Research in Industrial Projects for Students

Final Report, July 2019

Exploring a Type Theoretic Library for Python Environment

In collaboration with Nvidia

Project team

Ryan Brill, U.C. Berkeley

Tianyu Liu, National University of Singapore

Arshay Sheth, National University of Singapore

Yutong Zou, National University of Singapore

Academic mentor

Ziyuan Gao, National University of Singapore

Industry mentors

Zhangsheng Lai

Aik Beng Ng　　Vincent Tang

This projected was jointly supported by Nvidia, the Institue for Mathematical Sciences (NUS), and the Institute for Pure and Applied Mathematics (UCLA).

Abstract

In a recent paper, Leon Buttou makes a distinction between machine learning and machine reasoning, prefering the latter because it provides the reasons *why* a statement holds true. Type theory, an alternative to set theory, seems to be a promising start to achieve machine reasoning. The primary advantages of type theory is that its proofs are constructive and it is easier for machines to check type-theoretic proofs. Currently, a few robust type-theoretic theorem provers like Coq and Isabelle exist. However, these programs are esoteric and difficult to learn, and they are difficult to interface with AI/ML algorithms which are mostly implemented in Python. Hence our goal: we want to explore a type-theoretic environment in Python. Conceretely, in this project, we conduct a literature review on type theory, conduct a literature review on Coq, build a small type-theoretic library in Python, dissect a type-theoretic Python theorem prover we found online called Holpy, and compare these different type-theoretic environments.

Acknowledgements

We would like to express our very great appreciation to Mr. Zhangsheng Lai, our industry mentor, for his constructive suggestions and inspiring discussions during the planning and development of this project. We would also like to thank Mr. Ziyuan Gao, our academic mentor, for his advice and assistance in keeping our progress on schedule.

Our grateful thanks are also extended to Prof. Adrian Roellin and Dr. David Chew for their enthusiastic encouragement, useful critiques, and help in offering us resources during this program. We would like to offer our special thanks to Prof. Ilya Sergey. His willingness to give his time and share his knowledge and experience in the related field is very much appreciated.

We would also like to thank Institute for Mathematical Sciences at the National University of Singapore for hosting this program and Nvidia for sponsoring our project.

Finally, we would like to thank the Institute for Pure and Applied Mathematics at UCLA, and NSF grant DMS-1440415, for supporting our project.

Contents

Chapter 1

Introduction

In recent years, there has been an increasing interest in advancing artificial intelligence in the reasoning field. Though deep learning performs exceptionally well at pattern recognition, its ability to emulate reasoning is still underdeveloped. This observation forms the backbone of our project. In this chapter, we give a brief background on the key concepts our project.

1.1 The proposed problem

Nvidia is interested in understanding and adapting a body of mathematics known as Type Theory towards enabling **type theoretic machine reasoning**. This is a reasoning system based on type theory through which a machine can generate conclusions or inferences from available knowledge. In type theoretic machine reasoning, one can think of questions posed to the computer as theorems to be proven and answers to the questions as proofs of the theorem. We shall explain this by presenting an explicit example. Suppose we input the following sentences to a computer: (1) Mary went to the bedroom. (2) Mary took the football there. (3) John went to the bedroom. (4) John took the football to the garden. We now ask: where is the football? To answer this question using machine reasoning, we would break it into two smaller subquestions: Who was the last person with the football? What was the last location this person has been to? Then the fact John was the last person with the football would be a theorem to be proven. Similarly, the fact that John was last at the garden would also be a theorem to be proven. Once both these theorems have been proven, the computer will conclude that the football is at the garden.

Nvidia has used Coq, an interactive theorem prover, to implement machine reasoning for such examples. In contrast to frameworks such as Python, Coq is not widely usable by others in the research community due to its steep learning curve. Thus, the main goal of the project is: **analyze and extract the key components of Coq to be represented as an easily used library in Python for the purpose of enabling type theoretic machine reasoning**. Nvidia has found that **inductive types** and **record types** played an important role when they implemented machine reasoning in Coq. Thus, we will especially focus on these types and implement them into Python. We shall give a detailed explanation of these types in Chapter 5 and Chapter 6.

1.2 Prior related work

Type theory is a powerful mathematical language enabling us to perform formal design and verification, optimize computer software and hardware systems, and reason about them in a systematic and rigorous way. In the past decades, many other computer scientists and research teams have been working on adapting type theory to theorem proving.

The focus of our research, Coq, is a formal proof management system developed by *Inria* that allows us to express and verify complex mathematical definitions, theorems and proofs based on dependent type theory (Barendregt and Geuvers, 2001). Coq adopted *CIC*, the Calculus of Inductive Constructions (Coquand and Huet, 1986), as the logical framework and was implemented in the programming language *OCaml*. Besdies Coq, some other proof assistants which we found relevant to our research include Isabelle and Lean:

Isabelle, written in ML, is a widely used proof assistant. It was first released in 1986 written by Larry Paulson at the University of Cambridge and Tobias Nipkow at Technische Universitt Mnchen. The main difference between Isabelle and Coq is that the former is based on the logical system of higher order logic. Roughly speaking, higher order logic is an extension of propositional and predicate logic.

Lean is a project launched by Leonardo de Moura at Microsoft Research Redmond in 2013 (de Moura et al., 2015). It has a powerful logical foundation for expressing complex mathematical objects and supports mathematical reasoning as well as reasoning about complex systems, and proving assertions in these realms based on dependent type theory. We believe this is close to our aim of exploring the possibility of such a theorem prover in a Python environment.

