Structure and Equality in Type Systems
CNRS Research Proposal

Eric Finster

January 8, 2019

1 Motivation

Among the most ubiquitous and useful tools in the theory of programming languages is that of a *type system*. In its most elementary incarnation, a type system organizes the data manipulated by programs into *types*, allowing the functions and procedures comprising a program to be annotated with information about the kinds of data they consume as input and subsequently produce as output. Analysis of these annotations by a compiler or interpreter then allows for some basic sanity checks about the well-formedness of programs, ensuring, for example that functions are only applied to arguments which are in their domain. What is not at all obvious upon first encountering this simple and natural idea is that, carried to its inevitable conclusion, it leads to a unification of programming, logic and constructive mathematics. This unification takes place through the correspondence between logic and type theory known as *propositions as types*, and it has some strong consequences. A sufficiently strong type system thus serves a dual role, both pratical and theoretical: it is at once a functional programming language suited to the implementation of algorithms, as well as sophisticated mathematical tool for specifying and proving properties said programs.

The past decade has seen an even more startling development in this story: as first observed by Awodey and Warren [5] and independently by the Field's Medal winning mathematician Vladimir Voevodsky, the subtle properties of *equality* in type systems lead to deep connections with a branch of mathematics known as *homotopy theory*. Moreover, the implications of this connection for the treatment of mathematical *structures* prompted Voevodsky to propose that type theory, rather than the current foundations based on set theory, or a hypothetical one based on category theory, is the proper foundation for the mathematics of the 21st century, a point of view he codified in his Univalent Foundations Program. As we will see below, his ideas in many respects complete and clarify the propositions as types paradigm, pointing the way to a new generation of proof assistants and type theories with a more refined view of both equality and structure.

This research proposal seeks to follow up on the implications of these ideas, both as applied in computer science to the design of more sophisticated programing languages and proof assistants, as well as in mathematics, where as we will see, a similar revision in our way of thinking about structure and equality is taking place with the advent of tractable theories of higher categories. I begin with a review of some of the basic ideas motivating recent developments, along the way tracing the twin themes of *structure* and *equality*, themes which I feel will play a major role in the next generation of programming languages and proof assistants.

1.1 Propositions as Types

The source of this rich interplay between programming and logic is known as the *Curry-Howard correspondence*, or more colloquially as the doctrine of *propositions as types* [18]. Though many variants and extensions of this basic idea exist and are well studied in the literature, the orginal and motivating example connects the *logical connectives* of intuitionistic logic (for example, implication $\Rightarrow$, conjunction $\wedge$ and disjunction $\vee$) with corresponding *type constructors* in the simply typed λ-calculus, (function types $\to$, product types $\times$, and sum types $\sqcup$). Under this correspondence, *proofs* of intuitionistic formulae can be recorded as *terms*

of the λ-calculus. The corresponding λ-terms are then seen as programs, computing a value of their return type and under this correspondence, cut elimination of natural deduction style proofs corresponds to the normalization of the associated λ-terms.

An enormous body of work in computer science attests the the fruitfulness of this perspective. By considering other logics, for example, we can ask if they have computational interpretations so that logic can suggest new ideas in programming language theory (Girard's linear logic, is a famous example of this technique). And conversely, we can consider ideas which arise naturally in programming languages and use the Curry-Howard correspondence to extract new logics, whose rules naturally capture ideas of a given problem domain (recent examples are ideas like session types and separation logic).

Propositions as types is also at the center of the connection between functional programming and *category theory*, and this connection has had an enormous impact both practcally in terms of how functional programs are constructed and structured, but also theoretically as categorical methods have come to dominate our understanding of the semantics of both programming languages and logic: indeed, type systems themselves can be regarded as presentations of structured categories.

Finally, the propositions as types paradigm is at the center of the design of the type systems of many popular languages in wide use today such as Ocaml, Haskell, Scala, F# and more. As programmers seek more sophisticated and expressive languages, the proposition as types paradigm sets out natural design principles and leads to more regular and less ambiguous specificaions.

1.2 Structure and Equality

Despite the uncontestable success of the propositions as types paradigm in programming language theory, its restriction to the simply typed case described above has some limitations. As these limitations lead naturally to the themes explored in this proposal, we pause here to describe them. Specifically, the logics corresponding to simply typed languages are not rich enough to described *structured types* nor to support a consistent theory of *equality*. As we will see, modern functional languages take an ad-hoc appraoch to both of these ideas.

To take a specific example, consider the addition function on the natural numbers: $+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$. This function can of course be defined in any modern programming language. And the paradigm of propositions as types gives this function a logical reading: the *assumption* of a pair of natural numbers $(n, m) : \mathbb{N} \times \mathbb{N}$ *implies* the existence of a third: namely, $n + m$. The proof of this statement is the program computing their sum. Mathematically speaking, however, more is true: this operation equips the type $\mathbb{N}$ with the structure of a commutative monoid. And in fact, many statements about addition of natural numbers generalize to the case of any commutative monoid. A natural application of the propositions of types paradigm, then, would be to use the logic it furnishes us with to make and prove statements about this structure. Unfortunately, the language of formulae we obtain from simple types is simply not rich enough to even describe what a commutative monoid is.

On the other hand, this idea of a *structure*, that is, a type or collection of types equipped with operations satisfying some axioms, is pervasive both in mathematics and computer science. One might go so far as to say that mathematics *is* the study of such structures. Similarly, in computer science we find this idea incarnated, for example, in the notion of an *abstract data type*, and it is arguably at the heart of ideas like abstraction and encapsulation in object-oriented programming. The fact is that the idea of structured types arises so naturally that most modern languages provide at least some facilities for defining and manipulating them. In Haskell, for example, such structures are implemented using type-classes; Ocaml uses its system of modules and functors. And of course object-oriented languages implement these ideas using classes and interfaces. But by separating the implementation of structured types from the type system itself, we lose the ability to reason about them, and this, after all, was what the propositions as types should have provided us with.

