

Formal Foundations for Informal Mathematics Research

William DeMeo

WORK IN PROGRESS. DO NOT DISTRIBUTE. (compiled on 2018/12/11 at 19:01:56)

“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 [32]

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, lattice theory, and category theory. A distinguishing feature of this effort is the high priority placed on *usability* of the formal mathematical libraries that we produce. 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*.

1. INTRODUCTION AND MOTIVATION

A significant portion of the professional mathematician’s time is typically occupied by tasks other than *Deep Research*, by which we mean such activities as discovering truly non-trivial, publishable results, inventing novel proof techniques, or conceiving of compelling 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 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. In fact, 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 the effective proving and communicating of new mathematical results would be 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 new mathematics and for communicating these developments painlessly. 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.

To verify mathematical arguments by computer, the arguments must be written so that machines can interpret and check them. We refer to the practice of writing such proofs as *interactive theorem proving*, and to the programming language and software system that checks such proofs as a *proof assistant* or an *interactive theorem prover*. Such systems are increasingly used in academia and industry to verify the correctness of hardware, software, and protocols. *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* [19] and the *Feit-Thompson Odd Order Theorem* [20], as well as settling major open problems, such as the *Kepler Conjecture* [21].

Another justification for 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 (see, e.g., [16]). Moreover, the emergence of large computational proofs (like Thomas Hales' proof of the Kepler Conjecture) lead to referee assignments that are impossible burdens on individual reviewers [23]. Worse than that, even when such work survives peer review, there remain disputes over correctness, completeness, whether nontrivial gaps exist, etc. Some recent examples include Zhuk's proof of the *CSP dichotomy conjecture* of Feder and Vardi [35], and Fasel's proposed solution to *Murthy's conjecture* [16].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.

“The mathematical community today invests a good deal of time and resources in the refereeing process... surely any computational tools that can help in that regard should be valued.”

—Jeremy Avigad

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.

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

¹Fasel'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 [16].

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.

Finally, modern proof assistants support automated proof search and this can be used to discover long sequences of first-order deduction steps in a proof. Consequently, mathematicians can spend less time carrying out the parts of an argument that are more-or-less obvious (once discovered), and more time contemplating deeper questions.

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.

One question that leads to insight into what we call the “usability gap” that plagues most modern proof assistants is why *computer algebra systems* do not suffer as much 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 an interactive theorem prover than for a computer algebra system, and this is likely because the latter is typically conceived of by a mathematician whose primary aim is to build a system that presents things “as they should be,” that is, as a mathematically educated user would expect. To a mathematician, such systems are *well-designed*.

In many theorem proving systems, things are often not presented as we would expect or like them to be. In [27], 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 a few of the major roadblocks to wide-spread adoption of proof assistants; specifically,

- *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 cover only a small fraction of mathematics.

- *Weak automation of mathematics.* Proof assistants typically combine general-purpose logical automation and procedures for arithmetic. But without *domain-specific automation* (DSA), a single sentence in a proof in the Informal Language can correspond to dozens or hundreds of lines in a formal language.
- *Poor interoperability with computer algebra systems.* Notwithstanding the existence of a few prototypes [9, 22], 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 certificates produced by the algebra systems. However, most procedures do not generate 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 the main proof assistants have evolved over several decades and are difficult to learn. For example, to extend Coq, 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, to the point where he stopped believing paper-and-pencil proofs [28].

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. 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.

2.1. Foundations of mathematics and computing science. Universal algebra, lattice theory, and category theory have had a deep and lasting impact on the development of theoretical computer science, particularly the subfields of domain theory, denotational semantics, and programming languages research [29]. Dually, progress in theoretical computer science has informed and inspired a substantial amount of pure mathematics in the last half-century [2, 3, 5, 6, 26, 29], just as physics and physical intuition motivated so many mathematical discoveries of the last two centuries.

The availability and accessibility of *functional* programming languages supporting *dependent types* has given rise to 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 also to express programming constructs that would be incomprehensible or inconceivable in an *imperative* or *procedural* programming language. (For examples, see [4], [13, Chs. 5 & 10].) These are exciting developments for the design and implementation of computer programs, and this has some relevance to our work. However, our focus is on the development and codification of the foundations of general algebra for research in pure mathematics.