1.3 Our team's approach

We began our project by investigating Coq, and especially looked into how it defines inductive and record types, performs type checking and conducts simple theorem proving. Then, we conducted a literature review in different versions of type theory and higher order logic and their various implementations. We then developed a Python framework encapsulating the basic logical framework of type theory which enabled us to implement the basics of inductive and record types into Python. Subsequently, we came across the work done by researcher Bohua Zhan at the Institute of Software at the Chinese Academy of Sciences. Zhan has a similar goal to ours, but he is working on a much larger scale: he is currently building a proof assistant in Python called Holpy(Higher order logic in Python). Having across Holpy, we studied it and did a comparision & contrast study with Coq.

1.4 Overview of the report

In chapter 2, we give a brief overview of type theory. In chapter 3, we explain our type theoretic framework for Python. In Chapter 4, we talk about implementing a type checker. In chapter 5, we introduce inductive types, and we compare their implementation in Coq and Python. In chapter 6, we do the same for Record Types. In Chapter 7, we introduce Holpy and in Chapter 8, we discuss the advantages and disadvantages of Coq and Holpy.

Chapter 2

Brief Overview of Type Theory

In this chapter, we will give a short introduction to type theory.

2.1 Historical background

Type theory was originally proposed by Bertrand Russell as a solution to the paradox bearing his name in naive set theory. This paradox prompted mathematicians to put mathematics on a stronger axiomatic and logical foundation. Two competing theories, namely set theory and type theory, emerged. Set theory is currently more popular and widespread. However, due to the pioneering work of Per Martin-Löf, Thierry Coquand, and other mathematicians, type theory is slowly developing into an area of active research. We now proceed to explain some of the basic concepts and ideas of type theory. We would like our readers to gain a feel of the subject; our exposition will thus not be completely rigorous and we will only focus on the big picture.

2.2 Some Basic Concepts

Type theory has a single primitive relation $x : T$, which is read as " term x has type T". Thus, the two most important notions in type theory are **types** and **terms**. For example, $0 : \mathbb{N}$ means 0 has type $\mathbb{N}$, representing the fact 0 is a natural number. In type theory, every typable term has a unique well-defined type (Thompson, 1991).

In type theory, all theorems are types. For instance, "for all $n : \mathbb{N}, n = n$" is a type. To prove a theorem T in the framework of type theory, you must provide a term having type T. Hence in type theory, we have a correspondence between terms and proofs, and another correspondence between types and theorems. These correspondences are known as the **Curry-Howard Isomorphism** (Howard, 1980). The Curry-Howard Isomorphism is one of the most important principles in type theory and we will make extensive use of it in the discussion that follows.

We now explain briefly how to encode propositional logic in type theory. We first start with two types which we call 0 and 1, or *False* and *True*. By the Curry-Howard Isomorphism, we can interpret these types as propositions and so terms of these types should be the proofs of these propositions. Since 0 should have no proofs, 0 is a type that has no terms. Since 1 is a tautology, it should have just one term which we will call *.

The next connective that we would like to encode in type theory would be the implication $P \implies Q$. Again, via the Curry-Howard Isomoprhism, we can interpret P and Q as types.

Having done so, we introduce an important class of types in type theory which is that of a **function type**; we write this as $P \to Q$ for the given types P and Q. Thus, a term of $P \to Q$ would look like $f : P \to Q$. Intuitively, this means that f is a proof of the fact that from P we can deduce Q. By the Curry-Howard Isomorphism, we can use such terms: if we have a term $p : P$ then we can use $f : P \to Q$ to construct a term $f(p) : Q$.

So far, we have considered 0, 1 and function types. These types can interact with each other. For instance, if X is any type, we can consider the function type $0 \to X$. Since 0 has no terms, the type $0 \to X$ has only one term. This corresponds to the vacuous proof of a proposition. In mathematics, we often write B^A to denote the set of all functions $A \to B$. Since the function type $0 \to X$ has only one term, we have an equivalence $X^0 \simeq 1$ in type theory. This is analogous to the fact that if we raise any number to the power of 0, we get 1. Similarly, we can also consider the function type $1 \to X$. Any term of $1 \to X$ must be a proof of the fact that from 1 we can deduce X. Since 1 is just a tautology consisting of the single term $*$ and since the terms of X are the proofs of X, the terms of $1 \to X$ correspond exactly to functions $\{*\} \to X$. The latter turns out to be equivalent to X since such functions are determined by where they map $*$. In summary, we have an equivalence $X^1 \simeq X$ in type theory. Again we have analogy: any number raised to the power of one equals the number itself. Function types can also be used to encode the connective $\neg$. To see this, recall that in mathematics $\neg P$ is equivalent to $P \implies \mathit{False}$; so in type theory $\neg X$ can be encoded as the function type $X \to 0$.

Let us now consider the propositional connectives $\wedge$, $\vee$. Suppose we have a type T_1 with term x_1 and also a type T_2 with term x_2. Then, regarding T_1 as a proposition, x_1 is a proof of T_1 and similarly, x_2 is a proof of T_2. If we put these two proof together, for example as a pair (x_1, x_2), we would then get a proof of $T_1 \wedge T_2$. But such pairs would naturally lie in the direct product $T_1 \times T_2$. The direct product is represented as a **product type** in type theory and is also denoted as $T_1 \times T_2$. Thus, the connective $\wedge$ can be encoded as a product type in type theory. The idea behind implementing the connective $\vee$ is similar. Suppose we have types T_1 and T_2, both interpreted as propositions. We can produce a proof of $T_1 \vee T_2$ if we have a proof of T_1 or a proof of T_2. Such a proof would lie in the disjoint union $T_1 \coprod T_2$. Disjoint unions are implemented in type theory as a **sum type** and so sum types can encode the connective $\vee$.