Moreover, there is a second limitation of the simply typed case: while propositions as types connects logical assertions with types, in the simply typed case, there is no type corresponding to one of the most important assertions of all, namely, the assertion that two terms are *equal*. As a consequence, our simply typed logics are too weak to support equational reasoning. In practice, this means that most languages,

though they may provide support for structures in other ways, completely ignore the axioms that these structures must satsify, leaving them to the programmer to verify separately. Worse, in most languages, equality *itself* is left to be implemented in an ad-hoc manner: the programmer is responsible for implementing his own notion of equality for each newly introduced type. Misunderstandings, misuses and incorrectly specified notions of equality are at the heart of innumerable software bugs and inconsistencies. In situations where formally verified software is important, leaving a notion as fundamental as equality to be implemented in such a way is clearly unreasonable.

1.3 Dependent Types

Both of these limitations can be to a certain extent overcome by passing to *dependent type systems*. Anticipated already by Howard in describing the correspondence which bears his name, this project was taken up by the Swedish logician Per Martin-Löf who described the first examples of such systems. If types correspond to propositions, then *dependent* types correspond to *predicates*, and the passage from simple to dependent types is analagous to the passage from propositional logic to predicate logic. Equipped with a much richer language of types, in a dependent type system, we can state and prove essentially *arbitrary* statements about programs.

A fundamental idea in dependent type theory is the existence of a *universe* of types, which I will denote Type, and whose inhabitants are themselves types [1]. Given a type X, a dependent type can then be modeled as a function $P : X \to \mathsf{Type}$, that is, it is an assignment to every element $x : X$ of a type $P\,x$ so that P may be seen as a family of types varying over the elements of X. From here, one can introduce the dependent sum $\sum_{x:X} P\,x$, whose elemets are pairs (x, p) where $x : X$ and $p : Px$, and the dependent product $\prod_{x:X} P\,x$ whose elements are λ-terms $\lambda x.p(x)$ where $p(x) : P\,x$. These two constructions will serve as the existential and universal quantifiers in our enriched logic of types.

Finally, the introduction of type dependency, by which terms may themselves appear in types, allows for the introduction of the *identity type*. That is, for elements $x, y : X$ we suppose the existence of a new type $\mathsf{Id}_X\,x\,y$ whose inhabitants are proofs that x and y are equal. Notice how this type is necessarily dependent on the terms we are comparing, which explains its absense in a simple type systems. It was Martin-Löf who first introduced these types, explaining their introduction and elimination rules.

Armed with these innovations, we can now start to describe structured types in the sense of the last section. For example, we might consider a basic structure: that of a type X equipped with an endomorphism $\alpha : X \to X$. In dependent type theory, we can use the dependent sum to write

$$\mathsf{Endo} := \sum_{X:\mathsf{Type}} (X \to X)$$

so that inhabitants of this type are exactly *pairs* as desired. Notice how this definition internalizes this notion of structure in the sense that there is a *type* of "types equipped with an endomorphism". Hence dependent types allow use to integrate the notion of structure into the type system itself and subsequently reason about these structures.

In principle, all such structures can be expressed as iterated dependent sums, but for more sophisticated examples, it will be convenient, in order to reduce clutter, to express them as dependent record types (of which Σ can be see as a special case). Figure 1 uses this method to define one of the most useful structures in computer science, that of a *category*. Notice in particular the use of the identity type to specify the axioms of the structure. It may be surprising then that this direct translation of the notion of a category from mathematics turns out *not* to be the correct definition in general, an idea we will return to presently.

Equipped with the notion of type dependency, and a workable theory of equality, the propositions as types paradigm reaches its full potential. We now have a language in which we can not only write programs, but simultaneously a logic in which we can make, and prove, essentially arbitrary statements about the behavior of our programs. Moreover, we can define various notions of structured types, and explore the properties which follow from the axioms of our structure, thus providing a powerful means of abstraction.

[1] Universes must, of course, be stratified by size to avoid paradox, but I will ignore these complications here.

$$
\begin{aligned}
&\mathsf{record}\ Category :=\\
&\quad Ob : \mathsf{Type} \qquad\qquad Hom : Ob \to Ob \to \mathsf{Type}\\
&\quad \_\circ\_ : \prod_{x,y,z:Ob} Hom\, y\, z \to Hom\, x\, y \to Hom\, x\, z \qquad id : \prod_{x:Ob} Hom\, x\, x\\
&\quad unit\text{-}l : \prod_{x,y:Ob} \prod_{f:Hom\, x\, y} \mathsf{Id}_{Hom\, x\, y}\, (f \circ id\, x)\, f \qquad unit\text{-}r : \prod_{x,y:Ob} \prod_{f:Hom\, x\, y} \mathsf{Id}_{Hom\, x\, y}\, (id\, y \circ f)\, f\\
&\quad assoc : \prod_{x,y,z,w:Ob} \prod_{f:Hom\, x\, y} \prod_{g:Hom\, y\, z} \prod_{h:Hom\, z\, w} \mathsf{Id}_{Hom\, x\, w}(h \circ (g \circ f))\, ((h \circ g) \circ f)
\end{aligned}
$$

Figure 1: Naive definition of a category

This idea is at the heart of proof assistants such as Coq, Agda and Lean as well as the dependently typed programming language Idris. These systems have enjoyed spectacular successes, both in the formalization of highly non-trivial theorems such as the Feit-Thompson theorem and Four-color theorem, as well as in formal methods as applied in computer science as witnessed by the CompCert certified C compiler.

1.4 Univalent Type Theory

At first sight, it may seem that the introduction of dependent types has solved our problem: we now have a type system rich enough to describe arbitrary structures and prove facts about them, with these facts implemented as programs, just as one might expect from the propositions as types paradigm. But the dependent type system described above turns out to have some strange features, particularly with respect to the treatment of equality. And as we will see, this realization necessitates a serious revision in the way we think about structured types.

