FORMALIZING CHEMICAL THEORY USING THE LEAN THEOREM PROVER

A PREPRINT

Maxwell P. Bobbin[1], Samiha Sharlin[1], Parivash Feyzishendi[1], An Hong Dang[1], Catherine M. Wraback [1], and Tyler R. Josephson[1,2]

[1]Department of Chemical, Biochemical, and Environmental Engineering, University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250
[2]Department of Computer Science and Electrical Engineering, University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250

ABSTRACT

Chemical theory can be made more rigorous using the *Lean theorem prover* and its associated math library, `mathlib`. We formalize the Langmuir [38] and BET [11] theories of adsorption, making every step of derivation explicit based on scientific premises, and finding mathematical constraints that were not identified by the original authors. We also show how Lean flexibly enables reuse of proofs that build on more complex theories through *definition*, and we use this to define Lennard-Jones potential function and proof that it tends to infinity as the radius approaches zero. Finally, we show how Lean type classes can construct scientific frameworks for interoperable proofs, by creating a *class* for thermodynamics and kinematics, and using it to formalize gas law relationships like Boyle's Law and equations of motion that underlie Newtonian mechanics respectively. This approach can be extended to other fields in science and engineering, enabling the formalization of rich and complex theories in our literature.

Keywords Theorem proving · Proof assistants · Formal verification · Logic · Langmuir · BET · Lennard Jones · Boyle's Law · Kinematics

1 Introduction

1.1 Theorem provers for chemical theory

Theorem provers are a type of computer program that aid in the derivation of formal proofs through human-machine interaction [31]. They provide a way to interact with the current state of the proof, while the computer verifies each step [51, 58, 8, 28, 45, 46]. Although formal methods of proof are difficult to write, they use strict syntax that not only guarantees correctness, but is also machine-readable and interpretable (see Table 1).

Hand-written proofs	**Formal proofs**
Informal syntax	Strict, computer language syntax
Only for human readers	Machine-readable and executable
Might exclude information	Cannot miss assumptions or steps
Might contain mistakes	Rigorously verified by computer
Requires human to proofread	Automated proof checking
Easy to write	Challenging to write

Table 1: Comparison of hand-written and formalized proofs.

An interactive theorem prover may appear to be performing a symbolic manipulation of mathematical expressions, like a computer algebra system (CAS) such as SymPy [43], but such manipulations are only permitted supported by a proof with axiomatic foundation. For example, whether multiplication is commutative depends on the types of objects being multiplied: $a * b = b * a$ when a and b are scalars, but $A \times B \neq B \times A$ when A and B are matrices. In CAS systems, matrix multiplication is made non-commutative by imposing special conditions [43] whereas theorem provers only allow changes that are proven to be valid. So rearranging $a * b$ to $b * a$ is only possible in theorem provers once their equivalence has been proven true and changing $A \times B$ to $B \times A$ is not permitted because it has not been proven true. Consequently, theorem provers provide the utmost degree of confidence by allowing all maths to be constructed from their base axioms (see Table 2).

Interactive theorem provers	**Computer algebra systems**
Symbolically transform formulae	Symbolically transform formulae
Only permit correct transformations	Human-checked correctness
Verification tool	Computational tool
Explicit assumptions	Hidden assumptions
Built off a small, trusted, kernel	Large program with many algorithms
LEAN THEOREM PROVER, Isabelle	Mathematica, SymPy, MATLAB

Table 2: Interactive Theorem Provers [22, 58] vs. Computer Algebra Systems [60, 43, 2].

Historically formal verification and the axiomatization of theories have mainly been observed in mathematics [4, 27, 10, 30, 13, 34], however recently there have been a few notable attempts at formalizing physics theories such as versions of relativity theory [56, 40], electromagnetic optics [35], geometrical optics [54] and design of optical quantum experiments using logic artificial intelligence [16]. Our focus here is in formalizing fundamental theories in chemistry which has received very little attention. Like Paleo's work [47], we want to create a formal basis for chemical theories, and use the Lean theorem prover to verify them.

1.2 The Lean theorem prover

We have selected the Lean theorem prover for its power as an interactive theorem prover, the coverage of its mathematics library, `mathlib` [42], and the supportive online community of Lean enthusiasts [1] with an aim to formalize the entire undergraduate math curriculum. [3]. Interesting projects have emerged from its foundations, including Perfectoid Spaces [14], Cap Set Problem [21] and Liquid Tensor [53] and the team also created a web-based game, called Natural Number Game [15], that has been widely successful in introducing newcomers to Lean. As executable code, Lean proofs can be read by language modeling algorithms that find patterns in math proof databases, enabling automated proofs of formal proof statements [33, 48]

We anticipate that Lean is expressive enough to formalize diverse and complex theories across quantum mechanics, fluid mechanics, reaction rate theory, statistical thermodynamics, and more. Lean gets its power from its ability to define mathematical objects and prove their properties, rather than just assuming premises for the sake of individual proofs. Lean is based on `type theory` [59, 26] where both mathematical objects and the relation between them are modeled with types (see Figure 5 in supplementary text). Everything in Lean is a *Type*, whether natural numbers or real numbers, functions, Boolean conditions, or even entire proofs. *Types* also have a type of *Type* and we can define *Types* of our own, as well [6].

In this paper, we suggest how that may look, by demonstrating the tools of Lean through illustrative proofs in the chemical sciences. First, we introduce variables, types, premises, conjectures, and proof steps through a single derivation of the Langmuir adsorption model. Next, we show the use of functions and definitions of mathematical objects by defining the Lenanrd-Jones potential and showing the potential energy tends to infinity as the radius approaches zero. Finally, we show more advanced topics, such as using geometric series to formalize the derivation of the BET equation, and using classes to define the basis of thermodynamics.

2 Methods

Below we outline the proofs we formalized in Lean version 3.45.0. We hosted all the proofs in our website that provides a semi-interactive platform connecting to the Lean codes in our GitHub repository ⚛. An extended methods section introducing Lean is in the Supporting Information section 5.1.

3 Proofs

3.1 Langmuir Adsorption: Introducing Lean Syntax and Proofs

The Langmuir adsorption model explains loading of an adsorbate onto a surface under isothermal conditions [38] given by the Equation 1:

$$\theta_A = \frac{K_{eq} p_A}{1 + K_{eq} p_A} \tag{1}$$

where θ is the fractional of sites occupied by A, K_{eq} is equilibrium constant, and p_A is partial pressure of A. Several derivations have been developed [36, 57, 41] and here we formalize the derivation of Langmuir model through kinetics [41].

The central premises of the proof are expressions of adsorption rate (*hrad*), desorption rate (*hrd*), the equilibrium relation (*hreaction*) and the adsorption site balance (hS_0). The premises hK and $h\theta$ are local definitions that comes from the first four premise while hc1, hc2, and hc3 are mathematical constraints that appear during the formalization. The model assumes the system is in equilibrium, so the adsorption rate, $r_{ad} = k_{ad} p_A [S]$ and desorption rate, $r_d = k_d p_A [A]$ are equal to each other, where k_{ad} and k_d are the adsorption and desorption rate constant respectively, $[S]$ is concentration of empty sites, and $[A]$ is concentration of sites filled with A.

Figure 1: A formalization of Langmuir's adsorption model, shown as screenshots from Lean operating in VSCode. The left side of the figure shows the code window, while the right side shows variables and goals at each step in the tactic state. When the user places the cursor at one of the numbered locations in the "Code Window," VSCode displays the "Tactic State" of the proof. The turnstile symbol represents the state of the goal after each step. As each tactic is applied, hypotheses and/or the goal is updated in the tactic state as the proof proceeds. For clarity, we only show the hypothesis that changes after a tactic is applied and how that changes the goal. As an example, the goal state is the same in steps 1 and 2, since the first tactic rewrites (rw) the equation of adsorption (*hrad*) and desorption (*hrd*) into the premise that equilibrium (*hreaction*) exists. Next we rewrite (rw), simplify (field_simp), and otherwise rearrange the variables to exactly equal the goal state (steps 3-5). When the proof is finished, a celebratory message and party emoji appears (6).

As shown in figure 1, we can state every premise explicitly in Lean. While this is a more natural way of writing but we illustrate later why using local definitions is more desirable than this approach.

However these premises can be condensed through careful construction. For instance, the first two premises *hrad* and *hrd* can written into *hreaction* to yield *k_ad*P*S = k_d*A* and we can also write expressions of *h*θ and *hK* in the goal statement. While *hrad*, *hrd*, *h*θ, and *hK* each have their own scientific significance, in mathematics they are just a combination of variables. More details about two other improvised versions of Langmuir proof is in the supplementary information section 5.2.1.