The preceding paragraphs show how propositional logic can be completely encoded within type theory. However, type theory is more powerful; for instance, we could also encode predicate logic within type theory. Indeed, as mentioned at the start of this chapter, type theory is emerging as an alternative theory that can serve the role of being a foundational theory for mathematics. However the following question remains unanswered: if set theory can already provide a strong foundation for mathematics, why should we study type theory?

2.3 Advantage of Type Theory over Set Theory

In the last section, we saw the important role that the Curry-Howard Isomorphism plays in type theory. We saw that proving a theorem is equivalent to constructing a term which has the type of the theorem. If we translate this to the world of programming languages, then since programs can be used in a step by step procedure to construct such terms, we see that proofs can essentially be regarded as programs. Thus, the key advantage of type theory is that **it is easier for a computer to process type theoretic proofs**. Indeed, many proof assistants such as Coq are based on type theory. As mentioned in the last chapter, Nvidia is currently interested in pursuing the following analogy to achieve machine reasoning: questions asked to the computer should be regarded as theorems to be proven, and proofs of the theorem should constitute correct answers to the question. Therefore, it seems even more natural for us to delve into type theory and type-theoretic proofs.

Chapter 3

Type Theoretic Framework in Python

In this chapter, we explain our idea to create a type-theoretic framework in Python. Our main idea is to create two basic abstract classes of type and term expressions respectively. We denote them by `Type` and `Term`. Recall from the last chapter that in type theory, each term has a unique type. Thus, a type must be defined first before any term of that type is defined. The typing judgment and user-defined constructors are stored in the abstract class `Rule`.

3.1 Type

We created three classes `BaseType`, `UserDefType` and `Connective` that inherit from `Type`. We now proceed to describe each of them.

Suppose we want to prove a statement about all types. We would accomplish this by proving the statement for an arbitrary type. When dealing with an arbitrary type, say A, we know nothing about A, besides the fact that it is a type. We encode this idea via the class `BaseType`. To establish that A is a type, simply create a `BaseType` with name `"A"`.

The class `UserDefType` is used to implement inductive types in type theory. The class `Record` inherits from `UserDefType` because a record type is a special form of an inductive type that requires only a single constructor. We explain inductive and record types in detail in Chapter 5 6; for now, we encourage the reader to think of inductive types as sets that can be created by rules that are known as constructors.

The class `Connective` is an abstract class that forms expressions in propositional logic. It consists of three derived classes `Implication`, `Conjunction` as well as `Disjunction`.

3.2 Term

Terms are the building blocks of type theory. The class `Term` has several child classes including `Variable`, `Const`, `Application`, `Abstraction`, `OrderedPair` and `RecordTerm`.

A `Variable` is an individual variable in a lambda-expression, representing a parameter or mathematical/logical value. It is important to encode variables/parameters via the class `Variable` because it gives us a systematic way to substitute/reduce the variable in a function call.

A `Const` (constant) is a variable introduced by the user in defining an inductive or record type. For instance, constructors (like `O` and `S` for `nat`) are constants. Constructors are constants because we need to somehow record the type of constructors, while also giving the system a way to keep track of the constructor names and ensure they are not reused.

`Abstraction` is function definition: it encodes lambda function notation. An Abstraction is comprised of an argument, the type of the argument, and the function body. `Application` of F to X means applying a function F to an argument X. Both F and X are lambda terms.

An `OrderedPair` in typed lambda calculus corresponds to Conjunction in Type, meaning that only an `OrderedPair` term can have type `Conjunction`.

A `RecordTerm` (record term) is a term of a record type.

More terms will be developed for different uses in the future work.

3.3 Rule

In type theory, a *rule* is a known typing judgment. For example, when we define the natural numbers, we have the rules: `nat` is a type, `O` is a `nat`, and `S` is a `nat->nat`. For us, the abstract class `Rule` represents the rules in type theory and the user-defined constructors. You can use either `BaseTypeIntro` or `UserDefTypeIntro` to introduce new rules into the system.

Chapter 4

Type Checker Implementation

One of the key components of the project was to implement a working kernel which checks the type of a given term. This is made possible by the decidability of type checking in the Calculus of Inductive Constructions (CIC), which is the type theory underlying Coq. In this chapter, we will explain the basic design and implementation of type checker in Python.

4.1 Type Checking in Type Theory

In our type theoretic framework, all terms inherit the abstract class $Term$. The input of type checking should be a term, and the checker returns the type of the given term, which is an object of a class inheriting the abstract class *Type*. One of the main difficulties of the type checker is to decide whether a given term is typable (Nederpelt and Geuvers, 2014).

Before going over a few examples of type checking, we discuss notation. In *λ-calculus*, *abstraction* has the form $\lambda x_A.e$. In this notation, x is a variable (parameter) of the function, and $x : A$. The function can only be called on a term of type A - for example, $y : A$ with $(\lambda x_A.e)y$ - and the result of this function call is replacing all instances of x in e with y.

