139–145.

Foster, J Nathan et al. (2005). “Combinators for bi-directional tree transformations: a linguistic approach to the view update problem”. In: *ACM SIGPLAN Notices* 40.1, pp. 233–246.

ISO (Feb. 2020). *ISO/IEC 14882:2020 Information technology — Programming languages — C++*. Geneva, Switzerland: International Organization for Standardization. URL: `https://www.iso.org/standard/79358.html`.

Mac Lane, Saunders (1970). *Categories for the working mathematician*. 2nd ed. Springer-Verlag New York.

Milewski, Bartosz (2017). *Profunctor Optics: The Categorical View*. URL: `https://bartoszmilewski.com/2017/07/07/profunctor-optics-the-categorical-view/` (visited on 07/07/2017).

Pickering, Matthew, Jeremy Gibbons, and Nicolas Wu (2017). “Profunctor optics: Modular data accessors”. In: *arXiv preprint arXiv:1703.10857*.

Wadler, Philip (2015). “Propositions as types”. In: *Communications of the ACM* 58.12, pp. 75–84.