A first thing to notice in this regard is that, the formation rule for the identity type means that it can be *iterated*. Indeed, if elements $p, q : \mathsf{Id}_A\, x\, y$ represents proofs that the terms x and y are equal, then an element $\alpha : \mathsf{Id}_{\mathsf{Id}_A\, x\, y}\, p\, q$ is a proof that these two *proofs* are equal. On the other hand, the introduction rule for the identity type essentially says that there is only one way to inhabit it: by the unique term refl when x and y are definitionally equal. One might be justified, therefore, in thinking that these *higher* identity types must all be trivial. This turns out to be far from the case. It was first shown by Hoffman and Streicher using a model of type theory in *groupoids* that it was indeed possible to have non-trivial higher equalities, so that at least their triviality could not be proven it type theory itself. They also suggested that more sophisticated models could be obtained by thinking about the objects mathematicians know as *weak higher groupoids*. Later, Awodey and Warren noticed that the identity elimination rules made them into *path objects* in a structure known to topologists as a Quillen model category, thus uncovering a link with homotopy theory.

But it was with his introduction of Univalent Foundations that Voevodsky finally realized the implications of these ideas: the types of dependent type theory should be though of as constructive *spaces* and not, as evisioned by Martin Löf as constructive *sets*. The elements of their identity types, then, can be imagined as paths, paths between paths, ... and so on. This allows one to apply ideas coming from topology to the study of types.

Moreover, Voevodsky's insights have serious ramifications for the propositions as types paradigm we have been examining. To see why, first we introduce the notion of a *contractible* type, which is defined by the following pleasant formula combining the three essential type constructors of dependent type theory:

$$\mathsf{is\text{-}contr} X := \sum_{x:X} \prod_{y:X} \mathsf{Id}_X\, x\, y$$

We can now use this definition to control the complexity of the hierarchy of iterated identity types described above by introducing the notion of the h-level of a type. The definition is given by induction (which for

historical reasons, it is convenient to start at -2):

$$\mathsf{is\text{-}of\text{-}level}\, X\, (-2) := \mathsf{is\text{-}contr} X$$
$$\mathsf{is\text{-}of\text{-}level}\, X\, (S\, n) := \prod_{x,y:X} \mathsf{is\text{-}of\text{-}level}\, (\mathsf{Id}_X\, x\, y)\, n$$

Intuitively speaking, then, a type is of h-level n for $n \geq -2$ if its identity types become contractible after $n+2$ iterations.

Examining the first few cases of this definition is quite instructive. The case $n = -2$ conicides with contractibility by definition, but it is nonetheless worth pointing out that if a type is of level n, then it is also of level $n+k$ for any $k \geq 0$. In particular, *all* of the higher path types of a contractible type are themselves contractible. In other words, in a contractible type, not only is every element equal to a given element, but any two witnesses of these equalities are themselves equal and so on. In this respect, contractibility is a strong form of uniqueness which propagates throughout the tower of higher identity types.

Types of level -1 have the property that, if they are inhabited, then any two inhabitants are necessarily equal, and in a unique way. [2]. We thus say that such types are *propositions* and they play the role of *truth values*. If the propositions as types paradigm started out as an analogy between logic and types, the univalent persepctive lifts the analogy to a formal statement: logic is a *special case* of type theory, and this special case can be distinguished, in the present of equality, inside of type theory itself.

The next two levels are also familiar objects: types of level 0 are *sets*. Their defining property is that, if two elements x and y of a set are equal, then they are equal in a unique way. Otherwise stated, the sets are exactly those types for which being equal is a *proposition* in the sense just introduced. This is why we can think of elements of a set as having no internal structure and why, in classical mathematics where all objects are sets, we do not typically worry about whether "two proofs that two elements are equal are themselves equal". In a theory consisting only of sets, this question never arises.

Finally, types of level 1 correspond to the mathematical notion of a *groupoid*. In a groupoid, between any two elements x and y, we have a *set* of ways in which they might be equal. In particular, any element x in a groupoid has a possibly non-trivial set of automorphisms, and for this reason, we can imagine groupoids as collections of objects with symmetries. In particular, the theory of groups, so central in classical mathematics, is recovered by the theory of *connected* 1-types. It is remarkable that elementary considerations in computer science lead naturally to this enrichment of the basic vocabulary of mathematics.

The existence of this tower of h-levels is inspired by a similar construction in topology (the *Postnikov* tower of a spce), showing how intuitions from this subject can now be applied to the study of types. Indeed, I was a participant in the year long program at the Institute for Advanced Study in Princeton which was organized to investigate these ideas. It should be noted that it was not immediately clear at the outset that types would in fact bahave in a way consistent with their interpretation as homotopy types, as only the rudiments of the theory had been developed at that time. By now, however, there is no longer any doubt: an amazing number of sophisticated ideas and constructions from topology have now been applied to the types of type theory. The connection is indeed so close that it is probably not an exaggeration to say that the *constructive theory of equality* and *homotopy theory* are essentially one and the same.

1.5 The Coherence Problem

While the univalent perspective greatly clarifies the behavior of equality in dependent type theory, it unfortunately complicates the situation for describing structured types. The problem is well known in the literature on homotopy theory and higher category theory, and can be summed up as follows: from our experience with set theoretic foundations, we are accustomed to specifying structures as consisting of sets equipped with operations. We then impose axioms on these operations in the form of equations. But in the presence of higher equalities, we must also specify equations *among the equations*. And then further equations among these new equations, and so on, infintely often. In short, to have well behaved structured

[2] Classically, we would say the such types are *either* empty, *or* contractible, which can be a helpful intuition. Intuitionistically, however, the provided description is more accurate.

types in the presence of higher equalities, it is not sufficient to explain the equations at the next level, which is to say using just a single application of the identity type to the types involved. Rather we must explain the equations of our structure throughout *the whole tower of higher equalities*.

