BALKAN JOURNAL OF ELECTRICAL & COMPUTER ENGINEERING, Vol. 14, 2026
Research Article

Exploring Concepts of Interactive Theorem Proving

Burak Ekici[1]

***Abstract*—This paper formally presents the Calculus of Inductive Constructions (CiC), the expressive type system behind the Coq proof assistant. We begin with a brief review of the untyped λ-calculus and progressively build the CiC framework. As a case study, we formalize in Coq the proof that $\binom{n}{k}$ is a natural number for all $n \geq k \geq 0$, and relate it to its traditional counterpart. We then extract the Coq proof to Haskell, illustrating the Curry-Howard Isomorphism. The paper also examines the role of dependent types and heterogeneous equality, discusses the controlled use of axioms in dependently typed proofs, and outlines strategies for avoiding axioms to simplify reasoning.**

Index Terms **λ—Calculus, Pure Type Systems, Calculus of Inductive Constructions, Interactive Theorem Proving, Curry-Howard Isomorphism, The Coq Proof Assistant.**

I. Introduction

In mathematics, there are generally two ways to approach functions or maps: (i) from an extensional viewpoint and (ii) from an intensional viewpoint. The extensional viewpoint treats a map as a black box that consumes inputs and produces outputs. It does not consider the internal structure of the map. In this approach, a map can be seen as a directed graph from inputs to outputs.

In contrast, the intensional viewpoint zooms into the structure of functions. It defines functions as pairs consisting of an argument and a body. From the extensional perspective, two functions are equal if they return the same result for every input. This idea is known as the functional extensionality principle.

The intensional viewpoint, however, defines equality differently. Here, two functions are equal only if they have the same syntactic form—that is, if their bodies are identical. In this paper, we briefly describe the interactive theorem prover Coq [1]. Coq is based on a formal intensional calculus of functions. It allows one to (i) encode mathematical objects using a specification language with a strong type system, and (ii) formally reason about these objects using a tactic language [2]. It is worth noting that Coq's expressiveness is equivalent to that of ZFC [3].

Section II provides a concise overview of the untyped λ-calculus. We use a Haskell interpreter available at here for demonstration purposes. In Section III, we introduce a series of type systems over the untyped λ-calculus. We present them systematically and and incrementally, starting from the Simply Typed λ-calculus and building up to the Calculus of Inductive Constructions, which underlies Coq.

Section IV-A covers a case study. It presents both a pen-and-paper proof and a formal Coq proof of the same fact. We relate the two proofs as closely as possible. The formal Coq proof (tested to compile with the Coq compiler coq-8.20.0) is available at here.

In Section IV-B, we examine the relationship between dependent types and heterogeneous equality. We suggest ways to apply feasible axioms in proofs involving dependently typed structures. We also explore strategies to avoid using axioms to streamline proofs. The accompanying code for this section is available at here.

This work aims to be accessible to readers with a basic familiarity in logic or programming languages. The early sections serve as a self-contained introduction to foundational concepts. Later parts engage with more advanced features of the Coq proof assistant. Consequently, the paper can function both as a teaching resource for newcomers and as a bridge toward more sophisticated formal proof development.

An intuitive diagram in Figure 1 illustrates the hierarchy and relationships between variables, terms, types, expressions, and proofs. It organizes the concepts in layers, from basic building blocks (variables) up to complex constructs (proofs).

II. Basics of λ-Calculus

λ-calculus is a formal system that underlies the functional programming paradigm. It models computation as the evaluation of mathematical functions and adopts an intensional viewpoint. In λ-calculus, the principle of functional extensionality is not provable. However, it is not contradictory by design and can therefore be safely assumed if needed.

The system is based on the following key observations: (i) Functions do not need names; all functions are represented uniformly using the λ symbol. (ii) The names of function arguments are not significant. Renaming argument variables consistently throughout the function body does not affect its computational behavior. This property is known as α-equivalence. (iii) A function with multiple arguments can be expressed as a chain of single-argument functions. This transformation is known as Currying (or Schönfinkeling).

[1] **Burak Ekici**, is with Department of Computer Science, Oxford University, Oxford, UK, (burak.ekici@cs.ox.ac.uk).
https://orcid.org/0000-0002-6602-7906
Manuscript received Jan 10, 2025; accepted July 10, 2025. DOI: 10.17694/bajece.1617429
Ekici, B. (2026). Exploring Concepts of Interactive Theorem Proving. Balkan Journal of Electrical and Computer Engineering, 14, 83-100.

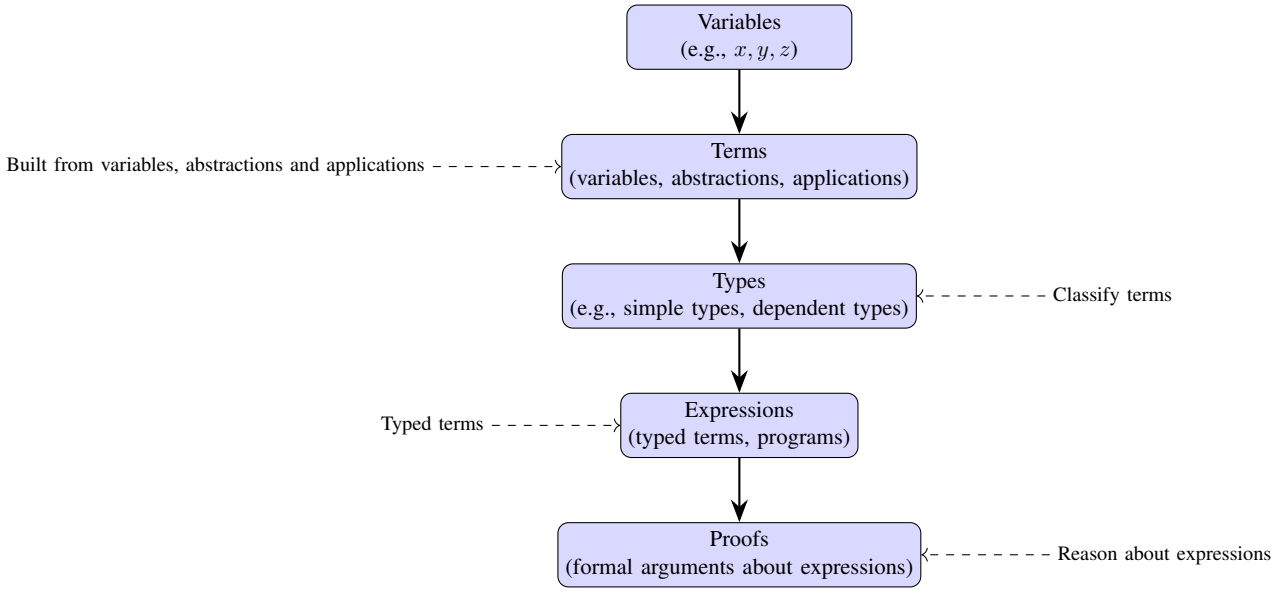


Fig. 1. Hierarchy illustrating variables, terms, types, expressions, and proofs.

Below is the syntax for the inductively defined set of lambda terms:

$$\begin{array}{lcll} t & := & x & \text{term variable/identifier} \\ & | & \lambda x.\, t & \text{function abstraction} \\ & | & t\, t & \text{function application} \end{array}$$

Fig. 2. Syntax for λ-calculus

The lambda abstraction $\lambda x.t$ defines an anonymous function, where x is a bound argument and t is the function body. For example, the function $f(x, y) = x + y$ can be represented in lambda terms as the abstraction $\lambda x.\lambda y.x + y$.

Function applications of the form $t\ t$ are the core mechanism of computation. If the term on the left is an abstraction, it is applied to the argument on the right. This process replaces the bound variable in the abstraction with the given argument, yielding a new term through substitution.

A. A Simple λ Interpreter

In the untyped λ-calculus, *computation* is fundamentally a process of syntactic manipulation. It can be viewed as a (partial) function from terms to terms. To illustrate how computation works in practice, we have implemented an interpreter in Haskell.

The term representation in this implementation follows the *named* approach, where variables are represented by strings. As a result, term manipulation must respect α-equivalence—that is, the specific choice of variable names should not affect the computation. Additionally, one must carefully track free and bound variables to avoid *variable capture*, a common issue that can arise during term *substitution*.

Alternatively, variable representation can use *deBruijn indices* (or levels), or the *locally nameless* approach. These representations avoid naming issues entirely and simplify reasoning about bindings and substitutions.

```haskell
data Term where
    Var    :: String -> Term
    Lambda :: String -> Term -> Term
    App    :: Term   -> Term -> Term
```

The `String` instance that occurs in the `Lambda` constructor represents the variable whose scope is defined by the `Lambda` binder, and is referred to as a *bound variable*.

Definition II.1 (free variables)**.** *The variables that are not bound by any binder is called free. The set of free variables of a term could be computed as follows:*

$$\mathcal{F}(x) = \{x\} \tag{II.1}$$
$$\mathcal{F}(t\ u) = \mathcal{F}(t) \cup \mathcal{F}(u) \tag{II.2}$$
$$\mathcal{F}(\lambda x.\, t) = \mathcal{F}(t) \setminus \{x\} \tag{II.3}$$

In the Haskell implementation, we use lists instead of sets. The approach involves recursively accumulating all variables in a list (empty `[]` *to start with) employing the* `filter` *function to exclude the bound variables in the meantime.*

```haskell
fvH :: Term -> [String] -> [String]
fvH t acc = case t of
  Var s       -> s : acc
  Lambda x t1 -> filter (\a -> a /= x) (fvH t1 acc)
  App t1 t2   -> unique (fvH t1 acc ++ fvH t2 acc)

fv :: Term -> [String]
fv t = fvH t []
```

Example II.2. *Let m be the following lambda term:* $\lambda x.\lambda y.\Big(\big(\lambda z.\lambda v.z\ (z\ v)\big)\ (x\ y)\ (z\ u)\Big)$.

$$\mathcal{F}(m) = \mathcal{F}\Big(\big(\lambda z.\lambda v.z\ (z\ v)\big)\ (x\ y)\ (z\ u)\Big) \setminus \{x, y\} \tag{II.4}$$
$$= \Big(\mathcal{F}\big(\lambda z.\lambda v.z\ (z\ v)\big) \cup \mathcal{F}(x\ y) \cup \mathcal{F}(z\ u)\Big) \setminus \{x, y\} \tag{II.5}$$
$$= \Big(\big(\mathcal{F}(z\ (z\ v)) \setminus \{z, v\}\big) \cup \{x, y\} \cup \{z, u\}\Big) \setminus \{x, y\} \tag{II.6}$$
$$= \Big(\big(\{z, v\} \setminus \{z, v\}\big) \cup \{x, y\} \cup \{z, u\}\Big) \setminus \{x, y\} \tag{II.7}$$
$$= \{z, u\} \tag{II.8}$$

One can encode `m` *in terms of* `Term`*, call* `let fvm = fv m in print fvm` *and get* `["z","u"]` *printed.*

Definition II.3 (α-equivalence)**.** *A term t is said to be* α-equivalent *to a term u, written $t \equiv_\alpha u$, if u can be obtained by* consistently *renaming the* bound *variables in t.*

$$\frac{x \neq y \quad y \notin \mathcal{F}(e)}{\lambda x.\, e \equiv_\alpha \lambda y.\, (e\{y/x\})}$$

The rule above formalizes how to rename the bound variable x to a fresh variable y in the abstraction $\lambda x.\, e$. To perform this renaming correctly, two conditions must be satisfied: (1) x and y must be distinct, ensuring that the renaming actually changes the name; (2) y must not occur freely in e, to prevent variable capture.