Moreover, we need to discuss $\vdash$ notation. Each step of a proof results in a *sequent*, consisting of a set of assumptions (*antecedents*) and a conclusion (*consequent*). A sequent with antecedent $A_1, ..., A_n$ and consequent C is written

$$A_1, ..., A_n \vdash C$$

The antecedents of a theorem correspond to your hypotheses, or what you know/assume, and the consequent corresponds to what you want to conclude/show. In general, we use Γ to denote an arbitrary antecedent.

Example 4.1.1

$$\Gamma \vdash \lambda x_A.x : A \to A$$

The abstraction $\lambda x.x$ is a typable term and has type $A \to A$.

Example 4.1.2

$$\Gamma, y : A \vdash \big(\lambda x_A.x\ y\big) : A$$

Let $I \equiv \lambda x_A.x$, the application $I\ y$ is valid and has type A as $y : A$ is in the context. By β-conversion we have:

$$I\ y \to_\beta\ y$$

Then $\Gamma, y : A \vdash y : A$ gives

$$\Gamma, y : A \vdash (\lambda x_A.x\ y) : A$$

Example 4.1.3 The application $\lambda x_A.x\ y$ is not typable in the context $y : B$ and A, B distinct types. This is because the abstraction $\lambda x_A.x$ has type $A \to A$ and requires an applicant of type A. The term y fails the criteria.

Example 4.1.4

The term $\boldsymbol{x}\ \boldsymbol{x}$ is not typable in $\lambda_\to$

We give a simple proof of the example here. The application requires the first x to be an abstraction. Suppose $x : A \to B$, where A and B are distinct types. In order to make the original term typable, we need $x : A$ for the second x. Then we have

$$x : A \to B$$

$$x : A$$

By uniqueness of types, this leads to a contradiction. So the term $x\ x$ is not typable (Thompson, 1991).

It is worth noticing that type checking is decidable (meaning there is an effective method to check whether arbitrary terms are of a given type) in $\lambda_\to$ as well as λ_C, which enables the implementation of a type checker. However, the current prototype does not contain any components of automated theorem proving such as tactics or hints.

4.2 Implementation of Type Checker

The type-checking function is implemented as a method of `TypeChecker` class. The type checker is separated as a module to reduce coupling. The method `type_check` checks whether the input term is a *Variable*, *Const*, *Abstraction* or *Application*. If it is not a *Variable* or *Const*, the type checker goes further to check the type of its components. The method `type_check` is called *recursively* to check the components in the terms.

Here is the Type Checker implementation in Python:

```
class TypeChecker:
    """
        The kernel of type checking
    """

    def __init__(self):
        self.assumptions = {}
        self.rules = []
        self.base_rules = []

    def type_check(self, term):
        """
            Type_check recursively check the type of a given term.
            A term is either an individual variable, an abstraction or an implication.
        :param term: Term
```

```python
:return: Type
"""
assert isinstance(term, Term)

if isinstance(term, Variable):
    """"Individual variable of base type"""
    if term.expression() in self.assumptions:
        return self.assumptions[term.expression()]
    for rule in self.base_rules:
        if rule.forward(term):
            assert isinstance(rule.typ, BaseType)
            return rule.typ
    raise Exception("Variable not defined")

if isinstance(term, Const):
    """"terms which are user defined constants"""
    for rule in self.rules:
        if rule.forward(term):
            assert isinstance(rule.typ, Type)
            return rule.typ
    raise Exception("Const not defined")

elif isinstance(term, Abstraction):
    """"typed abstraction"""
    term_arg = term.argument
    term_type = term.argument_type
    term_body = term.body
    assert isinstance(term_arg, Term)
    if isinstance(term_arg, Variable):
        term_str = term_arg.expression()
        self.assumptions[term_str] = term_type
        return Implication(term_type, self.type_check(term_body))
    else:
        raise Exception("Invalid Abstraction")

elif isinstance(term, Application):
    """terms of Application"""
    term_left = term.left
    term_right = term.right
    type_left = self.type_check(term_left)
    type_right = self.type_check(term_right)
    if isinstance(type_left, Implication) and type_left.left == type_right:
        return type_left.right
    else:
        raise Exception("Invalid Implication")
raise Exception("Not Implemented yet")
```

Chapter 5

Implementation of Inductive Types

One of the key components of the project was to replicate inductive types into Python. In this chapter, we will explain how inductive types work in Coq, and our approach to implementing them in Python.

5.1 Inductive Types in Coq

An arbitrary expression $a : A$ in type theory tells us that term a is of type A, but this expression gives us no information on the nature, number or properties of these terms. This observation motivates the notion of an **inductive type**: an inductive type is a type equipped with rules that explain how the terms of a type are built. Formally, these rules are known as **constructors**. We remark that any term of an inductive type must be constructed by some constructor and moreover every term must be constructed by using only a finite number of such constructor applications. We now proceed to give some examples of inductive types. Our first example, which is a prototypical example of an inductive type, is the type of natural numbers $\mathbb{N}$.