Far from being a mere theoretical curiosity, this problem has direct consequences for the axiomatization of structures inside dependent type theory. To illustrate, recall the naive definition of category given in Figure 1. On its face, this appears to be a perfectly natural translation of the definition of category into type theory. But as we begin developing the theory, problems start to arise. Consider, for example, the problem of defining one of the most elementary constructions of category theory: the *slice category*. For this construction, we fix a category C and an object $x : Ob\, C$. The objects and morphisms of the new category C/x can be defined as

$$Ob\,(C/x) := \sum_{y:Ob\,C} \sum_{f:Hom\,y\,x}$$
$$Hom\,(C/x)\,(y,f)\,(z,g) := \sum_{h:Hom\,y\,z} \mathsf{Id}_{Hom\,y\,x}\,(g \circ h)\,f$$

In other words: objects are morphisms of C with codomain x and morphisms are commutative triangles. Notice how the appearence of the identity type means that that the definition carries a *witness* that the triangle is commutative.

Now, when we attempt to define a category structure with the above definitions, things begin smoothly: we find that we can define composition, but that the definition uses the associative law of the original category C. When we try to show that this new composition is associative, we find that we cannot succeed. What is missing is exactly the information that the various proofs of associativity are *compatible with each other*. It is well known that a sufficient condition for this to be the case is MacLane's pentagon identity (Figure 2), which relates two *different* ways of reassociating a composite of four morphisms. Working out the type of this equation shows that it lives in a doubly iterated identity type (we say that it is a 2-*cell*).

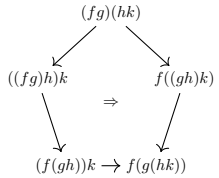

Figure 2: Pentagon

We could try adding this axiom as part of the definition of category, but then we would be required to prove it during the construction of the slice category. And here again, we would find that we needed a compatibility between the various instances of the pentagon identity, which itself would be a 3-cell, living in a triply iterated identity type. This quickly leads to an infinite regress: we find that even just to perform the elementary construction of the slice category, our definition of category requires infinitely many equations, arbitrarily high in the tower of identity types. One way around this problem is to assume that all the types involved are sets. [3] In this case, any extra "coherence conditions" are automatically satisfied since the higher identity types are contractible. But this simply avoids the problem rather than solving it. The question remains: what is the *correct* definition of a category structure on an arbitrary type?

This problem is characteristic of structures in the presence of a proof relevant equality such as we find in dependent type theory. To illustrate the prevalence of the problem, consider an example from functional programming. Recall that a common way of organizing functional programs is via the structure of a *monad*. A monad is a type constructor equipped with two operations, often called *bind* and *return*. Moreover, these

[3] Indeed, this is what is done, either explicitly or implicitly, in all the formalizations of category theory in type theory which exist today. A common way of accomplishing this before the introduction of h-levels was to assume the types supported a decidable equality, which implies they are sets.

operations are required to satisfy some equations referred to as the *monad laws* or *triangle identities*. Now suppose we wanted to reason formally about a functional program which employed some form of monadic programming. We would naturally need, at some point, to show that the type constructor was, indeed, a monad. In other words, we would need to verify the monad laws. But here is the problem: generalized to arbirary types, the traditional monad laws *are not complete*, for the same reason that the category laws discussed above are not. Indeed, until recently (see Section 2) the correct definition of monad in a proof-relevant setting *was not even known.* We hope the reader will agree that this is not a satisfactory situation for formal software verification.

Luckily, similar problems are well-studied in mathematics, and a theory of these structures is already known in that setting. The correct definition of category in a proof relevant setting is what mathematicians call an $(\infty, 1)$-category. I cannot here explain how mathematicians deal with the problem, but the reader can get an intuition for what such an object is from the discussion above: it is what arises if we replace the hom *sets* in the definition of category with hom *types* and then specify all the higher equations controlling associativity throughout the tower of identity types.

1.6 Summary

The doctrine of propositions as types connects types and logic, suggesting a way to reason about programs using type theory itself. The introduction of dependent types furnishes us with a language rich enough to describe structured types and reason about them equationally using the identity type. These capabilities play a crucial role in the pursuit of formally verified software.

But the identity type of Martin-Löf turns out to be significantly more complicated than might first be expected, and in order to have better control of dependently typed languages and their capabilities, we need to understand its behavior, including its deep connections with homotopy theory. The perspective offered by univalent type theory gives us a way to understand and predict how this equality should behave, and clarifies the connection between logic and type theory, while at the same time uncovering subtle problems with the most naive notions of structured types.

The design and implementation of the next generation of proof assistants and strongly typed languages will surely require a deeper understanding of these phenomena and their interactions with mathematics.

2 Project

In view of the ideas introduced above, my reasearch proposal seeks to continue to investigate the properties of equality and structure, not only in type theory, but more generally as these topics appear in computer science.

2.1 Coherent Structures in Type Theory

As we have seen, correctly describing structured types in the presence of a proof relevant equality is significantly more subtle than might first have been imagined. Indeed, it was these considerations that led Voevodsky already in 2013 to propose a new type system, reintroducing a "strict" equality, which enabled him to get around this problem. Variations on this idea such as [4] have also been proposed. It had been widely believe for some time that a modification of type theory of some sort was necessary to allow to properly deal with coherent structures.

This turns out not to be the case. Recently, I showed how one can in fact solve this problem in ordinary dependent type theory by introducing a definition of a *polynomial monad*. [4] This definition has been fully formalized [8], meaning that we now have a complete definition of coherent structure in a number of special cases. To date, it is the only such definition available in type theory. There are a number of immediate and important application of these ideas, which I will now describe.

[4] Unfortunately, as this work is quite recent, it has not had time to appear in print. Online descriptions are available, however, as notes [9] and a video lecture [11].

2.1.1 Polynomial Monads and Their Algebras

While I cannot describe fully the solution to this problem in this proposal, a couple of remarks are in order to motivate the projects which will follow. The overall approach is inspired by the work of Baez-Dolan on the categorification of algebraic structures [6]. Their point of departure is a structure known in mathematics as a symmetric operad. Recent developments [13] have shown that, in type theory, this notion becomes equivalent to that of a *polynomial monad* in the sense of [12], and hence I will freely apply their ideas to this closely related structure. As polynomial functors are well studied in the computer science literature (where they are commonly employed in the study of the semantics of inductive types) this shows that the Baez-Dolan approach is closely connected with the theory of inductive definitions and hence naturally adapted to type theory.