The second condition is crucial: if y were free in e, the substitution would alter the meaning of the term by binding what was previously free. By ensuring $y \notin \mathcal{F}(e)$, the transformation preserves the syntactic structure and meaning of the original term.

```
alpha :: String -> Term -> Term
alpha x t = case t of
 Lambda y t1 ->
   if (y /= x && (find x (fv t1) == False))
   then Lambda x (replace y x t1)
   else t
 App t1 t2   -> App (alpha x t1) (alpha x t2)
 Var y       -> Var y
```

where the function `replace y x t` *replaces every single occurrence of the variable* `y` *with variable* `x` *inside the term* `t` *with no precondition.*

Remark II.4. *α-equivalent terms are considered equal.*

Computation in the λ-calculus is defined by the manipulation of terms, primarily through substitution. The central idea is to represent "computable" functions as λ-terms, perform specific substitutions within those terms, and interpret the resulting term as the outcome of the computation.

Definition II.5 (substitution)**.** *Term substitution is handled by replacing all* free occurrences *of variable x in the term t by some term s, denoted $t[s/x]$, and formally defined by*

$$x[s/x] = s \tag{II.9}$$

$$y[s/x] = y \quad \text{if } x \neq y \tag{II.10}$$

$$(t\ u)[s/x] = (t[s/x])\ (u[s/x]) \tag{II.11}$$

$$(\lambda y.\, t)[s/x] = \lambda y.\, (t[s/x]) \quad \text{if } y \neq x \text{ and } y \notin \mathcal{F}(s) \tag{II.12}$$

Notice that the freshness constraint (II.12) can always be satisfied by α renaming. Namely, it is always possible to find a term t' such that $t' \equiv_\alpha \lambda y.\, t$ and perform the intended substitution safely on t' with no variable captured.

```
subst :: Term -> String -> Term -> Term
subst t x s = case t of
  Var y       -> if x == y then s else t
  Lambda y t1 -> if x /= y &&  (find y (fv s) == False)
                 then Lambda y (subst t1 x s)
                 else let z  = freshVar y
                          t' = alpha z t
                      in subst t' x s
  App t1 t2   -> App (subst t1 x s) (subst t2 x s)
```

Notice that in the Haskell implementation of the `subst` *function above, we make sure to generate a fresh variable* `z`*, thanks to* `freshVar` *function, rename the term* `t` *with* `z` *into* `t'` *and only then perform the substitution* `subst t' x s` *when the freshness constraint fails to be met.*

We now have the foundational elements required to define single-step term reduction, which amounts to one step of evaluation (or execution) in the computation process.

Definition II.6 (β-reduction)**.** *Single step term reduction $\to_\beta$ is a binary relation over lambda terms and governed by the rules listed below.*

$$\text{ROOT}\ \frac{}{(\lambda x.\, t)\ u \to_\beta t[u/x]} \qquad \text{APP-L}\ \frac{s \to_\beta t}{s\ u \to_\beta t\ u}$$

$$\text{ABS}\ \frac{s \to_\beta t}{\lambda x.\, s \to_\beta \lambda x.\, t} \qquad \text{APP-R}\ \frac{u \to_\beta t}{s\ u \to_\beta s\ t}$$

The rule root *intuitively states that if s has subterm of form $(\lambda x.\, t)\ u$, then replacing it by $t[u/x]$ is indeed a β-step. Terms of the shape $(\lambda x.\, t)\ u$ are called* redex *(short for* reducible expression*). The reflexive, transitive closure of $\to_\beta$, $s \to_\beta^* t$, denotes existence of sequence $s = t_1 \to_\beta t_2 \to_\beta \cdots \to_\beta t_n = t$ with $n \geqslant 0$; say "s (β-)reduces to t".*

The rule abs *allows reductions to occur within the body of a lambda abstraction. In contrast, the rules* app-l *and* app-r *enable reductions within applications that are not themselves redexes. These latter two rules introduce* non-determinism *into the reduction process, as they permit multiple valid reduction paths from the same term.*

To manage this non-determinism, one can impose a specific evaluation strategy—such as always reducing the leftmost outermost *redex or the* rightmost innermost *one. Choosing a consistent strategy helps ensure predictable and reproducible reductions. Alternatively,* parallel reduction *may be used, allowing independent subterms to reduce simultaneously, which can further mitigate non-determinism and improve efficiency.*

```
beta :: Term -> Term
beta e = case e of
  App (Lambda x t) s -> subst t x s
  Lambda x t         -> Lambda x (beta t)
  App s t            -> App (beta s) (beta t)
  _                  -> e
```

The `beta` *function shown above adopts a parallel reduction strategy, as indicated by its recursive calls on the* `App` *constructor. For additional variants of the* beta *function and alternative reduction strategies, see the file* `beta.hs`*.*

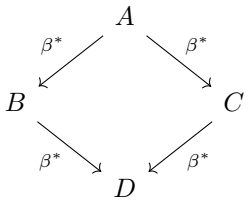
The function `refl_trans_beta` *implements the reflexive and transitive closure $\to_\beta^*$. It repeatedly applies β-reduction to a term* `e` *until it reaches its* normal form *(NF)—that is, a term on which no further reductions are possible.*

```
refl_trans_beta :: Term -> Term
refl_trans_beta e = if e == (beta e) then e
                    else let e' = beta e in
                    refl_trans_beta e'
```

 ISSN: 2147-284X https://dergipark.org.tr/en/pub/bajece

The choice of reduction strategy does not affect the overall outcome as different reduction strategies will ultimately converge to the same result: for any term A that can be reduced to B and C using different strategies, there always exists a term D such that both B and C can be further reduced to D.

Theorem II.7 (Church-Rosser). *$\forall A, B, C$, if $A \to_\beta^* B$ and $A \to_\beta^* C$ then $\exists D$ s.t. $B \to_\beta^* D$ and $C \to_\beta^* D$.*



Terms that are in NF are particularly significant because they could represent the final results of computations. This idea is feasible since the uniqueness of normal forms can be established as a corollary to the Church-Rosser Theorem.

Corollary II.8 (uniqueness of normal forms). *$\forall A, B, C$, if $A \to_\beta^* B$, $A \to_\beta^* C$ and B and C are in NF then $B \equiv_\alpha C$.*

We can now encode simple mathematical objects into the language of λ-calculus to facilitate computations.

Definition II.9 (Church numerals). *Alonzo Church defines natural numbers and operations on them in lambda terms.*

$$0 := \lambda s.\lambda z.z \tag{II.13}$$
$$1 := \lambda s.\lambda z.s\,z \tag{II.14}$$
$$2 := \lambda s.\lambda z.s\,(s\,z) \tag{II.15}$$
$$3 := \lambda s.\lambda z.s\,(s\,(s\,z)) \tag{II.16}$$
$$n := \lambda s.\lambda z.s^n\,z \tag{II.17}$$
$$add := \lambda M.\lambda N.\lambda s.\lambda z.N\,s\,(M\,s\,z) \tag{II.18}$$
$$pred := \lambda n.\lambda s.\lambda z.n\,(\lambda g.\lambda h.h\,(g\,s))(\lambda u.z)(\lambda u.u) \tag{II.19}$$
$$subtr := \lambda m.\lambda n.n\,pred\,m \tag{II.20}$$

The number of applications of the variable s to the variable z represents the corresponding natural number in this encoding. The term `add` denotes the addition function over natural numbers and is handled by applying the term M times the function s to z, then applying the result to N to compute $M+N$. Subtraction is defined with a cutoff behavior, meaning that $0-N=0$. The term `pred` is designed to handle this feature and is used within the term `subtr` to perform subtraction with this cutoff behavior.

```
numH :: Integer -> Term
numH n = if n == 0 then (Var "z") else (App (Var "s") (
    numH (n-1)))

num :: Integer -> Term
num n = Lambda "s" (Lambda "z" (numH n))

add :: Term
add = Lambda "M" (Lambda "N" (Lambda "s" (Lambda "z"
      (App (App (Var "N") (Var "s")) (App (App (Var
    "M") (Var "s")) (Var "z"))))))

addition :: Term -> Term -> Term
addition t1 t2 = App (App add t1) t2
```

With the above implementation in Haskell, one can run

```
let two = num 2
    three = num 3
    r = refl_trans_beta (addition two three)
```

and get `(λs. (λz. [s [s [s [s [s z]]]]]))` *printed out which encodes* `5`*. We employ a very simple pretty printing function named* `term2String` *that could be found inside the file* `Terms.hs`*. Also, please refer to the file* `Church.hs` *for Church encodings of naturals and a few operations over them.*

Another key aspect of handling computation through lambda term reductions is the ability to encode recursion. This capability enables more complex computations and operations to be represented and executed.

Definition II.10 (the $\mathcal{Y}$ combinator). *The $\mathcal{Y}$ combinator is a closed term (a term with no free variables) that given a function F as an argument, it not only reproduces F but also passes F onto itself, enabling the function to refer to itself and thereby achieve recursion.*

$$\mathcal{Y} := \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x)) \tag{II.21}$$
$$\mathcal{Y}\,F := \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))\,F \tag{II.22}$$
$$\to_\beta (\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x)) \tag{II.23}$$
$$\to_\beta F\,\big(\underbrace{(\lambda x.F\,(x\,x))\,(\lambda x.F\,(x\,x))}_{\mathcal{Y}\,F}\big) = F\,(\mathcal{Y}\,F) \tag{II.24}$$
$$\to_\beta F\,(F\,(\mathcal{Y}\,F)) \tag{II.25}$$
$$\to_\beta F\,(F\,(F\,(\mathcal{Y}\,F))) \tag{II.26}$$
$$\to_\beta \ldots \tag{II.27}$$

```
yComb :: Term
yComb = Lambda "f" (App (Lambda "x" (App (Var "f")
                    (App (Var "x") (Var "x"))))
                    (Lambda "x" (App (Var "f")
                                (App (Var "x")
                                     (Var "x")))))
```

We can now encode and compute recursively defined functions within untyped lambda terms. A classic example of this is the factorial function.

Example II.11 (factorial).

```
fact :: Term
fact = Lambda "f" (Lambda "x"( App (App (App
         (ite) (App (isZeroH) (Var "x")))
               (one))
               (App (App (mult) (Var "x"))
                    (App (Var "f")
                         (App (pred) (Var "x"))))))

factorial :: Term -> Term
factorial n = App (App yComb fact) n
```

The term factorial uses the $\mathcal{Y}$ combinator, called `yComb`*, applied to the term* `fact`*, then applied to the argument term* `n`*, which represents the input natural number. The use of* `yComb` *ensures that* `fact` *unfolds recursively for each positive value of* `n`*. This recursion terminates when* `n` *reaches* `0`*. The term* `ite` *encodes the* if-then-else *construct (defined in* `Boolean.hs`*),* `isZeroH` *(from* `Church.hs`*) checks whether a term is* `0`*, and* `mult` *(also in* `Church.hs`*) encodes multiplication of natural numbers. When running*

```
let t = refl_trans_beta (factorial three) in print t
```

we obtain (λs. (λz. [s [s [s [s [s [s z]]]]]])), *representing the number* 6.

Table I serves as a quick reference with these concepts covered in this section.

TABLE I
SUMMARY OF LAMBDA CALCULUS SYNTAX AND REDUCTION RULES

Syntax:	
Variable	x
Abstraction	$\lambda x.t$ (function definition)
Application	$t_1\ t_2$ (function application)
Alpha Conversion	Renaming bound variables, e.g., $\lambda x.t \equiv \lambda y.t[y/x]$
Beta Reduction	Applying functions: $(\lambda x.t)\ u \to t[u/x]$

Remark II.12. *There is a need to guarantee that function parameters are "valid" in terms of function application to prevent evaluation errors. This requirement becomes particularly evident when we consider terms that, while syntactically correct in the untyped setting, have no meaningful interpretation. For instance, the application* one one *makes no sense in a reasonable sort of mathematics: the constant* one *typically denotes a numeral rather than a function, so applying it to itself is semantically incoherent. Despite this, such an application is admissible and reducible in the untyped λ-calculus, which treats all terms uniformly and imposes no restrictions on how functions are applied.*

To avoid such unintended and potentially erroneous applications, one may embark on a syntactic approach that categorizes terms according to the types *of values they compute, and then judge whether to reduce them based on these types. This approach introduces a static discipline that enforces constraints on how terms can be composed and applied. Under this regime, the term* one one *would be rejected outright at the syntactic level, as it violates the typing rules—*one *is not a function type, and hence cannot appear in a function position.*

The benefit of this type-based stratification is twofold. First, it helps ensure that evaluation proceeds only on well-formed and semantically meaningful terms, thereby eliminating a large class of runtime errors. Second, and perhaps more profoundly, it provides a logical structure that allows for reasoning about programs and their behavior. With types in place, it becomes possible to state and prove formal properties about terms, such as their termination, equivalence, or compliance with a specification.