An interesting part of the proof is that only certain variables or their combinations are required to be not zero. When building this proof, Lean imports the real numbers and the formalized theorems and tactics for them in `mathlib`. Lean does not permit division by zero, and it will flag issues when a real number being divide might be equal to zero. Consequently, we must include additional hypothesis *hc1-hc3* in order to complete the proof. These ambiguities are better solved when we switch to using *classes* and *definitions*, because then we can prove properties about the object, such as variables that must not be zero will come from scientific significance rather than mathematical manipulation. However, this version of the Langmuir proof is still a formalized proof. Though not reusable, it is still machine readable and executable.

3.2 Lennard-Jones Potential: Introducing Functions and Definitions in Lean

When a mathematical object is formally defined in Lean, multiple theorems can be written about it with certainty that all proofs pertain to the same object. Functions in Lean are similar to functions in imperative programming languages like Python and C, in that they take in arguments and map them to outputs. However, functions in Lean (like everything in Lean) are also objects with properties that can be formally proved.

We illustrate using the Lennard-Jones potential (Eq. 2), a popular equation in molecular simulations for modeling intermolecular interactions. The potential energy $V(r)$ is a function of the radius between two particles, *r*, [24].

$$V(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \tag{2}$$

Two parameters, ϵ and σ, govern the strength and distance of interaction, respectively. As well, the Lennard-Jones function exhibits a minimum energy at the radius $r_m = 2^{1/6}\sigma$.

Formally, a function is defined as a mapping of one set (the domain) to another set (the co-domain). The notation for a function is given by the arrow "$\rightarrow$". For instance, the function conventionally written as $Y = f(X)$ or $Y(X)$ maps from set *X* to set *Y* is $X \rightarrow Y$ in arrow notion.[1]

Importantly, the arrow "$\rightarrow$" is also used to represent the conditional statement (if-then) in logic, but this is not a duplication of syntax. Because everything is a type in Lean, functions map type *X* to type *Y*; when each type is a proposition, the resulting function is an if-then statement.

As stated in the introduction, Lean's power comes from the ability to define objects globally, not just postulate them for the purpose of a local proof. In Lean, we use *def* to define new objects and then prove statements about these objects. The *def* command has three parts: the arguments it takes in (which are the properties of the object), the type of the output, and the proof that the object has such a type. In Lean:

```
def name properties : type := proof of that type
```

For instance, we can define a function that doubles a natural number:

```
def double : ℕ → ℕ := λ n : ℕ, n + n
```

The λ symbol comes from lambda-calculus and is how an explicit function is defined. After the lambda symbol is the variable of the function, *n* with type ℕ. Then after the comma is the actual function. By hand, we would write this as $f(n) = n + n$. This definition allows us, as the name suggests, to double any natural number. So, using it, we could show that:

```
double (3 : ℕ) = (6 : ℕ)
```

which we could easily show to be true.

[1]These types are easily extended to functionals, which are central to density functional theory. A function that takes a function as an input can be simply defined by $(\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R}$

Returning to the Lennard-Jones potential, we define the Lennard-Jones equation as a function mapping from the real numbers to the real numbers, $\mathbb{R} \to \mathbb{R}$. In this case, we are assuming, for now, that the radius between the two molecules is given as a one-dimensional real number. In the future, this could be generalized so that radius is given as an n-dimensional real vector, and the function has type $\mathbb{R}^n \to \mathbb{R}$ (the energy returned is still a scalar $\mathbb{R}$). Now that we know the type of the Lennard-Jones object, the proof of the type is the Lennard-Jones equation. In Lean:

```
def LJ (ϵ minRadius : ℝ)  : ℝ → ℝ :=
let σ := (1/2^(1/6))*minRadius
inλ r, 4*ϵ*(‖σ/r‖^(12)-‖σ/r‖^(6))
```

We note a couple things. First, while it is common for simulators to use σ as the characteristic length for LJ, we instead use the minimum radius (*minRadius*), which naturally describes more mathematical properties, leading to shorter and simpler proofs. Instead of taking in σ, we use a local definition of sigma, which is just a rearrangement of the equation defining the minimum radius. Second, the double bars signify the norm function. The norm takes in a general vector and outputs a real number. For an n-dimensional real vector, this is just the distance function. For this definition, the norm isn't as important since radius is one-dimensional, but in the future, this is necessary to convert an n-dimensional vector of radius to a scalar number.

Now with the Lennard-Jones model defined, we can prove properties about it. For this example, we show a theorem proving that the Lennard-Jones function tends to infinity as the radius approaches zero from the right. For this, we use the *tendsto* function (which is also an object defined in Lean). In Lean:

```
theorem tendsto_at_top_at_zero_radius
(ϵ minRadius: ℝ )
(hx : ∀ x, x ∈ {r : ℝ  | r ≠ 0})
(hϵ : 0 < ϵ)
(hm :  0 < minRadius)
: filter.tendsto  ( LJ ϵ minRadius ) (nhds_within 0 (set.Ioi 0) ) filter.at_top:=
```

The first part of the *tendsto* object takes in the function we are analyzing. Here, we see how the Lennard-Jones object we defined can be used. The next part is the filter along which we approach the point. *nhds_within* means the intersection between a neighborhood (an open set around a point) and another set. Here we have the intersection between the neighborhood around 0 and the set interval $(0, \infty)$, which specifies we are approaching from the right.

The final part is the filter the function approaches. *filter.at_top* means the top most element of a type, which we call positive infinity for the numbers we usually deal with. This proof is rather long, since it requires applying L'Hôpital's rule six times, but it can nonetheless be rigorously proved using the definitions and theorems previously formalized in `mathlib`. Several other properties of *this* LJ function can likewise be proved, including properties about the derivative of the Lennard-Jones function.

3.3 BET Adsorption: Advanced Proof using Definitions

Brunauer, Emmett, and Teller introduced the BET theory of multilayer adsorption (see Figure 2) in 1938 [11]. We formalize this derivation, beginning with Equation 26 from the paper, which is shown here in Equation 3:

$$\frac{V}{A * V_0} = \frac{cx}{(1-x)(1-x+cx)} \tag{3}$$

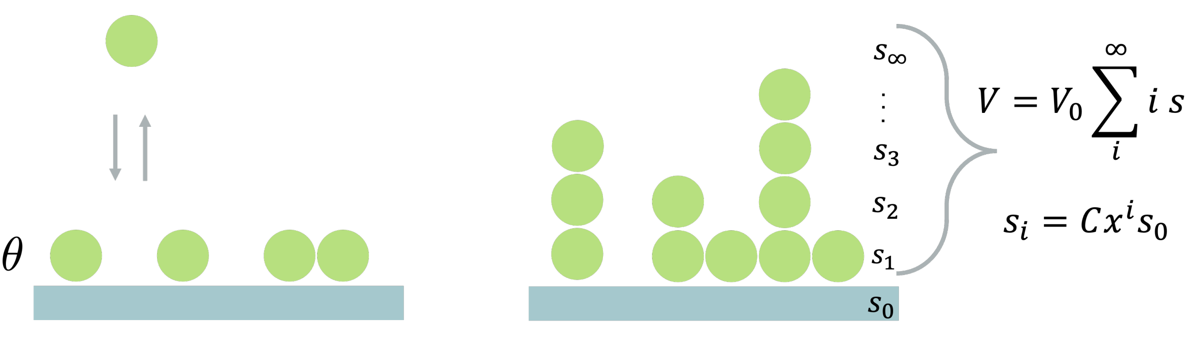

Figure 2: Langmuir model vs BET model. The BET model, unlike Langmuir, allows particles to create infinite layers on top of previously adsorbed particles. Here θ is fraction of the surface adsorbed, V is total volume adsorbed, V_0 is the volume of a complete unimolecular layer adsorbed in unit area, s_i is the surface area of the i$^{\text{th}}$ layer, s_0 is the surface area of the zeroth layer, and *x* and *C* are constants that relates heats of adsorption of the molecule in layers.

Here A is the total area adsorbed by all (infinite) layers expressed as sum of infinite series:

$$A = \sum_{i=0}^{\infty} s_i = s_0(1 + c\sum_{i=1}^{\infty} x^i) \tag{4}$$

and V is the total volume adsorbed is given by:

$$V = V_0 \sum_{i=0}^{\infty} i s_i = c s_0 V_0 \sum_{i=1}^{\infty} i x^i \tag{5}$$

The variables *y*, *x* and *C* are expressed in the original paper as shown through Equation 6 to 8:

$$y = PC_1,\ where\ C_1 = (a_1/b_1)e^{E_1/RT} \tag{6}$$

$$x = PC_L,\ where\ C_L = e^{E_L/RT}/g \tag{7}$$

$$C = y/x = C_1/C_L \tag{8}$$

where a_1, b_1, and g are fitted constants, E_1 is the heat of adsorption of the first layer, E_L is for the second (and higher) layers (also the same as heat of liquefaction of the adsorbate at constant temperature), R is the universal gas constant, and T is temperature. In Equation 6 and 7, everything besides the pressure term is constant, since we are dealing with an isotherm, so we group the constants together into one term.