The main innovation of Baez-Dolan is the introduction of what they call the *plus construction*, which, starting from a polynomial monad, produces a new polynomial monad whose multiplication can be thought of as encoding the *relations* present in the original. Intuitively speaking, we can think of these relations as living "one dimension higher". Iterating this construction generates an infinite tower of monads, each of whose multiplication witnesses the associativity and unicity of the previous monad. That is, the monad *laws* at one level are encoded by the monad *multiplication* at the next. The key insight needed to adapt this approach to type theory is to recognize that the existence of this infinite tower of derived monads can be taken as the *definition* of what it means for the original monad to be coherent. Moreover, an important aspect of this definition is that special cases recover structures of crucial importance: $(\infty, 1)$-operads, $(\infty, 1)$-categories and ∞-groupoids can all be seen as examples of polynomial monads.

Recall from categorical algebra that a monad can be seen as a way of encoding a particular kind of algebraic structure. Hence it is quite convenient that a coherent notion of polynomial monad can be defined in type theory, since this means that we can take this as *definition* of what a coherent structure on types is: to define a coherent structure on types is to define a polynomial monad. The *models* of the structure so defined are then the *algebras* for the monad in question.

As mentioned above, the definition of polynomial monad has already been implemented in dependent type theory, showing its feasibility. The definition of algebra, on the other hand, remains to be finished. Thus an immediate short term goal is:

Goal 1 *Finish the formalization of the definition of coherent algebra over a polynomial monad.*

Among all the polynomial monads, there is one which is distinguished: the *terminal monad*. As it happens, the terminal example of a polynomial monad turns out to be quite interesting: it is the universe Type of type theory itself, equipped with the operation of dependent sum $\sum$. In some respects this structure is more fundamental than the $(\infty, 1)$-category structure on the universe in which the morphims are functions between types, though the two structures are closely related. As it is the terminal example, one might be tempted to think that it should be quite easy to construct. But as the construction of any polynomial monad requires the specification of an infinite number of coherence conditions, things are not quite so simple. I have already what looks like a promising approach based on Voevodsky's univalence axiom, but the details remain to be fully checked.

Goal 2 *Construct the monad structure on* Type *and show that it is the terminal example. Use this to show that* Type *is an internal* $(\infty, 1)$*-category.*

2.1.2 The Groupoid Structure on Types

We have see that the tower of identity types endows types with an extremely rich algebraic structure known in mathematics as as an ∞-groupoid. Roughly, one can think of these objects as what is left of a topological space after one removes all the information about "nearness" of points, retaining only the algebraic structure of composition which exists on paths, paths between paths, and so on. These structures are notoriously difficult to define rigorously, even in classical mathematics. It has been a long standing open problem to

describe these structures internal to type theory. In other words, to *characterize* inside of type theory itself, the structure that one finds on types.

As it happens, a special case of the definition of polymonial monad is in fact a definition of ∞-groupoid. As the model one naturally obtains from this definition is based on the geometry of opetopes, I will refer to these as *opetopic ∞-groupoids* in what follows. One is thus led naturally to the following conjecure:

Conjecture 1 *The type of opetopic ∞-groupoids is equivalent to* Type.

In other words, every type admits the structure of an ∞-groupoid internally, and that structure is unique. In fact, there is ample evidence for this conjecture, and I believe a clear path to proving it. Moreover, a proof of this conjecture would be an indication that we had thoroughly understood how to handle infinite towers of coherence, and could describe explicitly all the operations one has in the tower of higher equalities on a type.

2.1.3 Simplicial Types and Internal Kan Complexes

A long standing open problem in type theory has been the definition of *simplicial types*. In fact, a special case of the definition of polynomial monad is that of an ∞-category. An algebra over such a monad turns out to be exactly a Type-valued diagram on the category. This leads to a resolution of the problem of defining simplicial types: a simplicial type is an algebra for the category Δ, regarded as a monad with only unary operations. A formalization of this definition is already underway, but not yet complete.

Goal 3 *Finish the formalization of simplicial types.*

As a result, it should be possible to internalize many of the simplicial techniques most commonly used for the definition coherent structures in mathematics. For example, a well known definition of ∞-groupoids starting from simplicial types asks that they satisfy the *Kan lifting condition*. Repeating this definition in type theory presents no theoretical challenge once a definition of simplicial type is established (indeed, this was the most commonly proposed plan for defining coherent objects in the first place.) Again, one has a natural conjecture for how these objects behave:

Conjecture 2 *The type of internal Kan complexes is equivalent to* Type.

Observe that, the combination of Conjectures 1 and 2 would give a proof of the following:

Corollary 1 *The type of Kan complexes and the type of opetopic ∞-groupoids are equivalent.*

Note that this statement is not known in the mathematical literature, but it seems quite likely that it can be demonstrated using the techniques I have outlined. And in fact, this example illustrates a more general point: there are many variations (cubical, globular, cellular, etc.) on the definition of ∞-groupoid in the literature. In type theory, however, we have a *canonical* definition: an ∞-groupoid is just a type. Hence all proposed definitions of ∞-groupoid can be tested in this way: the definition is coherent exactly when one can show that the type of its models is equivalent to Type. This may not immediately seem useful, but in fact it is an important sanity check: often we are interested in structures which reduce to ∞-groupoids in a special case (as with the case of polynomial monads) and constructing such an equivalence is good evidence that the definition is correct.

2.1.4 $(\infty, 1)$-Category Theory

Higher category theory itself has seen something of a revolution in the past decade owing to the widespread adoption of the theory of *quasicategories* developed by André Joyal and Jacob Lurie, who showed that essentially all the constructions of ordinary category theory could be extended to this model of $(\infty, 1)$-categories, occassionally with some modifications or clarifications. These results have been hugely influential in homotopy theory and algebraic geometry. With a working definition of $(\infty, 1)$-category in type theory, it now becomes possible to start developing this theory. Hence:

Goal 4 *Develop the theory of* $(\infty, 1)$*-categories in type theory.*

Of course category theory is a huge subject, so this is a long term project which is likely to take years of development and even in that case, remain relatively open-ended. Nevertheless, even the basic definitions and fundamental constructions like limits and colimits, slice categories, categories of functors and so on would significantly expand the possibilities for what we can formalize in type theory. Putting these definitions in place using the techniques described above is an important medium-term goal in the study of structures in type theory.

