Haskell -> Coq → Lean

Chrisil Ouseph

3rd July 2020

A project report submitted in partial fulfilment of the requirements of the MTH302 course of the Spring semester 2020 at IISER Pune

Table of Contents

Abstract

This project report chronicles my study of the functional programming language Haskell as well as the theorem provers Coq and Lean (in that order) under Dr Vivek Mallick from December 2019 to June 2020. The report sheds light primarily on the novel attributes and underlying principles of the languages and not much on the details, including syntax and semantics.

1. Introduction

1.1. About the Report

The foundational principles of the languages are expounded throughout **§**1. Then the focus will shift to specific aspects of each language in the respective sections. The GitHub repository accompanying the report may be found at https://github.com/chrisilouseph/Jan-20-Sem-Proj. All parts of the report regarding Haskell and Functional Programming as well as Lean, CHI and CC have been sourced from [1] and [3] respectively, whereas the table of proof styles in **§**4 was sourced from [2] and [4]. The sections on Coq in the report as well as the repository are minimal since [2] (my chief resource for Coq) concentrates solely on tactics with almost no theory or exercises as well as due to the lack of time to properly practice Coq before having to start Lean.

1.2. Functional Programming

Functional programming is a programming paradigm whose salient features include composing and applying pure functions, referential transparency, immutability, higher-order functions as well as avoiding side effects and shared states. Instead of creating and modifying objects, the program involves data flowing through various pure functions. This yields concise and powerful code and helps developers reason about their programs more effectively.

- **Side effects** are any changes a function makes other than returning an output, like modifying a global variable or writing on the screen.
- **Pure functions** are simply functions without side effects. Function application and composition is the direct analogue of applying and composing mathematical functions.
- **Immutability** refers to having objects that cannot be modified after creation.
- Lack of side effects and immutability ensure that functions return the same output given the same input, i.e. a function call can be replaced with its output. This is **referential transparency**.
- **Higher-order functions** are those which either take other functions as arguments or return a function as output.
- **Shared states** are objects which can pass between scopes, like global variables.

Haskell is a purely functional programming language, i.e. all objects are immutable and side effects and impure functions, whenever unavoidable, must be specifically mentioned.

1.3. Constructive mathematics and the Curry-Howard Isomorphism

Intuitionistic logic or constructive mathematics is that part of mathematics which deals with providing constructive proofs and detailed algorithms to deduce theorems. Essentially, it is the usual mathematics barring the law of excluded middle and the axiom of choice. Haskell Curry and Alvin Howard showed that there is an isomorphism (CHI) between computer programs and constructive mathematics, the **propositions-as-types** paradigm. Under CHI, propositions can be thought of as types, proofs as elements of these types and theorems as functions among these types. After including the missing axioms, this makes interactive theorem proving and type theory almost natural for writing and formalising mathematics.

1.4. Interactive Theorem Provers

Interactive theorem proving is the use of computers to write mathematical proofs. They have a small, trusted kernel program which checks the proofs as well as a library of already formalised mathematics. Over conventional pen-and-paper mathematics, interactive theorem proving has myriad advantages:

- It ensures that a given a proof is indeed correct and catches errors a human might overlook.
- It alleviates the work for referees for research papers as well as teaching assistants for quiz and examination answer sheets, who can then devote more time to research themselves.
- The mathematical library will facilitate searching for specific results in specialised fields.
- Automation in theorem provers can faithfully dispose of easy side cases and lemmas.
- It paves the way for automated theorem proving, where computers will be able to prove theorems themselves or even claim and prove original conjectures.

1.5. The Calculus of Constructions

Type theory is an alternate foundation of mathematics, in which every well-formed expression is associated with a type in a way to circumvent paradoxes like those infamous in naïve set theory. While simple type theory only permits basic functions between types, dependent type theory also allows the creation of functions where the type of the output can change depending on the input (Pi types). The Calculus of Constructions (CC) is a version of dependent type theory used by interactive theorem provers like Coq and Lean as their foundation. CC uses inductive types as well as a countable hierarchy of non-cumulative type universes. **Inductive types** are those whose constructors take as arguments predefined types and/or the type being defined itself. A **type universe** is a collection of types closed under the arrow constructor (->) in which each type itself has a type belonging to the next higher type universe. All propositions form a type universe (the lowest one: Sort 0 in Lean).

