Comprehensive Exam Report 1 - Foundations of Theorem Provers

Arjun Viswanathan

1 Introduction

In this report, we will compare two historic interactive theorem provers - LCF (Logic for Computable Functions) and Automath. Both proof assistants are driven by a small kernel, and reduce proof checking to type checking but their type systems vary significantly. The Curry-Howard Isomorphism is central to the Automath type system, whereas type checking in LCF is reduced to type checking in its implementation language of abstract data type instances. We take a look at the fundamentals of these theorem provers and how these have manifested in modern proof assistants that have been inspired by these two systems.

2 De Bruijn's Automath

N. G. de Bruijn introduced the Automath language in 1968 [9, 8] as a means to formalize mathematics in computers so that it can be mechanically checked. This led to the Automath project that, among other things, produced a proof-checker for the Automath language [7, 20].

Inspired by colleague A. Heyting, De Bruijn realized that in a typed language, logical propositions can also be considered types of their corresponding proofs, which constitute terms of the respective types. This crucial insight allowed logic to be done in a typed language. This correspondence between propositions and types, and proofs and terms was discovered independently by H. B. Curry and W. Howard and is thus termed as the Curry-Howard(-de Bruijn) Isomorphism.

2.1 The Curry-Howard Isomorphism

The Curry-Howard Isomorphism encapsulates the correspondence between proofs and terms (programs) and between propositions and types. In doing so, it equates the notion of proving to computing. Additionally, it reduces proof-checking to type-checking. Initially, Haskell Curry noticed the correspondence between minimal implication logic and a simple $\lambda-$ calculus, which was extended by William Howard [16].

Minimal implication logic is a subset of propositional logic with only implication and its introduction and elimination rules. It can be characterized by the following natural deduction rules:

$$\frac{}{\Delta \vdash A}\ (\texttt{ax}) \qquad \frac{\Delta \cup \{A\} \vdash B}{\Delta \vdash A \to B}\ (\to \texttt{-in}) \qquad \frac{\Delta \vdash A \to B \quad \Delta \vdash A}{\Delta \vdash B}\ (\to \texttt{-el})$$

Here, Δ is a set of implicational formulas and A and B range over implicational formulas.

Untyped $\lambda-$calculus gives a basic mechanism for dealing with functions, making it the foundation of functional programming. The syntax of terms in untyped $\lambda-$ calculus is as follows:

$$\Lambda := \nu \mid (\lambda\nu.\Lambda) \mid (\Lambda\ \Lambda)$$

where ν represents a variable, λ is used to denote function abstraction, and two juxtaposed terms represent the application of the first one to the second. Given a context Γ that gives types to free variables, the rules for deriving types of λ-terms are as follows:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ (\texttt{var}) \qquad \frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}\ (\texttt{abs}) \qquad \frac{\Gamma \vdash M : A \to B \quad \Gamma \vdash N : A}{\Gamma \vdash M\ N : B}\ (\texttt{app})$$

Curry noticed an isomorphism between these 2 systems of rules. Howard extended this to predicate logic and to Heyting arithmetic. The Curry-Howard isomorphism for the systems of natural deduction for minimal implication logic and the typing rules of untyped lambda calculus terms consists of two mappings, one of which can be stated as:

If Σ is a derivation in natural deduction with conclusion $\Delta \vdash A$, then $\bar{\Delta} \vdash \bar{\Sigma} : A$,

where $\bar{\Sigma}$ is a $\lambda-$term, and $\bar{\Delta}$ is a context such that for each $B \in \Delta$, there exists some x such that $x : B \in \bar{\Delta}$. There is also a reverse mapping that completes the isomorphism:

If $\bar{\Delta} \vdash \bar{\Sigma} : A$, is a typable term, then there exists a natural deduction derivation Σ with conclusion $\Delta \vdash A$.

2.2 Automath to Coq

The Curry-Howard Isomorphism is prevalent in Automath and subsequent theorem provers that were motivated by Automath. Some of these include Agda [22], Coq [6], and the Lean theorem prover [10]. One of the most popular of this lineage of Automath-style theorem provers is Coq. In Coq, a theorem is a formula which is proven using functions called tactics. To prove a conjecture in Coq, it suffices to (and is necessary to) provide to Coq, a term whose type is the proposition being proved. In most cases, this is done using tactics that continuously

modify the proof term. Initially, the entire term is a "hole" to be filled, that has the goal's type, and tactics break this down into smaller and smaller holes, until all of them are filled.

