Formal Foundations for Informal Mathematics Research

William DeMeo

Abstract. This document describes a research program, the primary goal of which is to develop and implement new theory and software libraries to support computer-aided mathematical proof in universal algebra and related fields. A distinguishing feature of this effort is the high priority placed on *usability* of the formal libraries produced. We aim to codify the core definitions and theorems of our area of expertise in a language that feels natural to working mathematicians with no special training in computer science. Thus our goal is a formal mathematical library that has the look and feel of the informal language in which most mathematicians are accustomed to working.

This research is part of a broader effort currently underway in a number of countries, carried out by disparate research groups with a common goal—to develop the next generation of **practical formal foundations for informal mathematics**, and to codify these foundations in a formal language that feels natural to mathematicians. In short, our goal is to present *mathematics as it should be.*

"Systems of axioms acquire a certain sanctity with age, and in the *how* of churning out theorems we forget *why* we were studying these conditions in the first place... Even when the mathematical context and formal language are clear, we should not perpetuate old proofs but instead look for new and more perspicuous ones."

–Paul Taylor
in *Practical Foundations of Mathematics*

Date: 2018-12-16.

1. Introduction and Motivation

A significant portion of the professional mathematician's time is typically occupied by tasks other than *Deep Research.* By Deep Research we mean such activities as discovering truly non-trivial, publishable results, inventing novel proof techniques, or conceiving new research areas or programs. Indeed, consider how much time we spend looking for and fixing minor flaws in an argument, handling straightforward special cases of a long proof, or performing clever little derivations which, if we're honest, reduce to mere exercises that a capable graduate student could probably solve. Add to this the time spent checking proofs when collaborating with others or reviewing journal submissions and it's safe to say that the time most of us spend on Deep Research is fairly limited.

It may come as a surprise, then, that computer-aided theorem proving technology, which is capable of managing much of the straight-forward and less interesting aspects of our work, has not found its way into mainstream mathematics. The reasons for this are simple to state, but difficult to resolve. For most mathematicians, the potential benefit of the currently available tools is outweighed by the time and patience required to learn how to put them to effective use. The high upfront cost is due to the fact that most researchers carry out their work in a very *efficient informal dialect* of mathematics—a common dialect that we share with our collaborators, and without which proving and communicating new mathematical results is difficult, if not impossible.

One hopes that published mathematical results *could* be translated into the rigorous language of some system of logic and formally verified. Nonetheless, few would relish spending the substantial time and energy that such an exercise would require. A mathematician's job is to discover new theorems and to present them in a language that is rigorous enough to convince colleagues, yet informal enough to be efficient for developing and communicating new mathematics. Such a language is what we refer to as the *Informal Language* of mathematics research. A relatively recent trend is challenging this status quo, however, and the number of mathematicians engaging in computer-aided mathematics research is on the rise [34, 36, 48]. Indeed, at the *Computer-aided Mathematical Proof* workshop of the *Big Proof Programme*, held in 2017 at the Isaac Newton Institute of Cambridge University, we witnessed the serious interest that leading mathematicians (including two Fields Medalists) showed in computer-aided theorem proving technology [43].

To verify mathematical arguments by computer, the arguments must be be written in a language that allows machines to interpret and check them. We refer to the practice of writing such proofs as *interactive theorem proving*, and to the programming languages and software systems that check such proofs as *proof assistants* or *interactive theorem provers.* Such systems are increasingly used in academia and industry to verify the correctness of hardware, software, and protocols. In fact, *constructive type theory* and *higher-order logics*, on which most modern proof assistants are based, have played significant roles in formalizing and mechanically confirming the solutions to important problems, such as the *Four-Color Theorem* [32] and the *Feit-Thompson Odd Order Theorem* [33], as well as settling major open problems, such as the *Kepler Conjecture* [35].

Another kind of computer-based theorem proving tool is called an *automated theorem prover*, which is quite different from a proof assistant in that the former is designed to independently search for a proof of a given statement with little or no help from the user. (Contrast this with the *interactive* nature of a proof assistant.) Unfortunately, when a proof is found by an automated theorem prover, it tends to be very difficult to read and understand, and it often seems impossible to translate an automatically generated proof into an Informal Language proof.

Further justification for the use of proof assistants is their potential for improving the referee and validation process. The main issues here are human fallibility and the high opportunity cost of human talent. Indeed, a substantial amount of time and effort of talented individuals is expended on refereeing journal submissions. Despite this the typical review process concludes without supplying any guarantee of validity of the resulting publications [28]. Moreover, the emergence of large computational proofs (e.g., Hales' proof of the Kepler Conjecture) lead to referee assignments that are impossible burdens on individual reviewers [39]. Worse than that, even when such work survives peer review, there remain disputes over correctness, completeness, and whether nontrivial gaps exist. Some recent examples include Zhuk's proof of the *CSP dichotomy conjecture* of Feder and Vardi [58], and Fasel's proposed solution to *Murthy's conjecture*.[1] By relying to a greater extent on computers to develop and check proofs, researchers can raise reviewing standards, even for computational proofs, thereby increasing confidence in the validity of new results.

As further justification for enlisting the help of computers for discovering and checking new mathematics, consider the space of all proofs comprehensible by the unassisted human mind, and then observe that this is but a small fraction of the collection of all potential mathematical proofs in the universe. The real consequences of this fact are becoming more apparent as mathematical discoveries reach the limits of our ability to confirm and publish them in a timely and cost effective way. Thus, it seems inevitable that proof assistants will have an increasingly important role to play in future mathematics research [36].

Beyond their importance as a means of establishing trust in mathematical results, formal proofs can also expose and clarify difficult steps in an argument. Even before one develops a formal proof, the mere act of expressing a theorem statement (including the foundational axioms, definitions, and hypotheses on which it depends) in a precise and (when possible) computable way almost always leads to a deeper understanding of the result. Moreover, by keeping track of changes across a collection of results (axioms, hypotheses, etc), proof assistants facilitate experiments with variations and generalizations. When changing a definition, a mathematician equipped with a proof assistant is alerted to the inferences that need repair, redundant definitions, unnecessary hypotheses, etc. [16].

Finally, modern proof assistants support automated proof search and this can be used to discover long sequences of first-order deduction steps. Consequently, mathematicians can spend less time carrying out the parts of an argument that are more-or-less obvious, and more time contemplating deeper questions. Indeed, Fields Medalist Sir Timothy Gowers expects collaboration with a "semi-intelligent database" to "take a great deal of the drudgery out of research." [34].

1.1. **The usability gap.**

Despite the many advantages and the noteworthy success stories mentioned above, proof assistants remain relatively obscure. There are a number of obvious reasons for this. First and foremost, proof assistant software tends to be tedious to use. Most mathematicians experience a significant slow down in progress when they must not only formalize every aspect of their arguments, but also express such formalizations in a language that the software is able to parse and comprehend.

[1] Fasel's solution survived peer review and was published in the *Annals of Mathematics* before it emerged that a lemma was flawed and the entire proof collapsed [28].

One question that leads to insight into this "usability gap" that plagues most modern proof assistants is why *computer algebra systems* do not suffer from the same problem. Simply put, computer algebra systems are more popular than interactive theorem provers. One reason is that the up-front cost to end users seems substantially higher for interactive theorem provers than for computer algebra systems, and this is because the latter are typically conceived of by mathematicians whose primary aim is to build a system that presents things "as they should be," that is, as a mathematically educated user would expect.

In many proof assistants and automated theorem provers, the mathematics are often not presented as we would expect or like them to be. In [50], Pollet and Kerber argue that this is not just a deficiency of the user interface. The problem with theorem provers goes much deeper; it goes to the core of these systems, namely to *the formal representation of mathematical concepts and knowledge.* How easy or hard it is to codify theorems and translate informal mathematical arguments into formal proofs in a particular system depends crucially on the formal foundations of that system, and the way in which these foundations are represented in the system.

1.2. **Closing the usability gap.** Our research program addresses the major roadblocks to wide-spread adoption of proof assistants. We describe some of these in the following list.

- *Incomplete mathematical libraries.* Formal libraries—consisting of definitions, lemmas, and proofs—are a prerequisite for most formal developments. Before users can apply proof assistants to their own research, they need to formalize a lot of undergraduate- and graduate-level mathematics. Even the largest formal libraries currently available (e.g., Mizar [45, 31]) cover only a small fraction of mathematics.
- *Weak automation of mathematics.* Proof assistants typically combine general-purpose logical automation and procedures for arithmetic. However, without *domain-specific automation* (DSA), a single sentence in a proof in the Informal Language can correspond to dozens or hundreds of lines in the formal language.
- *Poor interoperability with computer algebra systems.* Notwithstanding the existence of a few prototypes [15, 38], interoperability between computer algebra systems and proof assistants is an open problem. A trustworthy integration of algebra systems in proof assistants requires the validation of results produced by the algebra systems. However, most procedures do not generate proof certificates; for some algorithms, it is not even clear what certificates would look like [24].
- *Steep automation learning curve.* Extending a proof assistant to support *domain-specific automation* requires a high level of expertise. The programmatic interfaces of most proof assistants have evolved over several decades and are difficult to learn. For example, to extend the Coq proof assistant [21], one must learn both the Ltac tactic language and the low-level OCaml interfaces, in addition to the Gallina specification language.

Despite the many obstacles, there is a strong feeling in various parts of the research community that mathematics deserves to be formalized. Fields medalist Vladimir Voevodsky was one of the strongest advocates of this view. His research area was plagued by flawed theorems, and eventually he stopped believing paper-and-pencil proofs [51].