2.1.5 Internal Semantics of Type Theory

Another major application of the theory of polynomial monads is to the study of the semantics of type theory itself. The fact is that, prior to this theory, the only method available to construct models of type theory was to pass via either (constructive) set theory, or else 1-category theory, which implicitly employs set-truncated objects. This is the reason that much of the meta-theory of type theory relies on "strictification" results: in these models, we must quotient the syntax of type theory by a set-level equivalence relation in order to interpret it in a set-level model.

On the other hand, the univalent perspective suggests that, these set-truncated models are not the most natural: after all, type theory is a theory of *types*, which are more general than sets. Indeed, it is an often repeated mantra that type theory should serve as an *internal language* for $(\infty, 1)$-categories. But this statement is difficult to make precise exactly because, in classical mathematics, our only way of accessing the theory of $(\infty, 1)$-categories is to model them as set theoretic structures. This is a crucial point: the *only* theory we know of which can *directly* make sense of higher structures axiomatically is type theory itself! Hence we should expect that the natural home for describing the semantics of type theory is an internal theory of categories built on types, and not on sets.

In fact, many attempts have been made to give a semantics of type theory in type theory [2] [7] [17], and many partial results have been obtained. On the other hand, none is completely satisfactory: after defining type theory internally, it should be possible to show that Type itself is a model of the theory, and this has faced coherence problems of the kind described in the introduction. In retrospect, this is not surprising: internally, Type is *not* a 1-category, it is an $(\infty, 1)$-category, and this mismatch is the source of the coherence problems.

Now that an internal definition of $(\infty, 1)$-category is available, it seems entirely reasonable to expect to make progress on this problem. Indeed, it has been speculated that a way forward would be to generalize a well known tool in the categorical semantics of type theory, that of a *category with families*, to the case of an $(\infty, 1)$*-category with families*. Using the theory of polynomial monads and their algebras, such a definition is clearly possible. Viewed as a formalization project in type theory, working out the details of such a construction is likely to take some time, but it seems clear that we should be able to construct fully weak internal models of type theory within the next couple of years.

Goal 5 *Give a definition of* $(\infty, 1)$*-category with families. Use this to describe an internal semantics of type theory.*

Furthermore, an extremely well known and flexible class of models of type theory are *sheaf models*. Using sheaf models we can typically show results such as strong normalization and canonicity, as well as prove independence results by constructing models which violate various principles. Again, these models are typically done at the level of sets, exactly because there has previously been no reasonable definition of an internal category of Type-valued sheaves. We now have a clear path to developing just such a definition.

Goal 6 *Develop internal sheaf models of type theory. Use this to prove normalization and canonicity results in type theory without any set-truncation hypotheses.*

2.1.6 Higher Inductive Types

As remarked above, there are close connections between the theory of polynomial monads and the theory of inductive types. Indeed, a well-known semantic description of inductive types realizes them as *initial algebras* for polynomial functors. But an equivalent characterization is to say that they are initial algebras (in the monadic sense) for the free monad *generated* by a polynomial functor. This is consistent with the point of view that monads describe algebraic structure: in this case, its initial algebra is the *syntactic model*, and we can think of inductive definitions in type theory as syntactic models of algebraic theories generated by the type's constructors.

Higher inductive types generalize inductive types by allowing us to add *equations* to the types we are defining. That is, a specification of a higher inductive type gives not only constructors of the type itself, but can also provide contructors for elements in the *identity type* of the type being defined. As described in [14], a natural semantics of higher inductive types is to view them as initial algebras for *non-free* monads. Given an internal description of general polynomial monads as we have been discussing, we expect to be able to describe a semantics of higher inductive types inside type theory, similar to the way we can describe ordinary inductive types internally using the theory of *containers* [1]. A more thorough understanding of the semantics of higher inductive types could have very real consequences for the design of the next generation of proof-assistants, which one would hope could support such definitions natively.

Goal 7 *Develop an internal semantics of higher inductive types using the theory of polynomial monads.*

2.2 Synthetic Homotopy Theory

While the previous section dealt with understanding how to define and manipulate structures in the presence of higher equalities, studying the properties of higher equalities themselves is also important for a deep understanding of dependent type theories. Moreover, as we have argued, the properties of higher equalities are already quite well studied in the branch of mathematics known as homotopy theory. Hence we can regard the project of *synthetic homotopy theory* as an exercise in probing the properties of proof-relevant equality. Here we outline various projects connected with this goal.

2.2.1 Loop Spaces

A classic question in homotopy theory is the following: when is a space X homotopy equivalent to the space of *loops* on some other space Y? This question has a direct translation in type theory:

Problem 1 *Find conditions on a type X sufficient to construct a type Y and an element $y : Y$ such that*

$$X \simeq \mathsf{Id}_Y\, y\, y$$

In other words, the type theoretic question asks: when is a type equivalent to an identity type?

This question has a well known answer in topology: the key observation is that if X is equivalent to a loop space, then it must admit a multiplication operation $\mu : X \times X \to X$. This is because loops can be *composed* (and type theoretically, because equality is *transitive*). Moreover, as we always have a constant loop, and since loops can be inverted, this multiplication should make X into a *group up to homotopy*. The delicate question, then, is to make sense of what one means by "group up to homotopy". The traditional answer to this question passes by the theory of *operads* (in fact, operads were *invented* to answer this question.).