Example 5.1.1 The natural numbers $\mathbb{N}$ are constructed by using the following two constructors:

$$0 : \mathbb{N}, \quad S : \mathbb{N} \to \mathbb{N}$$

The first constructor declares that 0 is of type $\mathbb{N}$ while the second constructor declares that S is of function type $\mathbb{N} \to \mathbb{N}$. In this way, we can interpret the number 1 as $S(0)$, 2 as $S(S(0))$ and so on. The analogous code for implementing the natural numbers in Coq is as follows:

```
Inductive nat :  Set := | O : nat | S : nat -> nat.
```

Another example of an inductive type would be that of a binary tree:

Example 5.1.2 In type theory, we can create a type *btree* whose terms are constructed by the following constructors:

$$\textit{emptytree} : \textit{btree}, \quad N : \textit{btree} \times \textit{btree} \to \textit{btree}$$

The first constructor declares that there is a term *emptytree* which is of type *btree*. The second constructor declares that N is of function type $\textit{btree} \times \textit{btree} \to \textit{btree}$. Thus, whenever we have two terms of type *btree*, we can create a third term of type *btree*. In Coq, we would implement this construction as follows:

```
Inductive btree :  Type := | emptytree :  btree | N : btree -> btree -> btree.
```

Thus far it seems that all inductive types are recursively defined and have infinitely many terms. However, this may not always be the case since we can also have very simple examples of an inductive type defined by enumeration:

Example 5.1.3 We can define a type *Bool* which consists of just two terms: *true* and *false*. Thus, *Bool* will have the following two constructors which say that *true* is of type *Bool* and *false* is of type *Bool*.

$$\mathit{true} : \mathit{Bool}, \quad \mathit{false} : \mathit{Bool}$$

Implementing this in Coq will be very similar:

```
Inductive Bool :  Set := | true:  Bool | false:  Bool.
```

We end this section by remarking that one of the most important properties of inductive types is that they enable us to do proofs by induction. For instance, to prove a property P holds for all natural numbers, we need to prove that P holds for 0 and, if P holds for k then P holds for $S(k)$.

5.2 Implementing Inductive Types in Python

Examining the above examples of inductive types, we see that an inductive type is completely specified by its name, sort, constructor names, and constructor types. For instance, `nat` is completely specified by the name `nat`, the sort `Set`, the constructor names `O` and `S`, and the constructor types `nat` and `nat -> nat`. Furthermore, the constructor types of an Inductive type must satisfy some extra conditions: well-formed constraint, positivity constraint, and universe constraint (van Staal, 2012).

Because Inductive types have a common structure, it is clear that the easiest way to encode inductive types into Python is via *Object Oriented Programming* (OOP). Specifically, OOP is a fitting framework for Inductive types because:

1. We can store and access all the relevant fields, such as name, sort, constructor names, and constructor types.
2. The type checker can manipulate Inductive types with ease, via `isinstance()` and accessing its fields.
3. We can store Inductive type objects into the rules/type checker and access it from there with ease.

Here is the Inductive Type implementation in Python:

```python
class UserDefType(Type):
    def __init__(self, typename):
        self.typename = typename

    def __repr__(self):
        return self.typename

    def __eq__(self, other):
        """Overrides the default implementation"""
        if isinstance(other, UserDefType):
            return self.typename == other.typename
        return False

    def name(self):
        return self.typename
```

Here is how we would construct the booleans:

```python
bool_test = UserDefType("bool_test")
tru = Const("tru")
fal = Const("fal")
rule_b0 = UserDefTypeIntro(tru, bool_test)
rule_b1 = UserDefTypeIntro(fal, bool_test)
check.rules.append(rule_b0)
check.rules.append(rule_b1)
```

And here is how we would construct the natural numbers:

```python
nat = UserDefType("nat")
S = Const("S")
O = Const("O")
rule1 = UserDefTypeIntro(O, nat)
rule2 = UserDefTypeIntro(S, Implication(nat, nat))
check.rules.append(rule1)
check.rules.append(rule2)
print(check.type_check(S).name())
print(check.type_check(O).name())
print(check.type_check(Application(S, O)).name())
```

The output of the print statements above would be:

```
(nat => nat)
nat
nat
```

Chapter 6

Implementation of Record Types

Another key component of the project was to replicate record types into Python. In this chapter, we will explain how record types work in Coq, and our approach to implementing them in Python.

6.1 Record Types in Coq

A **Record Type** is an inductive type with one constructor, having additional properties. The purpose of a Record Type is to encode mathematical objects that are comprised of objects of different types.

The classic example of a record type is the rational numbers. The rationals are comprised of fields of four different types. Specifically, the top and bottom of a rational are natural numbers; the sign of a rational is a boolean; dividing by 0 is undefined, which is given by the type "bottom not equal to 0"; and a rational must be irreducible, which is given by a verbose type shown below.

Here, we show how the record type for rational numbers is written in Coq:

```
Record Rat : Set := mkRat {
    sign : bool ;
    top : nat ;
    bottom : nat ;
    rat_bottom_cond : 0 <> bottom ;
    rat_irred_cond : forall x y z : nat,
                     (x*y) = top /\ (x*z) = bottom
                     -> x = 1
}.
```

6.2 Implementing Record Types in Python

Looking at this example, we see that a record type is completely specified by its name (Rat), sort (Set), constructor name (mkRat), field names (sign, top, bottom, etc.), and field types (bool, nat, etc.).

As with inductive types, we found that object oriented programming is the easiest and best way to reproduce the functionality of record types in Python. OOP in Python is superior because:

1. We can use `isinstance()` to make type checking easier.
2. We may abstract away the details of record types into the class, allowing us to treat a record type as a type and not as its internals.
3. We may structure the record type in exactly the way we want it - which is to store its name, constructor name, field names, and field types.