The overriding goal of our project is to re-examine and formalize the foundations of mathematics, with a particular focus on our primary areas of expertise, universal algebra, to do so in a *practical* and *computable* way, and to codify these foundations and advance the state-of-the-art in computer-aided theorem proving technology. The goal will be achieved when the software becomes a natural, if not necessary, part of the working mathematician's toolbox. We envision a future in which we can hardly imagine proving new theorems, completing referee assignments, or communicating and disseminating new mathematics without the support of a proof assistant.

2. The Lean Theorem Prover and its Role in the Project

Given our motivation, the choice of proof assistant was easy; we chose the *Lean Programming Language* [2]—developed by Leonardo de Moura (Microsoft Research), Jeremy Avigad (Carnegie Mellon), and their colleagues—for a number of reasons. First, Lean is designed and developed by logicians and computer scientists working together to create a language and syntax that presents things *as they should be*, so that the working in the language feels almost as natural as working in the Informal Language. Thus it is reasonable to expect mathematicians, even those lacking special training in computer science, to adopt the system and use the libraries we develop.

Other considerations that make Lean ideally suited to this project are the following:

- **Clean and efficient design.** Lean's design and engineering is unusually clean and efficient, as the Lean developers have combined the best features from existing proof assistants (e.g., Coq, Isabelle/HOL, and Z3), drawing on decades of experience in interactive and automatic theorem provers.
- **Powerful and extensible proof automation support.** Lean's logic is very expressive, with emphasis placed on *powerful proof automation.* In fact, the proof automation system is easy to extend via *metaprograms that one can implement right in the Lean language itself!* In this way, Lean aims to *bridge the gap between interactive and automated theorem proving.*
- **Easy-to-read proofs.** Lean is unique among computer-based theorem proving tools in that its *proofs tend to be easy to read and understood*, without any special training. In fact, working in Lean often leads to formal proofs that are cleaner, more concise, and shorter than the corresponding proofs in the Informal Language. (We provide examples in Section 4 below.) This is a crucial feature if we expect the system to be adopted by mathematicians.
- **A rich type system supporting type classes.** For the design of basic algebraic libraries, Lean's developers turned to Isabelle/HOL for inspiration. The libraries rely on *type classes*, a mechanism to categorize types and their operations (e.g., "$\langle \mathbb{Z}, 0, -, + \rangle$ forms a group"). Type classes interact well with Lean's dependent types. By contrast, in Isabelle/HOL, there is no way to use type classes to reason about the integers modulo n as a ring, as observed in [4, Section 3.1].
- **A rich type system supporting dependent types and quotient types.** Lean's logical foundation is a variant of Coq's—a dependent type theory called the *Calculus of Inductive Constructions* (CiC) [22]. However, Lean has a small inference kernel and strong automation; independent proof checkers provide additional guarantees. Lean's support for dependent types is smoother than Coq's, thanks to flexible pattern matching and a generalized congruence closure algorithm [54]. Finally, Lean supports *quotient types* and a *transfer tool* to facilitate reasoning up to isomorphism without resorting to setoids [6] or homotopy type theory [5].

2.1. **Domain specific automation.** To support the formalization of theorems, we will develop libraries that contain formal statements and proofs of all of the core definitions and theorems of universal algebra. We will automate proof search in the specific domain of universal algebra, and develop tools to help find and formalize theorems emerging from our own mathematics research. We are currently involved in four research projects in universal algebra [14, 24, 25, 26], and an invaluable tool for our work would be a proof assistant with rich libraries for general algebra, equipped with *domain-specific proof tactics for automatically invoking the standard mathematical idioms from our field.* The latter is called *domain-specific automation* (DSA), and one of our primary goals is to demonstrate the utility of DSA for proving new theorems.

As Lean is a very young language, its domain-specific libraries are relatively small, but they are growing rapidly. *It is vital for mathematicians to get involved at this early stage and play a leading role in the development.* If we leave this entirely to our colleagues in computer science, they will base the development on their perception of our needs, history will likely repeat itself, and the resulting libraries and tools may fail meet the needs and expectations of working mathematicians.

3. Universal Algebra and its Role in the Project

Universal (or *general*) *algebra* has been invigorated in recent years by a small but growing community of researchers exploring the connections between algebra and computer science. Some of these connections were discovered only recently and were quite unexpected. Indeed, algebraic theories developed over the last 30 years have found a number of important applications in both of the two main branches of theoretical computer science—*Theory A*, comprising *algorithms* and *computational complexity*, and *Theory B*, comprising *domain theory*, *semantics*, and *type theory* (the theory of programming languages).[2] The present proposal falls within the scope of Theory B.

3.1. **Foundations of mathematics and computing science.** Universal algebra, lattice theory, and category theory have had deep and lasting impacts on theoretical computer science, particularly in the subfields of domain theory, denotational semantics, and programming languages research [42, 53]. Dually, progress in theoretical computer science has informed and inspired a substantial amount of pure mathematics in the last half-century [7, 8, 10, 11, 49, 53], just as physics and physical intuition motivated so many of the mathematical discoveries of the last two centuries.

Functional programming languages that support *dependent* and *(co)inductive types* have brought about new opportunities to apply abstract concepts from universal algebra and category theory to the practice of programming, to yield code that is more modular, reusable, and safer, and to express ideas that would be difficult or impossible to express in *imperative* or *procedural programming languages* [9, 40, 20, Chs. 5 & 10]. The *Lean Programming Language* [2] is one example of a functional language that supports dependent and (co)inductive types, and it is the language in which we will carry out our practical foundations program.

In the remainder of this project description, we give some background on interactive theorem proving technology, introduce dependent and inductive types, and describe the Lean language, which will be the main vehicle for this project. We will explain how these

[2] The Theory A–Theory B dichotomy was established by "The Handbook of Theoretical Computer Science" [56, 57], Volume A of which includes chapters on algorithms and complexity theory; Volume B covers domain theory, semantics, and type theory.

technologies can be used to advance pure mathematics in general, and universal algebra in particular. We then present the concrete goals of the project, with some discussion of how we intend to accomplish them, and some examples of the achievements we have already made in pursuit of these goals. Before proceeding, let us summarize in broad terms, using nontechnical language, the main objectives of the project. We intend to

(1) present the core of universal algebra using *practical logical foundations*; in particular, definitions, theorems and proofs shall be constructive and have computational meaning, whenever possible;
(2) develop software that extends the *Lean Mathematical Components Library* [3] to include the output of (1), implementing the core results of our field as *types* and their proofs as *programs* (or *proof objects*) in the *Lean Programming Language* [2, 27].
(3) develop *domain-specific automation (DSA)* tools that make it easier for working mathematicians harness the power of modern proof assistant technology;
(4) teach mathematicians how to use the assets developed in items (1)–(3) to do the following:
    a. translate existing or proposed Informal Language proofs (typeset in LaTeX, say) into Lean so they can be formally verified and tagged with a certificate of correctness;
    b. construct and formally verify proofs of new theorems using Lean;
    c. import (into Lean) software packages and algorithms used by algebraists (e.g. UACalc or GAP) so that these tools can be certified and subsequently invoked when constructing formal proofs of new results.

4. Proof of Concept: Lean Universal Algebra

This section demonstrates the utility of dependent and inductive types by expressing some fundamental concepts of universal algebra in Lean. In particular, we will formally represent each of the following: *operation*, *algebra*, *subuniverse*, and *term algebra*. Our formal representations of these concepts will be clear, concise, and computable, and we will develop a notation and syntax that should seem natural and self-explanatory to algebraists. Our goal here is to demonstrate the power of Lean's type system for expressing mathematical concepts precisely and constructively, and to show that if we make careful design choices at the start of our development, then our formal theorems *and their proofs* can approximate the efficiency and readability of their Informal Language analogs.

4.1. Operations and Algebras.

We use ω to denote (our semantic concept of) the natural numbers. The symbols $\mathbb{N}$ and `nat` are synonymous, both denoting the *type of natural numbers*, as implemented in Lean. A *signature* $S = (F, \rho)$ consists of a set F of *operation symbols*, along with a *similarity type* function $\rho \colon F \to N$. The value $\rho f \in N$ is called the *arity* of f. In classical universal algebra we typically assume $N = \omega$, but for most of the basic theory this choice is inconsequential and, as we will see when implementing general operations in Lean, it is unnecessary to commit in advance to a specific *arity type*.[3]

Classical universal algebra is the study of *varieties* (or *equational classes*) of algebraic structures where an *algebraic structure* is denoted by $\mathbf{A} = \langle A, F^{\mathbf{A}} \rangle$ and consists of a set A, called the *carrier* of the algebra, along with a set $F^{\mathbf{A}}$ of operations defined on A, one for

[3] An exception is the *quotient algebra type* since, unless we restrict ourselves to finitary operations, lifting a basic operation to a quotient requires some form of choice.

each operation symbol; that is,

$$F^{\mathbf{A}} = \{f^{\mathbf{A}} \mid f \in F \text{ and } f^{\mathbf{A}} \colon (\rho f \to A) \to A\}.$$

Some of the renewed interest in universal algebra has focused on representations of algebras in categories other than **Set**, multisorted algebras, and higher type universal algebra [1, 12, 29, 30, 46]. These are natural generalizations that we plan to integrate into future versions of our `lean-ualib` software library, once we have a working implementation of the core of classical (single-sorted, set-based) universal algebra.