These constants, along with the surface area of the zeroth layer, given by s_0, saturation pressure, and the three constraints are defined using the *constant* declaration in Lean. Mathematical objects can also be defined in other ways such as *def* or *class* [6] but for this proof we will use *constant* which is simple and easy to use for beginners. We will illustrate later in our thermodynamics proof how constants can be merged into a Lean *class* for reusability.

In Lean, this is ⚛:

```
constants (C_1 C_L s_0 P_0: ℝ)
(hCL : 0 < C_L) (hC1 : 0 < C_1) (hs_0 : 0 < s_0)
```

With these *constant* declarations, we can now define *y, x,* and *C* in Lean as ⚛:

```
def BET_first_layer_adsorption_rate (P : ℝ) := (C_1)*P
local notation ‘y’ := BET_first_layer_adsorption_rate

def BET_n_layer_adsorption_rate (P : ℝ):= (C_L)*P
local notation ‘x’ := BET_n_layer_adsorption_rate

def BET_constant := C_1/C_L
local notation ‘C’ := BET_constant
```

Since *y* and *x* are both functions of pressure, so their definitions require pressure as an input. Alternatively, the input can be omitted if we wanted to deal with x as a function, rather then a number. Notice that the symbols we declared using *constant* do not need to be supplied in the inputs as they already exist in the global workspace.

We formalize Equation 3 by recognizing that the main math behind the BET expression is an infinite sequence which describes the surface area of adsorbed particles for each layer. The series is defined as a function that maps the natural numbers to the real numbers; the natural numbers representing the indexing. It is defined on two cases: if the index is zero, it outputs the surface area of the zeroth layer, and if the index is the $n+1$, it outputs $x^{n+1}s_0C$.

$$s_i = Cx^i s_0 \; for \; i : [1, \infty) \tag{9}$$

In Lean, we define this sequence as ⚛:

```
def seq (P : nnreal) : ℕ → ℝ
|(0 : ℕ)    := s_0
|(nat.succ n)    := x^(n+1)*s_0*C
```

Where s_i is the surface area of the i$^{\text{th}}$ layer, C and x are given by Equation 8 and Equation 7, respectively, and s_0 is the surface area of the zeroth layer. The zeroth layer is the base surface and is constant.

We now have the area and volume equations both in terms of geometric series with well defined solutions. The BET equation is defined as the ratio of volume absorbed to the volume of a complete unimolecular layer, given by Equation 10.

$$\frac{V}{A * V_0} = \frac{cs_0 \sum_{i=1}^{\infty} ix^i}{s_0(1 + c\sum_{i=1}^{\infty} x^i)} \tag{10}$$

The main math of BET is simplifying this sequence into a simple fraction which involves solving the geometric series. The main math goal is given by Equation 11.

$$\frac{c\sum_{i=1}^{\infty} ix^i}{(1 + c\sum_{i=1}^{\infty} x^i)} = \frac{cx}{(1-x)(1-x+cx)} \tag{11}$$

Before doing the full derivation, we prove Equation 11, which we call *sequence_math*. We define a separate proof because there are many ways to rearrange the BET equation, but all of them rely on the same theorem. So, rather then derive Equation 11 each time, we can derive it once and use it in other proofs. In Lean, this is ⚛:

```
lemma BET.sequence_math {P : ℝ} (hx1: (x P) < 1) (hx2 : 0 < (x P)) :
    (∑' k : ℕ, ((k + 1)*(seq P (k+1))))/(s_0 + ∑' k, (seq P (k+1))) =
    C*(x P)/((1 -  (x P))*(1 - (x P) + (x P)*C)) :=
```

In Lean, the apostrophe after the sum symbol denotes that its an infinite sum, which are defined to start at zero since they are indexed by natural numbers, which start at zero. Since Equation 11 has an infinite sum that starts at one, we add one to all the indexes, *k*, so that when k is zero, we get one, etc. We also define two new theorems that derive the solution to these geometric series with an index starting at one. After expanding *seq*, we use those two theorems, and then rearrange the goal to get two sides that are equal. We also use the tag *lemma* instead of *theorem*, just to specify that its used to prove other theorems. The tag *lemma* has no functional difference from *theorem* in Lean, its just there for mathematicians to label proofs.

With this we can formalize the derivation of Equation 3. First we define Equation 3 as a new object and then prove a theorem showing we can derive this object from the sequence. In Lean, the definition looks like this ⚛:

```
def brunauer_26 := λ P :  ℝ, C* (x P)/((1-(x P))*(1-(x P)+C*(x P)))
```

Here, we explicitly define this as a function, because we want to deal with Equation 3 normally as a function of pressure, rather then just a number. Now we can prove a theorem that formalizes the derivation of this equation ⚛:

```
theorem brunauer_26_from_seq
{P V_0 : ℝ}
(hx1: (x P) < 1)
(hx2 : 0 < (x P))
:
  let Vads :=  V_0 * ∑' (k : ℕ), ↑k * (seq P k),
      A :=  ∑' (k : ℕ), (seq P k) in
  Vads/A = V_0*(brunauer_26 P)
:=
```

Unlike the Langmuir proof introduced earlier 1, the BET uses definitions that allowed reusability of those definitions across the proof structure. The proof starts by showing that *seq* is summable. This just means the sequence has some infinite sum and the $\sum'$ symbol is used to get the value of that infinite series. We show in the proof that both *seq* and *k*seq* are summable, where the first is needed for the area sum and the second is needed for the volume sum. After that, we simplify our definitions, move the index of the sum from zero to one so we can simplify the sequence, and apply the *BET.sequence_math* lemma we proved above. Finally, we use the *field_simp* tactic to rearrange and close the goal. With that, we were able to formalize the derivation of Equation 3, just at Brunauer et. al did in 1938.

In the supplementary section, we continue with the remaining derivation of the BET theory, by deriving Equation 28 from the paper, given by Equation 12

$$\frac{V}{A * V_0} = \frac{CP}{(P_O - P)(1 + (C-1)(P/P_0)} \tag{12}$$

This follows from recognizing that $1/C_L = P_0$. While Brunauer et. al. attempt to show this in the paper, we go over the trouble with implementing such logic. Instead, we show a similar proof that Equation 3 approaches infinity as pressure approaches $1/C_L$, and assume as a premise in the derivation of Equation 12 that $1/C_L = P_0$.

3.4 Classical Thermodynamics and Gas Laws: Introducing Lean Class

Lean is so expressive because it enables relationships between mathematical objects and we can use this functionality to precisely define and relate *scientific concepts* with mathematical certainty. We illustrate this by formalizing proofs of gas laws in classical thermodynamics.

We can prove that the ideal gas law, $PV = nRT$ follows Boyle's Law, $P_1V_1 = P_2V_2$, following the style of our derivation of Langmuir's theory: demonstrating that a conjecture follows from the premises ⚛. However, this proof style doesn't facilitate interoperability among proofs and limits the mathematics that can be expressed.

In contrast, we can prove the same, more systematically, by first formalizing the concepts of thermodynamic system and thermodynamic states, extending that system to a specific ideal gas system, defining Boyle's Law in light of these thermodynamic states, and then proving that the ideal gas obeys Boyle's Law (see Figure 3).

Classical thermodynamics describes the macroscopic properties of thermodynamic states and relationships between them. [20, 52]. We formalize the concept of "thermodynamic system" by defining a Lean class called *thermo_system* over the real numbers , with thermodynamic properties (e.g. pressure, volume, etc.) defined as functions from the natural numbers to the real nummbers $\mathbb{N} \to \mathbb{R}$. This represents picking out the state of the system, where the natural numbers are the states. So, state one and state two can map to different values of pressure, or the same value depending on how the systems changes from state one to two. Since these are state variables, we are only concerned with what happens at specific states, not what happens between state one and two. In Lean, this is ⚛:

```
class thermo_system :=
(pressure : ℕ → ℝ)
(volume :   : ℕ → ℝ)
(temperature : ℕ → ℝ)
(substance_amount : ℕ → ℝ)
(energy : ℕ → ℝ)
```

We define six descriptions of the system: isobaric (constant pressure); isochoric (constant volume); isothermal (constant temperature); adiabatic (constant energy); closed (constant mass); and isolated (constant mass and energy). Each of