Otherwise put, while untyped λ-calculus alone cannot be considered a proof system, introducing types *allows for the possibility of proving statements therein. The addition of types transforms the calculus into a system with both computational and logical dimensions. This shift is not merely a matter of convenience or safety; it fundamentally changes the expressive power of the system, enabling it to serve as the foundation for typed programming languages and formal logic alike.*

In the next section, we will summarize a few extensions that incorporate types *and can be viewed as proof systems. These extensions demonstrate how typing disciplines not only prevent erroneous computations but also provide a rigorous framework for constructing and verifying formal proofs.*

III. PURE TYPE SYSTEMS (PTSS)

While the untyped λ-calculus elegantly captures the essence of computation, it also permits ill-formed expressions—for instance, applying a numeral to itself. To prevent such nonsensical constructs, we introduce type systems. These systems enforce syntactic constraints that allow only meaningful compositions of terms.

Before delving into the technical details of typing λ-terms, we briefly discuss the well-known Curry-Howard Isomorphism [4]. This correspondence reveals a deep connection between *formal logic* and *constructive type systems*. The term *constructive* here refers to systems that avoid classical axioms such as the *Law of Excluded Middle* ($\forall P, P \vee \neg P$).

Logic	**Type Systems**
propositions	types
proofs	programs
$A \implies B$	$A \to B$ – ordinary function type
$\forall x \in A,\ B(x)$	$\Pi x\colon A.\, B(x)$ – dependent function type
false	empty type
true	singleton type
⋮	⋮

Fig. 3. Curry-Howard Isomorphism

The correspondence between logic and type systems—known as the Curry-Howard Isomorphism—was discovered by Haskell Curry and William Alvin Howard. It establishes that a logical proposition can be interpreted as a type, and a proof of that proposition corresponds to a program that inhabits the associated type.

In this view, logical implication corresponds to ordinary function types, while universal quantification aligns with *dependent function types*, which we explore further in a dedicated subsection. The logical constant *false* corresponds to the *empty type*, since it has no inhabitants and cannot be constructed. Conversely, *true* corresponds to a type with exactly one inhabitant, reflecting the idea that truth has a canonical, trivial construction.

From this perspective, the existential quantifier naturally corresponds to *dependent sum types*. These are introduced and formalized in Section III-J using inductive types and universal quantification.

We now turn to a family of formal systems known as *pure type systems* (PTSs), which provide a framework for typing λ-terms. We begin with the Simply Typed λ-calculus and gradually extend it, culminating in the Calculus of Inductive Constructions—the expressive type system that underlies the Coq proof assistant.

Figure 4 presents the syntax of terms in PTSs. These extend the untyped λ-calculus with (dependent) function types and *type universes* (also called *sorts*), which can be understood as types of types.

Note that the *typing relation* is denoted by the colon ': ' placed between a term and its type, which itself is also a term. For example, $x\colon t$ means that the term x has type t.

t	:=	Ident x	term variable/identifier
	|	$\lambda x\colon t.\, t$	function abstraction
	|	$t\, t$	function application
	|	$\Pi x\colon t.\, t$	(dependent) function types
	|	$\mathcal{U}$	universes

Fig. 4. Syntax for PTSs

Similarly, the term $\lambda x\colon t.t$ denotes a function that takes an argument x of type t and returns a term of type t. The term $\Pi x\colon t.t$ represents the type of functions from type t to type t.

Recall that ordinary function types are usually written using an arrow $\rightarrow$' from the domain type to the codomain type. The notation Π' here denotes *dependent function types*, which generalize ordinary function types by allowing the codomain to depend on the argument.

A. Dependent Types.

A type that depends on values from other types is called a *dependent type*. In other words, it is a type indexed by arbitrary values. While it is usually not advisable to interpret types strictly as sets, an intuitive way to understand dependent types is as *indexed families of sets*. For example, consider the set of people born on August 25th. This set can be seen as indexed by the days of the year, where each day corresponds to the subset of people born on that date. A classic example in type theory to illustrate dependent types is that of *vectors*: lists whose length is fixed and part of their type.

$$\texttt{vector} \quad : \quad \Pi n\colon \mathbb{N}.\, \texttt{List String}\ n$$

Observe that a `vector` instance is a list of strings whose size is bounded by a natural number n. In other words, the type of `vectors` is indexed both by the base type `String` and the natural number n. With this in mind, dependent function types can be understood as function types where the codomain is indexed by values from the domain type. For example, the `revert` function defined over `vectors` has a dependent type with the following signature:

$$\texttt{revert} \quad : \quad \Pi n\colon \mathbb{N}.\, \texttt{List String}\ n \rightarrow \texttt{List String}\ n$$

The function `revert` inputs a fixed-size list of strings, reverts it, and returns the reverted list which is of the same size with the input list. Also, dependent types play a crucial role when it comes to implement algebraic structures in type theory. For instance a set G, with a binary operation $+\colon G \rightarrow G$, qualifies as an algebraic group, among others, if the binary '+' operation is associative. Namely, $\forall a, b, c \in G, , (a + b) + c = a + (b + c)$ which translates in type theory (thanks to Curry-Howard Isomorphism) as a type $\Pi a\colon G.\Pi b\colon G.\Pi c\colon G., (a + b) + c = a + (b + c)$ depending on instances a, b and c. One last point to draw attention here is that a dependent function type with a constant (or not indexed) co-domain type is in fact an ordinary function (or arrow) type. Otherwise put, dependent function types subsume ordinary function types.

The syntax for PTSs presented in Figure 4 is enriched by a set of specifications and typing judgments. These govern the system at the meta-level, defining how types are assigned to terms.

B. Specifications

A specification of a PTS $\mathcal{P}$ is a triple $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ made of a set $\mathcal{S}$ of universes, a set $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{S}$ of axioms and a set $\mathcal{R} \subseteq \mathcal{S} \times \mathcal{S} \times \mathcal{S}$ of rules. The axioms clarify the universe organizations while the rules govern which kind of (dependent) function types are allowed to be constructed, and in which universe they live. For instance, considering the setting $(s_1, s_2, s_3, (s_1\colon s_2), (s_2\colon s_3), (s_1, s_2, s_3))$ for some PTS $\mathcal{P}$, one can reason that in $\mathcal{P}$ there are three sorts (or universes) s_1, s_2 and s_3 such that s_1 is given as an instance of s_2, and s_2 is an instance of s_3. And, $\mathcal{P}$ allows for the construction of types of functions that are defined from the instances of universe s_1 to the instances of universe s_2, and the newly generated function type lives in the universe s_3.

C. Typing Judgments

Typing judgments are indeed structurally established meta-level rules governing PTSs, and describing how to (i) form types, (ii) construct instances of types, and (iii) use instances of types. We below present the rules governing dependent function types only. One however needs to keep in mind that every newly introduced type has a similar set of rules governing its functionality:

$$\frac{\Gamma \vdash A\colon s_1 \quad \Gamma, x\colon A \vdash B(x)\colon s_2}{\Gamma \vdash \Pi x\colon A.\, B(x)\colon s_3} \text{ if } (s_1, s_2, s_3) \in \mathcal{R}$$

$$\frac{\Gamma, x\colon A \vdash b\colon B(x) \quad \Gamma \vdash \Pi x\colon A.\, B(x)\colon s}{\Gamma \vdash \lambda x\colon A.\, b\colon B(x)\colon \Pi x\colon A.\, B(x)} \text{ if } s \in \mathcal{S}$$

$$\frac{\Gamma \vdash f\colon \Pi x\colon A.\, B(x) \quad \Gamma \vdash a\colon A}{\Gamma \vdash (f\, a)\colon B(a)}$$

Fig. 5. Typing judgments

The topmost rule in Figure 5 states that if (i) there is a type A living in universe s_1 under some context Γ, and (ii) there is a dependent type $B(x)$ living in universe s_2 under the context Γ extended with the fact that some variable x ranges over the type A, then one can speak of the valid type $\Pi x\colon A.\, B(x)$ living in universe s_3 thanks to the rule $(s_1, s_2, s_3) \in \mathcal{R}$. Briefly, $\Pi x\colon A.\, B(x)$ simply forms the type $B(x)$ provided $x\colon A$, and lives in s_3. The second rule dictates how to instantiate a valid dependent function type. The one at the bottom is indeed a dependent version of modus-ponens ensuring that only well-typed function applications can be performed. Combining this rule with the β-reduction, it is possible to reason that evaluations of only well-typed function applications are allowed. For a complete overview, please refer to [5].

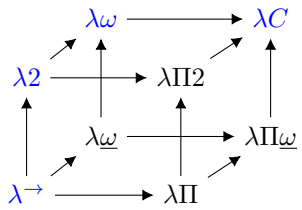
Remark III.1. *"$\vdash a : A$", with nothing on the left of $\vdash$, is read as the term t is of type A under* the empty context.

D. The λ-Cube

The λ-Cube [6] is a framework involving eight PTSs with $\mathcal{S} := \{\star, \Box\}$, $\mathcal{A} := \{(\star : \Box)\}$ and $\mathcal{R} := \{(s_1, s_2, s_2)\}$ where s_1 and s_2 vary over $\mathcal{S}$. That is, function types from instances of universe s_1 to instances of universe s_2 are allowed, and newly generated types live in s_2 as well. Hence, for a PTS present in the λ-Cube, we write the rules omitting the second s_2 as (s_1, s_2). Therefore, for any member of the λ-Cube, the above left rule presented in Figure 5 is refined to be

$$\frac{\Gamma \vdash A : s_1 \quad \Gamma, x : A \vdash B(x) : s_2}{\Gamma \vdash \Pi x : A.\, B(x) : s_2}.$$

PTSs appearing in λ-Cube, given by the side, differ from one another in the set of rules they have. We summarize such differences below for blue colored PTSs starting from the Simply Typed λ-calculus ($\lambda^{\to}$) all the way up to the Calculus of Constructions (λC).

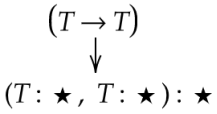


E. The Simply Typed λ-Calculus ($\lambda^{\to}$)

The Simply Typed λ-Calculus [7] is a PTS that appears at the bottom left of the λ-Cube with terms stated in Figure 4, and with the specification given below:

$$\lambda^{\to} \left\| \begin{array}{lcl} \mathcal{S} & := & \{\star, \Box\} \\ \mathcal{A} & := & \{(\star : \Box)\} \\ \mathcal{R} & := & \{(\star, \star)\} \end{array} \right.$$

That is, types of functions defined from instances of universe $\star$ to the instances of universe $\star$ are allowed to be constructed, and newly generated types live in $\star$ as well. For instance, types like $T : \star \vdash \Pi x : T.\, T$ can be built. See below picture for a verification in which we use the arrow symbol to denote function type just to increase the readability score. We embark on this use in the rest of the section.



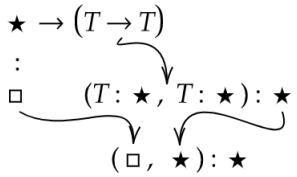
All in all, we could inhabit the type $T : \star \vdash \Pi x : T.\, T$, for example, with the $\lambda x : \tau.\, x$ which indeed is the identity function defined over instances of a fixed type T.