Suppose A is a set and f is a ρf-ary operation on A. In this case, we often write $f \colon A^{\rho f} \to A$. If N happens to be $\mathbb{N}$, then ρf denotes the set $\{0, 1, \dots, \rho f - 1\}$ and a function $g \colon \rho f \to A$, identified with its graph, is simply a ρf-tuple of elements from A. The domain $A^{\rho f}$ can be represented by the type $\rho f \to A$ of functions from ρf to A, and we will represent operations $f \colon A^{\rho f} \to A$ using the function type $(\rho f \to A) \to A$.

Fix $m \in \mathbb{N}$. An m-tuple, $a = (a_0, a_1, \dots, a_{m-1}) \in A^m$ is (the graph of) the function $a \colon m \to A$, defined for each $i < m$ by $a\,i = a_i$. Therefore, if $h \colon A \to B$, then $h \circ a \colon m \to B$ is the tuple whose value at i is $(h \circ a)\,i = h\,a\,i = h\,a_i$, which has type B. On the other hand, if $g \colon A^m \to A$, then $g\,a$ has type A. If $f \colon (\rho f \to B) \to B$ is a ρf-ary operation on B, if $a \colon \rho f \to A$ is a ρf-tuple on A, and if $h \colon A \to B$, then $h \circ a \colon \rho f \to B$, so $f(h \circ a) \colon B$.

4.2. Operations and Algebras in Lean (`lean-ualib/basic.lean`).

This section presents our implementation of operations and algebras in Lean, highlighting the similarity between the formal and informal rendering of these concepts. We start with the *type of operation symbols* and the *type of signatures*.

```
import data.set
definition op (β α) := (β → α) → α
```

An example of an operation of type `op (β α)` is the projection function π, of arity β on the *carrier type* α, which we define in Lean as follows:

```
definition π {β α} (i) : op β α := λ a, a i
```

The operation `π i` maps a given tuple `a : β → α` to its value at input `i`. For instance, suppose we have types α and β of arbitrary *height*,[4] and variables `i : β` and `f : β → α`.

```
variables (α : Type*) (β : Type*) (i : β) (f : β → α)
```

Then the command `#check π i f` shows that the type of `π i f` is α, as expected, since `π i f = f i`.

We define a signature as a structure with two fields, the type `F` of *operation symbols* and an *arity function* `ρ : F → Type*`, which takes each operation symbol f to its arity ρf.

```
structure signature := mk :: (F : Type*) (ρ : F → Type*)
```

Next we define the *type of interpretations of operations* on the carrier type α. First, let us fix a signature `S` and define some convenient notation.[5]

[4] The *height* of a type is simply type's *level* (see Section D.1), and the syntax `Type*` indicates that we do not wish to commit in advance to a specific height.

[5] The `section` command allows us to open a section throughout which our signature `S` will be available. The `section` ends when the keyword `end` appears below.

```
section
  parameter {S : signature}
  definition F := S.F
  definition ρ := S.ρ
  definition algebra_on (α : Type*) := Π (f : F), op (ρ f) α
  -- (section continued at * below)
```

The first definition allows us to write `f : F` (instead of `f : S.F`) to indicate that the operation symbol `f` inhabits `F`; similarly, the second definition allows us to denote the arity of `f` by `ρ f` (instead of `S.ρ f`). In these two cases, the Lean syntax matches our Informal Language notation exactly.

The definition of `algebra_on` makes sense; if we are given a signature `S` and a carrier type α, then an `S`-algebra over α is determined by its operations on α.[6] An inhabitant of the type `algebra_on` assigns an interpretation to each `op` symbol `f : F`, which yields a function of type `(ρ f → α) → α`.

Finally, we define an algebra. Since an algebra pairs a carrier with an interpretation of the operation symbols, we use the *dependent pair type*, `Σ (x : A), B x`, also known as a *Sigma type*. This is the type of ordered pairs `<a, b>`, where `a : A`, and `b` has type `B a` which may depend on `a`. Just as the *Pi type* `Π (x : A), B x` generalizes the notion of function type `A → B` by allowing the codomain `B x` to depend on the value `x` of the input argument, a Sigma type `Σ (x : A), B x` generalizes the cartesian product `A × B` by allowing the type `B x` of the second argument of the ordered pair to depend on the value `x` of the first.[7]

Since an algebra $\langle A, F^{\mathbf{A}}\rangle$ is an ordered pair where the type of the second argument depends on the first, it is natural to encode an algebra in type theory using a Sigma type, and we do so in the `lean-ualib` library as follows:

```
  -- (section continued from * above)
  definition algebra := sigma algebra_on
  instance alg_carrier : has_coe_to_sort algebra := ⟨_, sigma.fst⟩
  instance alg_operations : has_coe_to_fun algebra := ⟨_, sigma.snd⟩
end
```

The last two lines are tagged with `has_coe_to_sort` and `has_coe_to_fun`, respectively, because here we are using a very nice feature of Lean called *coercions*. Using this feature we can write programs using syntax that looks very similar to our Informal Language. For instance, the standard notation for the interpretation of the operation symbol f in the algebra $\mathbf{A} = \langle A, F^{\mathbf{A}}\rangle$ is $f^{\mathbf{A}}$. In our implementation, the interpretation of f is denoted by `A f`. While `A f` is not identical to the Informal Language's $f^{\mathbf{A}}$, it is arguably just as elegant, and we believe that adapting to it will not be a great burden on the user. To see this notation in action, let us look at how the `lean-ualib` represents the assertion that a function is an `S`-homomorphism.

[6] plus whatever equational laws it may models; our handling of *theories* and *models* in Lean is beyond the scope of this project description; for more information, see https://github.com/UniversalAlgebra/lean-ualib/

[7] Lean's built-in `sigma` type is defined as follows:

```
structure sigma α : Type u (β : α → Type v) := mk :: (fst : α) (snd : β fst)
```

```
definition homomorphic {S : signature}
{A : algebra S} {B : algebra S} (h : A → B) :=
∀ f a, h (A f a) = B f (h ∘ a)
```

Comparing this with a common Informal Language definition of a homomorphism, which is typically something similar to $\forall f\ \forall a\ h(f^{\mathbf{A}}(a)) = f^{\mathbf{B}}(h \circ a)$, we expect working algebraists to find the `lean-ualib` syntax quite satisfactory.

4.3. **Subuniverses.** In this section, we describe another important concept in universal algebra, the *subuniverse*, and use it to illustrate one of the underlying themes that motivates this research project. Indeed, subuniverses give us our first opportunity to demonstrations the power of *inductively defined types*. Such types are essential for working with infinite objects in a constructive and computable way and for proving (by induction of course!) properties of these objects.

A *subuniverse* of an algebra $\mathbf{A} = \langle A, F^{\mathbf{A}}\rangle$ is a subset $B \subseteq A$ that is closed under the operations in $F^{\mathbf{A}}$. We denote by $\mathsf{S}\mathbf{A}$ the set of all subuniverses of $\mathbf{A}$. If B is a subuniverse of $\mathbf{A}$ and $F^{\mathbf{A}\restriction B} = \{f^{\mathbf{A}} \restriction B \mid f \in F\}$ is the set of basic operations of $\mathbf{A}$ restricted to B, then $\mathbf{B} = \langle B, F^{\mathbf{A}\restriction B}\rangle$ is a *subalgebra* of $\mathbf{A}$. Conversely, all subalgebras are of this form.

If $\mathbf{A}$ is an algebra and $X \subseteq A$ a subset of the universe of $\mathbf{A}$, then the *subuniverse of* $\mathbf{A}$ *generated by* X is defined as follows:

$$\mathrm{Sg}^{\mathbf{A}}(X) = \bigcap\{U \in \mathsf{S}\mathbf{A} \mid X \subseteq U\}. \tag{4.1}$$

To give another exhibition of the efficiency and ease with which we can formalize basic but important mathematical concepts in Lean, we now present a fundamental theorem about subalgebra generation, first in the Informal Language, and then formally in Section 4.4. Notice that the added complexity of the Lean implementation of this theorem is not significant, and the proof seems quite readable (especially when compared to similar proofs in Coq).

Theorem 4.1 ([13, Thm. 1.14])**.** *Let $\mathbf{A} = \langle A, F^{\mathbf{A}}\rangle$ be an algebra in the signature $S = (F, \rho)$ and let $X \subseteq A$. Define, by recursion on n, the sets X_n as follows:*

$$X_0 = X; \tag{4.2}$$

$$X_{n+1} = X_n \cup \{f\,a \mid f \in F,\ a \in X_n^{\rho f}\}. \tag{4.3}$$

Then $\mathrm{Sg}^{\mathbf{A}}(X) = \bigcup X_n$.

Proof. Let $Y = \bigcup_{n<\omega} X_n$. Clearly $X_n \subseteq Y \subseteq A$, for every $n < \omega$. In particular $X = X_0 \subseteq Y$. Let us show that Y is a subuniverse of $\mathbf{A}$. Let f be a basic k-ary operation and $a \in Y^k$. From the construction of Y, there is an $n < \omega$ such that $\forall i,\ a\,i \in X_n$. From its definition, $f\,a \in X_{n+1} \subseteq Y$. Thus Y is a subuniverse of $\mathbf{A}$ containing X. By (4.1), $\mathrm{Sg}^{\mathbf{A}}(X) \subseteq Y$. For the opposite inclusion, it is enough to check, by induction on n, that $X_n \subseteq \mathrm{Sg}^{\mathbf{A}}(X)$. Well, $X_0 = X \subseteq \mathrm{Sg}^{\mathbf{A}}(X)$ from its definition. Assume that $X_n \subseteq \mathrm{Sg}^{\mathbf{A}}(X)$. If $b \in X_{n+1} - X_n$, then $b = f\,a$ for a basic k-ary operation f and $a \in X_n^k$. But $\forall i,\ a\,i \in \mathrm{Sg}^{\mathbf{A}}(X)$ and since this latter object is a subuniverse, $b \in \mathrm{Sg}^{\mathbf{A}}(X)$ as well. □

4.4. **Subuniverses in Lean (`lean-ualib/subuniverse.lean`).**

The argument in the proof of Theorem 4.1 is of a type that one encounters frequently throughout algebra. It has two parts. First that Y is a subuniverse of $\mathbf{A}$ that contains X. Second that any subuniverse containing X must also contain Y.