In the remainder of this project description, we give some background on interactive theorem proving technology, introduce dependent type theory, and describe the *Lean* proof assistant. We will demonstrate how and why these technologies are well suited to research in universal algebra. Thereafter we present the concrete goals of our research program and explain how we intend to accomplish them. Along the way, we will see some examples of the achievements we have already made in pursuit of these goals.

Before proceeding, however, let us summarize in broad terms and using nontechnical language the main objectives of this research project.

1. To present the core of universal algebra using *practical* logical foundations, which means that, when possible, definitions, theorems and proofs should be constructive and have computational meaning.
2. To develop software that extends the *Lean Mathematics Library* [12] to include the output of task (1), *implementing* the core results of our field as *types* and their proofs as *programs* (or *proof objects*) in Lean’s *dependent type theory*.
3. To develop *domain-specific automation (DSA)* tools that help working mathematicians harness the power of modern proof assistant technology.
4. To teach mathematicians how to use the assets developed in items (1)–(3) to do the following:

²In the sequel, we use the fairly standard acronym TCS for *theoretical computer science*, broadly construed. The Theory A–Theory B dichotomy was established by “The Handbook of Theoretical Computer Science” [33, 34], Volume A of which includes chapters on algorithms and complexity theory; Volume B covers domain theory, semantics, and type theory.

- a. translate existing or proposed Informal Language proofs (typeset in L^AT_EX, 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.

3. THE LEAN THEOREM PROVER AND ITS ROLE IN THE PROJECT

Given our motivation, the choice of language and proof system to use was easy; we chose the *Lean proof assistant* because it 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 and is easily adopted by mathematicians who lack special training in computer science.

Lean is a relatively new programming language and proof assistant developed at Microsoft Research and Carnegie Mellon University. Lean draws on decades of experience in interactive and automatic theorem provers (e.g., Coq, Isabelle/HOL, and Z3). Its logic is very expressive, and emphasis is placed on *powerful proof automation*. The system is easy to extend via *metaprograms* developed in the same logical language as is used to express specifications and proofs in Lean. In this way, Lean aims to *bridge the gap between interactive and automated theorem proving*.

There are many other reasons Lean is an ideal platform for this project. For instance, it is unique among computer-based theorem proving tools in that its *proofs tend to be easy to read and understood* without special training. In fact, working in Lean usually leads to formal proofs that are cleaner, more concise, and shorter than the corresponding proofs in the Informal Language.

To support the formalization of theorems, we will develop libraries that contain formal statements and proofs of all of the core definitions and results universal algebra, and we will explore how to automate proof search specifically in these foundational areas. We will also formalize theorems emerging from our own mathematics research. We are currently engaged in three other research projects, two in universal algebra and one in lattice theory. A proof assistant equipped with special libraries of definitions and results from algebra and lattice theory, as well as specialized tactics to automate the standard *proof idioms* of our field, would be extremely useful for our other research projects. One of our goals is to demonstrate (to ourselves and to our colleagues around the globe) the utility of such libraries and tactics for proving “real world” theorems.

We will develop techniques and tools that help mathematicians perform accurate computations and deductive reasoning using proof assistants, integrating procedures from computer algebra systems in a 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*.

Lean is a very young language, and its domain-specific libraries for special disciplines are small but growing. *It is vital for mathematicians to get involved at this early stage and play a leading role in its development.* If we leave all of this to our colleagues in computer science, they will base the development on their perception of our needs, history will likely repeat itself, and it is unlikely that the libraries and tools that come out of the effort will meet the needs and expectations of most working mathematicians.

3.1. Lean’s Hierarchy of Sorts and Types. 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*.³ 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 $n \geq 0$, we have **Sort n+1** = **Type n**, whose type is **Type n+1**. To summarize, we have the following type identities and relations:

$$\begin{array}{ll} \text{Sort } 0 = \text{Prop} & \text{Prop} : \text{Type } 0 \\ \text{Sort } 1 = \text{Type } 0 & \text{Type } 0 : \text{Type } 1 \\ \text{Sort } 2 = \text{Type } 1 & \text{Type } 1 : \text{Type } 2 \\ \vdots & \vdots \\ \text{Sort } n+1 = \text{Type } n & \text{Type } n : \text{Type } n+1 \end{array}$$

See Section C of the appendix for some examples demonstrating Lean’s ramified hierarchy of sorts and types.

We also want some operations to be polymorphic over type universes. For example, **list** α should make sense for any type α , no matter which universe α lives in. Thus **list** : **Type u_1** \rightarrow **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,

- *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;

³Although 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.