F. System F (λ2)

Girard's System F [8] extends $\lambda^{\to}$ with the ability to construct types of functions defined form the instances of the universe $\Box$ to the instances of universe $\star$, and these types live in $\star$.

$$\lambda 2 \left\| \begin{array}{lcl} \mathcal{S} & := & \{\star, \Box\} \\ \mathcal{A} & := & \{(\star : \Box)\} \\ \mathcal{R} & := & \{(\star, \star), (\Box, \star)\} \end{array} \right.$$

This brings out the ability to build types as $\vdash \Pi T : \star.\, T \to T$:



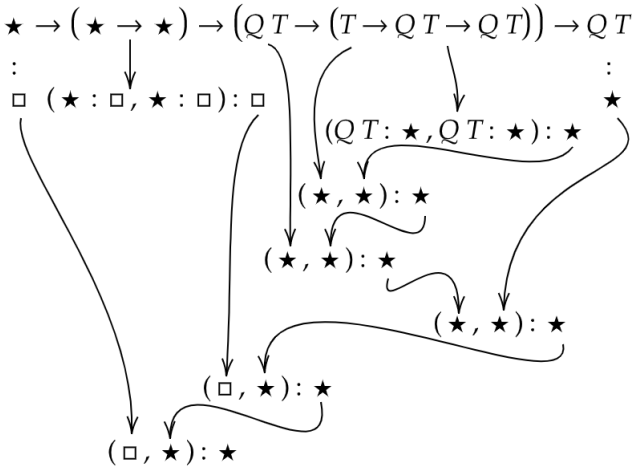
It is now possible to generate lambda terms parameterized by the instances of universe $\star$, known as *polymorphic* lambda terms. For instance, the lambda term $\vdash \lambda T : \star.\, \lambda x : T.\, x$ inhabiting $\vdash \Pi T : \star.\, T \to T$ is indeed the polymorphic identity function defined for all types T ranging over $\star$.

G. System Fω (λω)

The System Fω extends λ2 with the power to construct types of functions defined form the instances of the universe $\Box$ to the instances of universe $\Box$, and these types live in $\Box$.

$$\lambda\omega \left\| \begin{array}{lcl} \mathcal{S} & := & \{\star, \Box\} \\ \mathcal{A} & := & \{(\star : \Box)\} \\ \mathcal{R} & := & \{(\star, \star), (\Box, \star), (\Box, \Box)\} \end{array} \right.$$

Informally speaking, this extension allows for building new types parameterized by existing types. Such kind of types are known as *type constructors*. For example, we can now construct types such as $\vdash \Pi T : \star.\, \Pi Q : \star \to \star.\, Q\,T \to (T \to Q\,T \to Q\,T) \to Q\,T$ which denotes the type of lists Q of elements coming from some type T ranging over $\star$.



The term "$\lambda Q : \star \to \star.\, \lambda \mathrm{nil} : Q\,\mathbb{N}.\, \lambda \mathrm{cons} : \mathbb{N} \to Q\,\mathbb{N} \to Q\,\mathbb{N}.\, \mathrm{cons}\, 5\, \mathrm{nil}$", of type "$\vdash \Pi Q : \star \to \star.\, Q\,\mathbb{N} \to (\mathbb{N} \to Q\,\mathbb{N} \to Q\,\mathbb{N}) \to Q\,\mathbb{N}$", is the function that constructs a list of natural numbers including the element 5 only, namely [5], while the term "$\lambda Q : \star \to \star.\, \lambda \mathrm{nil} : Q\,\mathbb{N}.\, \lambda \mathrm{cons} : \mathbb{N} \to Q\,\mathbb{N} \to Q\,\mathbb{N}.\, \mathrm{cons}\, 10\, (\mathrm{cons}\, 5\, \mathrm{nil})$" is the function constructing the list [10; 5].

 ISSN: 2147-284X https://dergipark.org.tr/en/pub/bajece

H. Calculus of Constructions (λC)

The Calculus of Constructions [9] extends $\lambda\omega$ with the ability to construct new types of functions defined from instances of universe $\star$ to instances of universe $\Box$ such that newly built types live in $\Box$.

$$\lambda C \left\| \begin{array}{lcl} \mathcal{S} & := & \{\star, \Box\} \\ \mathcal{A} & := & \{(\star : \Box)\} \\ \mathcal{R} & := & \{(\star,\star), (\Box,\star), (\star,\Box), (\Box,\Box)\} \end{array} \right.$$

It is possible in λC to generate types that are parameterized by terms living in instances of $\star$. Such kind of types are indeed known as *dependent types*. With this extension, we could, for instance, construct types like $\vdash \Pi T\colon \star.\, \Pi Q\colon \star \to \mathbb{N} \to \star.\, Q\,T\,0 \to (\Pi n\colon \mathbb{N}.\, T \to Q\,T\,n \to Q\,T\,(n+1)) \to \Pi m\colon \mathbb{N}.\, Q\,T\,m$ given that $\mathbb{N}$ lives in $\star$. This is in fact the type of lists Q of elements coming from some type T ranging over $\star$ with the length $m\colon \mathbb{N}$. These lists are rather called *vectors*.

The term "$\lambda Q\colon \star \to \mathbb{N} \to \star.\, \lambda \text{nil}\colon Q\,\mathbb{Z}\,0.\, \lambda \text{cons}\colon \mathbb{N} \to \mathbb{Z} \to Q\,\mathbb{Z}\,\mathbb{N} \to Q\,\mathbb{Z}\,\mathbb{N}.\, \text{cons}\,0\,5\,\text{nil}$" of type $\vdash \Pi Q\colon \star \to \mathbb{N} \to \star.\, Q\,\mathbb{Z}\,0 \to (\Pi n\colon \mathbb{N}.\, \mathbb{Z} \to Q\,\mathbb{Z}\,n \to Q\,\mathbb{Z}\,(n+1)) \to \Pi m\colon \mathbb{N}.\, Q\,\mathbb{Z}\,m$ is indeed the function that constructs a list of integers with length 1 including 5; namely [5] while "$\lambda Q\colon \star \to \mathbb{N} \to \star.\, \lambda \text{nil}\colon Q\,\mathbb{Z}\,0.\, \lambda \text{cons}\colon \mathbb{N} \to \mathbb{Z} \to Q\,\mathbb{Z}\,\mathbb{N} \to Q\,\mathbb{Z}\,\mathbb{N}.\, \text{cons}\,1\,10\,(\text{cons}\,0\,5\,\text{nil})$" is the function constructing the integer list [10; 5] ensuring that its length is 2. Notice that in constructing a vector, the number of `cons` use must be equal to the type indexing length. Otherwise, the vector construction gets rejected. This nicely exhibits the power brought by the use of dependent types.

Note however that the type $\vdash \Pi Q\colon \Box.\, Q \to Q$ cannot be formed in λC. This is simply because, it is not known in which universe the sort $\Box$ lives.

$$\begin{array}{l} \Box \to (Q \to Q) \\ : \\ ? \qquad (Q\colon \Box,\ Q\colon \Box) : \Box \end{array}$$

To solve this out, the idea is to introduce infinitely many $\Box$ universes given in a stratified fashion.

$$\lambda C \left\| \begin{array}{lcl} \mathcal{S} & := & \{\star, \Box_{\mathtt{i}} \text{ s.t. } \mathtt{i} \in \mathbb{N}\} \\ \mathcal{A} & := & \{(\star : \Box_0), (\Box_{\mathtt{i}} : \Box_{\mathtt{i}+1}) \text{ s.t. } \mathtt{i} \in \mathbb{N}\} \\ \mathcal{R} & := & \{(\star,\star), (\Box_{\mathtt{i}},\star), (\star,\Box_{\mathtt{i}}), (\Box_{\mathtt{i}},\Box_{\mathtt{j}},\Box_{\mathtt{max(i,j)}}) \\ & & \text{s.t. } \mathtt{i},\mathtt{j} \in \mathbb{N}\} \end{array} \right.$$

Remark that there is now a hierarchy over universes: $\Box_i$ is an inhabitant of $\Box_{i+1}$ for all natural numbers i, and $\star$ lives at the bottom of the hierarchy in $\Box_0$. Therefore, the type $\vdash \Pi Q\colon \Box_0.\, Q \to Q$ could now be built, as depicted by the side, and lives in $\Box_1$, thanks to the rule $(\Box_{\mathtt{i}}, \Box_{\mathtt{j}}, \Box_{\mathtt{max(i,j)}})$.

$$\begin{array}{l} \Box_0 \to (Q \to Q) \\ : \\ \Box_1 \qquad (Q\colon \Box_0,\ Q\colon \Box_0) : \Box_0 \\ \qquad\qquad (\Box_1,\ \Box_0) : \Box_1 \end{array}$$

This obviously generalizes as such types of the form $\vdash \Pi Q\colon \Box_i.\, Q \to Q$ are allowed to be constructed, and live in $\Box_{i+1}$.

I. A Variant of Calculus of Constructions (λC^ω)

In this section, we present a variant λC^ω of λC in which the universe $\star$ is renamed to be `Prop` while the hierarchical universes $\Box$ are called `Type`:

$$\lambda C^\omega \left\| \begin{array}{lcl} \mathcal{S} & := & \{\mathtt{Set}, \mathtt{Prop}, \mathtt{Type_i} \text{ s.t. } \mathtt{i} \in \mathbb{N}\} \\ \mathcal{A} & := & \{(\mathtt{Set} : \mathtt{Type_0}), (\mathtt{Prop} : \mathtt{Type_0}), \\ & & (\mathtt{Type_i} : \mathtt{Type_{i+1}}) \text{ s.t. } \mathtt{i} \in \mathbb{N}\} \\ \mathcal{R} & := & \{(\mathtt{Prop},\mathtt{Prop}), (\mathtt{Set},\mathtt{Prop}), (\mathtt{Type_i},\mathtt{Prop}), \\ & & (\mathtt{Set},\mathtt{Set}), (\mathtt{Prop},\mathtt{Set}), (\mathtt{Type_i},\mathtt{Set},\mathtt{Type_i}) \\ & & (\mathtt{Set},\mathtt{Type_i}), (\mathtt{Prop},\mathtt{Type_i}), \\ & & (\mathtt{Type_i},\mathtt{Type_j},\mathtt{Type_{max(i,j)}}), \text{ s.t. } \mathtt{i},\mathtt{j} \in \mathbb{N}\} \end{array} \right.$$

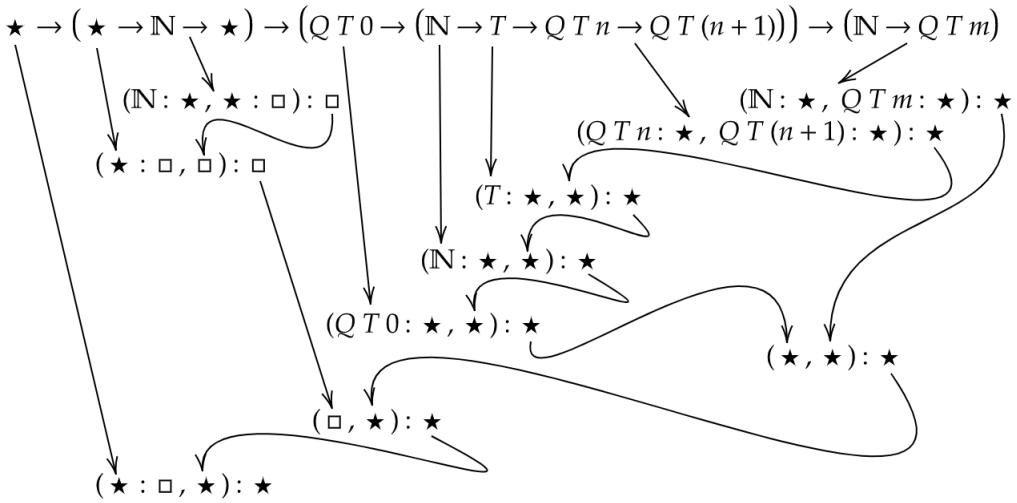
Remark that there is an additional universe called `Set` located at the bottom of the hierarchy along with `Prop`. The reason why such a universe exists has to do with predicativity/impredicativity feature of universes. A universe $\mathcal{U}$ is called impredicative if types like $\Pi P\colon \mathcal{U}.\, P \to P$ are living in $\mathcal{U}$. The proof is beyond the scope of this paper but it is well-known that in an impredicative universe if one assumes classical axioms as *Law of Excluded Middle* (LEM) and at the same time allows for *large elimination* (that is, computing terms that live in universes higher in the hierarchy out of terms inhabited in universes lower in the hierarchy), then it is possible to prove false. In fact, to avoid this problem following ideas could be employed:

1) having a predicative universe in which the possibility of assuming variants of LEM is allowed and large elimination is kept. This is `Set`, and is known as the universe of computations or informative objects.
2) having an impredicative universe in which the possibility of assuming variants of LEM is allowed but large elimination is turned off. This is `Prop`. In this universe, any instance p of some type P cannot be used to compute (or eliminated into) instances of types that live in universes `Set` and `Type`. For that reason, `Prop` is known as universe of *non-computational* propositions or logical objects.