We now show how the subalgebra concept and the foregoing argument is formally implemented in Lean.

```
import basic
import data.set
namespace subuniverse
  section
    open set
    parameter {α : Type*}       -- the carrier type
    parameter {S : signature}
    parameter (A : algebra_on S α)
    parameter {I : Type}        -- a collection of indices
    parameter {R : I → set α}   -- an indexed set of sets of type α
    definition F := S.F         -- the type of operation symbols
    definition ρ := S.ρ         -- the opereation arity function

    -- Definition of subuniverse
    definition Sub (β : set α) : Prop :=
    ∀ (f : F) (a : ρ f → α), (∀ x, a x ∈ β) → A f a ∈ β

    -- Subuniverse generated by X
    definition Sg (X : set α) : set α := ⋂₀ {U | Sub U ∧ X ⊆ U}
```

Lean syntax for the intersection operation on collections of *sets* is $\bigcap_0$.

Next we need "introduction" and "elimination" rules for arbitrary intersections, plus the useful fact that the intersection of subuniverses is a subuniverse.

```
    -- Intersection introduction rule
    theorem Inter.intro {s : I → set α} :
    ∀ x, (∀ i, x ∈ s i) → (x ∈ ⋂ i, s i) :=
    assume x h t ⟨a, (eq : t = s a)⟩, eq.symm ▹ h a

    -- Intersection elimination rule
    theorem Inter.elim {x : α} (C : I → set α) :
    (x ∈ ⋂ i, C i) →  (∀ i, x ∈ C i) :=
    assume h : x ∈ ⋂ i, C i, by simp at h; apply h

    -- Intersection of subuniverses is a subuniverse
    lemma sub_of_sub_inter_sub (C : I → set α) :
    (∀ i, Sub (C i)) → Sub ⋂i, C i :=
    assume h : ∀ i, Sub (C i), show  Sub ⋂(i, C i), from
      assume (f : F) (a : ρ f → α) (h₁ : ∀ x, a x ∈ ⋂i, C i),
      show A f a ∈ ⋂i, C i, from
        Inter.intro (A f a)
        (λ j, (h j) f a (λ x, Inter.elim C (h₁ x) j))
```

The next three lemmas show that `Sg X` is the smallest subuniverse containing `X`.

```
-- X is a subset of Sg(X)
lemma subset_X_of_SgX (X : set α) : X ⊆ Sg X :=
assume x (h : x ∈ X),
show x ∈ ⋂₀ {U | Sub U ∧ X ⊆ U}, from
  assume W (h₁ : W ∈ {U | Sub U ∧ X ⊆ U}),
  show x ∈ W, from
    have h₂ : Sub W ∧ X ⊆ W, from h₁,
    h₂.right h

-- A subuniverse that contains X also contains Sg X
lemma sInter_mem {X : set α} (x : α) :
x ∈ Sg X  →  ∀ {R : set α }, Sub R → X ⊆ R → x ∈ R :=
assume (h₁ : x ∈ Sg X) (R : set α)  (h₂ : Sub R) (h₃ : X ⊆ R),
show x ∈ R, from h₁ R (and.intro h₂ h₃)

-- Sg X is a Sub
lemma SgX_is_Sub (X : set α) : Sub (Sg X) :=
assume (f : F) (a : ρ f → α) (h₀ : ∀ i, a i ∈ Sg X),
show A f a ∈ Sg X, from
  assume W (h : Sub W ∧ X ⊆ W), show A f a ∈ W, from
    have h₁ : Sg X ⊆ W, from
      assume r (h₂ : r ∈ Sg X), show r ∈ W, from
        sInter_mem r h₂ h.left h.right,
        have h' : ∀ i, a i ∈ W, from assume i, h₁ (h₀ i),
        (h.left f a h')
```

A primary motivation for this project was our observation that, on the one hand, many important constructs in universal algebra can be defined inductively, and on the other hand, type theory in general, and Lean in particular, offers excellent support for defining inductive types and powerful tactics for proving their properties. These two facts suggest that there could be much to gain from implementing universal algebra in an expressive type system that offers powerful tools for proving theorems about inductively defined types.

So, we are pleased to present the following inductive type that implements the *subuniverse generated by a set*; cf. the Informal Language definition (4.2), (4.3).

```
inductive Y (X : set α) : set α
| var (x : α) : x ∈ X → Y x
| app (f : F) (a : ρ f → α) : (∀ i, Y (a i)) → Y (A f a)
```

Next we prove that the type `Y X` defines a subuniverse, and that it is, in fact, equal to $\mathrm{Sg}^{\mathbf{A}}(X)$.

```
-- Y X is a subuniverse
lemma Y_is_Sub (X : set α) : Sub (Y X) :=
assume f a (h: ∀ i, Y X (a i)), show Y X (A f a), from
Y.app f a h
```

```
-- Y X is the subuniverse generated by X
theorem sg_inductive (X : set α) : Sg X = Y X :=
have h0 : X ⊆ Y X, from
  assume x (h : x ∈ X),
  show x  ∈ Y X, from Y.var x h,
have h1 : Sub (Y X), from
  assume f a (h : ∀ x, Y X (a x)),
  show Y X (A f a), from Y.app f a h,
have inc_l : Sg X ⊆ Y X, from
  assume u (h : u ∈ Sg X),
  show u ∈ Y X, from (sInter_mem u) h h1 h0,
have inc_r : Y X ⊆ Sg X, from
  assume a (h: a ∈ Y X), show a ∈ Sg X, from
    have h' : a ∈ Y X → a ∈ Sg X, from
      Y.rec
      --base: a = x ∈ X
      ( assume x (h1 : x ∈ X),
        show x ∈ Sg X, from subset_X_of_SgX X h1 )
      --inductive: a = A f b for some b with ∀ i, b i ∈ Sg X
      ( assume f b (h2 : ∀ i, b i ∈ Y X) (h3 : ∀ i, b i ∈ Sg X),
        show A f b ∈ Sg X, from SgX_is_Sub X f b h3 ),
    h' h,
subset.antisymm inc_l inc_r
```

Observe that the last proof proceeds exactly as would a typical informal proof that two sets are equal—prove two subset inclusions and then apply the `subset.antisymm` rule: `A ⊆ B → B ⊆ A → A = B`. We proved `Y X ⊆ Sg X` in this case by induction using the *recursor*, `Y.rec`, which Lean creates for us automatically whenever an inductive type is defined. The Lean keyword `assume` is syntactic sugar for λ; this and other notational conveniences, such as Lean's `have...from` and `show...from` syntax, make it possible to render formal proofs in a very clear and readable way.

4.5. **Terms and Free Algebras.** Fix a signature $S = (F, \rho)$ and let X be a set disjoint from F. The elements of X are called *variables*. For every $n < \omega$, let $F_n = \rho^{-1}\{n\}$ be the set of n-ary operation symbols. By a word on $X \cup F$ we mean a nonempty, finite sequence of members of $X \cup T$. We denote the concatenation of sequences by simple juxtaposition. We define, by recursion on n, the sets T_n of words on $X \cup F$ by

$$T_0 = X \cup F_0;$$
$$T_{n+1} = T_n \cup \{fs \mid f \in F,\ \ s \colon \rho f \to T_n\}.$$

Define the set of *terms in the signature* S *over* X by $T_S(X) = \bigcup_{n<\omega} T_n$.

The definition of $T_S(X)$ is recursive, indicating that *the set of terms in a signature can be implemented in Lean as an inductive type.* We will confirm this in the next subsection, but before doing so, we impose an algebraic structure on $T_S(X)$, and then state and prove some basic but important facts about this algebra. These will also be formalized in the next section, giving us another chance to compare Informal Language proofs to their formal Lean counterparts, and to show off inductively defined types in Lean.

If w is a term, let $|w|$ be the least n such that $w \in T_n$, called the *height* of w. The height is a useful index for recursion and induction. Notice that the set $T_S(X)$ is nonempty iff either X or F_0 is nonempty. As long as $T_S(X)$ is nonempty, we can impose upon this set an algebraic structure. For every basic operation symbol $f \in F$ let $f^{\mathbf{T}_S(X)}$ be the operation on $T_S(X)$ that maps each tuple $t : \rho f \to T_S(X)$ to the term ft. We define $\mathbf{T}_S(X)$ to be the algebra with universe $T_S(X)$ and with basic operations $f^{\mathbf{T}_S(X)}$ for each $f \in F$.

The construction of $\mathbf{T}_S(X)$ may seem to be making something out of nothing,but it plays a crucial role in the theory. Indeed, Part (2) of Theorem 4.3 below asserts that $\mathbf{T}_S(X)$ is *universal for S-algebras*. To prove this, we need the following basic lemma, which states that a homomorphism is uniquely determined by its restriction to a generating set.