- *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 than 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.

3.2. Dependent Types. Lean is a functional programming language that supports *dependent types*. Here we explain what dependent types are. Later, when we present some examples of constructions in universal algebra, we will see exactly how dependent types support precise representations of many of the basic objects of interest in universal algebra. Besides being more precise and elegant than other representations, dependent types are constructive, and (provided we work in a language that supports them) 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 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, $\alpha : \text{Type}$, 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 `a` : α and a list `l` : `list α` and returns `cons α a l = a :: l` (that is, `l` with `a` prepended).

While it's clear that `cons α` should have type $\alpha \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha$, less clear is what type `cons` should have; perhaps $\text{Type} \rightarrow \alpha \rightarrow \text{list } \alpha \rightarrow \text{list } \alpha$. 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* $\alpha : \text{Type}$ is the first argument to `cons`, then the type of the next two arguments will be α and `list α` , which are types that depend on the first input argument.

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

the **Pi** construction is fundamental, and $\alpha \rightarrow \beta$ is just notation for $\Pi(x : \alpha), \beta x$ in the special case in which β is constant.

3.3. 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/>

3.4. Domain Specific Automation. One of the primary objectives of Lean is to bridge the gap between automatic and interactive theorem proving. A cornerstone for this vision is *white-box automation*: the key techniques underlying SMT solvers, such as the *congruence closure* for reasoning about equality [30] and the quantifier instantiation heuristics [15], are available to users via the metaprogramming framework. Despite this, several reasoning components, as well as some important metaprogramming “tape and glue,” are missing. For all its potential, Lean is not as automated as Isabelle/HOL.

We will design a cohesive set of reasoning components. As a young system, Lean is an ideal test-bed for new ideas. A strong reconstruction tactic based on a procedure for first- or even higher-order logic is critical to automation. The existing super tactic [30] is unfortunately orders of magnitude too slow to be useful. Developing a monadic prover with efficient data structures and finding solutions to speed up the execution of all metaprograms will require innovative ideas.

We also want to contribute to the Lean simplifier’s support for arithmetic and the envisioned algebraic normalizer, based on our experience working towards the first two objectives. The main difficulty is to balance raw deductive power, extensibility, and predictability. Furthermore, we will design a relevance filter that selects lemmas from background libraries, for use with tactics. One of the challenges is to perform the heuristic selection in a robust way, so that filtering can be used both for proof search and for rechecking the proof. Existing filters, whether they rely on machine learning [12] or not, suffer from a lack of “diversity:” they tend to select many equivalent lemmas while ignoring some key lemmas. Ideas from research in search engines will likely apply.

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 $\rho: F \rightarrow N$. 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*.⁴

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 each operation symbol; that is,

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

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, 7, 17, 18, 25]. These are natural generalizations that we plan to integrate into future versions of our `lean-uialib` 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 $f: A^{\rho f} \rightarrow A$. If N happens to be \mathbb{N} , then ρf denotes the set $\{0, 1, \dots, \rho f - 1\}$ and a function $g: \rho f \rightarrow A$, identified with its graph, is simply a ρf -tuple of elements from A . The domain $A^{\rho f}$ can be represented by the type $\rho f \rightarrow A$ of functions from ρf to A , and we will represent operations $f: A^{\rho f} \rightarrow A$ using the function type $(\rho f \rightarrow A) \rightarrow A$.

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 $a: m \rightarrow A$, defined for each $i < m$ by $a i = a_i$. Therefore, if $h: A \rightarrow B$, then $h \circ a: m \rightarrow B$ 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 $g: A^m \rightarrow A$, then $g a$ has type A . If $f: (\rho f \rightarrow B) \rightarrow B$ is a ρf -ary operation on B , if $a: \rho f \rightarrow A$ is a ρf -tuple on A , and if $h: A \rightarrow B$, then $h \circ a: \rho f \rightarrow B$, so $f(h \circ a): B$.

⁴An 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.

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 $\pi \ i$ maps a given tuple $a : \beta \rightarrow \alpha$ to its value at input i . For instance, suppose we have types α and β of arbitrary *height*,⁵ and variables $i : \beta$ and $f : \beta \rightarrow \alpha$.