A proof showing the existence of some term $x\colon T$, such that T lives in either `Set` or `Type`, and satisfying certain property P of `Prop` could be extracted out as a functional program implied by Curry-Howard isomorphism. The extraction of a program out of a proof implemented in λC^ω is mainly based on deletion of proof terms living in `Prop` (as they are non-computational) while keeping informative/computational ones living in `Set` or `Type`. Therefore, program extraction is indeed a good reason to keep `Prop` and `Set` separated. The case study in Section IV-A, among others, exemplifies the *program extraction* facility.

J. The Coq Proof Assistant

The theoretical frameworks described so far are not merely abstract constructs—they form the backbone of modern proof assistants. Among these, *Coq* provides a practical environment



based on the Calculus of Inductive Constructions (CIC), enabling both programming and formal verification.

Coq is an interactive theorem prover that has been developed with Curry-Howard Isomorphism in mind. It is based on the λC^{ω} type system extended with *inductive* type declarations. This is why its underlying type system is called the Calculus of Inductive Constructions (CiC).

In an inductive declaration, the type is constructed via structures called constructors that may employ self-referencing. For example, the type of natural numbers can be inductively built from two constructors named *zero* and *successor* function, and looks, in Coq notation, like

```
Inductive nat : Set ≜
  | O : nat
  | S : nat → nat.
```

By this, we *inductively* build in Coq a type named `nat` designated to live in universe`Set` out of constructors: `O` (representing zero) and `S` (denoting the successor function). Notice that the latter constructor uses self-referencing. That is, it employs in the process of type construction the type that is being constructed.

One point to note here is that the inductive type construction may diverge and not terminate. Given that non-termination proves false in the Coq system, it needs to be discarded somehow. To do so, Coq employs *strict positivity* [10] check.

Besides, Coq automatically generates the *structural induction* and the *structural recursion* principles for inductively declared types. For the `nat` type, such principals `nat_ind` and `nat_rec` are given as follows:

```
nat_ind : ∀ P : nat → Prop, P 0 → (∀ n : nat, P n →
     P (S n)) → ∀ n : nat, P n
nat_rect: ∀ P : nat → Set,  P 0 → (∀ n : nat, P n →
     P (S n)) → ∀ n : nat, P n
```

The structural induction principle `nat_ind` (whose correctness proof is skipped here) states that to construct an inhabitant of some dependent type `Πn: nat. P n` in `Prop`, otherwise put to prove some proposition `P` for all natural numbers, one needs to show that (i) it is provable for `O` and that (ii) it holds for the successor `S n` provided that it holds for some number `n`. Remark that the sole difference in between `nat_ind` and `nat_rec` is that the predicate `P` lands in `Prop` in the former, and in `Type` in the latter. Therefore, to construct instances of the dependent type `Πn: nat. P n`, living in `Type`, one needs to provide information on how `nat_rec` computes with (i) the constructor `O`, and (ii) the successor `S n` provided that it knows how to compute with some number `n`. We below give an example to the use of recursion principle `nat_rec` when it comes to define addition over natural numbers:

```
Definition addition ≜
  nat_rec (fun _: nat ⇒ nat → nat)
          (fun m: nat ⇒ m)
          (fun (n: nat) (p: (fun _: nat ⇒ nat → nat)
    n) (m: nat) ⇒ S (p m)).
```

The first argument λ `_: nat ⇒ nat → nat` of `nat_rec` is in fact the signature of the `addition` function, and expresses that it takes a natural number and returns another function which itself inputs a natural number and outputs a natural number. Thanks to Currifying or Schönfinkeling, we view `addition` as a function inputting a pair of natural numbers and returning a natural number. The second argument λ `m: nat ⇒ m` clarifies that if the first parameter of the `addition` is `O` then it immediately returns the second parameter `m`. The third argument (λ (`n: nat`) (`p: (λ _: nat ⇒ nat → nat) n`) (`m: nat`) ⇒ (`p m`)) explains the strategy of the `addition` function if its first parameter is `S n`, provided that it already has a computational strategy `p` for the number `n`. That is, in the first step, the function reduces into `S (p m)`. In the second step, if the number `n` is still a successor `S k` of some number `k`, it reduces into `S (S (p m))`. It applies this strategy until the number `k` goes down to `O`. Otherwise put, the function applies the successor function `S` first argument (`n`) times over the second argument (`m`). Given that a recursor can be described as pattern-matching with a fixpoint, one could program an addition function as follows:

```
Fixpoint addition' (n m: nat) ≜
  match n with
    | O   ⇒ m
    | S n ⇒ S (addition' n m)
  end.
```

Similar to the termination-guaranteed inductive types, Coq also ensures termination of recursive functions; otherwise, one could prove false. The built-in termination checker uses the strategy of *structurally decreasing arguments*. For functions that do not work with structurally decreasing arguments, one can also employ *well-founded relations* to convince Coq about function termination.

To exemplify a proof employing the structural induction principle $\texttt{nat\_ind}$, we prove in Coq that both functions `addition` and `addition'` indeed compute the same value on the same input.

The sameness measure we consider here is in fact the structural equality (also known as Leibniz equality), which is inductively constructed by reflexivity ($\texttt{eq\_refl}$), named `eq`, and denoted by the '=' sign.

```
Inductive eq (A : Type) (x : A) : A → Prop ≜
  | eq_refl: eq A x x.
```

Having this said, let us now look into the statement we want to prove:

```
Lemma add_eq:
     ∀ n: nat, ∀ m: nat, addition n m = addition' n m.
```

Proof. We aim at structurally constructing an instance (named `add_eq` following the `Lemma` vernacular) of the dependent type Π `n: nat.` Π`m: nat.` `addition n = addition' n m`. We argue by structural induction (`nat_ind`) over the input number `n`, and have two goals to close:

1) (base case) `addition 0 m = addition' 0 m`. Recall by definition that, both functions evaluate in a single step into `m` if `n` is `O`. Therefore we need to show that `m = m` holds which is trivial as Leibniz equality is reflexive.
2) (step case) `addition (S n) m = addition' (S n) m`. In this case, we have an induction hypothesis `IHn` of type Π`m : nat, addition n m = addition' n m`. It is known again by definition that the function `addition (S n) m` reduces into `S (addition n m)` in a single step of evaluation. Similarly, `addition' (S n) m` evaluates into `S (addition' n m)` in one step. Therefore, the goal takes the following shape: `S (addition n m) = S (addition' n m)`. Rewriting the induction hypothesis `IHn`, with `m`, into the goal, we get to show `S (addition' n m) = S (addition' n m)` which is trivial as Leibniz equality is reflexive.

□

In order to formalize proofs in Coq, one may employ *tactic*s. A tactic is (usually a tiny little) code block that manipulates proof states. We briefly describe a few of them that appear in the Coq proof of the sameness of addition functions:

- `intro n`: applies the introduction rule (left above rule in Figure 5) of dependent product types. For instance, executing `intros n m` over a goal like Π`n: nat.` Π`m: nat.` `addition n m = addition' n m` would turn the goal into `addition n m = addition' n m` introducing `n` and `m` into the proof context. Note that `intros` is the plural version of `intro`.
- `induction m`: applies the relevant induction principle over `m` if the type of `m` is inductively constructed.
- `simpl`: reduces the definitions inside the goal employing call-by-value evaluation.
- `reflexivity`: closes the goal if it is in the form of `R x x` where `R` is a relation respecting reflexivity.
- `rewrite H`: rewrites the term `H` in the goal provided that `H` is an hypothesis in the form of an equivalence.

The Coq proof follows exact same steps taken in the above pen-and-paper proof of `add_eq`, and looks like:

```
Proof. intro n.  (* introducing n to the context *)
       induction n.  (* applying {nat_ind} to n *)
       - intro m. simpl. reflexivity.  (* base case *)
       - intro m. simpl. (* step case *)
         rewrite IHn.
         reflexivity.
Qed.
```

Notice that the proof is composed of tactics and written between the vernacular commands `Proof`" and `Qed`". The former begins the proof process, while the latter closes and saves it if the proof is completed correctly.

Tactic applications and vernacular commands are separated (or ordered) by the period `.`" sign. The dash `-`" signs are used to focus on the current goal.

Please refer to Appendix A for a demonstration of the proof state manipulation performed after each tactic application.

We carry on with a few other Coq structures that take place in the case study detailed in the following Section IV-A. The type of integers, denoted `Z`, is built in Coq as follows:

```
Inductive Z : Set ≜
  | Z0  : Z
  | Zpos: positive → Z
  | Zneg: positive → Z.

Inductive positive : Set ≜
  | xI : positive → positive
  | xO : positive → positive
  | xH : positive.
```

The constructor `Z0` is indeed the integer zero. The `Zpos` and `Zneg` are respectively used to construct positive and negative integers employing the `positive` type instances which are inductively built in the style of unsigned binary numbers: the `xH`" constructors represent the binary number 1, the `xO n`" places a binary 0 to the end of the binary number `n` while `xI n`" appends a binary one 1 to `n`. For instance, `xO (xO xH)`" is binary 100" (decimal 4) while `xI (xO xH)`" is 101" (decimal 5). It is then obvious that `Zpos (xO (xO xH))`" implements the integer +4 while "`Zneg (xI (xO xH))`" constructing -5.

In Coq, the type of rational numbers `Q` is encoded by a *record* (an inductive type with a single constructor) with two

fields as follows:

```
Record Q : Set ≜
Qmake { Qnum : Z;
        Qden : positive }
```

```
Qeq (p q: Q) ≜
  (Qnum p * QDen q)%Z =
  (Qnum q * QDen p)%Z
```

The field `Qnum` of type `Z` encodes the nominator while `Qden` of type `positive` encoding the strictly positive denominator for any rational number. The notation `x # y` is reserved to denote $\frac{x}{y}$, for some integer `x` and positive `y`. The equality `Qeq` over instances of `Q` is given by the term $(\texttt{Qnum p} \times \texttt{QDen q}) = (\texttt{Qnum q} \times \texttt{QDen p})$, and denoted by the infix "==". The $\times$ symbol in `Qeq` represents integer multiplication, and `QDen (p: Q) := Z.pos (Qden p)` maps the `positive` type inhabitant denominator `p` of some `Q` into `Z` using the injection `Z.pos`.

It is also possible to construct the existential quantifier in Coq employing dependent product types (Π) and inductive type declarations as follows:

```
Inductive ex (A : Type) (P : A → Prop): Prop ≜
  | ex_intro : ∀ x : A, P x → ex A P
```

```
Inductive sig (A : Type) (P : A → Prop): Type ≜
  | exist : ∀ x : A, P x → sig A P
```

The Coq type `ex` depending on some propositional predicate `P` over an arbitrarily given type `A` encodes the existential quantification in mathematics. That is, the mathematical statement $\exists x \in A,\ P(x)$ is reflected in Coq as the type `ex A` $(\lambda$ `x: A` $\Rightarrow$ `P x`$)$ with the constructor `ex_intro` building some sub-type of `A` inhabiting `xs` such that `P x` holds. Notice that the type `sig` is constructed in the exact same fashion with that of `ex` with a sole difference that `ex` inhabits `Prop` while `sig` lives in `Type`. The Coq notation "`exists x, P x`" denotes the type "`ex A` $(\lambda$ `x: A` $\Rightarrow$ `P x`$)$" while "$\{$`x: A | P x`$\}$" denoting "`sig A` $(\lambda$ `x: A` $\Rightarrow$ `P x`$)$". Please refer to [1], for other useful constructions, such as logical operators `and`, `or`, ..., and tactics of Coq.