Lemma 4.2. *Let f and g be homomorphisms from $\mathbf{A}$ to $\mathbf{B}$. If $X \subseteq A$ and X generates $\mathbf{A}$ and $f\big|_X = g\big|_X$, then $f = g$.*

Proof. Suppose the subset $X \subseteq A$ generates $\mathbf{A}$ and suppose $f\big|_X = g\big|_X$. Fix an arbitrary element $a \in A$. We show $f(a) = g(a)$. Since X generates $\mathbf{A}$, there exists a (say, n-ary) term t and a tuple $(x_1, \dots, x_n) \in X^n$ such that $a = t^{\mathbf{A}}(x_1, \dots, x_n)$. Therefore,

$$\begin{aligned} f(a) = f(t^{\mathbf{A}}(x_1, \dots, x_n)) &= t^{\mathbf{B}}(f(x_1), \dots, f(x_n)) \\ &= t^{\mathbf{B}}(g(x_1), \dots, g(x_n)) = g(t^{\mathbf{A}}(x_1, \dots, x_n)) = g(a). \end{aligned}$$

□

Theorem 4.3 ([13, Thm. 4.21])**.** *Let $S = (F, \rho)$ be a signature.*

(1) $\mathbf{T}_S(X)$ is generated by X.

(2) For every S-algebra $\mathbf{A}$ and every function $h \colon X \to A$ there is a unique homomorphism $g \colon \mathbf{T}_S(X) \to \mathbf{A}$ such that $g\big|_X = h$.

Proof. The definition of $\mathbf{T}_S(X)$ exactly parallels the construction in Theorem 4.1. That accounts for (1). For (2), define $g(t)$ by induction on ρt. Suppose $\rho t = 0$. Then $t \in X \cup F$. If $t \in X$ then define $g(t) = h(t)$. For $t \notin X$, $g(t) = t^{\mathbf{A}}$. Note that since $\mathbf{A}$ is an S-algebra and t is a nullary operation symbol, $t^{\mathbf{A}}$ is defined.

For the inductive step, let $|t| = n + 1$. Then $t = f(s_1, \dots, s_k)$ for some $f \in F_k$ and $s_1, \dots, s_k$ each of height at most n. We define $g(t) = f^{\mathbf{A}}(g(s_1), \dots, g(s_k))$. By its very definition, g is a homomorphism. Finally, the uniqueness of g follows from Lemma 4.2. □

4.6. Terms and Free Algebras in Lean (`lean-ualib/free.lean`).

As a second demonstrate of inductive types in Lean, we define a type representing the (infinite) collection $\mathbb{T}(X)$ of all terms of a given signature.

```
import basic
section
  parameters {S : signature} (X :Type*)
  local notation ‘F‘ := S.F
  local notation ‘ρ‘ := S.ρ

  inductive term
```

```
    | var : X → term
    | app (f : F) : (ρ f → term) → term

    def Term : algebra S := ⟨term, term.app⟩
  end
```

The set of terms along with the operations $F^{\mathbf{T}} := \{\mathtt{app}\, \mathtt{f} \mid f : F\}$ forms an algebra $\mathbf{T}(X) = \langle \mathbb{T}(X), F^{\mathbf{T}}\rangle$ in the signature $S = (F, \rho)$. Suppose $\mathbf{A} = \langle A, F^{\mathbf{A}}\rangle$ is an algebra in the same signature and $h \colon X \to A$ is an arbitrary function. We will show that $h \colon X \to A$ has a unique *extension* (or *lift*) to a homomorphism from $\mathbf{T}(X)$ to $\mathbf{A}$. Since $\mathbf{A}$ and $h \colon X \to A$ are aribtrary, this unique homomorphic lifting property holds universally; accordingly we say that the term algebra $\mathbf{T}(X)$ is *universal* for S-algebras. Before implementing the formal proof of this fact in Lean, let us first define some domain specific syntactic sugar.

```
section
  open term
  parameters {S : signature} (X :Type*) {A : algebra S}
  definition F := S.F             -- operation symbols
  definition ρ := S.ρ             -- arity function
  definition 𝕋 := @Term S         -- term algebra over X
  definition 𝕏 := @var S X        -- generators of the term algebra
```

If `h : X → A` is a function defined on the generators of the term algebra, then the *lift* (or *extension*) of `h` to all of `𝕋(X)` is defined inductively as follows:

```
  definition lift_of (h : X → A) : 𝕋(X) →
  | (var x) := h x
  | (app f a) := (A f) (λ x, lift_of (a x))
```

To prove that the term algebra is absolutely free, we show that the lift of an arbitrary function `h : X → A` is a homomorphism and that this lift is unique.

```
  -- The lift is a homomorphism.
  lemma lift_is_hom (h : X → A) : homomorphic (lift_of h) :=
  λ f a, show lift_of h (app f a) = A f (lift_of h ∘ a), from rfl

  -- The lift is unique.
  lemma lift_is_unique : ∀ {h h' : 𝕋(X) → A},
  homomorphic h → homomorphic h' → h ∘ 𝕏 = h' ∘ 𝕏 → h = h' :=
  assume (h h' : 𝕋(X) → A) (h₁ : homomorphic h)
    (h₂ : homomorphic h')(h₃ : h ∘ 𝕏 = h' ∘ 𝕏),
    show h = h', from
      have h₀ : ∀ t : 𝕋(X), h t = h' t, from
        assume t : 𝕋(X),
        begin
          induction t with t f a ih₁ ,
          show h 𝕏( t) = h' 𝕏( t),
          { apply congr_fun h₃ t },
          show h (app f a) = h’ (app f a),
          { have ih₂  : h ∘ a = h’ ∘ a, from funext ih₁,
            calc h (app f a) = A f (h ∘ a) : h₁ f a
                         ... = A f (h’ ∘ a) : congr_arg (A f) ih₂
                         ... = h’ (app f a) : (h₂ f a).symm }
        end,
      funext h₀
end
```

5. Summary of Project Stages

In the introduction, we set out the goals of the project in broad strokes. Here we describe four stages of concrete activities that map out a plan for achieving our goals.

Stage 1. **lean-ualib.** Implement in Lean *formal statements and proofs* of the definitions and theorems that constitute the *core* of our field, universal algebra. We call this the *Lean Universal Algebra Library*, abbreviated `lean-ualib`.[8]

Stage 2. **domain-specific automation.** Develop *proof tactics* in Lean that carry out, in a highly automated way, the most common arguments and proof techniques of universal algebra; that is, make the *proof idioms* of our field readily available in Lean.

Stage 3. **lean-uaailib.** *Develop search and artificial intelligence tools to accelerate growth of the library either by guided user input or by allowing the library to grow itself.*

Stage 4. **Algebras, Categories and Types: with computer-aided proofs.** Create a textbook that presents (a) the theory that is formalized in `lean-ualib`, (b) a comprehensive *reference manual* for the library, and (c) *examples and instructions for working mathematicians.*

Although Stage 1 alone may seem like a massive undertaking, we will start with the basic results of the theory and formalize the lemmas and theorems that are most commonly used in our field to prove deeper results. In the process, we will

(1) figure out how to *automate proof search* in the specific domain of universal algebra;
(2) create libraries for operation clones, free algebras, homomorphisms, and congruences, and design reasoning procedures for these;
(3) explore how to best exploit Lean’s *metaprogramming framework* and develop techniques and tools that help mathematicians perform accurate deductions and computations using the proof assistant;
(4) integrate procedures from other computer algebra systems (e.g., the UACalc) in a formally verified way.
(5) contribute to Lean’s development with ideas and features designed to benefit all users, notably an efficient built-in procedure for first-order reasoning and an integration with automatic theorem provers.

Then we will formalize deeper results, prioritizing definitions, lemmas, and theorems according to how widely they are used in our field. Having already formalized the *free algebra in an arbitrary signature* and implemented a formal proof that it is *absolutely free*,[9] our next target is *Birkhoff’s HSP Theorem.* With these and other foundational results

[8] Work on the `lean-ualib` has already begun; the permanent residence of our open source repository is at https://github.com/UniversalAlgebra/lean-ualib.

[9] *ibid.* lean-ualib/tree/master/src/free.lean.

established, we will have a small mathematical arsenal at our disposal that we can exploit when formalizing proofs of deeper theorems. Thus, the library will expand until we have formalized the core of our subject.

With this strategy, we expect the library development will begin at a moderate pace but will quickly accelerate. To ensure that we don't get off to a slow start or risk "reinventing the wheel," besides developing domain-specific automation tools, we will we will also build upon existing Lean libraries, such as the *Lean Mathematical Components Library* [3], so we don't have to formalize everything from scratch. These strategies will combine to substantially reduce and limit the amount of work required to complete the first stage.

The ultimate goal is to advance the state of Lean and its mathematical libraries to the point at which it becomes *a proof assistant that helps working mathematicians*, by making them both more productive and more confident in their results.

6. Challenges and objectives

Interactive theorem proving is steadily gaining ground. In some areas of computer science, it is common for research papers to be accompanied by formal proofs. We have first hand experience deploying Lean in the classroom to help teach discrete math to first- and second-year math and computer science majors. (See our Teaching Statement [23].) Others have also reported success with proof assistants in a classroom setting [47, 48]. Kevin Buzzard at Imperial College London has the ambitious goal of formalizing the entire undergraduate mathematics curriculum in Lean [18]. These circumstances point to a future where computer-aided theorem proving ese tools will be routinely used, resulting in more reliable science. However, to have a substantial impact on mathematical practice, we must narrow the usability gap.