1.6. Proof Irrelevance

Both Lean and Coq are based on a version of CC which admits **proof irrelevance**, the concept that any two proofs of a proposition are equal. As far as the languages are concerned, what matters is whether or not the proof of a proposition exists and not its details. Lean has this built-it as a lemma:

```
lemma proof_irrel {a : Prop} (h₁ h₂ : a) : h₁ = h₂ := rfl
```

While Coq makes it a special axiom:

```
Axiom proof_irrelevance : forall (P : Prop) (p1 p2 : P), p1 = p2.
```

2. Haskell

2.2. Static-Typing and Laziness

Haskell is statically typed, as in, the types of all expressions and variables are known at compile-time, drastically reducing time spent debugging code. Although Haskell has type inference—the ability to infer the types of terms using context—in ambiguous situations, the type must be explicitly mentioned.

Also, Haskell is lazy. This means that the evaluation of an expression is delayed until its value is needed, avoiding repeated evaluations. For example, if `l` is a list and `f` and `g` are functions on lists, then while evaluating `f (g l)`, instead of evaluating `g` for the entire list and returning the output list to `f`, Haskell evaluates `f (g l1)`, `f (g l2)` and so on (where `li` is the i^{th} element of the list) and constructs the output list from them. This allows for infinite lists like `x = [1..]` as long as the output is finite and also makes evaluation more efficient.

2.3. Currying

Consider the addition function on integers, `add :: int -> int -> int`. Other languages would flag `add 1` as an error due to insufficient number of arguments. However, Haskell views this as a new function `add 1 :: int -> int` which adds 1 to the input. This is currying: calling a function without providing all the arguments to create a new function whose inputs are those which were not provided earlier and output is the same as if all the inputs were given together.

2.4. Typeclasses

Typeclasses are collections of types called instances which all have similar functions using them. For example, consider the typeclass `Fractional`, whose default instances are `Float` and `Double`. The typeclass has three functions associated to it which need to be defined for a type `a` to be made an instance of the typeclass, namely `(/) :: a -> a -> a`, `fromRational :: Rational -> a` and `recip :: a -> a`. A function `f :: (Fractional a) => a -> a` will now accept and return inputs of type `Float` as well as `Double`.

2.5. Functors and Monads

Functors and Monads are important typeclasses with several prominent instances like `Maybe`, `[]`, `Either` and `IO`. Their definitions are given below:

```
class Functor (f :: * -> *) where
  fmap :: (a -> b) -> f a -> f b
  (<$) :: a -> f b -> f a
```

```
class Applicative m =>
Monad (m :: * -> *) where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
```

Here, `Applicative` is the typeclass of applicative functors and `(* -> *)` is the kind of type constructors that take exactly one concrete type and return a concrete type. Although many types can be made an instance of these typeclasses using contrived functions of appropriate types, it is beneficial only when the types obey certain laws. These are the two **Functor Laws** and three **Monad Laws**:

F1 (Identity-to-Identity): `fmap id = id`

F2 (Homomorphism): For all appropriate functions, `fmap (f . g) = fmap f . fmap g`

M1 (Left Identity): For any monadic function, `return x >>= f = f x`, or `f <=< return = f`

M2 (Right Identity): For any monadic value m or function f,
`m >>= return = m`, or `return <=< f = f`

M3 (Associativity): For all monadic functions, `f <=< (g <=< h) = (f <=< g) <=< h`

3. Lean

3.1. Inheritance from Functional Programming

Several properties of functional programming languages have carried over into theorem provers. Like Haskell, Lean has **currying**, which may be used to partially specialize theorems to get useful implications. Lean also has a **typeclass system**; algebraic structures such as `group` and `ring` and several properties of types like `nonempty` and `has_add` are implemented as typeclasses. Lean's syntax is also heavily influenced by Haskell's, including the arrow operator, match-with-end, etc.

3.2. Decidable Propositions

`decidable` is the typeclass of decidable propositions (those that are either true or false) with two constructors: `is_false : ¬p → decidable` and `is_true : p → decidable`. In classical logic, all statements are decidable. However, in constructive logic, only certain propositions can be proved to be decidable, for example, equality and order in natural numbers.