IV. Case Studies

A. Code Extraction

In the below detailed Coq formalization, the approach we take is to utilize functions implemented in existing Coq libraries whenever possible, rather than formalizing them from scratch. This allows us to leverage already proven lemmas available in these libraries related to the functions we use in our formalization, without the need to re-prove them ourselves. For this case study, it suffices to import the Coq libraries `Factorial`, `QArith`, and `Lia`. Any proof term absent from the accompanying Coq file is definitively imported from one of these libraries.

```
Require Import Factorial QArith Lia.
```

As the name suggests, the library `Factorial` contains the definition of the factorial function `fact` over natural numbers, along with various properties that `fact` satisfies. Similarly, `QArith` provides the necessary definitions to construct rational numbers and many related facts about them. It also imports the libraries `ZArith` and `BinPos`, which formalize Coq integers and positive numbers, respectively, along with numerous useful property proofs. The library `Lia` implements the tactic `lia`, designed to automatically solve basic properties in *linear integer arithmetic*.

Definition IV.1 (binomial coefficients)**.** *For all natural numbers n and k, the binomial coefficients are defined by the following formula:*

$$\binom{n}{k} = \begin{cases} \dfrac{n!}{(n-k)! \times k!}, & \text{for } n \geq k \\ 0, & \text{otherwise} \end{cases}$$

```
Definition binom (n k: nat): Q ≜
  if Nat.leb k n
  then Z.of_nat (fact n) #
       Pos.of_nat (fact k *
                   fact (n - k))
  else 0.
```

Observe that in the Coq definition `binom` of binomial coefficients, the function takes a pair of natural numbers `n` and `k`, performs the Boolean check $n \geq k$ using the standard library function `Nat.leb`, and behaves according to the mathematical definition. The functions `Z.of_nat` (imported from `ZArith`) and `Pos.of_nat` (from `BinPos`) serve as injections from natural numbers to integers and from natural numbers to positive numbers, respectively. The symbol "$*$" denotes multiplication, while "$-$" denotes subtraction over natural numbers.

Let us carry on with explaining functionalities of a few more Coq tactics that appear in below proofs:

- `case_eq t`. Performs case analysis over the term `t`, of some inductive type, providing information about the structure of `t` in the form of Leibniz equality.
- `assert p: P`. Introduces an hypothesis named `p` into the context, and sets `P` as a sub-goal.
- `exists p`: Plugs the term `p` of type `A` into goals of the form "`exists m: A, P m`", and modifies the proof state into the following shape "`P p`".
- `pose proof H as H'`: doubles the hypothesis `H` under the name `H'` within the proof context.
- `specialize` $(\texttt{H t}_1 \cdots \texttt{t}_n)$: specializes the hypothesis `H` with properly typed terms from $\texttt{t}_1$ to $\texttt{t}_n$.

Lemma IV.2 (binomS)**.** $\forall n,\ k \in \mathbb{N}, \binom{n+1}{k+1} = \binom{n}{k} + \binom{n}{k+1}$.

Proof. The proof argues by a case distinction on whether $n \geq k$:

1) $n \geq k$. Here, we carry on with another case distinction on whether $n \geq (k+1)$:
   a) $n \geq (k+1)$. In this case, the goal we aim at proving turns out to be $\frac{(n+1)!}{(k+1)!((n+1)-(k+1))!} = \frac{n!}{k!(n-k)!} +$

$\frac{n!}{(k+1)!(n-(k+1))!}$ thanks to Definition IV.1 of binomial coefficients.

1. We rewrite the fact $(n+1)! = n!(k+1) + n!(n-k)$ within the goal, and turn it into $\frac{n!(k+1)+n!(n-k)}{(k+1)!((n+1)-(k+1))!} = \frac{n!}{k!(n-k)!} + \frac{n!}{(k+1)!(n-(k+1))!}$.
2. Splitting the left hand side considering the fact that $(n+1)-(k+1) = n-k$ gives $\frac{n!(k+1)}{(k+1)!(n-k)!} + \frac{n!(n-k)}{(k+1)!(n-k)!} = \frac{n!}{k!(n-k)!} + \frac{n!}{(k+1)!(n-(k+1))!}$.
3. The obvious fact $\frac{n!}{k!(n-k)!} = \frac{n!(k+1)}{(k+1)k!(n-k)!}$ yields $\frac{n!}{k!(n-k)!} + \frac{n!(n-k)}{(k+1)!(n-k)!} = \frac{n!}{k!(n-k)!} + \frac{n!}{(k+1)!(n-(k+1))!}$ when written inside the goal.
4. Similarly, it is easy to show $\frac{n!}{(k+1)!(n-(k+1))!} = \frac{n!(n-k)}{(k+1)!(n-k)!}$ provided that $(k+1)!(n-(k+1))! = (k+1)!(n-k)!$. Employing this fact within the goal, we obtain $\frac{n!}{k!(n-k)!} + \frac{n!(n-k)}{(k+1)!(n-k)!} = \frac{n!}{k!(n-k)!} + \frac{n!(n-k)}{(k+1)!(n-k)!}$ which is trivial thanks to the reflexivity of equality.

b) $(k+1) > n$. It is easy to deduce in this case that $k = n$. Therefore the goal takes the following shape: $\binom{n+1}{n+1} = \binom{n}{n} + \binom{n}{n+1}$ which reduces into $1 = 1 + 0$ due to Definition IV.1 of binomial coefficients. It is trivial to show $1 = 1 + 0$.

2) $k > n$. In this case, the goal turns out to be $0 = 0 + 0$ again thanks to Definition IV.1 of binomial coefficients.

□

Please refer to Appendix A for the Coq mechanization of this fact. The mechanized proof is annotated with comment lines to highlight its correspondence with the paper proof.

Theorem IV.3 (binomNat). *$\forall n, k \in \mathbb{N}, \exists m \in \mathbb{N}, m = \binom{n}{k}$ for some natural number m.*

Proof. We argue by induction on n, and need to prove below given sub-goals:

1) (base case) The goal we intend to show in this case is $\exists m \in \mathbb{N}, m = \binom{0}{k}$.
   a) If $k = 0$ then the goal turns out to be $\exists m \in \mathbb{N}, m = \binom{0}{0}$, and simplifies into $\exists m \in \mathbb{N}, m = 1$ thanks to Definition IV.1. There obviously exists a natural number m which closes the goal, and it is 1.
   b) If $k > 0$ then we need to show $\exists m \in \mathbb{N}, m = \binom{0}{k}$ which again by Definition IV.1 reduces int $\exists m \in \mathbb{N}, m = 0$. Similarly, plugging the natural number 0 for m closes the goal.
2) (step case) In this case, we aim at proving $\exists m \in \mathbb{N}, m = \binom{n+1}{k}$ for some natural number k, provided an induction hypothesis $\forall k \in \mathbb{N}, \exists m \in \mathbb{N}, m = \binom{n}{k}$ (IHn).
   a) If $k = 0$, the goal we have is $\exists m \in \mathbb{N}, m = \binom{n+1}{0}$, and reduces into $\exists m \in \mathbb{N}, m = 1$ due to Definition IV.1. We plug-in the natural number 1 for m, and get the goal proven.
   b) If $k > 0$, the goal takes the following shape: $\exists m \in \mathbb{N}, m = \binom{n+1}{k}$. Thanks to Lemma IV.2, we have $\binom{n+1}{k} = \binom{n}{k-1} + \binom{n}{k}$. Rewriting this into the goal gives $\exists m \in \mathbb{N}, \binom{n}{k-1} + \binom{n}{k}$. The induction hypothesis (IHn) entails that $\binom{n}{k-1}$ and $\binom{n}{k}$ are both natural numbers when specialized by $k-1$ and k respectively. Thus, $\binom{n}{k-1} + \binom{n}{k}$ is also a natural number.

□

```
Lemma binomNat : ∀ n k: nat, { m:nat | Z.of_nat m #
     Pos.of_nat 1 == (binom n k) }.
Proof. intro n.
       induction n; intros.
       - case_eq k; intros.  (*1*)
         + simpl. ∃ 1%nat. simpl. reflexivity.  (*1.a*)
         + simpl. ∃ 0%nat. simpl. reflexivity.  (*1.b*)
       - simpl. case_eq k; intros.  (*2*)
         + ∃ 1%nat.  (*2.a*)
           ...
         + pose proof IHn as IHn'.  (*2.b*)
           specialize (IHn' (k-1)%nat).
           specialize (IHn k).
           ...
           rewrite binomS.
           ...
           reflexivity.
Defined.
```

Executing below listed vernaculars, we extract this formal Coq proof out as an Haskell program that computes the number `m` satisfying for all natural numbers `n` and `k` that `Z.of_nat m # Pos.of_nat 1 == (binom n k)`, as it is implied by Curry-Howard Isomorphism (Figure 3): *proofs are programs*. Please refer to Appendix A for the complete Haskell code.

```
Require Import Extraction.
Extraction Language OCaml.
Recursive Extraction binomNat.
```

Mind that the `Defined` vernacular functions in the similar manner with that of `Qed` with a unique difference that it allows *transparency* of proof terms. That is, a proof closed with `Defined` can be reduced (by i.e., `simpl` tactic) if needed as opposed to those closed with `Qed` which are saved just as black-boxes with complete *opacity*.

On a last note, the technical machinery underlying program extraction is beyond the scope of this paper. Please refer to [11] for all related details. Loosely speaking, the main idea backing program extraction is in fact two fold: (i) keeping the proof terms that construct the existential object (`m` in our example) which lands in Coq's `Set` (or `Type`), while (ii) deleting the proof terms constructing the associated property (`Z.of_nat m # Pos.of_nat 1 == (binom n k)` in our statement) which lands in Coq's `Prop`. Recall from Section III-I that one of the reasons why Coq has `Set` and `Prop` distinction is to help program extraction.

B. Dependent Types

Another key feature of Coq is its use of dependent types. These types offer a high level of flexibility for expressing complex notions, allowing types to depend on values. However, this expressiveness can also introduce additional challenges when it comes to proving properties about dependently typed terms. In what follows, we examine two cases that present challenges due to the use of dependent types. We

demonstrate how to overcome these issues by employing the *extended pattern matching construct* with and without the use of *uniqueness of identity proofs* (UIP) axiom. This axiom can bridge the gap between *definitional* and *propositional equality* in Coq, providing a way to simplify proofs involving equality. However, the UIP axiom can be omitted when definitions do not involve propositional equality. To explicitly demonstrate that we use two different (dependently typed) definitions of vectors in Coq: (1) as lists with a specific length, and (2) as constructs defined using *cons* and *nil*.

1) Vector: lists with a specific length: In this case, we encode vectors using the concept of lists with a specific, fixed length. This approach allows vectors to have their lengths encoded as part of their type, ensuring that certain operations, such as concatenation, can be well-typed.

```
Class vector {A: Type} (n: nat) ≜ mk_vector
  { ul: list A; uc: len ul = n }.
```

In the Coq class declaration provided, a vector is conceptualized as a structured package comprising two components: an underlying list `ul` and a proof obligation (or condition) `uc` that ensures that the length of `ul` is indeed `n` over which the package has been parametrized. The parameterization of the package by the length n introduces a dependency in the type system, ensuring that operations and transformations on vectors adhere to strict size constraints, as enforced by the proof obligation.

This declaration, however, introduces challenges, particularly when comparing vector instances using Coq's Leibniz equality. Specifically, it is evident that two vectors should be considered identical if and only if their underlying lists are identical. However, the presence of the proof obligation (`uc`) complicates the matters, as Coq's standard equality mechanism does not automatically recognize the equivalence of vectors solely based on the equality of their underlying lists. Namely, Coq raises a flag rejecting the below statement

```
Lemma vector_eq: ∀ {A: Type} {n m: nat}
  (v1: @vector A n)
  (v2: @vector A m) (eq: n = m),
  @ul A n v1 = @ul A m v2 → v2 = v1.
```

claiming that `v2` and `v1` are of different type instances. This arises because `v1` is an instance of type `vector A n` while `v2` is of type `vector A m`. Coq is unable to automatically infer that these types are equivalent, even under the assumption that `m` is equal to `n`. To overcome this difficulty, we employ *extended pattern matching* construct and restate lemma as follows:

```
Lemma vector_eq: ∀ {A: Type} {n m: nat}
  (v1: @vector A n)
  (v2: @vector A m) (eq: n = m),
  @ul A n v1 = @ul A m v2 →
  v2 = match eq in _ = V return @vector A V with
       | eq_refl ⇒ v1
       end.
```

We use the fact `eq` that `n` is equal to `m` and intuitively ask Coq to expect a value of type `vector A m` even though `v1` is of type `vector A n`. This is possible under the condition that `eq` can be reduced to the single constructor `eq_refl` of Leibniz equality.