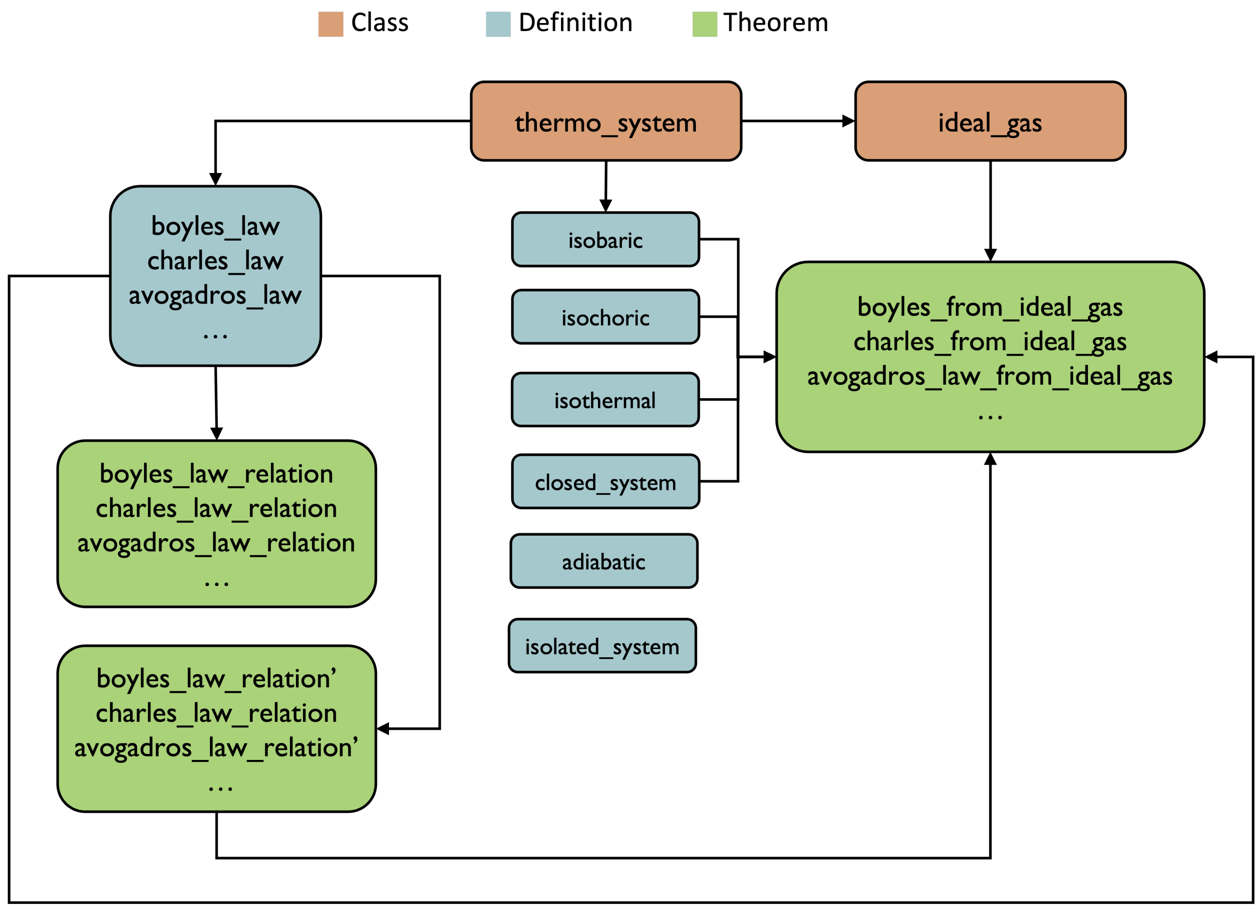

Figure 3: Thermodynamic system in Lean. Here the *thermo_system* and *ideal_gas* are Lean `classes` that describe different kinds of thermodynamic systems like *isobaric*, *isochoric*, *isothermal* etc. using Lean `definitions` to proof `theorems` relating to the *gas laws*.

these conditions has the type *Prop*, or proposition, considering them to be assertions about the system. We formally define these by stating that, for all (∀) pairs of states n and m, the property at those states is equal. We define these six descriptions to take in a *thermo_system*, since we need to specify what system we are ascribing this property to. In Lean, this is :

```
def isobaric  (M : thermo_system ) : Prop :=
∀ n m : ℕ, pressure n = pressure m
def isochoric(M : thermo_system ) : Prop :=
∀ n m : ℕ, volume n = volume m
def isothermal(M : thermo_system ) : Prop:=
∀ n m : ℕ, temperature n = temperature m
def adiabatic(M : thermo_system ) : Prop :=
∀ n m : ℕ, energy n = energy m
def closed_system (M : thermo_system ) : Prop:=
∀ n m : ℕ, substance_amount n = substance_amount m
def isolated_system (M : thermo_system ) : Prop :=
adiabatic M ∧ closed_system M
```

We define an isolated system to be just a closed system and (∧) adiabatic, rather then using the universal quantifier (∀), since it would be redundant.

Now that the basics of a thermodynamics system has been defined, we can define models that attempt to mathematically describe the system. These models can be defined as another class, which *extends* the *thermo_system* class. When a class extends another class, it inherits the properties of the class it extended. This allows us to create a hierarchy of classes so we don't have to redefine properties over and over again. The most well known model is the ideal gas model, which comes with the ideal gas law equation of state. We define the ideal gas model to have two properties, the universal gas constant, R, and the ideal gas law. In the future, we plan to add more properties to the definition, especially as we expand on the idea of energy. We define the ideal gas law as an equation relating the products of pressure and volume to the product of temperature, amount of substance, and the gas constant. In Lean, this is ⚛:

```
class ideal_gas
  extends thermo_system :=
(R : ℝ)
(ideal_gas_law : ∀ n : ℕ, (pressure n)*(volume n) = (substance_amount n)*R*(temperature n))
```

To define a system modeled as an ideal gas, we write in Lean: *(M : ideal_gas* ℝ*)*. Now we have a system, M, modeled as an ideal gas.

Boyle's law states that the pressure of an ideal gas is inversely proportional to the systems volume in an isothermal and closed system [39]. This is mathematically given by Equation 13, where P is pressure, V is volume, and k is a constant whose value is dependent on the system.

$$PV = k \tag{13}$$

In Lean, we define Boyle's Law as ⚛:

```
def boyles_law (M : thermo_system ) :=
∃(k : ℝ), ∀ n : ℕ, (pressure n) * (volume n)= k
```

We use the existential operator (∃) on k, which can be read as *there exists a* k, because each system has a specific constant. We also define the existential before the universal, so its logically correct. Right now it reads, *there exists a* k*, such that for all states, this relation holds*. If we write it the other way, it would say *for all states, there exists a* k*, such that this relation holds.* The second way means that k is dependent on the state of the system, which isn't true. The constant is the same for any state of a system. Also, even though Boyle's law is a statement about an ideal gas, we define it on a general system so, in the future, we can look at what assumptions are needed for other models to obey Boyle's Law.

Next, we prove a couple of theorems relating to the relations that can be derived from Boyle's law. From Equation 13, we can derive a relation between any two states, given by Equation 14, where n and m are two states of the system.

$$P_n V_n = P_m V_m \tag{14}$$

The first theorem we prove shows how Equation 14 follows from Equation 13. In Lean this looks like ⚛:

```
theorem boyles_law_relation  (M : thermo_system ) :
boyles_law M → ∀ n m : ℕ, pressure n * volume n = pressure m * volume m :=
```

The right arrow can be read as *implies*, so the statement says that *Boyle's law implies Boyle's relation*. This is achieved using modus ponens, introducing two new names for the universal quantifier, then rewriting Boyle's law into the goal, by specializing Boyle's law with n and m. We also want to show that the inverse relation holds, such that Equation 14 implies Equation 13. In Lean, this is ⚛:

```
theorem boyles_law_relation'  (M : thermo_system ) :
(∀ n m, pressure n * volume n = pressure m * volume m) → boyles_law M :=
```

We begin in the same way, by using modus ponens and simplifyin Boyle's law to be in the form of Equation 13. Next, satisfy the existential, by providing an old name. In our proof, we use $P_1 V_1$ as an old name for k, then we specialize the relation with n and 1 and close the goal.

Finally, with these two theorems, we show that Boyle's law can be derived from the ideal gas law, under the assumption of an isothermal and closed system. In Lean, this is ⚛:

```
theorem boyles_from_ideal_gas  (M : ideal_gas ) :
 (iso1 : isothermal M.to_thermo_system )  (iso2 : closed_system M.to_thermo_system ):
boyles_law M:=
```

This proof is completed by using the second theorem for Boyle's relation along with simplifying the ideal gas relation using the two *iso* constraints.

We have implemented this framework to prove both Charles and Avagadro's Law illustrating its expansiveness across thermodynamic systems. In the future, we plan to define energy and prove theorems relating to it including the laws of thermodynamics [5].