3.3. Added Axioms

As mentioned in the introduction, constructive logic is not sufficient for modern mathematics and Lean has three more default axioms:

- `propext : ∀ {a b : Prop}, (a ↔ b) → a = b`
  This is the propositional extension axiom, which states that equivalent propositions are actually equal. This lets us substitute a proposition with a more convenient equivalent form.
- `classical.choice : Π {α : Sort u}, nonempty α → α`
  The controversial axiom of choice produces an element of a type given that it is non-empty.
- `quot.sound : ∀ {α : Sort u} {r : α → α → Prop} {a b : α}, r a b → quot.mk r a = quot.mk r b`
  This axiom of quotients says that equivalent terms get mapped to the same equivalence class.

4. Proof Styles in Coq and Lean

4.1. Table of Common Proof Components

Coq Tactic	Lean tactic	Lean proof term	Use/Effect
intro(s)	intro(s)	assume	Fix an arbitrary element of a Pi type
exact	exact		Finish a proof with an exact proof term
pose	have	have	Add a new element to the context
apply	apply		Use implications to transform goals
refine	refine		Generates subgoals using implications and constructors
simpl	simp		Simplify goals and/or hypotheses
unfold	(d)unfold		Unfold definitions of terms
case/destruct/elim	cases	elim	Generate a subgoal for each constructor of an inductive type. If the hypothesis is an 'or' statement, generate 2 subgoals where the left and right propositions are true separately
admit	sorry	sorry	Create arbitrary proof term temporarily
induction	induction	cases_on	Proof by induction on an element of an inductive type
assumption	assumption		Finish a proof by a term directly from context
reflexivity	refl	rfl	Solve a trivial equality
trivial	trivial		Solve a variety of easy goals
contradiction	contradiction	absurd	Prove false using contradictory statements
rewrite	rw	▸	Replace a term with an equivalent one
ring	ring		Solve goals with just addition & multiplication
tauto	tauto		Solve tautologies in constructive logic
repeat	repeat		Apply the given tactic repeatedly until failure
;	;		Apply the tactic on the right to all subgoals produced by that on the left

4.2. A Proof using Coq tactics

```
Theorem not_exists :
(forall P : Set -> Prop, ~(exists x, P x) <-> (forall x, ~ (P x))).
Proof.
    intro P.
    split.

        intros h x hpx.
        apply h.
        exact (ex_intro P x hpx).

    intros h1 h2.
    destruct h2 as [x h2].
    case (h1 x h2).
Qed.
```

4.3. A Proof using Lean tactics

```
theorem not_exists : ∀ P : Type → Prop, ¬ (∃ x, P x) ↔ ∀ x, ¬ P x :=
begin
    intro P,
    split,

        {intros h x hpx,
        apply h,
        exact ⟨x, hpx⟩},

    intros h1 h2,
    cases h2 with w hw,
    exact h1 _ hw
end
```

4.4. A Proof using Lean proof terms

```
theorem not_exists : ∀ P : Type → Prop, ¬ (∃ x, P x) ↔ ∀ x, ¬ P x :=
λ P, ⟨λ hne x hpx, hne ⟨x, hpx⟩, λ hfn ⟨x, hpx⟩, hfn x hpx⟩
```

5. Acknowledgements

First and foremost, I extend my sincere gratitude to Dr Vivek Mallick for letting me undertake this exotic project and believing in me to see it through. I cannot put into words how much I appreciated his effort to recall Haskell and Coq syntax while deepening my understanding of such foreign concepts.

I thank my family whose financial and moral support have extended far longer than these seven months.

Finally, I would also underline my appreciation of Dr Scott Morrison of the Australian National University for accidentally introducing me to interactive theorem proving as well as Prof Siddhartha Gadgil of IISc for intentionally introducing me to Lean. These fine gentlemen might just have changed my entire life's course.

6. Bibliography

[1] Lipovača, Miran. *Learn You a Haskell for Great Good!: A Beginner's Guide*. N.p.: No Starch, 2012. Print.

[2] Nahas, Mike. "Mike Nahas's Coq Tutorial." *Nahas_tutorial*. N.p., 2012. Web.

[3] Avigad, Jeremy, Leonardo De Moura, and Soonho Kong. "Theorem Proving in Lean." *Theorem Proving in Lean - Theorem Proving in Lean 3.4.0 Documentation*. N.p., 2017. Web.

[4] Foster, Nate. "Coq Tactics Cheatsheet." *CS 3110 Coq Tactics Cheatsheet*. Cornell University, 2018. Web. 02 July 2020.