In addition to Automath, de Bruijn's contribution to the interactive theorem proving world includes an answer to the question 'Why is a mechanical proof, more dependable than a written one?'. A proof assistant satisfies the "de Bruijn criterion" [5] if it generates proof terms that can be checked independently from the system by a simple program that a skeptical user could write him/herself.

By reducing the proof checker of Coq to a type checker for propositions, the Curry-Howard isomorphism helps Coq fulfill the de Bruijn criterion. Type checkers are small and straightforward programs that are - in principal - easier to check externally. While this was likely more true for a simpler system like Automath than it is for a modern one like Coq (which has a much more complicated type system), it remains true that type-checkers are relatively simple in the space of all programs, which contributes to the widespread acceptance of properties proved in Coq.

Over the years, Coq has been enhanced with complicated tactics and even tactic languages. The soundness of these tactics doesn't affect the soundness of the Coq proof system, because it is only concerned with the well-typedness of the proof term corresponding to a proposition, not with how the term was constructed. A buggy tactic, by definition, would construct an ill-typed proof term, which wouldn't pass through the Coq type-checker.

To demonstrate how Coq utilizes the Curry-Howard Isomorphism, consider the following example, motivated by Chapter 9 of the Logical Foundations book [25].

Logical conjunction in Coq is defined as a proposition (`Prop` in Coq) that can be constructed only using the two conjuncts. It is defined as an inductive type with only one constructor:

```
Inductive and (P Q : Prop) : Prop :=
        | conj : P -> Q -> and P Q.
```

Noting that $\wedge$ is the infix notation for conjunction, a proof of `P ∧ Q -> Q` would involve using tactics that extract the evidence of `Q` from that of `P ∧ Q`. Specifically:

```
Theorem ex : forall P Q, P /\ Q -> Q.
Proof.
  intros P Q HPQ.
  destruct HPQ as [HP HQ].
  apply HQ.
Qed.
```

The proof script between `Proof.` and `Qed.` consists of tactics that complete the proof. The `intros` tactic localizes the variables `P` and `Q` and evidence `HPQ` for `P ∧ Q`. The propositions-as-types correspondence is on display here with `ex` having type `forall P Q, P ∧ Q -> Q` and `HPQ` having type `P ∧ Q`. The `destruct`

tactic separates the proof of the conjunction into proofs of its conjuncts, and the `apply` tactic plugs the right conjunct into the hole representing the goal. Finally, the `Qed` command calls Coq's type-checker to verify that the term constructed by the tactics has the type of the lemma trying to be proved.

To see the proof term that has `forall P Q, P ∧ Q -> Q` as its type, consider the following equivalent definition of `ex`.

```
Definition ex : forall P Q, P /\ Q -> P :=
  fun P Q HPQ =>
    match HPQ with
          | conj HP HQ => HQ
    end.
```

This definition defines a function that behaves exactly like the tactics do - given the evidence for `P ∧ Q`, it destructs this evidence by pattern matching on all things a conjunction can be. From our definition above, we know that it can only be a combination of the proofs of its conjuncts. From the only case of the match, it returns the right proof term. This function definition defines the exact proof term that is constructed using the tactics above, thus demonstrating the proofs-as-terms correspondence.

3 Milner's LCF

Dana Scott developed a logic for reasoning about computable functions at Stanford, and Robin Milner who visited Stanford and worked with Scott, later built the Edinburgh LCF (Logic for Computable Functions) system [13]. The LCF system provided a functional programming language ML, designed to implement the LCF logic and write proofs in it. ML abbreviates meta language, to distinguish itself from the object language it is encoding, that is, the proof calculus. The features of ML include a polymorphic type system, functions as first-class objects that can be passed around like terms, and abstract datatypes - data structures that restrict access to the internal via a fixed set of functions.

For Edinburgh LCF, Milner had the idea of recording proof results, and not entire proofs, once a proof had been checked. This avoided a lot of redundancy and saved space. He implemented this using an ML abstract datatype `theorem` for defining theorems in LCF. The predefined values of this ADT were instances of axioms and its operations were inference rules. Since this `theorem` type was the only way to create new theorems in the LCF system, type-checking would ensure that the only way to create theorems was via their axioms and inference rules. Since the type-checker guaranteed that only valid theorems could be created, once an instance of a theorem had been created by the type-checker, LCF didn't have to remember how it was created, just that it was created soundly.

Milner also introduced the concepts of tactics and tacticals, and imported the concept of backward proofs to proof assistants. While it was common for proofs to be constructed forward - from the axioms to the goal to be proved,

a backward approach breaks down the goals into simpler (sub-)goals and eventually to something resembling the axioms. A tactic is a function that helps in this reduction, and tacticals can combine tactics to perform tasks such as repetitions and branching. Tactics and tacticals add some automation to proof procedures, but they lie outside the trusted proof kernel. A tactic is a function operating on values of type `theorem` and a tactical is a combination of such functions. The functions will only have the intended effect on the value, if they are sound transformations, as deemed by the type-checker.