3.5 Kinematic equations: Calculus in Lean

Calculus and differential equations are ubiquitous in chemical theory, and the `mathlib` library has lots of it formalized that we can explore. To illustrate Lean's calculus capabilities and motivate future work in this, we formally prove that the kinematic equations follow from calculus-based definitions of the equations of motion. These equations are the basis for many chemical theories, such as reaction kinetics [25] and molecular dynamics [29] that use Newtonian mechanics.

The equations of motion are a set of two, coupled, differential equations that relate the position, velocity, and acceleration of an object in an n-dimensional vector space [9]. The differential equations are given by Equation 15 and 16, where **x**, **v**, and **a** represents position, velocity, and acceleration respectively. The bold type face is used to signify a vector quantity. All three variables are parametric equations, where each dimension of the vector is a function of time. So, while they are vector quantities, we don't have to deal with partial derivatives. It would have been possible to construct these proofs using partial differential equations, however mathlib currently have limited theorem of partial derivatives.

$$\mathbf{v}(t) = \frac{d(\mathbf{x}(t))}{dt} \tag{15}$$

$$\mathbf{a}(t) = \frac{d(\mathbf{v}(t))}{dt} \tag{16}$$

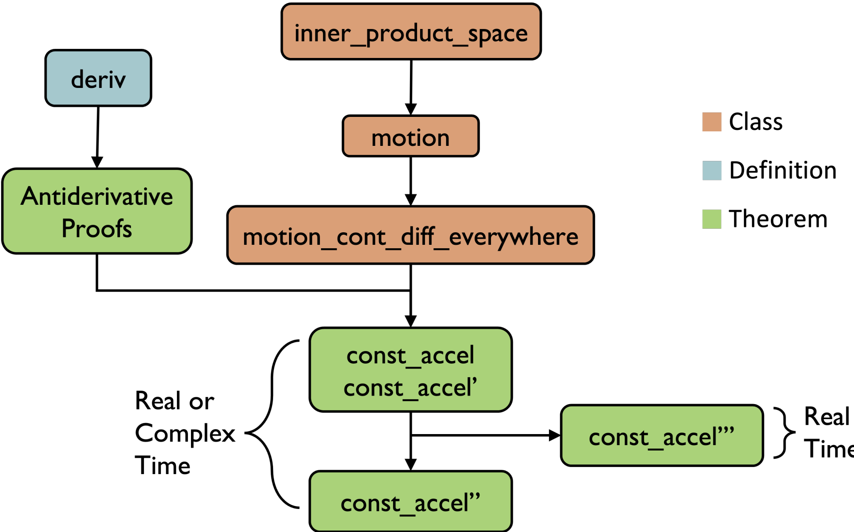

Figure 4: Kinematics in Lean. Here we define *motion* as Lean `class` that represents the relation between position, velocity and acceleration through differential equations that are proved using `definitions` of *derivative* functions.

Just like in the thermodynamics section, we can define a class that encompasses the main definition of motion. This class will define three new elements, representing position, velocity, and acceleration, which are functions. It will also define the two differential equation relating these three functions. This class also extends inner product space, which is a real or complex vector space with an operator (the inner product). The inner product is a generalization of the dot product for any vector space. By extending inner product space, the motion class inherits all of inner product space's properties, which are needed to use calculus. In Lean :

```
class  motion (𝕂 : Type u_1) (E : Type u_2) [is_R_or_C 𝕂]
  extends inner_product_space 𝕂 E :=
(position velocity acceleration : 𝕂 → E )
(hvel : velocity = deriv position)
(hacc : acceleration = deriv velocity)
```

$\mathbb{K}$ represents a field which we require to be either a real or complex number, and E symbolizes a general vector field. In mathematics, a field is an algebraic structure with addition, subtraction, multiplication, and division operations. We define a general vector space rather vector space could be an n-dimensional Euclidean vector space, to make the class as general as possible. This allows us to talk about motion, not just in a Euclidean vector space, but also a hyperbolic vector space, or a vector space with special properties.

Lean has two definitions of the derivative, the familiar "high school" derivative for single variable functions, called *deriv*, and the more general Fréchet derivative, called *fderiv*. The Fréchet derivative can be thought of as the generalization of the derivative of a function of a single variable, to the derivative of a function of multiple variables, which gives the total derivative of a function as a linear map. For the purposes of the kinematic equations, the "high-school" derivative suffices.

In Lean, if a function is not differentiable at a point, the derivative at that point returns zero. However, there is a special scientific importance when the derivative equals zero, especially for kinematics. Therefore, we define another class to require the equations of motion to be n-times continuously differentiable everywhere. We only require the equations to be n-time differentiable, instead of infinitely differentiable for generality reasons, however a theorem can instantiate this class and assume infinite differentiablity. We also declare this as a separate class, instead of in the *motion* class, because, in the future, we plan to define another class that only require the equations to be n-times continuously differentiable on a set, rather then everywhere. That way, depending on the theorem, the user can choose which extentsion they want. In Lean, this class looks like ⚛:

```
class  motion_cont_diff_everywhere (𝕂 :Type u_1) (E : Type u_2) [is_R_or_C 𝕂]
  extends motion 𝕂 E :=
(contdiff : ∀ n : with_top ℕ,  m : ℕ,
(m < n) ∧ (cont_diff 𝕂 n (deriv^[m] position)))
```

The field *contdiff* states that for all n, defined as a natural number including positive infinity, and for all m, defined as a natural number, m is less than n, and the mth derivative of position is continuously differentiable n-times. The carrot symbol is the logical symbol for *and*, which requries both sides to true.

When acceleration is constant, meaning the acceleration points linearly in one direction, there are four useful analytical solutions to this set of differential equations. These kinematic equations are given by Equations 17 - 20, where the subscript naught denotes that variables evaluated at time equals zero.

$$\mathbf{v}(t) = \mathbf{a}t + \mathbf{v}_0 \tag{17}$$

$$\mathbf{x}(t) = \frac{\mathbf{a}t^2}{2} + \mathbf{v}_0 t + \mathbf{x}_0 \tag{18}$$

$$v^2(t) = v_0^2 + 2\mathbf{a} \cdot \mathbf{d} \tag{19}$$

$$\mathbf{x}(t) = \frac{\mathbf{v}(t) + \mathbf{v}_0}{2} t + \mathbf{x}_0 \tag{20}$$

Under the assumption of one dimensional motion, these equations simplify to the familiar introductory kinematic equations. Equation 19, also known as the Torricelli Equation, uses the shorthand square to represent the dot product, $v^2(t) \equiv \mathbf{v(t)} \cdot \mathbf{v(t)}$.

With this, we now can begin deriving the four kinematic equations. The first three derivations, Equation 17 - 19, all use the same premises, given below:

```
(𝕂 : Type u_1) (E : Type u_2) [is_R_or_C 𝕂]
(M : motion_cont_diff_everywhere 𝕂 E)
(A : E) (n : with_top ℕ)
(accel_const : motion.acceleration = λ (t : 𝕂), A)
```

The first line contains three premises to declare the field and vector space the motion space is defined on. The next line defines a motion space, M. The third line contains two premises, a variable, A, which represents the value of constant

acceleration, and n, the number of times position can be differentiated. When applying these theorems, the *top* function, which means positive infinity in Lean, can be used to specify n. The final line is a premise that assumes acceleration is constant. The lambda function is constant, because A is not a function of t, so for any value of t, the function outputs the same value, A. The three kinematic equations in Lean are given below (note, the premises are omitted since they have already been given above).

```
theorem const_accel premises : velocity = λ (t : 𝕂), t·A + velocity 0 :=

theorem const_accel' premises :
position = λ (t : 𝕂), (t^2)/2·A + t·(velocity 0) + position 0 :=

theorem const_accel'' premises :
∀ t : 𝕂, position t = (t/2)·((velocity t) - (velocity 0)) + position 0 :=
```

The · symbol is used to indicate the scalar multiplication symbol, which is when a vector value is multiplied by a scalar value. Normally we are used to seeing the · symbol used to represent the dot product, but Lean uses the *inner* function for the dot product. Also, *velocity 0* means the velocity function evaluate at 0. Lean use parenthesises for orders of operations, not for function inputs, so $f(x)$ in normal notation converts to $f\,x$ in Lean. The proof of the first two theorems use the antiderivative, whose formalization we explain in the supplementary information **??**, along with the two differential equations from the motion class. The third theorem is proved using the previous two theorems and rearranging them.