As we have pointed out, the theory of $(\infty, 1)$-operads is a special case of the theory of polynomial monads. And in fact, the operad which describes associative multiplicative structures, known as the A_∞-operad, turns out to be relatively easy to construct (in computer science terms, it is exactly the List monad). An algebra over this monad is a type with an *infinitely coherent associative multiplication*. One says the algebra is *group like* if, in addition, all its elements are invertible under multiplication. This motivates the following type-theoretic translation of the classical theorem from topology:

Conjecture 3 *The type of group-like A_∞-algebras is equivalent to the type of pointed, connected types.*

It is important to note that, if we use the naive translation of the notion of a group, specifying only the associativity axiom, but leaving out the higher axioms, then this theorem is *false*. This is another example of the difference between the naive notions of structure we obtain by analogy with set theory, and the coherent structures which are correctly behaved in a proof relevant setting.

2.2.2 Iterated Loop Spaces

The question of the previous section has a natural generalization: given a type X, we can ask under what conditions we can construct a sequence $(Y_1, y_1), (Y_2, y_2), \ldots, (Y_n, y_n)$ of pointed types such that $X \simeq \mathsf{Id}_{Y_1}\, y_1\, y_1$ and for each i we have $Y_i \simeq \mathsf{Id}_{Y_{i+1}}\, y_{i+1}\, y_{i+1}$. In other words, when is X and *n-fold* identity type?

This question also has an answer in topology: it is related to the *commutativity* of the multiplication of the previous section. Here, one meets a well studied phenemenon in homotopy theory, which has a direct analog in type theory: unlike in set theory, in the presence of a proof relevant equality, there is an infinite tower of progressively stronger "notions of commutativity" which interpolate between merely associative (i.e., not at all commutative) and being *infinitely commutative* which corresponds to the notion we known in set theory.

This tower of increasingly commutative structures is controlled by a sequence of operads known as the E_n-operads. That is there is a sequence

$$A_\infty = E_1, E_2, E_3, \ldots, E_\infty$$

of operads beginning with the A_∞ operad of the previous section. Moreover, the answer to the question of when a space is an n-fold loop space is exactly that it admits the structure of a group-like E_n-algebra.

The construction of the E_∞ operad turns out to be quite straightforward. (In fact, it can be seen as the sub-operad of the terminal opeard, i.e. Type, consisting of those types where are merely equivalent to a finite set). The construction of the E_n operads for $1 < n < \infty$ is considerably more difficult. A definition by induction is given by the relation $E_{n+1} = E_1 \otimes E_n$ where $\otimes$ is the *Boardmann-Vogt tensor product* of operads.

Goal 8 *Define the Bordmann-Vogt tensor product, thus giving a construction of the E_n-operads. Show that a type is an n-fold identity type if and only if it is a group-like algebra over the operad E_n.*

We hasten to point out that, while the proof of the above theorems will undoubtly rely on the univalence axiom, the *phenomenon* that there exists a sequence of increasingly subtle notions of commutativity in the presence of a proof relevant equality is completely independent of this axiom: it is simply a property of the identity type Id_X, whether or not we use the univalence axiom to attempt to characterize it. It is in this sense that I feel topology can make contributions in computer science: without the intuition provided by the univalent perspective, it is likely that this phenomenon would have continued to go unnoticed. But it is in fact a fundamental property of equality in type theory. Indeed, the above theorem describes *exactly* what structure higher identity types carry.

2.3 Type Theory and Higher Topos Theory

In parallel with the development of univalent type theory in the past decade, mathematicians have developed a generalization of the notion of *topos* familiar from ordinary category theory. These two subjects are closely linked, and it is widely believed that one can view type theory as the internal language of a higher topos. Thus the simultaeous investigation of the theory of higher topoi is intimately related with out understanding of type theory itself. In collaboration with Mathieu Anel, Georg Biedermann and André Joyal, we have been investigating various questions in the theory of higher topoi, and our results have direct applications to type theory.

2.3.1 Left Exact Modalities

In the classical theory of topoi, the notion of *subtopos* is controlled by a *Lawvere-Tieney topology*. In the theory of higher topoi, that role is played by the notion of a *left-exact modality* in the sense of [15]. We recently proved a generation result, showing how *any* dependent family $P : X \to \mathsf{Type}$ can be used to generate a left exact modality in type theory. This opens the door to a number of useful constructions, making a formalization of this result desirable.

Goal 9 *Formalize the generation of left exact modalities.*

2.3.2 Goodwillie Calculus

The Goodwillie Calculus is an advanced technique and organizing principle in classical homotopy theory. It seems that this theory plays an intergral role in understanding the structure of higher topoi. Moreover, it has a direct and simple explanation in type theory in terms of modalities. A large portion of this theory can be developed internally using our techniques for generating left-exact modalities combined with the *Generalized Blakers-Massey Theorem* of [3]. In fact, the logical point of view on this subject appears nowhere else, and is unknown to most topologists, meaning that this is a real opportunity for type theory to contribute original theorems to mathematics.

Goal 10 *Develop the type-theoretic perspective on Goodwillie Calculus.*

2.3.3 Spectra and Linear Dependent Type Theory

In homotopy theory, the notion of a *specturm* plays a similar role to that of a *chain complex* in algebra, and these objects turn out to be the natural home for *(co)homology* theories. Spectra have already been defined in type theory and been used to construct *spectral sequences*, a method of calculating topological invariants of types. In fact, most modern homotopy theory is concerned with the properties of spectra, as opposed to spaces themselves, as these objects turn out to be considerably more tractable.

One consequence of the connection between the Goodwillie Calculus of the previous section and higher topos theory is that the category of *parameterized spectra* is in fact itself a higher topos. This means that we expect it to support a model of type theory, and we can ask what kind of extra principles it satisfies. Our work on the Goodwillie calculus provides exactly such a description, and it is possible to write down an extension of type theory which supports, in addition to the synthetic homotopy theory of ordinary type theory, a version of synthetic *stable homotopy theory* which allows one to formally manipulate spectra and their associated (co)homological invariants.

One way of interpreting spectra is that they are like *linearized* homotopy types, and indeed there is a close connection with linear logic. In fact, it seems that the topos of parameterized spectra is probably a natural semantic model for the notion of dependent linear type theory, and presents a real opportunity for extending some of the ideas coming from univalent type theory to linear logic. As of today, there are several proposals for this kind of type theory in the literature, and having this example in mind as a possible semantics maybe a useful guide to investigating their properties.