For example, consider the *Modus Ponens* rule, also represented as the implication elimination rule in the previous section:

$$\frac{\Gamma \vdash p \to q \quad \Gamma \vdash p}{\Gamma \vdash q}\ MP$$

This is a rule with two premises and a conclusion. In the LCF system, this is an ML function with type `theorem` $\to$ `theorem` $\to$ `theorem`. It takes two objects of type `theorem` and returns a `theorem` object.

3.1 The HOL Family

LCF moved to Cambridge [23] from Edinburgh through Milner's collaborators. The HOL (Higher-Order Logic) [12] theorem prover was created for hardware verification, but its support for higher-order logic made it and its descendants relevant for various applications. The first stable version of HOL - HOL88 - was written in Common Lisp. HOL90 was then implemented in Standard ML. HOL Light [15] was another variant implemented in Caml Light that prioritizes constructive logic over classical logic.

As new logics were designed, so were proof assistants that could reason about them. Isabelle [26] was born as an LCF-style proof assistant for Martin-Löf's constructive type theory and grew into a general framework to deal with multiple logics. Lawrence C. Paulson developed it as an LCF-like system that provided a meta-logic, within which other logical frameworks could be declared - in the previous HOL systems, the logic had to be implemented in ML. Higher-order unification along with backtracking is central to proof search in Isabelle. While higher-order unification is able to unify more expressive terms such as variables representing functions, it adds undecidability and non-determinism to the solution space. For example, $f\ 3$ and $3 + 3$ can be unified by substituting 4 possible function definitions for f: $\lambda x.x + 3$, $\lambda x.3 + x$, $\lambda x.x + x$, $\lambda x.3 + 3$. These issues are usually avoided because in practice, Isabelle has to deal with first-order unification more often. Additionally, a subset of higher-order terms, called higher-order patterns were found to behave like first-order terms for the purposes of unification. When there are a non-deterministic set of unifiers, backtracking allows different solutions to be explored. Paulson used a higher-order unification algorithm by Huet [17], who also proved its undecidability. Unification of higher-order terms was novel when Isabelle used it, but it is common in today's proof assistants.

In Isabelle, the `theorem` type can essentially be extended to reason about a particular object logic. Instances of Isabelle include Isabelle/CTT (constructive type theory), Isabelle/IFOL (intuitionistic first-order logic), Isabelle/FOL (classical FOL), Isabelle/FOL with Zermelo-Fraenkel set theory. The most popular of these is Isabelle/HOL [21] - the Isabelle instance for higher-order logic. Isabelle/HOL became a replacement for HOL due to its efficient automation via a simplifier and support for external first-order theorem proving tools [19], and its vast proof libraries which include a standard library and the Archive of Formal Proofs [1].

Thus, the reduction of proof-checking to type-checking of abstract datatypes in the meta language is at the core of LCF-style proof systems which includes the entire HOL family, including Isabelle which is a generic logical framework that can be instantiated to do proof checking in particular logics.

4 A Comparison - Automath vs LCF

While there isn't evidence that the Automath project was influenced by the LCF project (to the contrary, their dates overlap), the *LCF approach*, credited to Edinburgh LCF is the approach followed by most proof assistants today. Theorem provers that follow the LCF approach implement a small proof kernel that facilitates the creation of theorems. This proof kernel usually ensures soundness by some form of type-checking. In this sense, Automath, and its successors also follow the LCF approach with type-checking proof terms to have types as propositions, using the Curry-Howard Isomorphism. Some works claim that Coq is a descendant of LCF [24]. On the other hand, the de Bruijn criterion - that necessitates that either proof terms generated by a proof assistant or the proof checker itself are externally checkable by a simple program - is arguably satisfied by LCF-style theorem provers as well. Thus, while we look at some of the differences between Automath and LCF, and their respective successors, we will keep in mind that both introduced some principles that have become pervasive in modern proof assistants.