The fourth kinematic equation, Equation 20, uses the inner product. The inner product is a function that takes in two vectors from a vector space, and outputs a scalar. If the vector space is a Euclidean vector space, this is just the dot product. While the other three kinematic equations held for both real or complex time, we were unable to prove Equation 20 for complex time. Meaning, so far, it only holds for real time. This has to do with the complex conjugate that arises when simplifying the proof. The inner product is semi-linear, linear in its first argument, Equation 21, but sesquilinear in the second argument, Equation 22.

$$\langle ax + by, z\rangle = a\langle x, z\rangle + b\langle y, z\rangle \tag{21}$$

$$\langle x, ay + bz\rangle = \bar{a}\langle x, y\rangle + \bar{b}\langle x, z\rangle \tag{22}$$

The bar denotes the complex conjugate. For a complex number, $g = a + bi$, the complex conjugate is: $\bar{g} = a - bi$. If g is a real number, then $g = \bar{g}$. For the proof of Equation 20, we get to a form where one of the inner products has addition in the second term that we have to break up, and no matter which way we rewrite the proof line, one of the inner products ends up with addition in the second term. Therefore, we needed to define the final kinematic equation to hold only for real time. In Lean, this looks like :

```
theorem real_const_accel'''
(N : motion_cont_diff_everywhere ℝ E)
(accel_const : N.to_motion.acceleration =  (t :ℝ), A)
{n : with_top ℕ}
:

∀ t : ℝ, inner (motion.velocity t) (motion.velocity t) =
inner (motion.velocity 0) (motion.velocity 0) +
2 * inner A ((motion.position t) - (motion.position 0)) :=
```

While we haven't definitely proved that Equation 20 doesn't hold for complex time, we, so far, have run into contradictions when accounting for the possibility for complex time. Thus, Equation 20, currently only holds for real time. An imaginary-time framework can be used to derive equations of motion from non-standard Lagrangians [49, 50], which offers the opportunity to examine many hidden properties in classical and quantum dynamical systems that can be explored in the future.

4 Discussion and Outlook

In this paper, we demonstrate how the Lean theorem prover works and how it can be used to describe scientific relationships by formally proving fundamental scientific theories and engineering mathematics. We observed that in some cases when scientific statements are converted into formal language it reveals hidden assumptions behind mathematical derivations. Although formalization can be a slow process, the fundamental theory only needs to be

formally verified once, and it can then be used for the development of other theories. Thus we have not just proved a couple of theorems about scientific objects, but have begun to create an interconnected structure of proofs relating fields of science through type classes.

Mathematicians have traditionally produced lengthy hand-written proofs that required computers to verify their accuracy. For instance, it took 12 mathematicians years to prove Kepler's conjecture was 99% correct [32], and finally, the Flyspeck project headed by Hales [31] formalized the proof using Isabelle and HOL Light. Much of our motivation for this work is to assign the verification of complex theories to computers that can use a library of formally verified foundational theories. As science, engineering, and mathematics advance rapidly, it will become increasingly difficult to verify these manually with certainty. The limitations of hand-written proofs and their reliability have also been discussed in the literature [12, 31, 7], where formalization and automated theorem provers can help.

Though we are formalizing proofs for chemical theory, our proofs will have a cascading effect on the efforts of others aiming to formalize their fields. An ever-growing community of people from different backgrounds led by mathematicians has helped build `mathlib` [1], and we anticipate a similar group of scientists building a library of proofs relating to scientific theories. Our next goals for Lean are to set out the foundation for classical mechanics, to continue building out thermodynamics, and to look at some higher-level proofs like the Noether Theorem [37]. All our proofs for this paper uses Lean version 3 and its `mathlib` library which is not yet imported into the newer version of Lean 4 [44] that incorporates functional programming and domain-specific automation [44].

Acknowledgements

We are grateful to the Lean prover community and contributors of mathlib on whose work this project is built upon. We especially thank Kevin Buzzard, Patrick Massot, Tomas Skrivan, Eric Wieser, and Andrew Yang for helpful comments and discussions around our proof structure and suggestions for improvement. We thank Charles Fox, Mauricio Collares, and Ruben Van de Velde for helping us out with the website. This material is based upon work supported by the National Science Foundation under Grant No. (NSF #218938), as well as startup funds from the University of Maryland, Baltimore County.

Conflicts of Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Lean Prover Community on Zulip. `https://leanprover.zulipchat.com/`.

[2] MATLAB. `https://www.mathworks.com/products/simulink.html`.

[3] Undergraduate Mathematics in mathlib. `https://leanprover-community.github.io/undergrad.html`.

[4] Kenneth Appel, Wolfgang Haken, and John Koch. Every planar map is four colorable. Part II: Reducibility. *Illinois Journal of Mathematics*, 21(3):491–567, 1977.

[5] Peter Atkins. *The Laws of Thermodynamics: A Very Short Introduction*. OUP Oxford, 2010.

[6] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. Theorem proving in Lean. *Release*, 3(0):1–4, 2015.

[7] Jeremy Avigad and John Harrison. Formally verified mathematics. *Communications of the ACM*, 57(4):66–75, 2014.

[8] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[9] Joseph Stiles Beggs. *Kinematics*. CRC Press, 1983.

[10] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Mathematics in Computer Science*, 9(1):41–62, 2015.

[11] Stephen Brunauer, P. H. Emmett, and Edward Teller. Adsorption of gases in multimolecular layers. *Journal of the American Chemical Society*, 60(2):309–319, 1938.

[12] Alan Bundy, Mateja Jamnik, and Andrew Fugard. What is a proof? *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 363(1835):2377–2391, 2005.

[13] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 299–312, 2020.

[14] Kevin Buzzard, Johan Commelin, and Patrick Massot. Formalising perfectoid spaces, 2020.

[15] Kevin Buzzard and Mohammad Pedramfar. Natural number game. `https://github.com/ImperialCollegeLondon/natural_number_game`, 2020.

[16] Alba Cervera-Lierta, Mario Krenn, and Alán Aspuru-Guzik. Design of quantum optical experiments with logic artificial intelligence. *arXiv preprint arXiv:2109.13273*, 2021.

[17] Thierry Coquand and Jean H Gallier. A proof of strong normalization for the theory of constructions using a kripke-like interpretation, 1990.

[18] Thierry Coquand and Gérard Huet. *The Calculus of Constructions*. PhD thesis, INRIA, 1986.

[19] Thierry Coquand and Christine Paulin. Inductively defined types. In *International Conference on Computer Logic*, pages 50–66. Springer, 1988.

[20] Kevin D. Dahm and Donald P. Visco. *Fundamentals of Chemical Engineering Thermodynamics*. Cengage Learning, 2015.

[21] Sander R. Dahmen, Johannes Hölzl, and Robert Y. Lewis. Formalizing the solution to the Cap Set problem. In *10th International Conference on Interactive Theorem Proving (ITP 2019)*, volume 141 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:19, Dagstuhl, Germany, 2019. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[22] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). *Automated Deduction - CADE-25*, page 378–388, 2015.

[23] Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.

[24] Daan Frenkel and Berend Smit. *Understanding Molecular Simulation: From Algorithms to Applications*, volume 1. Elsevier, 2001.

[25] Arthur Frost and Ralph Pearson. Kinetics and Mechanism. *The Journal of Physical Chemistry*, 65(2):384–384, 1961.

[26] Paul G Goerss and John F Jardine. *Simplicial Homotopy Theory*. Springer Science & Business Media, 2009.

[27] Georges Gonthier et al. Formal proof–the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, 2008.

[28] Michael JC Gordon and Tom F Melham. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[29] James M Haile, Ian Johnston, A John Mallinckrodt, and Susan McKay. *Molecular Dynamics Simulation: Elementary Methods*, volume 7. American Institute of Physics, 1993.

[30] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Hoang Le Truong, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, et al. A formal proof of the Kepler conjecture. In *Forum of mathematics, Pi*, volume 5. Cambridge University Press, 2017.

[31] Thomas C Hales. Formal proof. *Notices of the AMS*, 55(11):1370–1380, 2008.

[32] Thomas C Hales. Historical overview of the Kepler conjecture. In *The Kepler Conjecture*, pages 65–82. Springer, 2011.

[33] Jesse Michael Han, Jason Rute, Yuhuai Wu, Edward W. Ayers, and Stanislas Polu. Proof artifact co-training for theorem proving with language models, 2021.

[34] Kevin Hartnett. Proof assistant makes jump to big-league math. `https://www.quantamagazine.org/lean-computer-program-confirms-peter-scholze-proof-20210728/`, 2021. Accessed: 2022-08-26.

[35] Sanaz Khan-Afshar, Umair Siddique, Mohamed Yousri Mahmoud, Vincent Aravantinos, Ons Seddiki, Osman Hasan, and Sofiène Tahar. Formal analysis of optical systems. *Mathematics in Computer Science*, 8(1):39–70, 2014.

[36] Maurice Kleman and Oleg D Lavrentovich. *Soft Matter Physics: An introduction*. Springer, 2003.

[37] Yvette Kosmann-Schwarzbach. The noether theorems, 2011.