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

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

We define a signature as a structure with two fields, the type F of *operation symbols* and an *arity function* $\rho : F \rightarrow \text{Type*}$, 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.⁶

```
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 $\rho \ f$ (instead of $S.\rho \ f$). In these two cases, the Lean syntax matches our Informal Language notation exactly.

⁵The *height* of a type is simply type's *level* (see Section 3.1), and the syntax `Type*` indicates that we do not wish to commit in advance to a specific height.

⁶The `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.

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 .⁷ An inhabitant of the type `algebra_on` assigns an interpretation to each `op` symbol $f : F$, which yields a function of type $(\rho \ f \rightarrow \alpha) \rightarrow \alpha$.

Finally, we define an algebra. Since an algebra pairs a carrier with an interpretation of the operation symbols, we use the *dependent pair type*, $\Sigma (x : A), B \ x$, also known as a *Sigma type*. This is the type of ordered pairs $\langle a, b \rangle$, where $a : A$, and b has type $B \ a$ which may depend on a . Just as the *Pi type* $\Pi (x : A), B \ x$ generalizes the notion of function type $A \rightarrow B$ by allowing the codomain $B \ x$ to depend on the value x of the input argument, a Sigma type $\Sigma (x : A), B \ x$ generalizes the cartesian product $A \times B$ by allowing the type $B \ x$ of the second argument of the ordered pair to depend on the value x of the first.⁸

Since an algebra $\langle A, F^A \rangle$ 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-uilib` 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 $A = \langle A, F^A \rangle$ is f^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^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-uilib` represents the assertion that a function is an S -homomorphism.

```
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 $\forall f \forall a \ h(f^A(a)) = f^B(h \circ a)$, we expect working algebraists to find the `lean-uilib` syntax quite satisfactory.

⁷plus 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-uilib/>

⁸Lean's built-in `sigma` type is defined as follows:

```
structure sigma  $\alpha$  : Type u ( $\beta$  :  $\alpha \rightarrow$  Type v) := mk :: (fst :  $\alpha$ ) (snd :  $\beta$  fst)
```

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 demonstrate 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 \mathbf{SA} the set of all subuniverses of \mathbf{A} . If B is a subuniverse of \mathbf{A} and $F^{\mathbf{A}} \upharpoonright^B = \{f^{\mathbf{A}} \upharpoonright B \mid f \in F\}$ is the set of basic operations of \mathbf{A} restricted to B , then $\mathbf{B} = \langle B, F^{\mathbf{A}} \upharpoonright^B \rangle$ 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:

$$(4.1) \quad \text{Sg}^{\mathbf{A}}(X) = \bigcap \{U \in \mathbf{SA} \mid X \subseteq U\}.$$

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 ([8, 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:*

$$(4.2) \quad X_0 = X;$$

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

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

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), $\text{Sg}^{\mathbf{A}}(X) \subseteq Y$. For the opposite inclusion, it is enough to check, by induction on n , that $X_n \subseteq \text{Sg}^{\mathbf{A}}(X)$. Well, $X_0 = X \subseteq \text{Sg}^{\mathbf{A}}(X)$ from its definition. Assume that $X_n \subseteq \text{Sg}^{\mathbf{A}}(X)$. 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 $\forall i, a_i \in \text{Sg}^{\mathbf{A}}(X)$ and since this latter object is a subuniverse, $b \in \text{Sg}^{\mathbf{A}}(X)$ as well. \square