The difficulties are as much social as technical. Proof assistants are developed primarily by computer scientists. Much of the formalized mathematics is motivated by hardware or software verification [17, 37, 41, 52]. Mathematicians largely dismiss proof assistants as impractical, so the technology improves only slowly. We want to resolve this problem by working closely with computer scientists. We aim to bring a proof assistant, its automation, and its libraries further, guided by our own research needs, as well as those of our fellow mathematicians who understand the value of proof assistants. Specifically, we will collaborate with Jasmin Blanchette and his team at the VU Amsterdam to formalize our latest research results in universal algebra and lattice theory and recent results in related areas, addressing usability issues as they arise.

Our overall aim will be met by pursuing four scientific objectives, presented below. Our starting point is that there is tremendous value in simultaneously using and developing a proof assistant. Part of the project's innovative character is that it combines these two activities, which are normally carried out by different people working in different teams. Especially for a young s ystem, close cooperation between developers and users is essential. Although formalization and implementation work can be satisfying in their own right, we will do everything possible to avoid sterile exercises in formalization and development of a large but useless library. Our first objective is to formally prove theorems about research mathematics.

Formalization can increase the trustworthiness of our results and the lucidity of our proofs, while raising awareness of formal verification in the mathematics community. From a computer science perspective, formalization will guide the development of proof assistant technology.

We will formalize theorems emerging from our research in universal algebra and lattice theory. The PI is involved in three different research projects, two in universal algebra and one that involves both algebra and lattice theory. These projects are briefly described in the appendix below. A proof assistant equipped with special libraries and tactics for formalizing universal algebra would be an invaluable tool for these research problems.

Furthermore, a number of results and ideas on which our work depends appear in journals and conference proceedings with many important details missing. Using Lean allows us to not only verify the correctness of theorems and proofs, but also determine the precise foundational assumptions required to confirm the validity of the result. Thus, when doing mathematics with the help of modern proof assustant technology, we are presented with the possibility of automating the process of generalizing results.

The idea is that implementing theorems as types and proofs as "proof objects" would produce the following:

(1) computer verified, and possibly simplified, proofs of known results
(2) better understanding of existing theory and algorithms
(3) new and/or improved theorems and algorithms

There is substantial evidence that Lean is the best platform for formalizing universal algebra. The type theory on which Lean is based provides a foundation for computation that is more powerful than first-order logic and is ideally suited to the specification of the basic objects and most common proof strategies found in of our field.

In first-order logic, higher-level (meta) arguments are second class citizens: they are interpreted as informal procedures that should be expanded to primitive arguments to achieve full rigor. This is fine for informal proofs, but becomes impractical in a formal one. Type theory supplies a much more satisfactory solution, by providing a language that can embed meta-arguments in types.

Indeed, algebraic structures are most naturally specified using typed predicate calculus expressions and explicitly requires variable-dependent sets, just like the specification language of Martin-Löf's Theory of Types [44]. In order to support this feature, programming constructs should be able to return set or type values, hence *dependent types*. This accommodates a stronger form of function definition than is available in most programming languages; specifically, it allows the type of a result of a function application to depend on a formal parameter, the value (not merely the type) of the input.

For formalizing long complex mathematical arguments Lean relies on *computational reflection*. Dependent types make it possible for data, functions, and *potential* computations to appear inside types. Standard mathematical practice is to interpret and expand these objects, replacing a constant by its definition, instantiating formal parameters, etc. In contrast, Lean supports this through typing rules that lets such computation happen transparently.

Arbitrarily long computations can thus be elided from a proof, as CiC has an ι-rule for recursion. This yields an entirely new way of proving results about specific calculations. With these tools at our disposal, we can quickly and efficiently formalize many of our results. Beyond merely certifying the correctness of our proofs, we expect that the proposed deliverables of this project will accelerate the pace of mathematical discoveries in our field.

6.1. **Research methods.** Methodological aspects of our project are listed below.

- *Collaborations:* Besides collaborating with our fellow mathematicians, we will also cooperate with computer scientists with expertise in Lean, including Jeremy Avigad (co-founder of Lean) and Jasmin Blanchette (PI of the "Lean Forward" project). Not only will they guide us through their field and help us carry out the formalization, they will provide frank feedback on the technology we develop and eventually integrate into the Lean Mathlib library [3].

- *Robust engineering:* The software and libraries will be developed with maintainability in mind, so that they can serve in future research. To ensure their longevity, we will initiate a repository of third-party Lean formalizations and tools, inspired by the Archive of Formal Proofs [13].

7. Conclusion

This effort requires a careful reconsideration of how to express the informal logical foundations of our subject, since this determines how smoothly we can implement the core mathematical theory in the formal language of the proof assistant, and this will in turn influence how accessible are the resulting libraries.

We are building tools that will make the common, informal *mathematical idioms* available in Lean, which will make this software more accessible and make it easier to discover new theorems, verify existing theorems, and test conjectures, all using a language that is relatively close to the informal language that is commonly used by those of us working in universal algebra and related fields.

Our goal is to formally prove theorems and do research mathematics while at the same time address the main usability roadblocks that stand in the way of widespread adoption of proof assistant technology. The theorems we decide to formalize and implement in the software are selected, together with our collaborators, to guide the development of theorem libraries and verified tools and methods (or *proof tactics*) for doing modern research in mathematics. The main objective of this project (and others like it [3, 16]) is to develop and codify the **practical formal foundations of informal mathematics** in which most modern research is carried out.

To support the formalization of theorems, we will develop formal libraries of fundamental universal algebra and lattice theory and explore how to automate proof search in these areas. We will create libraries for algebraic structures, free algebras, clones of polynomials and term operations, and varieties (equational classes), and design and implement reasoning procedures for these. We will explore how best to exploit Lean's *metaprogramming* framework. Moreover, we will develop techniques and tools that help mathematicians carry out correct natural deduction and induction proofs using Lean, integrating procedures from existing computer algebra systems (e.g., UACalc) in a fully verified way. Finally, we will contribute to Lean's development with ideas and features designed to benefit all users, notably an efficient built-in procedure for first-order reasoning and an integration with automatic theorem provers. The ultimate aim is to develop a proof assistant that actually helps mathematicians, by making them more productive and more confident in their results.

As scientists, we should take seriously any theory that may exposes weaknesses in our assumptions or our research habits. We should not be threatened by these disruptive forces; we must embrace them, deconstruct them, and take from them whatever they can offer our mission of advancing pure mathematics.

Appendix A. Metadata

Title	Formal Foundations for Informal Mathematics Research
Primary Field	03C05 Equational classes, universal algebra
Secondary Fields	03B35 Mechanization of proofs and logical operations 03B40 Combinatory logic and lambda-calculus 03F07 Structure of proofs 03F55 Intuitionistic mathematics 08B05 Equational logic, Malcev conditions 08B20 Free algebras 68N15 Programming languages 68N18 Functional programming and lambda calculus 68T15 Theorem proving (deduction, resolution, etc.) 68W30 Symbolic computation and algebraic computation

A.1. Project Personnel.

Principal Investigator.

William DeMeo (Burnett Meyer Instructor, University of Colorado, Boulder)

Collaborators.

Clifford Bergman (Professor, Iowa State University)

Ralph Freese (Professor, University of Hawaii)

Peter Jipsen (Professor, Chapman University)

Connor Meredith (Graduate Student, University of Colorado, Boulder)

Hyeyoung Shin (Graduate Student, Northeastern University)

Siva Somayyajula (Graduate Student, Carnegie Mellon University)

External Contributors and Advisors.

Jeremy Avigad (Professor, Carnegie Mellon University)

Andrej Bauer (Professor, University of Ljubljana)

Jasmin Blanchette (Assistant Professor, Vrije Universiteit Amsterdam)

Appendix B. complementary projects

B.1. **Lean Forward.** The Netherlands Organization for Scientific Research recently awarded a five-year grant to Jasmin Blanchette for his *Lean Forward* project, which is similar but complementary to our project. First, Dr. Blanchette is a computer scientist enlisting the support of mathematicians, whereas we are mathematicians enslisting support of computer scientists. Moreover, Jasmin works primarily with number theorists, whereas we are focused on universal algebra. We have had fruitful contact with Jasmin at the *Big Proof* workshop last summer at Cambridge University's Isaac Newton Institute, and will attend the inaugural meeting of the Lean Forward project in January 2019 in Amsterdamn. We look forward to productive future collaborations with the Lean Forward scientists.

B.2. **Existing libraries for universal algebra and lattice theory.** There has been prior work (mostly by computer scientists) on formalizing certain parts of universal algebra. The first significant example of this was initiated in Venanzio Capretta's phd thesis (see [19]) presenting the fundamentals of universal algebra using intuitionistic logic so that the resulting theorems have computational meaning (via the propositions-as-types/proofs-as-programs correspondence explained earlier). Capretta's formalizations were done in Coq.

Another more recent development in Coq is the `mathclasses` library, initiated by Bas Spitters and Eelis van der Weegen [55].

B.3. **Formal proof archives.** The Archive of Formal Proofs is a collection of proof libraries, examples, and larger scientific developments, mechanically checked in the theorem prover Isabelle. It is organized in the way of a scientific journal, is indexed by dblp and has an ISSN: 2150-914x. Submissions are refereed and companion AFP submissions to conference and journal publications are encouraged. A development version of the archive is available as well.

Appendix C. Dependent Types

Lean is a functional programming language that supports *dependent types*. Here we give just enough background on dependent types so that the unfamiliar reader can understand the role they play in the examples in Section 4 (where we implement some core concepts from universal algebra in Lean). Besides being more precise and elegant that other representations, dependent types are constructive, and (provided we work in a language like Lean that supports dependent types) we can compute them.