[38] Irving Langmuir. The adsorption of gases on plane surfaces of glass, mica and platinum. *Journal of the American Chemical society*, 40(9):1361–1403, 1918.

[39] Ira Noel Levine. *Physical Chemistry*. McGraw-Hill, 1978.

[40] Eric Hanqing Lu. *A formalization of elements of special relativity in Coq*. PhD thesis, Harvard University, 2017.

[41] Richard I Masel. *Principles of adsorption and reaction on solid surfaces*, volume 3. John Wiley & Sons, 1996.

[42] The mathlib Community. The Lean Mathematical Library, 2020.

[43] Aaron Meurer, Christopher P Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K Moore, Sartaj Singh, et al. Sympy: Symbolic computing in python. *PeerJ Computer Science*, 3:e103, 2017.

[44] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In *International Conference on Automated Deduction*, pages 625–635. Springer, 2021.

[45] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: A proof assistant for higher-order logic*. Springer, 2002.

[46] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752. Springer, 1992.

[47] Bruno Woltzenlogel Paleo. Physics and proof theory. *Applied Mathematics and Computation*, 219(1):45–53, 2012.

[48] Stanislas Polu, Jesse Michael Han, Kunhao Zheng, Mantas Baksys, Igor Babuschkin, and Ilya Sutskever. Formal mathematics statement curriculum learning, 2022.

[49] VS Popov. Imaginary-time method in quantum mechanics and field theory. *Physics of Atomic Nuclei*, 68(4):686–708, 2005.

[50] El-Nabulsi Ahmad Rami, Tirdad Soulati, and Hamidreza Rezazadeh. Non-standard complex lagrangian dynamics. *J. Adv. Res. Dyn. Control Syst*, 5:50–62, 2013.

[51] Piotr Rudnicki. An overview of the Mizar project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–330, 1992.

[52] Stanley I Sandler. *Chemical, biochemical, and engineering thermodynamics*. John Wiley & Sons, 2017.

[53] Peter Scholze. Lean liquid. `https://github.com/leanprover-community/lean-liquid`, 2020.

[54] Muhammad Umair Siddique. *Formal analysis of geometrical optics using theorem proving*. PhD thesis, Concordia University, 2015.

[55] Th Skolem. *Peano's Axioms and Models of Arithmetic*, volume 16. Elsevier, 1955.

[56] Mike Stannett and István Németi. Using isabelle/hol to verify first-order relativity theory. *Journal of Automated Reasoning*, 52(4):361–378, 2014.

[57] M Volmer. Thermodynamische folgerungen ans der zustandsgleichung für adsorbierte stoffe. *Zeitschrift für Physikalische Chemie*, 115(1):253–260, 1925.

[58] Markus M Wenzel. *Isabelle/Isar - A versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, 2002.

[59] George W Whitehead. *Elements of Homotopy Theory*, volume 61. Springer Science & Business Media, 2012.

[60] Stephen Wolfram. *Mathematica: A system for doing mathematics by computer*. Addison Wesley Longman Publishing Co., Inc., 1991.

5 Supplementary Information

5.1 Additional Background

Lean is an open source theorem prover developed by Microsoft Research and Carnegie Mellon University, based on dependent type theory, with the goal to formalize theorems in an expressive way [22]. Lean supports user interaction and constructs axiomatic proofs through user input, allowing it to bridge the gap between interactive and automated theorem proving. Like Mizar [51] and Isabelle [58], Lean allows user to state definitions and theorems but also combines more imperative tactic styles as in Coq [8], HOL-Light [28], Isabelle [45] and PVS [46] to construct proofs. The ability to define mathematical objects, rather then just postulate them is where Lean gets its power [6]. It can be used to create an interconnected system of mathematics where the relationship of objects from different fields can be easily shown without loosing generality.

Lean has a small kernel, based on dependent type theory, with just over 6000 lines of code that allows it to instantialize a version of the Calculus of Inductive Constructions (CoIC) [18, 19]. The strong normalizing characteristic of the CoIC [17] creates a robust programming language that is consistent. The CoIC creates a constructive foundation for mathematics allowing the entire field of mathematics to be built off of just 6000 lines of code.

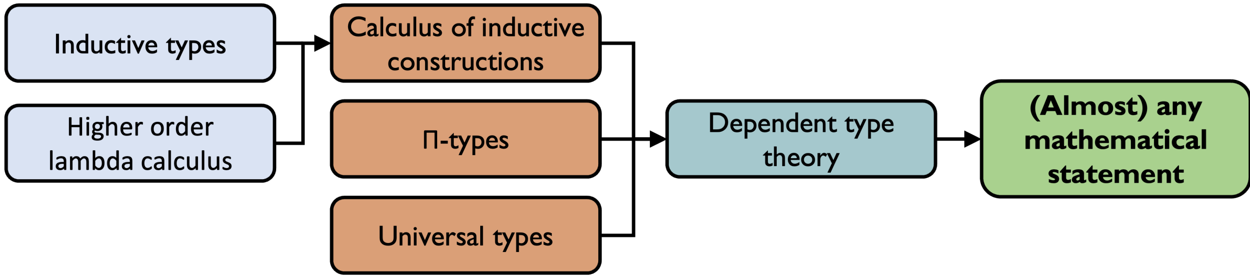

Figure 5: Overview of Lean Theorem Prover

As mentioned above, the power of Lean comes from the ability to define objects and prove properties about them. In Lean, there are three ways to define new Types: type universes, Pi types, and inductive types. The first two are used to construct the basis of dependent type theory, and are used for more theoretical, foundational stuff. Instead we will focus on the use of inductive types. Standard inductive types, known as just *inductive types*, are built from a set of constructors and well found *recursion*. Non-recursive inductive types that contain only one constructor are called *structures*.

Many mathematical objects in Lean can be constructed through inductive types, which is a type built from a set of constructors and proper recursion [23]. The natural numbers are an inductive type, defined using Peano's Encoding [55]. This requires two constructors, a constant element, 0 : nat, and a function called the successor function, S. Then one can be constructed as S(0), two can be constructed as S(S(0)), etc.

In Lean, the natural numbers are defined as:

```
inductive nat
| zero : nat
| succ (n : nat) : nat
```

Here, the type *nat* is defined through recursion by a constant element, zero, and a function. With this, the *def* command is used to define properties about the class, like addition or multiplication. For instance, addition of the natural numbers is defined as:

```
protected def add : nat → nat → nat
| a  zero     := a
| a  (succ b) := succ (add a b)
```

Addition is defined as a function that takes in two natural numbers and outputs a natural number. Since the natural numbers are created from two constructors, there are two cases of addition that must be shown. The first is a general natural number plus zero which yields the general natural number, and the next is a general natural number plus the successor of a general natural number. The second case used recursion and calls *add* again until it reduces to zero.

The other way to define types is using *structure*. More specifically, we will use *class*, which is an augmentation of *structure* [6]. *Class* allows us to add constraints to a type variable. For instance, the *class has_add* constrains a type to have a function called *add* which represents addition.

```
class has_add (α : Type u) :=
(add : α → α →  α)
```

This can be used for much more advanced ideas, like defining rings or abelian groups. We use class to define areas of science as new types with constraints to follow certain rules.

5.2 Additional Proofs

5.2.1 Langmuir Adsorption

The first Langmuir proof introduced earlier states every premise explicitly but however we can condense that by rewriting *hrad* and *hrd* into *hreaction* to yield $k_ad*P*S = k_d*A$ and we can then rewrite $h\theta$ and hK in the goal statement. While *hrad*, *hrd*, $h\theta$, and hK have scientific significance, they do not have any mathematical significance. In Lean it looks like:

```
theorem Langmuir_single_site2
(P k_ad k_d A S: ℝ)
(hreaction : k_ad*P*S = k_d*A)
(hS : S ≠ 0)
(hk_d : k_d ≠ 0)
: A/(S+A) = k_ad/k_d*P/(1+k_ad/k_d*P) :=
```

However, while those four variables do not have any mathematical significance, and only serve to hinder our proofs, they do have scientific significance, and we do not want to just omit them. Instead we can use the *let* command to create an in-line, local definition. This allows us to have the applicability of the theorem, while still having scientifically important variables. In Lean, this looks like:

```
theorem Langmuir_single_site
(P k_ad k_d A S : ℝ)
(hreaction : let r_ad := k_ad*P*S, r_d := k_d*A in r_ad = r_d)
(hS : S ≠ 0)
(hk_d : k_d ≠ 0)
:
let θ := A/(S+A),
    K := k_ad/k_d in
θ = K*P/(1+K*P) :=
```

The first line after the *theorem* statement, gives the variables use in the proof. Notice that r_{ad}, r_d, K_{eq}, and θ are not defined as variables. Instead, the *let* statement defines those four variables in their respective premise or goal. Then, in the proof we can simplify the *let* statement to get local definitions of those variables, just like *hrad*, *hrd*, *hθ*, and *hK*. While this version of proof follow the same proof logic minus the two initial rewrites from earlier version, however if we stick with the first proof, we will find it very difficult to use compared to using this proof above, because of all those hypothesises. Suppose we wanted to prove *langmuir_single_site2* and we already have proven *langmuir_single_site*. We would find it impossible to use *langmuir_single_site* because we are missing premises like *hrad* or *hrd*. Yet, we could prove the other way, ie. use *langmuir_single_site* to prove *langmuir_single_site2*. Having all of those extra premises that define the relation between variables only serves to hinder the applicability of our proofs.