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 operation 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.sym ▸ 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 $\text{Sg } X$ is the smallest subuniverse containing X .

```

-- X is a subset of Sg(X)
lemma subset_X_of_SgX (X : set  $\alpha$ ) :  $X \subseteq \text{Sg } X :=$ 
  assume x (h :  $x \in X$ ),
  show  $x \in \bigcap_0 \{U \mid \text{Sub } U \wedge X \subseteq U\}$ , from
    assume W ( $h_1 : W \in \{U \mid \text{Sub } U \wedge X \subseteq U\}$ ),
    show  $x \in W$ , from
      have  $h_2 : \text{Sub } W \wedge X \subseteq W$ , from  $h_1$ ,
       $h_2.\text{right } h$ 

-- A subuniverse that contains X also contains Sg X
lemma sInter_mem {X : set  $\alpha$ } (x :  $\alpha$ ) :
 $x \in \text{Sg } X \rightarrow \forall \{R : \text{set } \alpha\}, \text{Sub } R \rightarrow X \subseteq R \rightarrow x \in R :=$ 
  assume ( $h_1 : x \in \text{Sg } X$ ) (R : set  $\alpha$ ) ( $h_2 : \text{Sub } R$ ) ( $h_3 : X \subseteq R$ ),
  show  $x \in R$ , from  $h_1$  R (and.intro  $h_2$   $h_3$ )

-- Sg X is a Sub
lemma SgX_is_Sub (X : set  $\alpha$ ) : Sub (Sg X) :=
  assume (f : F) (a :  $\rho$  f  $\rightarrow \alpha$ ) ( $h_0 : \forall i, a \ i \in \text{Sg } X$ ),
  show  $A \ f \ a \in \text{Sg } X$ , from
    assume W ( $h : \text{Sub } W \wedge X \subseteq W$ ), show  $A \ f \ a \in W$ , from
      have  $h_1 : \text{Sg } X \subseteq W$ , from
        assume r ( $h_2 : r \in \text{Sg } X$ ), show  $r \in W$ , from
          sInter_mem r  $h_2$   $h.\text{left } h.\text{right}$ ,
          have  $h' : \forall i, a \ i \in W$ , from assume i,  $h_1$  ( $h_0 \ i$ ),
          ( $h.\text{left } f \ a \ h'$ )

```

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  $\alpha$ ) : set  $\alpha$ 
| var (x :  $\alpha$ ) :  $x \in X \rightarrow Y \ x$ 
| app (f : F) (a :  $\rho$  f  $\rightarrow \alpha$ ) : ( $\forall i, Y \ (a \ i)$ )  $\rightarrow Y \ (A \ f \ a)$ 

```

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

```

-- Y X is a subuniverse
lemma Y_is_Sub (X : set  $\alpha$ ) : Sub (Y X) :=
  assume f a (h:  $\forall i, Y\ X\ (a\ i)$ ), 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  $\alpha$ ) : Sg X = Y X :=
  have h0 : X  $\subseteq$  Y X, from
    assume x (h : x  $\in$  X),
    show x  $\in$  Y X, from Y.var x h,
  have h1 : Sub (Y X), from
    assume f a (h :  $\forall x, Y\ X\ (a\ x)$ ),
    show Y X (A f a), from Y.app f a h,
  have inc_l : Sg X  $\subseteq$  Y X, from
    assume u (h : u  $\in$  Sg X),
    show u  $\in$  Y X, from (sInter_mem u) h h1 h0,
  have inc_r : Y X  $\subseteq$  Sg X, from
    assume a (h: a  $\in$  Y X), show a  $\in$  Sg X, from
      have h' : a  $\in$  Y X  $\rightarrow$  a  $\in$  Sg X, from
        Y.rec
        --base: a = x  $\in$  X
        ( assume x (h1 : x  $\in$  X),
          show x  $\in$  Sg X, from subset_X_of_SgX X h1 )
        --inductive: a = A f b for some b with  $\forall i, b\ i \in Sg\ X$ 
        ( assume f b (h2 :  $\forall i, b\ i \in Y\ X$ ) (h3 :  $\forall i, b\ i \in Sg\ X$ ),
          show A f b  $\in$  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: $A \subseteq B \rightarrow B \subseteq A \rightarrow A = B$. We proved $Y\ X \subseteq Sg\ X$ 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

$$\begin{aligned} T_0 &= X \cup F_0; \\ T_{n+1} &= T_n \cup \{fs \mid f \in F, s: \rho f \rightarrow T_n\}. \end{aligned}$$

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 $t: \rho f \rightarrow T_S(X)$ 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 $f|_X = g|_X$, then $f = g$.*

Proof. Suppose the subset $X \subseteq A$ generates \mathbf{A} and suppose $f|_X = g|_X$. 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 ([8, 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 $h: X \rightarrow A$ there is a unique homomorphism $g: \mathbf{T}_S(X) \rightarrow \mathbf{A}$ such that $g|_X = h$.

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. \square

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 $F^{\mathbf{T}} := \{\text{app } f \mid f : F\}$ 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 $h : X \rightarrow A$ is an arbitrary function. We will show that $h : X \rightarrow A$ has a unique *extension* (or *lift*) to a homomorphism from $\mathbf{T}(X)$ to \mathbf{A} . Since \mathbf{A} and $h : X \rightarrow A$ are arbitrary, 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 $h : X \rightarrow A$ is a function defined on the generators of the term algebra, then the *lift* (or *extension*) of h to all of $\mathbb{T}(X)$ 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 $h : X \rightarrow A$ 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.