Remark IV.4. *There is unfortunately no* computational *way to reduce* `eq` *into* `eq_refl` *due to the way the* `vector` *class is defined. There, we impose the existence of Leibniz equality in the proof obligation which is indeed not definitional. That is, we cannot evaluate (simplify or compute)* `eq` *such that it takes the shape of* `eq_refl`*. To close this gap (thus the proof of the lemma* `vector_eq`*), we employ a version of UIP axiom,* `UIP_refl`*, which claims that*

```
UIP_refl: ∀ (U : Type) (x : U) (p : x = x), p = eq_refl.
```

all identity proofs are identical. It is a perfectly reasonable axiom which does not clash with Coq's type system but still is an axiom and better be omitted if possible.

1) With the current deceleration, we could prove the lemma statement where vectors are related with heterogenous equality (rather than Leibniz equality) but no axioms.
2) If we want to prove it with Leibniz equality and no axioms, we need to modify the initial declaration.

We demonstrate both cases in this section.

Another situation where a similar issue might arise involves the concatenation of vectors. Joining two vectors of type `vector A n` and `vector A m` produces a result of type `vector A (n+m)`. To achieve this, we take a slightly different approach and define the concatenation function using Coq tactics. This is made possible by the Curry-Howard Correspondence, which states that proofs and programs are essentially the same.

```
Definition appendV {A: Type} {n m}
   (v1: @vector A n) (v2: @vector A m):
   @vector A (plus n m).
Proof. refine(
            match (v1, v2) with
              | (mk_vector _ l1 p1, mk_vector _ l2 p2) ⇒
                 mk_vector (plus n m) (appendL l1 l2) _
            end).
       rewrite app_len, p1, p2. easy.
Defined.
```

The function `appendV` is stated such that, given vectors `v1` and `v2` of types `vector A n` and `vector A m`, respectively, it returns a vector of type `vector A (plus n m)`, where `plus` denotes the addition function for natural numbers. To construct the function body, we employ here the `refine` tactic, which is used to fill the holes in partial proofs, and state that if `l1` and `l2` are the underlying lists of `v1` and `v2` and `p1` and `p2` are poof obligations that `len l1 = n` and `len l2 = m`, we can build a new vector (using `mk_vector`) with the underlying list being the concatenation of `l1` and `l2`, given by `appendL l1 l2` function. Notice that within the `refine` tactic, we skip the proof obligation `len (appendL l1 l2) = plus n m` by leaving it as a hole in the definition, placing an underscore "_" instead. The hole has then been filled up with rewriting the facts `p1`, `p2` and `app_len` (inside `vector.v`). It is also important to note that we conclude the definition with the `Defined` keyword

rather than `Qed` to ensure that it remains transparent and can be reduced (executed) when necessary.

We can move onto the associativity of the `appendV` function. There, we need to again employ the extended pattern matching for the same reason above:

```
Lemma app_assoc:
  ∀ n m u A (v1: @vector A n) (v2: @vector A m) (v3:
     @vector A u),
    (appendV (appendV v1 v2) v3) =
    (match plus_assoc n m u in _ = V return @vector A V
     with
      | eq_refl ⇒ (appendV v1 (appendV v2 v3))
     end).
```

We make sure with the extended match that the addition function `plus` is associative as we are comparing vector instances of types `vector A (plus n (plus m u))` and s `vector A (plus (plus n m) u)`. This has been provided by the `plus_assoc` function (inside `vector.v`). The proof of the lemma proceeds by an application of `vector_eq` and a straightforward induction over the underlying list of the vector `v1`.

Now to address the item (1) at Remark IV.4, we retain the vector declaration as is and only adjust the equality statement over vectors to utilize *heterogeneous John Major's equality* (JMeq) [12].

```
Inductive eq_dep (p:U) (x:P p) : ∀ q:U, P q → Prop ≜
    | eq_dep_intro : eq_dep p x p x.

Inductive JMeq
     (A : Type) (x : A) : ∀ B : Type, B → Prop ≜
    | JMeq_refl : JMeq x x.

Lemma vector_jmeq: ∀ {A: Type} {n m: nat}
  (v1: @vector A n) (v2: @vector A m) (eq: n = m),
  @ul A n v1 = @ul A m v2 → JMeq v2 v1.
```

Notice that the `JMeq` allows us to compare instances of different types. To close this goal, we need not use any axiom as we can turn the goal into the following shape with sigma types. This is mainly because `JMeq` is implied by the dependent equality `eq_dep` which is equivalent to the equality between dependent sum (sigma) types. See lemmata `eq_dep_id_JMeq` (in `Coq.Logic.JMeq` library) and `eq_sigT_iff_eq_dep` (in the library `Coq.Logic.EqdepFacts`). The fact `vector_jmeq` definitely allows for the proof of `app_assoc_jm`.

```
Lemma app_assoc_jm: ∀ n m u A
  (v1: @vector A n) (v2: @vector A m) (v3: @vector A u),
  JMeq (appendV (appendV v1 v2) v3)
       (appendV v1 (appendV v2 v3)).
```

The proof of `app_assoc_jm` (in `vector.v.`) gives us the claim (1) of Remark IV.4. We can do more here and get the proof of `app_assoc` out of the fact `vector_jmeq` and the axiom `JMeq_eq`. See `jmeq_eq` in `vector.v` file.

```
JMeq_eq: ∀ (A : Type) (x y : A), JMeq x y → x = y
```

In brief, to address the challenges associated with equality proofs in dependent types, one approach is to use axiomatic schemas. Specifically, one can adopt heterogeneous equality, establish the necessary proofs within this framework, and apply `JMeq_eq` in the end to go to the Leibniz equality. Alternatively, extended pattern matching with the axiom `UIP_refl` can be employed unless the matched equality can be reduced to `eq_refl`. We plan to investigate and clarify the relationship between these two axioms in a future work.

2) Vector: construct with cons and nil: In this section, we address the item (2) in Remark IV.4, and prove the associativity of vector concatenation with no axioms in use. For that we need to modify the declaration of the vector type so that instances could be reduced when need be. We however do not define things from the starch but make use of the definition `t` in the standard library `Coq.Vectors.VectorDef`.

```
Inductive t (A : Type) : nat → Type ≜
  | nil : t A 0
  | cons: A → ∀ n : nat, t A n → t A (S n).
```

The declaration is pretty straightforward. The type `A` obviously makes the definition polymorphic. The dependency over natural numbers can also be seen from the declaration signature. The constructor `nil` represents the empty vector with the length 0 while `cons` generates a vector of length `S n` (or `n + 1`) out of a vector of length `n`, for some natural number `n`. The lemma `app_assoc` does not take much effort and space to close.

```
Lemma app_assoc:
  ∀ n m u A (v1: t A n) (v2: t A m) (v3: t A u),
   (append (append v1 v2) v3) =
   (match plus_assoc n m u in _ = V return t A V with
     | eq_refl ⇒ (append v1 (append v2 v3))
   end).
```

It applies structural induction on the instance `v1`.

- `v1 = nil` when evaluated, the lemma `plus_assoc` transforms into the proof of `m + u = m + u`, and further reduces into `eq_refl` due to the way that it is proven above (using simple induction and `eq_refl` itself) and saved as a transparent term in the state. Therefore, in the end, we need to show `append v2 v3 = append v2 v3` which is trivial.
- `v1 = cons` the goal takes the following shape.

```
IHv1 : append (append v1 v2) v3 =
       match plus_assoc n m u in (_ = V)
       return (t A V)  with
       | eq_refl ⇒ append v1 (append v2 v3)
       end
_____________________________________(1/1)
cons A h (n + m + u) (append (append v1 v2) v3) =
match eq_ind_r (fun n0 : nat ⇒ S n0 = S (n + m +
     u)) eq_refl (plus_assoc n m u) in (_ = V)
     return (t A V) with
| eq_refl ⇒ cons A h (n + (m + u)) (append v1
     (append v2 v3))
end
```

This case is a bit more involved in comparison with the base case but still easy. We just rewrite the induction hypothesis `IHv1` and get the goal as in the following.

```
cons A h (n + m + u)
match plus_assoc n m u in (_ = V) return (t A V)
with
  | eq_refl ⇒ append v1 (append v2 v3)
end =
match eq_ind_r
(fun n0 : nat ⇒ S n0 = S   (n + m + u)) eq_refl
     (plus_assoc n m u) in (_ = V) return (t A V)
with
  | eq_refl ⇒ cons A h (n + (m + u))
                       (append v1 (append v2 v3))
end
```

Observe within the goal that at both sides of the equality, we have a dependency over `plus_assoc` and also that all choices of the numbers `n`, `m` and `u`, the `plus_assoc` reduces to reduces into `eq_refl` again due to the way it is proven. Therefore, we can directly destruct `plus_assoc n m u` and left with `cons A h (n + (m + u)) (append v1 (append v2 v3)) = cons A h (n + (m + u)) (append v1 (append v2 v3))` to be shown. This is again trivial.

In summary, it is generally a better practice to avoid using propositional Leibniz equality in Coq type declarations, if possible, as it may not always reduce to `eq_refl` and could require additional axioms in use. Instead, adopting inherently structural definitions, where feasible, is preferable. Such definitions work effectively with dependent types and typically result in simpler proofs.

a) A short note on Coq, Lean and Isabelle: . Lean and Coq both employ cumulative hierarchies of universes to avoid paradoxes such as Girard's, organizing types as `Type_0 : Type_1 : Type_2 : ...` in Coq and `Type 0 : Type 1 : Type 2 : ...` in Lean. While both systems support cumulative universes, their handling differs in practice. Coq uses an implicit universe polymorphism mechanism, introduced in version 8.5, where universe levels are abstract and often inferred by the solver, though they can be explicitly annotated using `@i` syntax. Coq distinguishes between global and local universes and maintains universe constraints during typechecking, which can lead to inconsistencies if constraints conflict. In contrast, Lean favors a more explicit approach: users often declare and manage universe levels directly using universe u, and type definitions are annotated accordingly. This explicitness provides clarity and predictability at the cost of additional verbosity. Overall, Coq aims for a more automatic, solver-based universe management system, whereas Lean embraces manual control and transparency in universe handling.

Both Coq and Lean are dependently typed proof assistants based on variants of the Calculus of Inductive Constructions (CIC), and they offer rich support for dependent types—that is, types that depend on values. At a foundational level, their expressive power is similar: both allow encoding dependent functions (Π-types), dependent pairs (Σ-types), and inductive families (like length-indexed vectors). However, the user experience and syntax differ. Coq follows a more traditional and verbose ML-style syntax, often requiring more boilerplate, especially when dealing with dependent pattern matching and eliminators. In contrast, Lean offers a more lightweight, Haskell-inspired syntax with features such as more ergonomic pattern matching for dependent types, better elaboration support, and a powerful type class system that integrates smoothly with dependent reasoning. Moreover, Lean's tactic framework is more programmable and consistent when constructing terms with complex dependencies. While both systems offer full dependent pattern matching via eliminators, Lean tends to be more permissive and user-friendly out of the box, especially in interactive development. Nevertheless, Coq has a longer history, a more mature metatheory, and extensive libraries (like the standard library and MathComp) which use dependent types in sophisticated ways. In short, both systems support dependent types at similar expressive levels, but Lean often provides a smoother and more modern interface for working with them.