Here is the Record Type implementation in Python:

```python
class Record(UserDefType):
    def __init__(self, typename, field_dict, type_checker):
        """
        typename : str
        field_dict : dictionary { keys - str : values - Type}
        """
        # assert that each Type in field_dict is a valid type
        super().__init__(typename)
        self.field_dict = field_dict
        self.check = type_checker

    def __repr__(self):
        s = ""
        for field_name in self.field_dict:
            field_type = self.field_dict[field_name]
            s = s + ' '*2 + str(field_name) + " : " + str(field_type) + "\n"
        return self.typename + " { \n" + s + "}"
```

6.3 Implementing Record Terms in Python

Terms of a record type, which we call **Record Terms**, are created in Coq via the lone constructor of a record type. For example, the rational number 1/2 is created like so:

```
Definition half = mkRat true 1 2 bottom_cond_proof irred_cond_proof.
```

Looking at this example, we see that a record term is completely specified by the constructor and the field terms (for each field of a record type, supply a term with type matching the field type).

As with record types, we found it is easy to use OOP in Python to replicate the functionality of record terms. The additional benefits of OOP for record terms are:

1. The `__call__()` magic method allows us to extract specific fields of a record term.
2. The `__eq__()` magic method allows us to specify when two record terms are equal.
3. The `__repr__()` magic method allows us to display record terms to the user in the way we like.
4. The `__init__()` magic method allows us to type check each field term, and to store the field terms.

Here is the Record Term implementation in Python:

```
class RecordTerm(Term):
    field_dict = None

    def __init__(self, record_type, type_checker, *fields):
        """
        make a term of the record type record_type
        :param record_type: a record type
        :param type_checker: a type checker
        :param fields: a list of terms which have the correct types as defined by
                       the field_dict in record_type

        :self.field_dict: a dictionary {keys - str (field names) : values - terms,
                          those in *fields}
        """
        self.record_type = record_type
        self.check = type_checker
        self.field_dict = record_type.field_dict

        assert len(fields) == len(self.record_type.field_dict)
        counter = 0
        for field_name in self.record_type.field_dict:
            field_type = record_type.field_dict[field_name]
            field = fields[counter]
            assert self.check.type_check(field) == field_type
            self.field_dict[field_name] = field
            counter += 1

    def __repr__(self):
        s = ""
        for field_name in self.field_dict:
            field_value = self.field_dict[field_name]
            s = s + ' '*2 + str(field_name) + " = " + str(field_value) + "\n"
        return self.record_type.typename + " term " + "{ \n" + s + "}"

    def __call__(self, field_name):
        """
            :param field_name: str
        return the term associated with field_name
        """
        return self.field_dict[field_name]

    def __eq__(self, other):
        """Overrides the default implementation"""
        if isinstance(other, RecordTerm):
            return self.record_type == other.record_type \
                   and self.check == other.check \
                   and self.field_dict == RecordTerm.field_dict
        return False
```

Here is an example of how to use Record Types in our Python library:

```
rat = Record("rat",
             {"top": nat, "bottom": nat, "sign": bool_test},
             check)
print(rat)
half = RecordTerm(rat,
                  check,
                  Application(S, O),
                  Application(S, Application(S, O)),
                  tru)
print(half)
print(half("top"))
print(half("sign"))
```

The outputs are as follows:

```
rat {
  top : nat
  bottom : nat
  sign : bool_test
}
rat term {
  top = (S O)
  bottom = (S (S O))
  sign = tru
}
(S O)
tru
```

Chapter 7

Holpy

In the process of building our own type-theoretic library in Python, we discovered that a group of researchers in China led by Dr Bohua Zhan are also building a type-theoretic environment for Python: Holpy.

7.1 Introducing Holpy

Holpy (Higher Order Logic in Python) is a prototype proof assistant developed by Bohua Zhan at Institute of Software, Chinese Academy of Sciences (Zhan, 2019). Holpy is developed based on higher-order logic (simply-typed lambda calculus) that has been successfully employed in HOL4, HOL-Light, and Isabelle/HOL. Similar to the proof assistants mentioned above, Holpy does not produce explicit proof terms, but represents theorems as sequents with a set of antecedents and a single consequent. The primitive deduction rules, such as implication introduction and implication elimination, are similar to those in previous HOL-based systems. By introducing the type "Set", along with axioms of ZFC set theory, it is possible to construct an environment for formalizing mathematics in set theory within higher-order logic.

7.2 Overview of Holpy Implementation

The design of Holpy includes three interconnected ideas: extensive use of macros, a JavaScript Object Notation (JSON)-based format for theory files, and an Application Programming Interface (API) for implementing proof automation in Python.

The most important aspect of Holpy's design is its extensive use of macros for representing proofs. Traditionally, in the context of proof terms (representation of proofs by terms, via the Curry-Howard Isomorphism), a macro is a special proof term that can be expanded into a larger proof term using an associated procedure, and serves as an abbreviation of the latter for efficiency purposes. In Holpy, the Curry-Howard correspondence is not used explicitly, so that the proof terms are represented implicitly.

In Holpy, a *proof rule* is defined as a function which maps a list of input theorems to a new theorem. A proof item specifies a step of deduction using a proof rule. It consists of an identifier, the name of the proof rule used, the argument supplied to the proof rule, and a list of identifiers of other proof items, from which the input theorems are to be obtained. A proof is represented as an ordered list of proof items. Each proof item can only refer to previous items in the list. A theorem can be associated to each proof item in the proof by invoking the proof rules in sequence. The result of the proof is the theorem associated to the last proof item.