What makes dependent types *dependent*? The short explanation is that types can depend on *parameters*. For example,the type `list` α depends on the argument α, and this dependence is what distinguishes `list` $\mathbb{N}$ from `list bool`. Thus, dependent types are *polymorphic*. However, dependent types differ from so-called *generic types* (which also depend on *type parameters*). Let us demonstrate with another example. Consider the type `vec` α `n`, that is, the type of vectors of length `n` whose entries inhabit the type α. The type `vec` α `n` depends on one type parameter, α `: Type`, and one *value parameter*, `n :` $\mathbb{N}$. Thus, dependent types are generic types that can depend, not only on type parameters, but also on value parameters.

Suppose we wish to write a function `cons` that inserts a new element at the head of a list. What type should `cons` have? Such a function is polymorphic: we expect the `cons` function for $\mathbb{N}$, `bool`, or an arbitrary type α to behave the same way. Thus, it makes sense to take this type to be the first argument to `cons`, so that for any type α, `cons` α is the function that takes an element $\mathtt{a} : \alpha$ and a list $\ell : \mathtt{list}\ \alpha$ and returns $\mathtt{cons}\ \alpha\ \mathtt{a}\ \ell = \mathtt{a} :: \ell$ (that is, ℓ with a prepended).

While it's clear that $\mathtt{cons}\ \alpha$ should have type $\alpha \to \mathtt{list}\ \alpha \to \mathtt{list}\ \alpha$, less clear is what type cons should have; perhaps $\mathtt{Type} \to \alpha \to \mathtt{list}\ \alpha \to \mathtt{list}\ \alpha$. But this is nonsense since the type parameter α does not refer to anything, whereas it should refer to whatever is passed in for the argument of type Type. In other words, *assuming* $\alpha : \mathtt{Type}$ is the first argument to cons, then the type of the next two arguments will be α and $\mathtt{list}\ \alpha$, which are types that depend on the first input argument.

This example is an instance of a *dependent function type*, also known as a Pi type. Given $\alpha : \mathtt{Type}$ and $\beta : \alpha \to \mathtt{Type}$, view β as an indexed family of types over the index set α; for each $\mathtt{a} : \alpha$ we have a type $\beta\ \mathtt{a}$. The type $\Pi(\mathtt{x} : \alpha),\ \beta\ \mathtt{x}$ denotes the type of functions f with the property that, for each $\mathtt{a} : \alpha,\ \mathtt{f}\ \mathtt{a}$ is an element of $\beta\ \mathtt{a}$, so the type of the value returned by f *depends* on the input argument. In case β is a constant function, the type $\Pi(\mathtt{x} : \alpha),\ \beta\ \mathtt{x}$ is no different from the (non-dependent) function type $\alpha \to \beta$. Indeed, in dependent type theory, and in Lean, the Pi construction is fundamental, and $\alpha \to \beta$ is just notation for $\Pi(\mathtt{x} : \alpha),\ \beta\ \mathtt{x}$ in the special case in which β is constant.

Appendix D. More About Lean

D.1. **Lean's type hierarchy.** Like its more mature cousins Agda and Coq, Lean takes for its logical foundations *dependent type theory* with *inductive types* and *universes*. However, unlike Agda and Coq, Lean's universes are *not cumulative*.[10] At the bottom of the hierarchy is Type 0, which is usually denoted by Type. Think of Type as the universe of "small" or "ordinary" types. At the next level is Type 1, which is a larger universe of types that contains Type as an *element*. Type 2 is larger still and contains Type 1 as an element, and so on.

The upshot of this *ramified system* is that all of the types described in the last paragraph are *predicative*, which means that their definitions are not self-referential, so certain set-theoretic paradoxes (e.g. Russel's) are avoided. However, certain situations call for a self-referential type, so Lean supplies one. It is the *impredicative* type Prop. A synonym for Prop is Sort, which is shorthand for Sort 0. The type of Prop (and Sort and Sort 0) is Type 0; that is, Prop : Type 0. In general, for all $\mathtt{n} \geqq 0$, we have $\mathtt{Sort\ n{+}1} = \mathtt{Type\ n}$, whose type is $\mathtt{Type\ n{+}1}$. To summarize, we have the following type identities and relations:

$$
\begin{array}{rclcrcl}
\mathtt{Sort\ 0} & = & \mathtt{Prop} & \qquad & \mathtt{Prop} & : & \mathtt{Type\ 0} \\
\mathtt{Sort\ 1} & = & \mathtt{Type\ 0} & & \mathtt{Type\ 0} & : & \mathtt{Type\ 1} \\
\mathtt{Sort\ 2} & = & \mathtt{Type\ 1} & & \mathtt{Type\ 1} & : & \mathtt{Type\ 2} \\
 & \vdots & & & & \vdots & \\
\mathtt{Sort\ n{+}1} & = & \mathtt{Type\ n} & & \mathtt{Type\ n} & : & \mathtt{Type\ n{+}1}
\end{array}
$$

We also want some operations to be polymorphic over type universes. For example, $\mathtt{list}\ \alpha$ should make sense for any type α, no matter which universe α lives in. Thus $\mathtt{list} : \mathtt{Type\ u\_1} \to \mathtt{Type\ u\_1}$, where u_1 is a variable ranging type levels.

One aspect that makes the Lean kernel trustworthy is its size; at its core, it implements a very small set of components, namely,

[10] Although we are trying hard to avoid overly technical discussions here, we will point out that lack of cumulative universes in Lean does not reduce Lean's expressiveness; instances in which universe cumulativity would be exploited in Coq can be handled in Lean using *universe polymorphism* and the lift map.

- *dependent lambda calculus*;
- *universe polymorphism* in a hierarchy of ω-many universe levels;
- *inductive types* and *inductive families of types*, generating only the recursor for an inductive type;
- *pattern matching.*

Lean is easy to extend via *metaprogramming.* Briefly, a *metaprogram* is a program whose purpose is to modify the behavior of other programs. *Proof tactics* form an important class of metaprograms. These are automated procedures for constructing and manipulating proof terms. A distinguishing feature of Lean is that *metaprograms can be written in the Lean language*, rather that in the lower level language (C/C++) that was used to create Lean. Thus the metaprogramming language is the same logical language that we use to express specifications, propositions, and proofs.

D.2. **Lean's Elaboration Engine.** On top of the Lean kernel there is a powerful *elaboration engine* that can

(1) infer implicit universe variables,
(2) infer implicit arguments, using higher order unification,
(3) support overloaded notation or declarations,
(4) insert coercions,
(5) infer implicit arguments using type classes,
(6) convert readable proofs to proof terms,
(7) construct terms using tactics.

Lean does most of these things simultaneously. For instance, the term constructed by type classes can be used to find out implicit arguments for functions. A nice list of examples demonstrating of each of these key features of Lean can be found in the blog post by Floris van Doorn available at `https://homotopytypetheory.org/2015/12/02/the-proof-assistant-lean/`

References

[1] J. Adámek, J. Rosický, and E. M. Vitale. *Algebraic theories*, volume 184 of *Cambridge Tracts in Mathematics.* Cambridge University Press, Cambridge, 2011. A categorical introduction to general algebra, With a foreword by F. W. Lawvere.

[2] Jeremy Avigad, Mario Carneiro, Leonardo de Moura, Johannes Hölzl, and Sebastian Ullrich. The lean theorem proving language. GitHub.com, 2018. URL: `https://leanprover.github.io/`.

[3] Jeremy Avigad, Mario Carneiro, and Johannes Hölzl. The lean mathematical components library (mathlib). GitHub.com, 2018. URL: `https://github.com/leanprover/mathlib`.

[4] Jeremy Avigad, Johannes Hölzl, and Luke Serafin. A formally verified proof of the central limit theorem. *J. Automat. Reason.*, 59(4):389–423, 2017. `doi:10.1007/s10817-017-9404-x`.

[5] Steve Awodey. Type theory and homotopy. In *Epistemology versus ontology*, volume 27 of *Log. Epistemol. Unity Sci.*, pages 183–201. Springer, Dordrecht, 2012. `doi:10.1007/978-94-007-4435-6_9`.

[6] Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *J. Funct. Programming*, 13(2):261–293, 2003. Special issue on "Logical frameworks and metalanguages". URL: `http://dx.doi.org/10.1017/S0956796802004501`, `doi:10.1017/S0956796802004501`.

[7] Andrej Bauer. Five stages of accepting constructive mathematics. *Bull. Amer. Math. Soc. (N.S.)*, 54(3):481–498, 2017. `doi:10.1090/bull/1556`.

[8] Andrej Bauer. On fixed-point theorems in synthetic computability. *Tbilisi Math. J.*, 10(3):167–181, 2017. `doi:10.1515/tmj-2017-0107`.

[9] Andrej Bauer. Algebraic effects and handlers. OPLSS 2018, June 2018. Lecture notes available at: `https://github.com/OPLSS/introduction-to-algebraic-effects-and-handlers`; Recorded lecture available at: `https://youtu.be/atYp386EGo8`.

[10] Andrej Bauer and Jens Blanck. Canonical effective subalgebras of classical algebras as constructive metric completions. *J.UCS*, 16(18):2496–2522, 2010.

[11] Andrej Bauer, Gordon D. Plotkin, and Dana S. Scott. Cartesian closed categories of separable Scott domains. *Theoret. Comput. Sci.*, 546:17–29, 2014. doi:10.1016/j.tcs.2014.02.042.