Isabelle, in its most widely used instantiation Isabelle/HOL, is based on classical Higher-Order Logic (HOL), which does not natively support full dependent types. In HOL, types cannot depend on terms, so one cannot directly define, say, a vector type indexed by its length. Instead, dependent-like behavior must be encoded indirectly using predicates or type classes, which can lead to more verbose or less expressive formulations. That said, Isabelle does have experimental object logics—like Isabelle/CTT or Isabelle/Mizar—that support varying degrees of type dependency, but these are less used than Isabelle/HOL.

As a result, formalizations that rely heavily on dependent types—such as certified programming or intensional type-level reasoning—are often more natural and direct in Coq and Lean. Isabelle excels, however, in automation, structured proofs (via Isar), and integration with classical mathematics, but its foundational logic imposes stricter boundaries on type-level expressiveness compared to the dependently-typed paradigms of Coq and Lean.

V. Conclusion

We have formally described the underlying type system for the proof assistant Coq along with a case study that aims at relating a pen-and-paper mathematical proof to its Coq developed version. We extracted the formal Coq proof out as a Haskell program to have a concretely exemplify the consequence of Curry-Howard Isomorphism. We also examined the relationship between dependent types and heterogeneous equality. In our discussion, we demonstrated various strategies for utilizing reasonable axioms when working with proofs involving dependently typed structures. We concluded that it is advisable to avoid proof obligations associated with propositional Leibniz equality in Coq whenever possible.

References

[1] The Coq Development Team, *The Coq proof assistant reference manual*, https://coq.inria.fr/distrib/current/refman/, 2018, version 8.8.1.

[2] D. Delahaye, "A tactic language for the system coq," in *Logic for Programming and Automated Reasoning, 7th International Conference, LPAR 2000, Reunion Island, France, November 11-12, 2000, Proceedings*, ser. Lecture Notes in Computer Science, M. Parigot and A. Voronkov, Eds., vol. 1955. Springer, 2000, pp. 85–95. [Online]. Available: https://doi.org/10.1007/3-540-44404-1_7

[3] B. Werner, "Sets in types, types in sets," in *Theoretical Aspects of Computer Software, Third International Symposium, TACS '97, Sendai, Japan, September 23-26, 1997, Proceedings*, ser. Lecture Notes in Computer Science, M. Abadi and T. Ito, Eds., vol. 1281. Springer, 1997, pp. 530–346. [Online]. Available: https://doi.org/10.1007/BFb0014566
[4] W. A. Howard, "The formulæ-as-types notion of construction," in *The Curry-Howard Isomorphism*, P. D. Groote, Ed. Academia, 1995.
[5] H. Barendregt, "Lambda calculi with types," in *HANDBOOK OF LOGIC IN COMPUTER SCIENCE*. Oxford University Press, 1992, pp. 117–309.
[6] H. Barendregt, "Introduction to generalized type systems," *J. Funct. Program.*, vol. 1, no. 2, pp. 125–154, 1991.
[7] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 06 1940. [Online]. Available: https://projecteuclid.org:443/euclid.jsl/1183387805
[8] J. Girard, "The system F of variable types, fifteen years later," *Theor. Comput. Sci.*, vol. 45, no. 2, pp. 159–192, 1986. [Online]. Available: https://doi.org/10.1016/0304-3975(86)90044-7
[9] T. Coquand and G. P. Huet, "The calculus of constructions," *Inf. Comput.*, vol. 76, no. 2/3, pp. 95–120, 1988. [Online]. Available: https://doi.org/10.1016/0890-5401(88)90005-3
[10] T. Coquand and C. Paulin, "Inductively defined types," in *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, ser. Lecture Notes in Computer Science, P. Martin-Löf and G. Mints, Eds., vol. 417. Springer, 1988, pp. 50–66. [Online]. Available: https://doi.org/10.1007/3-540-52335-9_47
[11] P. Letouzey, "Extraction in Coq: An Overview," in *Logic and Theory of Algorithms, 4th Conference on Computability in Europe, CiE 2008, Athens, Greece, June 15-20, 2008, Proceedings*, ser. Lecture Notes in Computer Science, A. Beckmann, C. Dimitracopoulos, and B. Löwe, Eds., vol. 5028. Springer, 2008, pp. 359–369. [Online]. Available: https://doi.org/10.1007/978-3-540-69407-6_39
[12] C. McBride, "Elimination with a motive," in *TYPES*, ser. Lecture Notes in Computer Science, vol. 2277. Springer, 2000, pp. 197–216.

BIOGRAPHIES

Burak Ekici graduated from the Department of Computer Engineering at İzmir Institute of Technology in 2009. Between 2010 and 2011, he held a traineeship position at the European Commission Joint Research Centre in Ispra, Italy. He earned his PhD from Université Grenoble Alpes in France in late 2015. From 2016 to 2018, he held postdoctoral research positions at the University of Iowa, USA, and the University of Innsbruck, Austria. Between 2019 and 2025, he served as an assistant professor at İstanbul Kültür University, TED University, and Muğla Sıtkı Koçman University. Since October 2023, he has been working as a Senior Research Associate in the Department of Computer Science at the University of Oxford, UK.

APPENDIX

```
Lemma binom_S: ∀ n k: nat,
  binom (S n) (S k) == binom n k + binom n (S k).
Proof. intros n k.
       unfold binom.
       case_eq (k ≤? n); intros. (*1*)
       - case_eq (S k ≤? n); intros Ha.  (*1.a*)
         + ...
           assert (Hc: (Z.of_nat (fact (S n)) =
                       (Z.of_nat ((fact n) * (k+1) +
                        (fact n) * (n-k))))%Z).
           { ... }
           rewrite Hc.  (*1.a.1*)
           assert (He: ((S k * fact n + (n - k) *
                        fact n) =
                       (fact n * (k + 1) +
                        fact n * (n - k)))
           { ... }
           rewrite ← He. (*1.a.2*)
           ...
           assert (H1: (Z.of_nat (S k * fact n) #
                        Pos.of_nat ((S k) * fact k *
                        fact (n - k)))
                        ==
                       (Z.of_nat (fact n) #
                        Pos.of_nat (fact k *
                        fact (n - k)))).
           rewrite H1.  (*1.a.3*)
           assert (H2: (Z.of_nat ((n - k) * fact n) #
                        Pos.of_nat (S k * fact k *
                        fact (n - k)))
                        ==
                       (Z.of_nat (fact n) #
                        Pos.of_nat (S k * fact k *
                        fact (n - S k)))).
           rewrite H2.  (*1.a.4*)
           reflexivity.
         + assert (H1: n = k). (*1.b*)
           { ... }
           rewrite H1 in *.
           ...
       - assert (H1: S k ≤? S n = false). (*2*)
         { ... }
         rewrite H1.
         assert (H2: S k ≤? n = false).
         { ... }
         rewrite H2. reflexivity.
Defined.
```

APPENDIX

On the left, it is given a formal Coq proof made of tactic applications in such a way that the tactic in execution is colored red, and on the right we demonstrate the goal context above the dashed line with the current goal state below the line.

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       - intro m. simpl. reflexivity.
       - intro m. simpl. rewrite IHn.
         reflexivity.
Qed.


1 subgoal
n : nat
_____________________________________(1/1)
∀ m : nat, addition n m = addition' n m
```

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       - intro m. simpl. reflexivity.
       - intro m. simpl. rewrite IHn.
         reflexivity.
Qed.

2 subgoals
_____________________________________(1/2)
∀ m : nat, addition 0 m = addition' 0 m
_____________________________________(2/2)
∀ m : nat,
  addition (S n) m = addition' (S n) m
```

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       - intro m. simpl. reflexivity.
       - intro m. simpl. rewrite IHn.
         reflexivity.
Qed.

1 subgoal
_____________________________________(1/1)
∀ m : nat, addition 0 m = addition' 0 m
```

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       -intro m. simpl. reflexivity.
       - intro m. simpl. rewrite IHn.
         reflexivity.
Qed.

1 subgoal
m : nat
_____________________________________(1/1)
addition 0 m = addition' 0 m
```

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       - intro m.simpl. reflexivity.
       - intro m. simpl. rewrite IHn.
         reflexivity.
Qed.

1 subgoal
m : nat
_____________________________________(1/1)
m = m
```

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       - intro m. simpl.reflexivity.
       - intro m. simpl. rewrite IHn.
         reflexivity.
Qed.

This subproof is complete, but there are some unfocused
     goals:

_____________________________________(1/1)
∀ m : nat,
  addition (S n) m = addition' (S n) m
```

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       - intro m. simpl. reflexivity.
      - intro m. simpl. rewrite IHn.
         reflexivity.
Qed.

1 subgoal
n : nat
IHn : ∀ m : nat,
  addition n m = addition' n m
_____________________________________(1/1)
∀ m : nat,
  addition (S n) m = addition' (S n) m
```

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       - intro m. simpl. reflexivity.
       -intro m. simpl. rewrite IHn.
        reflexivity.
Qed.

1 subgoal
n : nat
IHn: ∀ m : nat, addition n m = addition' n m
m : nat
_____________________________________(1/1)
addition (S n) m = addition' (S n) m
```

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       - intro m. simpl. reflexivity.
       - intro m.simpl. rewrite IHn.
         reflexivity.
Qed.

1 subgoal
n : nat
IHn : ∀ m : nat, addition n m = addition' n m
m : nat
_____________________________________(1/1)
S (addition n m) = S (addition' n m)
```

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       - intro m. simpl. reflexivity.
       - intro m. simpl. rewrite IHn.
         reflexivity.
Qed.

1 subgoal
n : nat
IHn : ∀ m : nat, addition n m = addition' n m
m : nat
_____________________________________(1/1)
S (addition' n m) = S (addition' n m)
```

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       - intro m. simpl. reflexivity.
       - intro m. simpl. rewrite IHn.
         reflexivity.
Qed.

No more subgoals.
```

```
Lemma add_eq: ∀ n: nat, ∀ m: nat,
  addition n m = addition' n m.
Proof. intro n.
       induction n.
       - intro m. simpl. reflexivity.
       - intro m. simpl. rewrite IHn.
         reflexivity.
Qed.

add_eq defined.
```

The vernacular "`Qed`" tells Coq to finalize the construction of the proof term `add_eq`, and apply one last type-checking to see whether the proof terms is of the expected type, namely "`forall n: nat, forall m: nat, addition n m = addition' n m`", for our case here. If type-checking succeeds, Coq saves the proof object for potential further use; throws an exception otherwise.

APPENDIX

```
module Main where

import qualified Prelude

data Nat =
   O
 | S Nat

nat_rect :: a1 → (Nat → a1 → a1) →
            Nat → a1
nat_rect f f0 n =
  case n of {
```

```
    O → f;
    S n0 → f0 n0 (nat_rect f f0 n0)}

nat_rec :: a1 → (Nat → a1 → a1) →
           Nat → a1
nat_rec = nat_rect

type Sig a = a
-- singleton inductive, whose constructor
   was exist

add :: Nat → Nat → Nat
add n m =  case n of {  O   → m;
                        S p → S (add p m)}

sub :: Nat → Nat → Nat
sub n m = case n of {
  O   → n;
  S k →
  case m of { O → n; S l → sub k l}}

binomNat :: Nat → Nat → Nat
binomNat n =
  nat_rec (\k →
     case k of { O → S O;  S _ → O})
     (\_ iHn k →
        case k of {O   → S O; S _ →
          let {iHn' = iHn (sub k (S O))} in
          let {iHn0 = iHn k} in
          add iHn' iHn0}) n
```

In order to print out and observe the outcome of the `binomNat` function, please append the code given in below snippet.

```
printNat :: Nat → Prelude.String
printNat O = "O"
printNat (S k) = "S " Prelude.++ printNat k

main = Prelude.print (printNat
       (binomNat (S (S (S (S (S O)))))
                 (S (S O))))
```

One can then run `printNat` function to monitor the result returned by `binomNat`. For instance, "`binomNat (S (S (S (S (S O))))) (S (S O))`" encoding $\binom{5}{2}$ returns 10 which is encoded The code has been tested to compile fine with the Haskell compiler `ghc 8.0.2`. Navigate to the following link to test the code online:

https://rextester.com/YOV40992