The second component of Holpys design is a foundational format for theory files based on JSON. Most existing proof assistants choose a file format that is not supported by many tools . As these files are usually stored internally, any such tool must re-implement a large part of the core of the proof assistant. In contrast, Holpy is designed with a foundational format for theory files (containing definitions and theorems) based on JSON. While the format is human readable and partially human editable, it is not well-suited for direct editing by users. Instead, the intention is that the file will be read and edited by other tools, including user interfaces that display the content and reflect changes in the files.

The third aspect is the proof automation API in Python. Most proof assistants today support user-defined proof automation. Coq, for example, supports user-defined tactics with the Ltac language. Lean also has meta-programming language implemented in Lean itself for proof automation. However, these are all based on the meta-programming language of the proof assistants or functional programming language like ML. Holpy makes it possible to implement proof automation in Python, which is more user-friendly and flexible, with the support of thousands of powerful libraries. The separation of proof checking and macro expansion ensures the soundness of the proof automation despite the weak type and memory safety enforcement of popular imperative languages.

7.3 Example: Theorem Proving in Holpy

Holpy uses a method of describing proofs known as **natural deduction**. In natural deduction, each step of a proof results in a **sequent**, which consists of a set of assumptions called **antecedents** and a conclusion called **consequent**. A sequent with antecedents $A_1, \dots, A_n$ and consequent C is written as $A_1, \dots, A_n \vdash C$. We now give some examples of **primitive natural deduction rules** that Holpy uses; they are a set of rules that are always assumed to give valid sequents. We list the name of the rule, its notation in Holpy and finally the content of the rule itself:

- *Assume*, assume: $\frac{}{A \vdash A}$
- *Implication Introduction*, implies_intr: $\frac{A \vdash B}{\vdash A \to B}$
- *Implication Elimination*, implies_elim: $\frac{A \vdash B, \vdash A}{\vdash B}$
- *Conjuction Introduction*, conjI: $\vdash A \to B \to A \wedge B$
- *Conjunction Destruction*, conjD1: $\vdash$: $A \wedge B \to A$
- *Conjunction Destruction*, conjD2: $\vdash$: $A \wedge B \to B$

As an illustration, consider the code below. It shows (for instance) how the previous three rules are implemented in Holpy and then goes on to use the above rules to give a proof of the proposition $A \wedge B \to B \wedge A$.

```
thy = basic.load_theory('logic_base')
print("conjI:", printer.print_thm(thy, thy.get_theorem('conjI'), unicode=True))
# out[1]: conjI:  A -> B -> A /\ B
print("conjD1:", printer.print_thm(thy, thy.get_theorem('conjD1'), unicode=True))
# out[2]: conjD1:  A /\ B -> A
```

```
print("conjD2:", printer.print_thm(thy, thy.get_theorem('conjD2'), unicode=True))
# out[3]: conjD2:  A /\ B -> B

A = Var("A", boolT)
B = Var("B", boolT)
th0 = Thm.assume(conj(A, B))
th1 = thy.get_theorem('conjD1')
th2 = Thm.implies_elim(th1, th0)
th3 = thy.get_theorem('conjD2')
th4 = Thm.implies_elim(th3, th0)
th5 = thy.get_theorem('conjI')
th6 = Thm.substitution({"A": B, "B": A}, th5)
th7 = Thm.implies_elim(th6, th4)
th8 = Thm.implies_elim(th7, th2)
th9 = Thm.implies_intr(conj(A, B), th8)

print(printer.print_thm(thy, thm9, unicode = True))
# out[4]:  A /\ B -> B /\ A
```

Chapter 8

Comparing Holpy and Coq

We have now discussed two type-theoretic theorem provers, Holpy and Coq. To learn more about the programs, we found it prudent to compare and contrast them.

8.1 Remarks on Holpy

We like the fact that Python is the language of Holpy's implementation. It is nice that Holpy is implemented in Python because it is easy for curious coders to delve deeper into Holpy's implementation, whereas the Coq kernel is implemented in the abstruse OCaml language. It is nice that Holpy's user interface is in Python because it will be easier for people to learn how to use Holpy, and it will be easier for people to integrate Holpy with their other Python programs.

Earlier in the project, when we were trying to build a small type-theoretic Python library, we realized that an object oriented approach was a good approach because it would allow us to capture the functionality of types, terms, type checking, and inductive types, and record types. In creating Holpy, Bohua Zhan agrees with us, as Holpy is structured via object oriented programming (OOP). That two independent groups came to the same conclusion (that OOP is a good way to program type theory into Python) strengthens our belief in OOP for type theory, and it strengthens our belief in Holpy.

We like that Holpy is easy to integrate with other coding projects. Specifically, Theorem and Proof objects may be extracted as JSON files, which can be easily integrated into many programmming languages (such as Java, C++, and Swift). This may be helpful for running machine learning algorithms on proofs/theorems, or for proof checking massive files in a different proof checking system. Another advantage of Holpy is its extensive use of macros (abbreviations for potentially large proof terms), which thus makes Holpy scalable (able to handle large proofs).

As a fledgling program, the current version of Holpy also has its flaws. Holpy's primary flaw of concern to Nvidia is that inductive types are minimally implemented, record types are not implemented, and proofs on inductive and record types are not yet supported in Holpy. Another potential problem is that theorem proving is not interactive: users need to prove the theorem from scratch without any support from Holpy. Furthermore, predicate logic is underdeveloped: Holpy has minimal proof support for the $\forall$ and $\exists$ quantifiers. This is problematic because many theorems involve these quantifiers. Lastly, Holpy lacks a developer guide, which makes it difficult to extend.