[12] Mike Behrisch, Sebastian Kerkhoff, and John Power. Category theoretic understandings of universal algebra and its dual: monads and Lawvere theories, comonads and what? In *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVIII)*, volume 286 of *Electron. Notes Theor. Comput. Sci.*, pages 5–16. Elsevier Sci. B. V., Amsterdam, 2012. doi:10.1016/j.entcs.2012.08.002.

[13] Clifford Bergman. *Universal algebra*, volume 301 of *Pure and Applied Mathematics (Boca Raton)*. CRC Press, Boca Raton, FL, 2012. Fundamentals and selected topics.

[14] Clifford Bergman and William DeMeo. Universal algebraic methods for constraint satisfaction problems. *CoRR: arXiv preprint*, abs/1611.02867, 2016. URL: https://arxiv.org/abs/1611.02867, arXiv:1611.02867.

[15] P. G. Bertoli, J. Calmet, F. Giunchiglia, and K. Homann. Specification and integration of theorem provers and computer algebra systems. *Fund. Inform.*, 39(1-2):39–57, 1999. Symbolic computation and related topics in artificial intelligence (Plattsburg, NY, 1998).

[16] J. C. Blanchette. Lean forward: Usable computer-checked proofs and computations for number theorists. GitHub.com, 2018. URL: https://lean-forward.github.io/.

[17] S. Boldo, F. Clément, J. Filliâtre, M. Mayero, G. Melquiond, and P. Weis. Wave equation numerical resolution: A comprehensive mechanized proof of a c program. *Journal of Automated Reasoning*, 2013.

[18] Kevin Buzzard. Xena project. weblog, 2018. URL: https://xenaproject.wordpress.com/.

[19] Venanzio Capretta. Universal algebra in type theory. In *Theorem proving in higher order logics (Nice, 1999)*, volume 1690 of *Lecture Notes in Comput. Sci.*, pages 131–148. Springer, Berlin, 1999. URL: http://dx.doi.org/10.1007/3-540-48256-3_10, doi:10.1007/3-540-48256-3_10.

[20] P. Chiusano and R. Bjarnason. *Functional Programming in Scala*. Manning Publications, 2014. URL: https://books.google.com/books?id=bmTRlwEACAAJ.

[21] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.7*, October 2017. URL: http://coq.inria.fr.

[22] Thierry Coquand and Gérard Huet. The calculus of constructions. *Inform. and Comput.*, 76(2-3):95–120, 1988. URL: http://dx.doi.org/10.1016/0890-5401(88)90005-3, doi:10.1016/0890-5401(88)90005-3.

[23] William DeMeo. Teaching statement, December. URL: https://github.com/williamdemeo/job-app.

[24] William DeMeo, Ralph Freese, and Peter Jipsen. Representing finite lattices as congruence lattices of finite algebras, 2018. URL: https://github.com/UniversalAlgebra/fin-lat-rep.

[25] William DeMeo, Ralph Freese, and Matthew Valeriote. Polynomial-time tests for difference terms in idempotent varieties. GitHub.com, 2018. URL: https://github.com/UniversalAlgebra/term-conditions/tree/master/ijac.

[26] William DeMeo, Peter Mayr, and Nik Ruskuc. A new characterization of fiber products of lattices, 2018. URL: https://github.com/UniversalAlgebra/fg-free-lat.

[27] William DeMeo and Siva Somayyajula. The lean universal algebra library. GitHub.com, 2018. URL: https://github.com/UniversalAlgebra/lean-ualib.

[28] J. Fasel. Erratum on "on the number of generators of ideals in polynomial rings". *Annals of Mathematics*, 186(2):647–648, 2017.

[29] Eric Finster. Higher algebra in type theory. GitHub.com, 2018. URL: https://github.com/ericfinster/higher-alg.

[30] David Gepner, Rune Haugseng, and Joachim Kock. ∞-operads as analytic monads. arXiv, 2017. URL: https://arxiv.org/abs/1712.06469.

[31] G. Gonthier. Presented at Workshop on Computer-Aided Mathematical Proof, 2017. Video available at https://www.newton.ac.uk/seminar/20170713090010001.

[32] Georges Gonthier. Formal proof—the four-color theorem. *Notices Amer. Math. Soc.*, 55(11):1382–1393, 2008.

[33] Georges Gonthier, Andrea Asperti, Jeremy Avigad, and et al. A machine-checked proof of the odd order theorem. In *Interactive theorem proving*, volume 7998 of *Lecture Notes in Comput. Sci.*, pages 163–179. Springer, Heidelberg, 2013. doi:10.1007/978-3-642-39634-2_14.

[34] William Timothy Gowers, Natarajan Shankar, and Patrick Ion. Panel on future directions for big proof, July 2017. Panel discussion; recorded video available online. URL: `https://www.newton.ac.uk/seminar/20170714143015301`.

[35] Thomas Hales, et al. A formal proof of the Kepler conjecture. *Forum Math. Pi*, 5:e2, 29, 2017. `doi:10.1017/fmp.2017.1`.

[36] Michael Harris. Mathematician's of the future? *Slate*, 2015. URL: `https://goo.gl/RjM8yd`.

[37] J. Harrison. Formal verification at intel. In *Logic in Computer Science*, pages 45–54. IEEE Computer Society, 2003.

[38] John Harrison and L. Théry. A skeptic's approach to combining HOL and Maple. *J. Automat. Reason.*, 21(3):279–294, 1998. `doi:10.1023/A:1006023127567`.

[39] M. J. H. Heule and O. Kullmann. The science of brute force. *Communications of the ACM*, 60(8):70–79, 2017.

[40] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, April 1989. URL: `http://dx.doi.org/10.1093/comjnl/32.2.98`, `doi:10.1093/comjnl/32.2.98`.

[41] J. Hurd. Verification of the miller-rabin probabilistic primality test. *Journal of Logic and Algebraic Programming*, 56(1-2):3–21, 2003.

[42] Martin Hyland and John Power. The category theoretic understanding of universal algebra: Lawvere theories and monads. In *Computation, meaning, and logic: articles dedicated to Gordon Plotkin*, volume 172 of *Electron. Notes Theor. Comput. Sci.*, pages 437–458. Elsevier Sci. B. V., Amsterdam, 2007. URL: `https://doi-org.colorado.idm.oclc.org/10.1016/j.entcs.2007.02.019`, `doi:10.1016/j.entcs.2007.02.019`.

[43] Ursula Martin, Larry Paulson, and Andrew Pitts. Computer-aided mathematical proof. In *Big Proof Workshop*, Cambridge University, 2017. Isaac Newton Institute. URL: `https://www.newton.ac.uk/event/bprw01`.

[44] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory. Lecture Notes*. Bibliopolis, Naples, 1984. Notes by Giovanni Sambin.

[45] R. Matuszewski and P. Rudnicki. Mizar: The first 30 years. *Mechanized Mathematics and Its Applications*, 4(1):3–24, 2005.

[46] Karl Meinke. Universal algebra in higher types. *Theoret. Comput. Sci.*, 100(2):385–417, 1992. `doi:10.1016/0304-3975(92)90310-C`.

[47] T. Nipkow. Teaching semantics with a proof assistant: No more lsd trip proofs. pages 24–38. Springer, 2012.

[48] B. C. Pierce. Lambda, the ultimate TA: Using a proof assistant to teach programming language foundations. In G. Hutton and A. P. Tolmach, editors, *International Conference on Functional Programming*, pages 121–122. ACM, :2009.

[49] Don Pigozzi and Antonino Salibra. Lambda abstraction algebras: representation theorems. *Theoret. Comput. Sci.*, 140(1):5–52, 1995. Selected papers of AMAST '93 (Enschede, 1993). URL: `http://dx.doi.org/10.1016/0304-3975(94)00203-U`, `doi:10.1016/0304-3975(94)00203-U`.

[50] M. Pollet and Manfred Kerber. Informal and formal representations in mathematics. *Studies in Logic, Grammar and Rhetoric: From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, 10(23):75–94, 1 2007.

[51] J. Rehmeyer. Voevodsky's mathematical revolution. Scientific American Blogs, 2013. URL: `http://blogs.scientificamerican.com/guest-blog/`.

[52] D. M. Russinoff. A mechanically checked proof of correctness of the amd k5 floating point square root microcode. *Formal Methods in System Design*, 1999.

[53] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoret. Comput. Sci.*, 121(1-2):411–440, 1993. A collection of contributions in honour of Corrado Böhm on the occasion of his 70th birthday. URL: `http://dx.doi.org/10.1016/0304-3975(93)90095-B`, `doi:10.1016/0304-3975(93)90095-B`.

[54] Daniel Selsam and Leonardo de Moura. Congruence closure in intensional type theory. In *Automated reasoning*, volume 9706 of *Lecture Notes in Comput. Sci.*, pages 99–115. Springer, [Cham], 2016. `doi:10.1007/978-3-319-40229-1_8`.

[55] Bas Spitters and Eelis van der Weegen. Developing the algebraic hierarchy with type classes in Coq. In *Interactive theorem proving*, volume 6172 of *Lecture Notes in Comput. Sci.*, pages 490–493.

Springer, Berlin, 2010. URL: http://dx.doi.org/10.1007/978-3-642-14052-5_35, doi:10.1007/978-3-642-14052-5_35.

[56] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. MIT Press, Cambridge, MA, USA, 1990.

[57] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics*. MIT Press, Cambridge, MA, USA, 1990.

[58] Dmitriy Zhuk. The proof of CSP dichotomy conjecture. *CoRR arXiv*, abs/1704.01914, 2017. URL: http://arxiv.org/abs/1704.01914, arXiv:1704.01914.