Although both Automath-style and LCF-style systems reduce proof-checking to type checking, the type-checkers and indeed the type systems, of both tools are quite different. In LCF and the HOL systems, the type-checker of the implementation language or the meta-language guarantees the soundness of the proof assistant. For Automath-style ITPs, the type of propositions and the type-checking algorithm are implemented within the implementation language. For instance, in Coq, what it means to be a `Prop` and how a type-checker would check terms of type `Prop` are described in OCaml (the implementation language of Coq). This distinction offers some trade-offs. On the one hand, in LCF, the trust-base is the type-checker of the entire ML language, whereas in Automath, the trust-base is a type-checker that is implemented inside the meta language. In principle, this could be smaller than the type checker of the entire language. In practice, while this is a clear difference in approach, we have not seen this argument being made to claim that one tool is better than the

other. Additionally, because LCF's `theorem` type is abstract, proof objects i.e. instances of type `theorem` cannot be inspected or modified. Milner intended for it to be this way to make LCF space efficient - once a proof was type-checked, it only mattered *what* was proven, not *how* it was proven. In Automath, on the other hand, proof objects can be are available for inspection. For instance, in Coq, proof terms can be pattern matched on by advanced users who are able to manipulate Coq's OCaml code.

Automath and its descendents are based on Martin-Löf's dependent type theory, in contrast to the simple type theory implemented by LCF and most of its descendants. While polymorphic types allow types to be parameterized by other types - such as `list` α where α can be any type - dependent types allow types to *depend* on types or values. For instance, consider the $\texttt{cons}_{\mathbb{N}}$ function for `list` $\mathbb{N}$ (lists of natural numbers) that inserts elements into lists. `cons` has type $\mathbb{N} \to \texttt{list}\ \mathbb{N} \to \texttt{list}\ \mathbb{N}$. For example, inserting the element 1 into the empty list gives the singleton list with 1, expressed as $\texttt{cons}_{\mathbb{N}}\ 1\ [\,]\ = [1]$. We would need a $\texttt{cons}_{\alpha}$ for each type α. In a dependently typed system, we can type a generic `cons` function as $\Pi\ \alpha\ :\ \texttt{Type},\ \alpha \to \texttt{list}\ \alpha \to \texttt{list}\ \alpha$. The Π indicates that the second and third arguments to `cons` *depend* on the first, which is a type, or a value of type `Type` (types are often considered values in a hierarchy of types in dependently typed systems). Dependently-typed systems allow for more expressive types that could potentially avoid code reuse. Additionally, the type system could catch certain bugs at compile time that might not be as straightforward to do with simply-typed checkers. For instance, consider the vector type $\texttt{vec}\ :\ \texttt{Type} \to \mathbb{N} \to \texttt{Type}$. Here, the first argument specifies the type of elements in the vector, while the second is a natural number representing the length of the vector. Not only can we define a generic `append` function for such vectors, we can also enforce a constraint on the length of the resultant vector in the `append` function: $\texttt{append}\ :\ \Pi\ (\alpha\ :\ \texttt{Type})\ (m\ n\ :\ \mathbb{N}),\ \texttt{vec}\ \alpha\ m \to \texttt{vec}\ \alpha\ n \to \texttt{vec}\ \alpha\ (m+n)$. Given that the length of the first vector is m and that that of the second is n, the type system can enforce at compile time that the length of the result of appending the vectors is $m+n$. In contrast, such a constraint must be expressed externally in a simply-typed system. For example, John Harrison [14] modeled vectors $\texttt{vec}\ \alpha\ n$ in HOL as functions of type $N \to \alpha$ where N is a type with exactly n values. The examples for dependent types were taken from Chapter 2 of Theorem Proving in Lean [4].

LCF implements a classical logic in comparison to Automath's constructive logic. Every constructive proof is classically valid. Classical logic can be considered as an extension of constructive logic with the *law of the excluded middle* that states that for every proposition A, $A \vee \neg A$ holds. This can also be stated as the double negation elimination law or as proving a proposition by contradiction. As such, some statements that are true in classical logic might not have a proof in constructive logic. Additionally, some propositions might be easier to prove in classical logic. A common example is a classical proof of the following lemma.

Lemma : *There are irrational numbers a and b such that a^b is rational.*

Proof. $\sqrt{2}^{\sqrt{2}}$ is either rational or irrational, by the law of the excluded middle. If it's rational, then $a = b = \sqrt{2}$.
If $\sqrt{2}^{\sqrt{2}}$ is irrational, we know $(\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = 2$. Then, $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$.
Constructively, this proof is more complicated and involves proving that $\sqrt{2}^{\sqrt{2}}$ is irrational. However, constructive proofs *construct* a proof object [3]. For proofs of existential statements and disjunctions, we actually have a proof object as a witness to the existential and of one of the disjuncts. Due to its insistence on constructive objects, constructive logic has a direct relation to computation, which is, for instance, demonstrated by the Curry-Howard Isomorphism.