We briefly summarize the high-level goals of the project in the following list of four key stages. The presentation here is brief, and the language over-simplified, but further details about each of these stages are presented in later sections.

- Stage 1. **lean-ualib.** Implement in Lean (the proof assistant described below) *formal statements and proofs* of the definitions and theorems that constitute the *core* of our field (universal algebra and lattice theory). (We call this the *Lean Universal Algebra Library*, or **lean-ualib** for short.)⁹
- 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 available in Lean.
- Stage 3. **Algebras, Categories and Types: with computer-aided proofs.** Publish 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*.
- Stage 4. **lean-uaailib.** *Develop search and artificial intelligence tools that will accelerate growth of the library either by guided user input or by allowing the library to grow itself.*

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 *clones* of operations and terms, 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 (in particular, 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 go after the deeper results, prioritizing definitions, lemmas, and theorems according to how widely they are used in our field. With these foundations established, we will soon have a small mathematical arsenal at our disposal that we

⁹Work on the **lean-ualib** has already begun; the permanent residence of our open source repository is <https://github.com/UniversalAlgebra/lean-ualib>.

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, we will develop novel automation strategies, and “proof tactics” that are specifically tailored to the type of arguments that are most commonly used—the “proof idioms” of our field. Furthermore, we will build upon already existing Lean math libraries, such as the Lean Mathlib [12], so we will not have to formalize all of the basic theorems 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.

5.1. 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. Proof assistants are even deployed in the classroom [51, 57], replacing paper-and-pencil arguments. These circumstances point to a future where these tools will be routinely used, resulting in more reliable science. But to have a substantial impact on mathematical practice, we must narrow the usability gap.

Our overall aim is to make proof assistants usable by mathematicians, by initially focusing on number theory and related areas, developing the proof automation, tool integrations, and formal libraries guided by actual users' needs.

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, 40, 42, 61]. Mathematicians largely dismiss proof assistants as impractical, so the technology improves only slowly. Sadly, even routine operations such as factorization of polynomials can become small challenges when moving to a formal-logic environment [46, 66]. We want to break the vicious circle by working closely with mathematicians. We aim to bring a proof assistant, its automation, and its libraries further, guided by the needs of mathematicians who understand the value of proof assistants. Specifically, we will collaborate with Sander Dahmen and his team at the VU Amsterdam to formalize parts of the team's research in number theory and recent results in related areas, addressing usability issues as they arise.

Our vehicle will be the Lean proof assistant [27]. Lean is a new open source system developed by Leonardo de Moura (Microsoft Research, USA), Jeremy Avigad (Carnegie Mellon, USA), and their colleagues. The system's design and engineering is unusually clean and efficient. Lean attempts to combine the best from two leading proof assistants.

- Lean's logical foundation is a variant of Coq's calculus of inductive constructions (CiC) [14], a dependent type theory. Lean distinguishes itself with its

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 [30]. A mechanism for introducing quotient types and a transfer tool facilitate reasoning up to isomorphism without resorting to setoids [6] or homotopy type theory [5].

- 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 by Avigad et al. [3, Section 3.1]. I encountered similar difficulties with tensors and multidimensional measures [8, Section 5].

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 system, 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.

From the mathematician’s point of view, formalization can increase the trustworthiness of their results and the lucidity of their 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 could be extremely useful for these research problems. One of our goals is to demonstrate the utility of such libraries and tactic for proving these “real-world” theorems.

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 technology, we are presented with the possibility of automated process for 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 seems well suited to the specification of the basic objects and most common proof strategies found in 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 [24]. 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. while Lean supports this through typing rules that lets such computation happen transparently.

$$\frac{t : A \quad A \equiv_{\beta\iota\delta\zeta} B}{t : B}$$

This rule states that the $\beta\iota\delta\zeta$ -computation rules of CiC yield equivalent types. It is a subsumption rule: there is no record of its use in the proof term t .

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.

5.2. Research methods. The most noteworthy 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 []

- *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].
- *Flexiformality*: Russell quipped that “the method of ‘postulating’ what we want has many advantages; they are the same as the advantages of theft over honest toil” [60]. However, for most mathematicians, this fundamentalism is unappealing and unrealistic. Modern research on Diophantine equations often depends on the modularity of elliptic curves [18], which constitutes the final step in the proof of Fermat’s last theorem [69] and relies on an overwhelming amount of theory. Full formalization being completely unrealistic for the project, we must embrace flexible levels of formality [44].