We should bear in mind that since Holpy is under active development and is still a work in progress, many of these flaws may be remedied in the future. For instance, all the missing functionalities discussed above have been implemented in many other theorem provers, indicating that there is a high chance that it is possible to implement them in Holpy as well. Indeed, Zhan claimed, in response to an email from us, that he would be implementing inductive types in the future.

8.2 Remarks on Coq

We feel that Coq's biggest strength is its versatility: it can be used to prove a wide variety of statements about myriad mathematical structures. The natural numbers, groups, rings, graphs (among many others) can all be encoded into Coq, and theorems about these statements can be proved constructively. Beyond standard mathematical structures, customized ones can be encoded into Coq too. An example is English sentences - if we have a series of statements such as "Mary left the football in the bathroom", "John picked up the football", and "John gave the football to Joseph", we can encode this into Coq and prove theorems about the location of the football.

Furthermore, software verification is an important application of Coq. If one has a large and possibly byzantine computer program, too large to verify correctness by inspection, it may be prudent to verify the correctness of your program; software verification is the process of reasoning on the correctness of your software. Because of Coq's versatility, programmers are able to encode the structure of other computer programs into Coq, and are then able to reason on this software, especially for programs involving advanced specifications (such as language semantics or real numbers) (Paulin-Mohring).

Another of Coq's strengths is proof automation. In fact, there are theorems whose known proofs are so large that they have only been able to be carried out via a proof assistant. The infamous Four Color Theorem - "The regions of any simple planar map can be colored with only four colors, in such a way that any two adjacent regions have different colors" - is one such theorem (Gonthier, a). In 2008, George Gonthier and Benjamin Werner proved the Four Colour Theorem using Coq. The idea of the shortest known proof today is to reduce the theorem to around 600 cases and conduct a proof by exhaustion, but 600 cases is too much to do by hand, so it is done in Coq. What results is a proof with 0.2% done by a human and 99.8% done by a machine, whose proof is correct because Coq is certified to be bug free (Gonthier, b).

Lastly, Coq is widely used in academia, and has a strong community of dedicated users. Coq's strong academic community makes it great for research applications, because there will be a network of other smart and motivated people who can provide assistance and collaboration if needed.

Now let us discuss some disadvantages of Coq. Coq is robust due to deliberate planning and many years of in-depth development; thus, Coq does not have many flaws. Its primary pitfall, however, is that it has a steep learning curve and is difficult to use. It took us two weeks to get acquainted with the basic ideas of Coq, and we are still far from proficient in Coq. Unfortunately, this conflicts with Nvidia's goal of having people in industry use theorem provers to conduct machine reasoning, because Coq is just too difficult to use. The second problem with Coq is that it is not possible to interface with Python, which is used by most programmers and machine learners in industry, so in order to use Coq, one must learn Coq.

Solutions to these problems are easier in principle than in practice. For instance, to mitigate Coq's steep learning curve, Coq should be widely taught in university courses so that more people become familiar with it. Also, the Coq team should develop more online and user-friendly tutorials.

8.3 Key differences between Holpy & Coq

A stark difference between Holpy and Coq is that proofs in Coq involve “backward chaining” logic, whereas proofs in Holpy involve “forward chaining” logic (natural deduction). So, to prove a theorem in Holpy, we start from nothing, add the assumptions of the theorem, apply steps called “deduction rules” to build up to the theorem, and stop once we have reached a satisfactory conclusion. On the other hand, in Coq, we start from the theorem, and apply “tactics” to break the theorem down in to simpler subtheorems, which we continue to break down via tactics, until we reach the state “no more subgoals”. Natural deduction is easier for proofs which we have already written out, because most standard proofs are written via natural deduction. But we are dealing with automated theorem proving, which is beneficial for large and complicated proofs, and in this case backwards chaining logic is superior. This is because in building up a theorem via natural deduction, it is often unclear how to connect the current state to the final theorem statement, but in breaking down a theorem into subgoals, the current goal is precisely stated. This backwards-chaining environment is referred to as *interactive* theorem proving because we are essentially interacting with the machine - supply a tactic to Coq, and Coq sends back a new subgoal.

Bibliography

Barendregt, H. and Geuvers, H. (2001). Proof-assistants using dependent type systems. *Handbook of automated reasoning*, 2:1149–1238.

Coquand, T. and Huet, G. (1986). *The calculus of constructions.* PhD thesis, INRIA.

de Moura, L., Kong, S., Avigad, J., Van Doorn, F., and von Raumer, J. (2015). The lean theorem prover (system description). In *International Conference on Automated Deduction*, pages 378–388. Springer.

Gonthier, G. Formal proof - the four-color theorem. *Notices of the American Mathematical Society 55(11).*

Gonthier, G. The four colour theorem: Engineering of a formal proof. In *Computer Mathematics, ASCM 2007.*

Howard, W. A. (1980). The formulae-as-types notion of construction. *To HB Curry: essays on combinatory logic, lambda calculus and formalism*, 44:479–490.

Nederpelt, R. and Geuvers, H. (2014). *Type theory and formal proof: an introduction.* Cambridge University Press.

Paulin-Mohring, C. Introduction to the coq proof-assistant for practical software verification.

Thompson, S. (1991). *Type theory and functional programming.* Addison Wesley.

van Staal, W. (2012). Inductive types in coq. http://cs.ru.nl/freek/courses/tt-2012/talks/presentation.pdf.

Zhan, B. (2019). holpy: Interactive theorem proving in python. *arXiv preprint arXiv:1905.05970.*