It is possible to do classical reasoning in some Automath-based systems, and constructive reasoning in LCF-based ones. For instance, Isabelle was first created as a tool to reason in constructive logic, and its first object-logic was Isabelle/CTT, based on constructive type theory. While Coq's logic is fully constructive, classical axioms such as the law of the excluded middle and the axiom of choice can be added as additional axioms to Coq to do classical reasoning. Also, since constructive proofs are programs, extraction of programs that are correct by constructions from proof assistants like Coq is relatively straightforward.

While these theoretical aspects are quite fundamental in differentiating LCF and Automath and their respective descendants, more practical aspects might play a role in choosing between modern-day proof assistants. Isabelle, one of the most popular LCF-style provers today, offers a structured proof language called Isar. Isabelle/HOL, offers on top of its large standard library, the Archive of Formal Proofs [1] — a comprehensive collection of proofs contributed by the Isabelle community. In fact, these vibrant communities for each proof assistant have built extensive libraries for each of them over the years that might make one more attractive than the other. For example, the Coq proof assistant has plenty of libraries and verification projects available for reuse [2]. Similarly, the Lean Theorem Prover also has an active user community that maintains a library for mathematics called `mathlib` [18]. Other considerations may include utilities such as independent tactic languages, for example, Coq's LTAC [11].

References

[1] Archive of formal proofs. https://www.isa-afp.org/.

[2] Coq-community/awesome-coq. https://www.github.com/coq-community/awesome-coq.

[3] J. Avigad. Classical and constructive logic. 2000.

[4] J. Avigad, L. de Moura, and S. Kong. Theorem proving in lean. 2016.

[5] H. Barendregt and H. Geuvers. *Proof-Assistants Using Dependent Type Systems*, page 1149–1238. Elsevier Science Publishers B. V., NLD, 2001.

[6] T. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.

[7] N. de Bruijn. *A processor for PAL*. Technische Hogeschool Eindhoven, 1970. Notitie 1970/30.

[8] N. G. de Bruijn. The mathematical language automath, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, editors, *Symposium on Automatic Demonstration*, pages 29–61, Berlin, Heidelberg, 1970. Springer Berlin Heidelberg.

[9] N. G. de Bruijn. *AUTOMATH, a Language for Mathematics*, pages 159–200. Springer Berlin Heidelberg, Berlin, Heidelberg, 1983.

[10] L. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The lean theorem prover (system description). In A. P. Felty and A. Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[11] D. Delahaye. A tactic language for the system coq. In M. Parigot and A. Voronkov, editors, *Logic for Programming and Automated Reasoning*, pages 85–95, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[12] M. Gordon. *From LCF to HOL: A Short History*, page 169–185. MIT Press, Cambridge, MA, USA, 2000.

[13] M. Gordon, R. Milner, C. P. Wadsworth, and P. T. Christopher. Edinburgh lcf: a mechanized logic of computation. 1978.

[14] J. Harrison. A hol theory of euclidean space. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logics*, pages 114–129, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[15] J. Harrison. Hol light: An overview. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 60–66, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[16] W. A. Howard. The formulæ-as-types notion of construction. In P. D. Groote, editor, *The Curry-Howard Isomorphism*. Academia, 1995.

[17] G. Huet. A unification algorithm for typed lambda-calculus. *Theoretical Computer Science*, 1(1):27 – 57, 1975.

[18] T. mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, page 367–381, New York, NY, USA, 2020. Association for Computing Machinery.

[19] J. Meng and L. C. Paulson. Experiments on supporting interactive proof using resolution. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning*, pages 372–384, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[20] R. Nederpelt. Automath, a language for checking mathematics with a computer. In *Tagung über formale Sprachen (Oberwolfach, Germany, August 30-September 5, 1970)*, pages 27–29. Gesellschaft für Mathematik und Datenverarbeitung, 1970.

[21] T. Nipkow, M. Wenzel, and L. C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer-Verlag, Berlin, Heidelberg, 2002.

[22] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[23] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, USA, 1987.

[24] L. C. Paulson, T. Nipkow, and M. Wenzel. From LCF to isabelle/hol. *CoRR*, abs/1907.02836, 2019.

[25] B. C. Pierce, A. A. de Amorim, C. Casinghino, M. Gaboardi, M. Greenberg, C. Hriţcu, V. Sjöberg, and B. Yorgey. *Logical Foundations*. Software Foundations series, volume 1. Electronic textbook, May 2018.

[26] M. Wenzel, L. C. Paulson, and T. Nipkow. The isabelle framework. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '08, page 33–38, Berlin, Heidelberg, 2008. Springer-Verlag.