5.2.2 BET Adsorption

We continue the derivation of Equation 27 from the paper that aims to redefine *x* as $x = P/P_0$, by recognizing that the volume should approach infinity at the saturation pressure, and, mathematically, it approaches infinity as x approaches one from the left. For x to approach one, pressure must approach $1/C_L$. First, we show that Equation 26 from the paper approaches infinity as *P* approaches $1/C_L$. We specifically require it to approach from the left because volume approaches negative infinity if we come from the right. In Lean, this looks like ⚛:

```
lemma BET.tendsto_at_top_at_inv_CL
: filter.tendsto brunauer_26
(nhds_within (1/C_L) (set.Ioo 0 (1/C_L)))
filter.at_top:=
```

The function *filter.tendsto* is the generic definition of the limit. It has three inputs, the function, what the independent variable approaches, and what the function approaches, in that order. We split this into three lines to better visualize what is happening. First, we are using the object *brunauer_26*, which is the BET equation as a function of pressure in terms of x. Next, *(nhds_within (1/C_L) (set.Ioo 0 (1/C_L)))* is how we say approaches $1/C_L$ from the left. *nhds_within* means the intersection of a neighborhood, abbreviated as *nhds*, and a set. A neighborhood of a point is the open set around that point. *set.Ioo* designates a left-open right-open interval. Here we have the interval $(0, 1/C_L)$. The intersections of the neighborhood and this set constrains us to approach the neighborhood from the left. The final part is *filter.at_top* which is a generalization of infinity, and just says our function approaches infinity.

In the original derivation done by Brunauer et al, they wish to show that $P_0 = 1/C_L$ because as pressure approaches each of these values, volume approaches infinity, these two values are equal. It should be noted that this idea is only true if *C*, the BET constant, is greater than or equal to one. If not, the function has two points where it hits infinity in the positive pressure region. We also have problems showing the congruence of such a fact in Lean, since such a relation has yet to be formalized and the congruence of two *nhds_within* has not been shown. For now, we use the lemma above to prove a simplier version of the theorem where we assume $P_0 = 1/C_L$, and show that with this assumption, V approaches infinity. In Lean, this looks like ⚛:

```
theorem brunauer_27
(h1 : P_0 = 1/C_L)
: filter.tendsto brunauer_26  (nhds_within (P_0) (set.Ioo 0 (P_0))) filter.at_top:=
```

The proof of this theorem involves rewriting h1, and then applying the lemma proved above. While we would prefer to prove that $P_0 = 1/C_L$, this proof will serve as a placeholder, until Mathlib builds out more math related to the congruence of this subject. This theorem does not use a lcoal definition, like Langmuir, because P_0 is already defined as a variable using *constant*.

Finally, we formalize the derivation of Equation 28 from the paper, givne by Equation 23.

$$\frac{V}{A * V_0} = \frac{CP}{(P_O - P)(1 + (C-1)(P/P_0)} \tag{23}$$

Just like Equation 3, we first define Equation 23 at an object then formalize the derivation of this object. In Lean, the object looks like ⚛:

```
def brunauer_28 := λ P : R, C*P/((P_0-P)*(1+(C-1)*(P/P_0)))
```

Now we can prove a theorem that formalizes the derivation of this object ⚛:

```
theorem brunauer_28_from_seq
{P V_0: ℝ}
(h27 : P_0 = 1/C_L)
(hx1: (x P) < 1)
(hx2 : 0 <  (x P))
: let Vads :=  V_0 * ∑' (k : ℕ), ↑k * (seq P k),
      A :=  ∑' (k : ℕ), (seq P k) in
  Vads/A = V_0*(brunauer_28 P) :=
```

Rather then explicitly solving the sequence ratio, like we did for Equation 3, we can now use the theorem that derived Equation 3 to solve the left hand side of our new goal. We then have a goal where we show that Equation 23 is just a rearranged version of Equation 3, which is done through algebraic manipulation.

5.2.3 The antiderivative in Lean

For a function, f, the antiderivative of that function, given by F, is a differentiable function, such that the derivative of F is the original function f. In Lean, we formalize the general antiderivative and show how it can be used for several specific applications, including the antiderivative of a constant, of a natural power, and of an integer power. We generalize our functions as a function from a general field onto a vector field, $f : \mathbb{K} \to E$. This allows us to apply the theorems to any parametric vector function, including scalar functions.

Our goal is to show, from the assumption that $f(t)$ is the derivative of $F(t)$ and $f(t)$ is the derivative of $G(t)$, then we have an equation $F(t) = G(t) + F(0)$, which is the antiderivative of $f(t)$. $G(t)$ is the variable portion of the equation. For example, if the antiderivative is of the form $F(t) = t^3 + t + 6$, then $G(t) = t^3 + t$ and $F(0) = 6$. $F(0)$ is the constant of integration, but written in a more explicit relation to the function. Since $G(t)$ is the function of just variables, we have as another premise $G(0) = 0$.

The first goal is to show that a linearized version of the antiderivative function holds. We can rewrite $F(t)$ so that is linear by moving $G(t)$ to the left hand side, leaving us with an equation that equals a constant.

$$F(t) - G(t) = C \tag{24}$$

Thus, we can relate any two points along this function, $\forall xy, F(x) - G(x) = F(y) - G(y)$. To show this holds, we recognize that if Equation 24 is constant, then the derivative of this function is equal to zero.

$$\frac{d}{dt}(F(t) - G(t)) = 0 \tag{25}$$

Next, we apply the linearity of differentiation to Equation 25 to get a new form: $\frac{d}{dt}F(t) - \frac{d}{dt}G(t) = 0$, and rearrange to get:

$$\frac{d}{dt}F(t) = \frac{d}{dt}G(t) \tag{26}$$

From the first premise, we assumed that $f(t)$ is the derivative of $F(t)$. From our second premise, we assumed that $f(t)$ is also the derivative of $G(t)$. Thus, applying both premises, we can simplify Equation 26 to:

$$f(t) = f(t) \tag{27}$$

which we recognize to be correct.

Now that we have a new premise to use, given by Equation 28, we can specialize this function to get our final form.

$$\forall xy, F(x) - G(x) = F(y) - G(y) \tag{28}$$

We specialize the universals by supplying two old names. For x, we use t (the variable we have been basing our differentiation around), and for y we use 0. Thus, Equation 28 becomes:

$$F(t) - G(t) = F(0) - G(0) \tag{29}$$

Our third premise was that $G(0) = 0$, so we can simplify and rearrange Equation 28, to get our final form:

$$F(t) = G(t) + F(0) \tag{30}$$

Which satisfies the goal we laid out in the beginning. In Lean, the statement of this theorem looks like ⚛:

```
theorem  antideriv
{E : Type u_2}  {𝕂: Type u_3} [is_R_or_C 𝕂] [normed_add_comm_group E]
[normed_space 𝕂 E]
{f F G: 𝕂 → E} (hf : ∀ t, has_deriv_at F (f t) t)
(hg : ∀ t, has_deriv_at G (f t) t)
(hg' : G 0 = 0)
: F = λ t, G t + F(0) :=
```

Applying the *antideriv* theorem to examples is very straight forward. We will show an example by deriving the antiderivative of a constant function. In Lean, we would state this as :

```
theorem antideriv_const
(F : 𝕂 → E) k : E
(hf : ∀ t, has_deriv_at F k t):
(F =  (x : 𝕂)), x·k + F 0) :=
```

Here we say that the derivative of $F(x)$ is the constant k, and want to show that $F(x) = x \cdot k + F(0)$, where the "·" operator stands for scalar multiplication. To use the *antideriv* theorem, we must show that its premises follow, meaning we must show:

```
∀ t, has_deriv_at F k t
∀ t, has_deriv_at x·k k t
0·k = 0
```

The first goal is explicitly given in our premises, *hf*. The next goal can be derived by taking out the constant, and showing that the function x has a derivative equal to 1. The final goal can be easily proven by recognizing zero multiplied by anything is zero. Thus, we have formalized antiderivative of a constant function, and can use this same process for any other function. The antiderivative is especially important for deriving the kinematic equations, as seen in the next section.