6. 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, verifying 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 lattice theory.

Our goal is to formally prove theorems and do research mathematics while at the same time addressing 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 [12, 10]) 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 foundational, verified fashion. 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 ours. Dr. Blanchette and his team are coming from computer science with support from mathematicians, whereas we are coming from mathematics with support from computer scientists. Moreover, Blanchette 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 Jasmin's inaugural meeting in January 2019 in Amsterdam. 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 [11]) 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 [31].

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. EXAMPLES: LEAN'S RAMIFIED HIERARCHY OF SORTS AND TYPES

```

universes u v
variables A B : Prop
variable  $\alpha$  : Type u
variable  $\beta$  : Type v
variable  $\gamma$  : Type (u+1)

#check A  $\rightarrow$  B          -- Prop
#check A  $\rightarrow$  Prop       -- Type

#check A  $\rightarrow$   $\alpha$        -- Type u
#check  $\alpha$   $\rightarrow$   $\alpha$       -- Type u
#check  $\alpha$   $\rightarrow$   $\beta$       -- Type (max u v)
#check  $\alpha$   $\rightarrow$   $\gamma$       -- Type (u+1)
#check ( $\alpha$   $\rightarrow$   $\beta$ )  $\rightarrow$   $\gamma$  -- Type (max (max u v) (u+1))

#check  $\alpha$   $\rightarrow$  Prop      -- Type u
#check  $\alpha$   $\rightarrow$  Type u   -- Type (u+1)
#check  $\alpha$   $\rightarrow$  Type v   -- Type (max u (v+1))

#check Sort u  $\rightarrow$  Sort u -- Type u
#check Type u  $\rightarrow$  Type u -- Type (u+1)
#check Sort u  $\rightarrow$  Sort v -- Type (max u v)
#check Type u  $\rightarrow$  Type v -- Type (max (u+1) (v+1))

#check Prop          -- Type
#check Prop  $\rightarrow$  Prop -- Type
#check Sort          -- Type
#check Sort  $\rightarrow$  Sort -- Type

#check Type          -- Type 1
#check Type  $\rightarrow$  Type -- Type 1
#check Sort 1        -- Type 1
#check Prop  $\rightarrow$  Type -- Type 1
#check Sort 0  $\rightarrow$  Sort 1 -- Type 1

#check Type 1        -- Type 2
#check Type 1  $\rightarrow$  Prop -- Type 2
#check Type 1  $\rightarrow$  Sort 2 -- Type 2

```

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] Andrej Bauer. Five stages of accepting constructive mathematics. *Bull. Amer. Math. Soc. (N.S.)*, 54(3):481–498, 2017. URL: <https://doi-org.colorado.idm.oclc.org/10.1090/bull/1556>, doi:10.1090/bull/1556.
- [3] Andrej Bauer. On fixed-point theorems in synthetic computability. *Tbilisi Math. J.*, 10(3):167–181, 2017. URL: <https://doi-org.colorado.idm.oclc.org/10.1515/tmj-2017-0107>, doi:10.1515/tmj-2017-0107.
- [4] 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>.
- [5] Andrej Bauer and Jens Blanck. Canonical effective subalgebras of classical algebras as constructive metric completions. *J.UCS*, 16(18):2496–2522, 2010.
- [6] Andrej Bauer, Gordon D. Plotkin, and Dana S. Scott. Cartesian closed categories of separable Scott domains. *Theoret. Comput. Sci.*, 546:17–29, 2014. URL: <https://doi-org.colorado.idm.oclc.org/10.1016/j.tcs.2014.02.042>, doi:10.1016/j.tcs.2014.02.042.
- [7] 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. URL: <https://doi-org.colorado.idm.oclc.org/10.1016/j.entcs.2012.08.002>, doi:10.1016/j.entcs.2012.08.002.
- [8] Clifford Bergman. *Universal algebra*, volume 301 of *Pure and Applied Mathematics (Boca Raton)*. CRC Press, Boca Raton, FL, 2012. Fundamentals and selected topics.
- [9] 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).
- [10] J. C. Blanchette. Lean forward: Usable computer-checked proofs and computations for number theorists. GitHub.com, 2018. URL: <https://lean-forward.github.io/>.
- [11] Venzano 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.
- [12] Mario Carneiro and Johannes Hölzl. Lean mathlib. GitHub.com, 2018. URL: <https://github.com/leanprover/mathlib>.
- [13] P. Chiusano and R. Bjarnason. *Functional Programming in Scala*. Manning Publications, 2014. URL: <https://books.google.com/books?id=bmTRlwEACAAJ>.
- [14] 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](http://dx.doi.org/10.1016/0890-5401(88)90005-3), doi:10.1016/0890-5401(88)90005-3.
- [15] Leonardo de Moura and Nikolaj Bjørner. Efficient E-matching for SMT solvers. In *Automated deduction—CADE-21*, volume 4603 of *Lecture Notes in Comput. Sci.*, pages 183–198. Springer, Berlin, 2007. URL: https://doi-org.colorado.idm.oclc.org/10.1007/978-3-540-73595-3_13, doi:10.1007/978-3-540-73595-3_13.
- [16] J. Fasel. Erratum on “on the number of generators of ideals in polynomial rings”. *Annals of Mathematics*, 186(2):647–648, 2017.
- [17] Eric Finster. Higher algebra in type theory. GitHub.com, 2018. URL: <https://github.com/ericfinster/higher-alg>.