Goal 11 *Describe the type theory of parameterized spectra and make precise the link with linear logic.*

2.4 Opetopes in Computer Science

Recall that the definition of polynomial monad discussed above is based on an adaptation of Baez and Dolan's plus construction for polynomial monads. As we have already pointed out, polynomial monads are well known in computer science where they can be interpreted as *type constructors* or *data types*. Perhaps the most familiar is the List monad (which, as we have mentioned, is the type theoretic incarnation of the A_∞-operad). The Baez-Dolan plus construction then has the following somewhat surprising consequence: the List monad is first in an infinite sequence of polynomial monads generated by the plus construction. In fact, the List monad itself is obtained by applying the plus construction to Ident, the *identity monad*.

The reason this sequence is perhaps not so-well known in the theory of functional programming is that the remaining monads are all *indexed* in a non-trivial way, which is to say they are not endofunctors of Type but rather endofunctors of the slice category $I \to \mathsf{Type}$ for a specific type I. Hence the remaining monads in the sequence only appear once we have dependent types. For example, applying the Baez-Dolan plus construction to List gives the type constructor of *planar trees*, but regarded as a type constructor $\mathsf{Tree} : (\mathbb{N} \to \mathsf{Type}) \to (\mathbb{N} \to \mathsf{Type})$. Applied to a type family $X : \mathbb{N} \to \mathsf{Type}$, and natural number $n : \mathbb{N}$, the type $\mathsf{Tree}\, X\, n$ can be desribed as the type of planar trees with n leaves whose internal nodes are decorated by X and subject to the constraint that a node with k descendants is decorated by an element of the type $X\, k$.

In general, the type constructors obtained in this way can be regarded as *higher dimensional trees*. Baez and Dolan gave them the name *opetopes*. Moreover, it turns out that there is a natural diagrammatic way to represent these higher tree-like structures. And while a formal definition of these objects is somewhat difficult (but follows from the definition of polynomial monad discussed above), their close connection to inductive types makes them quite amenable to implementation in functional programming languages. I have developed an extensive library of routines for manipulating these objects, and I feel that they deserve to be more widely known in computer science.

2.4.1 Opetopic Proof Assistant

One of the interesting applications of the fact that opetopes admit a simple diagrammatic notation is that one can use them to design a graphical proof assistant for a definition of higher categories based on opetopic geometry. I developed a prototype implementation of such a system, called *Orchard*, in 2015. Since then, I have attempted to redesign the project for the web. The result is currently online ([10]), but not yet returned to a fully functional state.

Goal 12 *Finish developing a higher categorical proof assistant based on opetopes.*

2.4.2 Higher Grammars

A second application of opetopes in computer science is to the theory of formal languages. It has been known for some time that there is a natural family of language classes interpolating between the context-free languages and context-sensitive ones. The first family beyond the context-free languages in the hierarchy are sometimes called *mildly context-senesitive* and are captured, for example, by *tree adjoining grammars*. The linguist James Rogers showed that it is possible to capture this hierarchy using a notion of higher-dimensional tree [16].

Using opetopic diagrams, we can in fact give a graphical notation for these higher dimensional grammars. It has been a long standing goal of mine to develop this theory in more detail. For example, can one write an algorithm, possibly generalizing the Earley or CYK algorithms, for parsing such higher dimensional grammars? What are the complexities of such algorithms?

Goal 13 *Develop a theory of higher dimensional grammars based on opetopes.*

2.5 Laboratories of Insertion

Laboratoire des Sciences du Numérique à Nantes (LS2N) (UMR6004)
Laboratoire d'Informatique de l'Ecole Polytechnique (LIX) (UMR7161)
Institut de Recherche en Informatique Fondamentale (IRIF) (UMR8243)

References

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005.

[2] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *ACM SIGPLAN Notices*, volume 51, pages 18–29. ACM, 2016.

[3] M. Anel, G. Biedermann, E. Finster, and A. Joyal. A Generalized Blakers-Massey Theorem. `https://arxiv.org/abs/1703.09050`, March 2017.

[4] Danil Annenkov, Paolo Capriotti, and Nicolai Kraus. Two-level type theory and applications. *CoRR*, abs/1705.03307, 2017.

[5] Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. *Math. Proc. Cambridge Philos. Soc.*, 146(1):45–55, 2009.

[6] John C Baez and James Dolan. Higher-dimensional algebra iii. n-categories and the algebra of opetopes. *Advances in Mathematics*, 135(2):145–206, 1998.

[7] James Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, 2009.

[8] Eric Finster. Higher algebra in type theory. `https://github.com/ericfinster/higher-alg`.

[9] Eric Finster. Notes on a definition of higher structure. `http://ericfinster.github.io/`, 2018.

[10] Eric Finster. Opetopic. `http://opetopic.net/`, 2018.

[11] Eric Finster. Towards higher universal algebra in type theory. `https://www.youtube.com/watch?v=hlCVHVtAlqQ`, 2018.

[12] Nicola Gambino and Joachim Kock. Polynomial functors and polynomial monads. In *Mathematical proceedings of the cambridge philosophical society*, volume 154, pages 153–192. Cambridge University Press, 2013.

[13] D. Gepner, R. Haugseng, and J. Kock. ∞-Operads as Analytic Monads. *ArXiv e-prints*, December 2017.

[14] Peter LeFanu Lumsdaine and Mike Shulman. Semantics of higher inductive types. *arXiv preprint arXiv:1705.07088*, 2017.

[15] E. Rijke, M. Shulman, and B. Spitters. Modalities in Homotopy Type Theory. *ArXiv e-prints*, June 2017.

[16] James Rogers. Syntactic structures as multi-dimensional trees. *Research on Language and Computation*, 1(3-4):265–305, 2003.

[17] Mike Shulman. Homotopy type theory should eat itself. `https://homotopytypetheory.org/2014/03/03/hott-should-eat-itself/`, 2014.

[18] Philip Wadler. Propositions as types. *Communications of the ACM*, 58(12):75–84, 2015.