- [18] David Gepner, Rune Haugseng, and Joachim Kock. ∞ -operads as analytic monads. arXiv, 2017. URL: <https://arxiv.org/abs/1712.06469>.
- [19] Georges Gonthier. Formal proof—the four-color theorem. *Notices Amer. Math. Soc.*, 55(11):1382–1393, 2008.
- [20] 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. URL: https://doi-org.colorado.idm.oclc.org/10.1007/978-3-642-39634-2_14, doi:10.1007/978-3-642-39634-2_14.
- [21] Thomas Hales, et al. A formal proof of the Kepler conjecture. *Forum Math. Pi*, 5:e2, 29, 2017. URL: <https://doi-org.colorado.idm.oclc.org/10.1017/fmp.2017.1>, doi:10.1017/fmp.2017.1.
- [22] John Harrison and L. Théry. A skeptic’s approach to combining HOL and Maple. *J. Automat. Reason.*, 21(3):279–294, 1998. URL: <https://doi-org.colorado.idm.oclc.org/10.1023/A:1006023127567>, doi:10.1023/A:1006023127567.
- [23] M. J. H. Heule and O. Kullmann. The science of brute force. *Communications of the ACM*, 60(8):70–79, 2017.
- [24] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory. Lecture Notes. Bibliopolis*, Naples, 1984. Notes by Giovanni Sambin.
- [25] Karl Meinke. Universal algebra in higher types. *Theoret. Comput. Sci.*, 100(2):385–417, 1992. URL: [https://doi-org.colorado.idm.oclc.org/10.1016/0304-3975\(92\)90310-C](https://doi-org.colorado.idm.oclc.org/10.1016/0304-3975(92)90310-C), doi:10.1016/0304-3975(92)90310-C.
- [26] 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](http://dx.doi.org/10.1016/0304-3975(94)00203-U), doi:10.1016/0304-3975(94)00203-U.
- [27] 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.
- [28] J. Rehmeyer. Voevodsky’s mathematical revolution. Scientific American Blogs, 2013. URL: <http://blogs.scientificamerican.com/guest-blog/>.
- [29] 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](http://dx.doi.org/10.1016/0304-3975(93)90095-B), doi:10.1016/0304-3975(93)90095-B.
- [30] 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. URL: https://doi-org.colorado.idm.oclc.org/10.1007/978-3-319-40229-1_8, doi:10.1007/978-3-319-40229-1_8.
- [31] 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.
- [32] Paul Taylor. *Practical foundations of mathematics*, volume 59 of *Cambridge Studies in Advanced Mathematics*. Cambridge University Press, Cambridge, 1999.
- [33] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science (Vol. A): Algorithms and Complexity*. MIT Press, Cambridge, MA, USA, 1990.
- [34] Jan van Leeuwen, editor. *Handbook of Theoretical Computer Science (Vol. B): Formal Models and Semantics*. MIT Press, Cambridge, MA, USA, 1990.
- [35] Dmitriy Zhuk. The proof of CSP dichotomy conjecture. *CoRR arXiv*, abs/1704.01914, 2017. URL: <http://arxiv.org/abs/1704.01914>, arXiv:1704.01914.